CSC4005 Assignment 1 Report

Name: Ran Hu

Student ID: 116010078

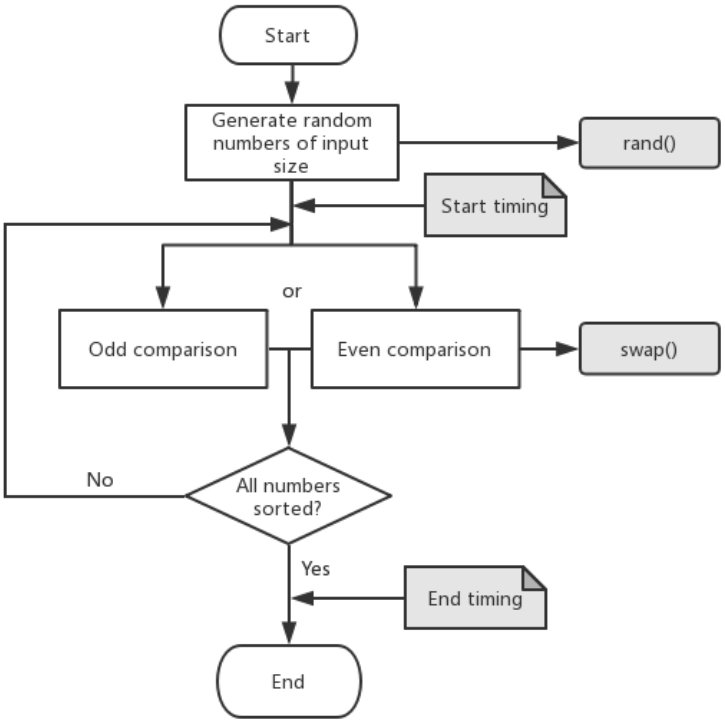
Introduction

Unlike bubble sort, odd-even (transposition) sort is not a sequential algorithm. It uses multiple processes to create a pipline of sorting. In this assignment, I implemented both the sequential and MPI version of odd-even sort. Also, the performances of different input array sizes and various numbers of processors are compared and analyzed.

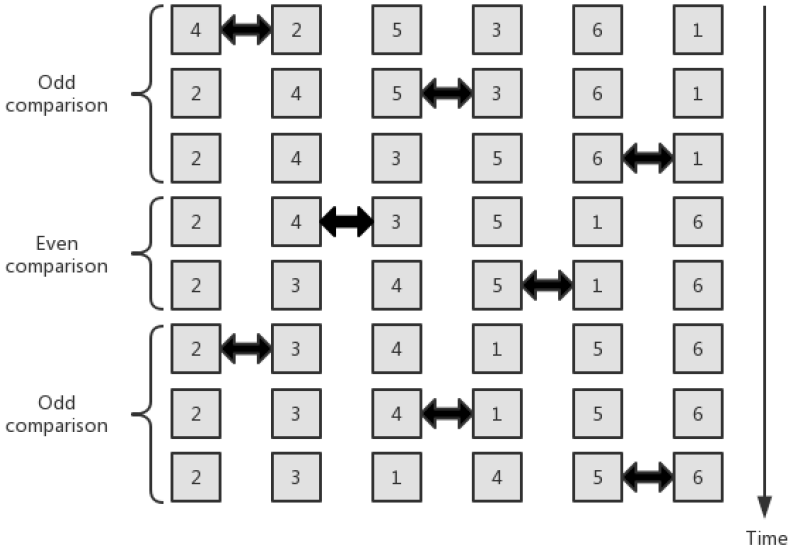
Program Design

I created two flow charts to describe the structures of my sequential program and MPI program.

Sequential Program



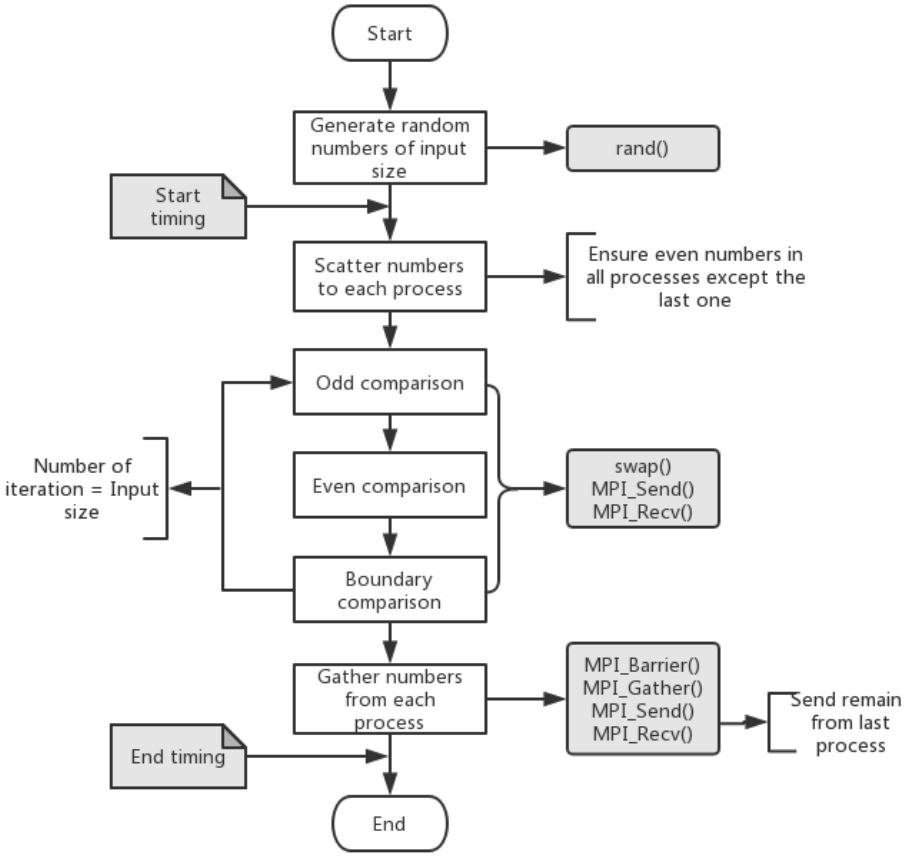
In the main function, I first generate random numbers. In order to make the input array the same for all experiments, I set a specific seed. The seed can be changed if I want to test different input arrays. Then the major information and the original array are printed out.



The major part is the sorting process. I set a flag `FINISH` to test whether the array has been sorted. If `FINISH == 0`, there will be an odd comparison or even comparison along the whole array (see the above figure). The odd comparison and even comparison are done alternately by changing the value of `ODD_EVEN` flag.

In the end, the sorted array and the running time will be printed out.

MPI Program



First, I use a certain seed to generate random numbers. Although it is not exactly "random", I would like to use the seed to ensure that all the processes get the same original array, and all the experiments I do have the same objective condition. Then after the initialization, the timing will begin.

Next, I scatter the numbers to all processes. In order to keep all the possible inputs undergo the same sorting process, i.e., **odd comparison -> even comparison -> boundary comparison**, I consider two scenerios and make each processes except the last one have even number of elements. Two features of my input array size are:

$$\begin{aligned} \text{common size} &= \text{input size} \,/\, \text{number of processes} \\ \text{remainder} &= \text{input size} \,\%\, \text{number of processes} \end{aligned}$$

1. If the common size is odd,

   $$\text{local size} = \text{common size} + 1$$

   In this case, the remainder can be contained in the local array. Since some processes may not have enough numbers to fill the local size, I complete empty positions using the largest possible number "1000", which is the upper bound of my input numbers. The resulting local array size is like this:

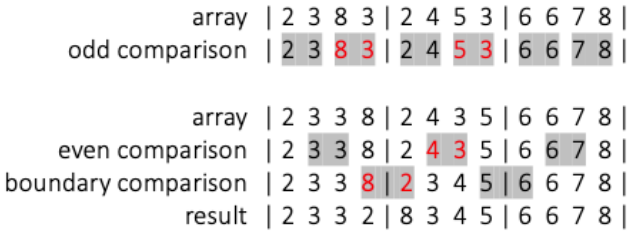
   $$| \text{odd} + 1 \,|\, \text{odd} + 1 \,|\, \text{odd} + 1 \,|\, \ldots \,|\, \text{xxx}, \text{xxx}, \ldots, 1000, 1000 \,|$$

2. If the common size is even, the common local size is just the local size. If the remainder > 0, I will put the remaining numbers into the last process. The resulting local array size is like this:

   $$| \text{even} \,|\, \text{even} \,|\, \text{even} \,|\, \ldots \,|\, \text{even} + \text{remainder} \,|$$

In this way, I can sort all kinds of input size using the same sorting order.

After printing out major information, the sorting process begins.



There are input array size of iterations in total. In each iteration, the operations in the above figure will happen. On the boundary, the right element will first be sent, and recieved by the previous process. After being compared with the left element, the previous process will send an element, either itself or the recieved one to the next process.

When the iteration has ended, I gather the data of same size from each process. Since in the second case, I may have some remainders that make the last local array especially large, I will send the remaining elements to the root process. I also eliminate the "1000"s that are added in the array, which are arranged in the end and can be abandoned easily.

The timing ends after gathering all the data. Related information will be printed out.

Instructions

In my submitted file `116010078.zip`, there are two source files.

Sequential Program

To run the sequential implementation `odd_even_seq.cpp`:

1. Compile the C++ source code `odd_even_seq.cpp` and generate an executable file `odd_even_seq` (you can name whatever you want) using

   ```
   $ g++ odd_even_seq.cpp -o odd_even_seq
   ```

2. Execute the executable file `odd_even_seq` and give an array input size

   ```
   $ ./odd_even_seq $INPUT_SIZE$
   ```

   For example, if I want to sort an array of 20 numbers, simply type

   ```
   $ ./odd_even_seq 20
   ```

   And the output will be printed out.

MPI Program

To run the MPI implementation `odd_even_MPI.cpp` on the server:

1. Compile the C++ source code `odd_even_MPI.cpp` and generate an executable file `odd_even_MPI` (you can name whatever you want) using

   ```
   $ mpic++ -o odd_even_MPI odd_even_MPI.cpp
   ```

2. Then create a script file `MPI_script.pbs` to run all the MPI case

   ```
   #!/bin/bash
   #PBS -l nodes=1:ppn=5,mem=1g,walltime=72:00:00
   #PBS -q batch
   #PBS -m abe
   #PBS -V
   #PBS -e /code/116010078/MPI_error.txt
   #PBS -o /code/116010078/MPI_out.txt

   for i in {2..16}
   do
       for j in 20 1000 100000
       do
           timeout 60 mpiexec -n $i -f /home/mpi_config
   /code/116010078/odd_even_MPI $j
       done
   done
   ```

   where `/code/116010078/odd_even_MPI` specifies the executable file. The `$i` value is the number of processes, and `$j` value is the input array size. In my case, I tested array size of 20, 1000 and 100000, using 2 to 16 processes, respectively.

3. Run the script. The output will be generated in `MPI_out.txt` file; the error will be generated in `MPI_error.txt`

```
$ qsub MPI_script.pbs
```

Performance Analysis

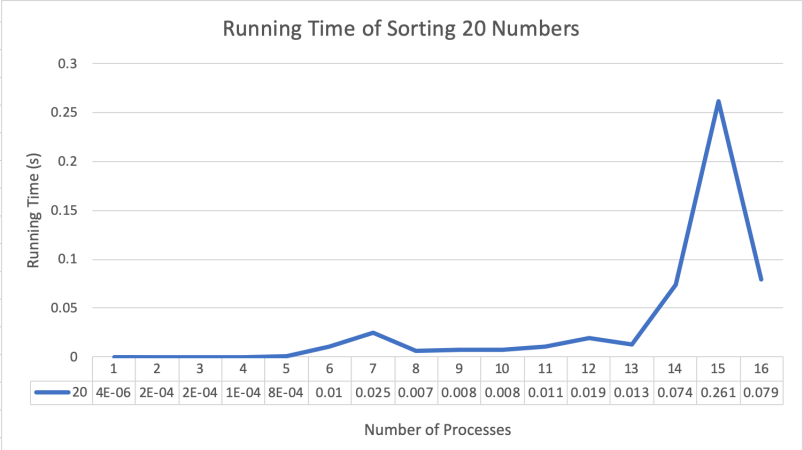
Below are the running time data I collected from all the experiments. Each column uses the same number of processes, i.e., the "column 1" is the sequential implementation and from "column 2" to "column 16" are the MPI implementations. Each row is of the same input array size, i.e., 20, 1000, 100000.

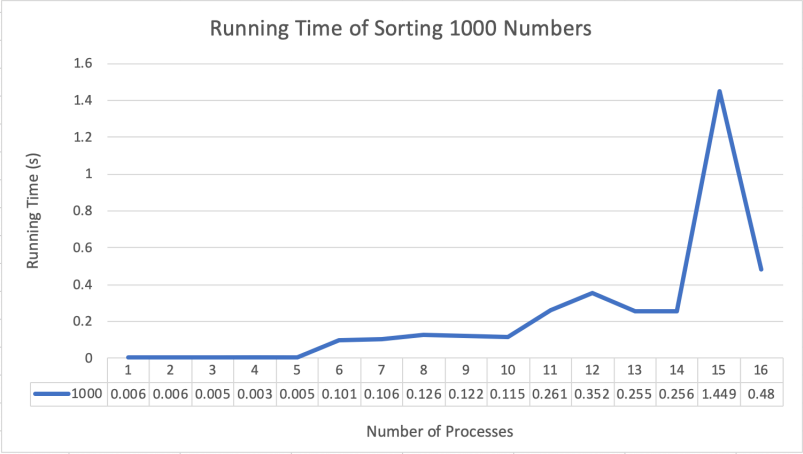
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
20	4E-06	0.00021	0.00017	0.00012	0.00083	0.01037	0.02476	0.00684	0.00763	0.00794	0.01119	0.01892	0.01261	0.07384	0.26133	0.0793
1000	0.006	0.00607	0.00512	0.00345	0.00464	0.1007	0.10568	0.12643	0.12225	0.115	0.26138	0.3517	0.25481	0.25559	1.44946	0.48
100000	34.8969	30.4746	21.8934	17.0201	10.7525	24.1268	23.291	19.7522	17.4877	15.7508	26.6247	24.8964	23.8862	51.7391	26.4747	37.5304

Table 1. Running Time Table

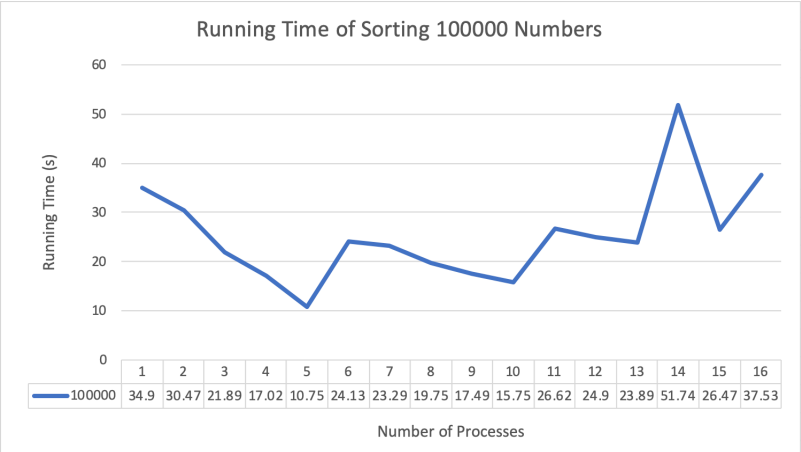
Light yellow color denotes that the running time is relatively short; the darker the color, the longer the running time; the red one is the longest.

As can be see clearly in the table, the general trend is that the larger the input array size, the longer the running time. But how does different numbers of processes influence the sorting time? I ploted three graphes for different input array sizes (small: 20; median: 1000; large: 100000):





When the input array number is small or median, although the iteration number is greatly reduced, more processes will make the running time longer. This is probably because the communication time between processes are much longer than the operation time in a single process.



When the input array size is large, the performance becomes interesting. Almost every 5 processes, the running time displays a ladder-like shape. This is because each machine has 5 cores, and each process may run on a single core. Within 1-5, 5-10... processes, more processes will make the sorting more effecient; when the process number exceed 5, 10..., the communication between different processes will need to across different machines, which is longer than that on a single machine.

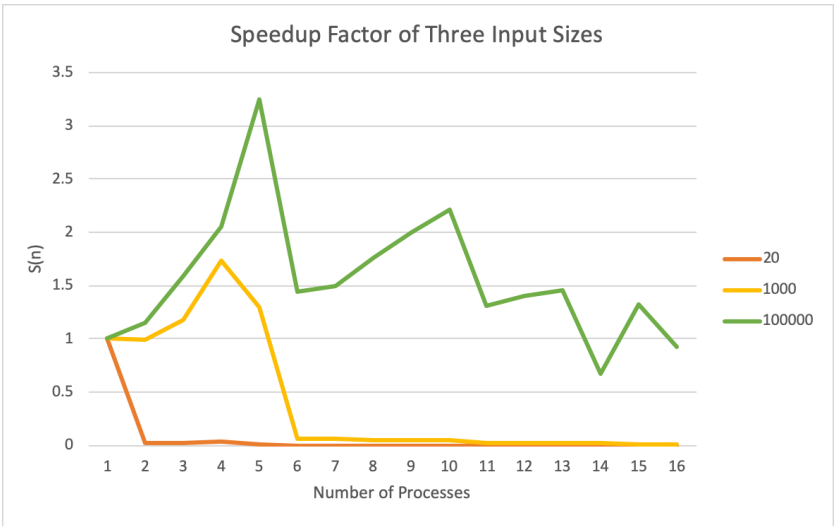
Furthermore, I calculated the speedup factor

$$S(n) = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor with n processors}} = \frac{t_s}{t_p}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
20	1	0.01951	0.02381	0.03419	0.00485	0.00039	0.00016	0.00058	0.00052	0.0005	0.00036	0.00021	0.00032	5.4E-05	1.5E-05	5E-05
1000	1	0.98846	1.17106	1.74028	1.29385	0.05955	0.05675	0.04743	0.04906	0.05215	0.02294	0.01705	0.02354	0.02346	0.00414	0.01249
100000	1	1.14512	1.59395	2.05034	3.24548	1.4464	1.4983	1.76674	1.99551	2.21557	1.3107	1.40168	1.46096	0.67448	1.31812	0.92983

Table 2. Speedup Factor Table

As can be clearly see, the larger the input array size, the greater the speedup.



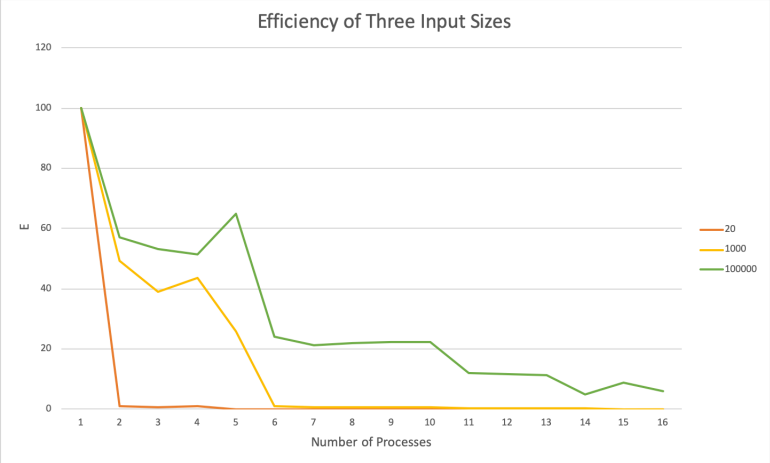
When the input array size is small or median, more than 2 or 5 processes will not have much speedup. When the input array size is large, increasing the process number from 1 to 5 will increase the speedup, but more processes do not have better performance.

Also, I calculated the efficiency

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}} = \frac{t_s}{t_p \times n} = \frac{S(n)}{n} \times 100\%$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
20	100	0.97561	0.79365	0.8547	0.09697	0.00643	0.00231	0.00731	0.00583	0.00504	0.00325	0.00176	0.00244	0.00039	0.0001	0.00032
1000	100	49.4231	39.0353	43.507	25.877	0.99251	0.81069	0.59291	0.54508	0.52147	0.20858	0.1421	0.18104	0.1676	0.02758	0.07809
100000	100	57.2558	53.1317	51.2584	64.9096	24.1066	21.4043	22.0842	22.1723	22.1557	11.9154	11.6807	11.2382	4.81771	8.7875	5.81145

Table 3. Efficiency Table



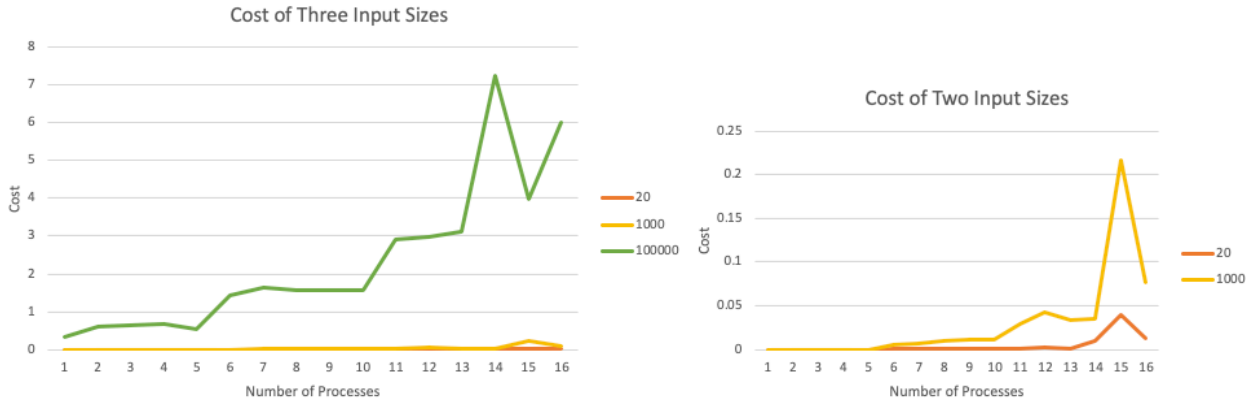
The larger the input array size, the greater the efficiency. And the general trend of efficiency is decreasing as the number of processes increases. This is because more processes will consume more resources.

Additionally, I calculated the cost

$$\text{Cost} = (\text{execution time}) \times (\text{total number of processors used}) = \frac{\mathrm{t_s n}}{\mathrm{S(n)}} = \frac{\mathrm{t_s}}{\mathrm{E}}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
20	4E-08	4.1E-06	5E-06	4.7E-06	4.1E-05	0.00062	0.00173	0.00055	0.00069	0.00079	0.00123	0.00227	0.00164	0.01034	0.0392	0.01269
1000	6E-05	0.00012	0.00015	0.00014	0.00023	0.00604	0.0074	0.01011	0.011	0.0115	0.02875	0.0422	0.03312	0.03578	0.21742	0.0768
100000	0.34897	0.60949	0.6568	0.6808	0.53762	1.44761	1.63037	1.58018	1.5739	1.57508	2.92872	2.98757	3.10521	7.24347	3.9712	6.00486

Table 4. Cost Table



The cost of three input sizes are shown on the left figure, since the trends of line of 20 and 1000 cannot be seen clearly, I ploted them on the right figure. Generally, the cost is increasing as the number of processes increases, but I am not sure about the performance trend when there are more processes. Also, the experiment may have some bias since for each input size and process number, I only tested once on the same input array.

Results

Here I printed the output of two implementations sorting the same 20 random numbers. For MPI implementation, the number of processes range from 2 to 16.

Sequential Program

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- Sequential Implementation

Sorting 20 random numbers

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.000004
```

MPI Program

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 2 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.000205
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 3 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.000168
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 4 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.000117
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 5 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.000825
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 6 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.010373
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 7 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.024763
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 8 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.006844
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 9 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.007625
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 10 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.007936
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 11 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.011185
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 12 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.018922
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 13 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.012607
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 14 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.073840
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 15 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.261332
```

```
Name: Ran Hu
Student ID: 116010078
Assignment 1: Odd-Even Transposition Sort -- MPI Implementation

Sorting 20 random numbers using 16 processors

Orignal array: 637 130 172 145 916 873 100 46 869 83 197 562 226 847 10 937 16 109 105 539

Sorted array: 10 16 46 83 100 105 109 130 145 172 197 226 539 562 637 847 869 873 916 937

Running time: 0.079297
```

Conclusion

In this assignment, I implemented the sequetial and MPI version of odd-even transposition sort. When the input array size is not large enough, parallel computing seldom reduces the running time; when the input array size is large, using more processes on a single machine will improve the performance. However, the cost of communication between different processes and different machines should be weighed in order to gain the best performance. All in all, speedup is small when the number of processes is large, indicating that this algorithm cannot achieve superlinear speedup.

Sorce Code

Sequential Program

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <chrono>
#include <vector>

using namespace std;
using namespace chrono;

const size_t ROOT = 0;

int swap(int *array, int i) {
    int tmp = array[i];
    array[i] = array[i+1];
    array[i+1] = tmp;
    return 0;
}

int main(int argc, char **argv) {
    string argvi(argv[1]);
    int global_size = atoi(argv[1]);
    int *global_array = (int *)malloc(sizeof(int)*global_size);
    int ODD_EVEN = 0;
    int FINISH = 0;

    /* generate random numbers to create array data of input size */

    srand(2019);  //set the seed
    for (int i = 0; i < global_size; i++) {
        global_array[i] = ((int)rand()) % 1000;  //generate random numbers
smaller than 1000
    }

    printf("\nName: Ran Hu\n");
    printf("Student ID: 116010078\n");
    printf("Assignment 1: Odd-Even Transposition Sort -- Sequential
Implementation\n\n");
    printf("Sorting %d random numbers\n\n", global_size);
    printf("Orignal array: ");
    for (int i = 0; i < global_size; i++) {
        printf("%d ", global_array[i]);
    }

    auto start_time = system_clock::now();
```

```
    /* sorting process */

    while (FINISH != 1){
        FINISH = 1;
        for (int i = ODD_EVEN; i < global_size-1; i += 2){
          if (global_array[i] > global_array[i+1]) {
          FINISH = 0;
              swap(global_array, i);
            }
        }
        ODD_EVEN = !ODD_EVEN;
    }

    auto end_time = system_clock::now();
    auto duration = duration_cast<microseconds>(end_time - start_time);

    printf("\n\nSorted array: ");
    for (int i = 0; i < global_size; i++) {
        printf("%d ", global_array[i]);
    }
    printf("\n\nRunning time: %f\n\n", double(duration.count()) *
microseconds::period::num / microseconds::period::den);

    return 0;
}
```

MPI Program

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <vector>

using namespace std;

const size_t ROOT = 0;

int swap(int *array, int i) {
    if (array[i] > array[i+1]) {
        int tmp = array[i];
        array[i] = array[i+1];
        array[i+1] = tmp;
    }
    return 0;
}
```

```
int main(int argc, char **argv) {
    int global_size = atoi(argv[1]);
    int number_of_processors, rank_of_processor, actual_global_size;
    int *global_array = (int *)malloc(sizeof(int)*global_size);
    int new_sorted_global_array[global_size];
    int remainder, local_size_max, local_size;
    int send_right, recv_right, send_left, recv_left;
    double start_time, end_time;

    /* generate numbers to create array data of input size */

    srand(2019);  //set the seed
    for (int i = 0; i < global_size; i++) {
        global_array[i] = ((int)rand()) % 1000;  //generate random numbers
smaller than 1000
    }

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_processor);

    start_time = MPI_Wtime();

    if (number_of_processors <= 1) {
        printf("\nError! Number of processors must be larger than 1\n" );
        MPI_Finalize();
        exit(0);
    }

    /* make each processor have even number of numbers
     * the last processor has the remaining numbers */

    remainder = global_size % number_of_processors;
    local_size = global_size / number_of_processors;
    local_size_max = local_size;
    actual_global_size = global_size;
    if (local_size % 2 == 1) {
      // case 1 array: |odd+1|odd+1|...|XXX+n*1000|
        local_size += 1;
        local_size_max = local_size;
        actual_global_size = local_size * number_of_processors;
    }
    else if (rank_of_processor == number_of_processors-1) {
        local_size_max = local_size + remainder;
      // case 2 array:|even|even|...|even+remainder|
    }
    int *sorted_global_array = (int *)malloc(sizeof(int)*actual_global_size);
```

```
    /* scatter the global array data to the local array in each process */

    int *local_array = (rank_of_processor == number_of_processors-1)
            ? (int *)malloc(sizeof(int)*local_size_max) : (int
*)malloc(sizeof(int)*local_size);
    for (int i = 0; i < local_size_max; i++) {
        if (local_size*rank_of_processor+i < global_size) {
            local_array[i] = global_array[local_size*rank_of_processor+i];
        }
        else{
            local_array[i] = 1000;  // 1000 is the maximum number
        }
    }

    if (rank_of_processor == ROOT) {
        printf("\nName: Ran Hu\n");
        printf("Student ID: 116010078\n");
        printf("Assignment 1: Odd-Even Transposition Sort -- MPI
Implementation\n\n");
        printf("Sorting %d random numbers using %d processors\n\n",
global_size, number_of_processors);
        printf("Orignal array: ");
        for (int i = 0; i < global_size; i++) {
            printf("%d ", global_array[i]);
        }
    }

    /* sorting process */

    for (int iteration_number = 0; iteration_number < global_size;
iteration_number++) {

        /* odd comparison */

        if (iteration_number % 2 == 1) {
            for (int local_array_index = 0; local_array_index <
local_size_max/2; local_array_index++) {
                swap(local_array, 2*local_array_index);
            }
        }

        /* even comparison & boundary comparison*/

        else {
            if (rank_of_processor == number_of_processors-1) {
                send_left = local_array[0];
                MPI_Send(&send_left, 1, MPI_INT, rank_of_processor-1, 0,
MPI_COMM_WORLD);
            } else if (rank_of_processor != ROOT) {
```

```
                MPI_Recv(&recv_right, 1, MPI_INT, rank_of_processor+1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                send_left = local_array[0];
                MPI_Send(&send_left, 1, MPI_INT, rank_of_processor-1, 0,
MPI_COMM_WORLD);
            } else if (rank_of_processor == ROOT && number_of_processors != 1)
{
                MPI_Recv(&recv_right, 1, MPI_INT, rank_of_processor+1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            for (int local_array_index = 0; local_array_index <
local_size_max/2; local_array_index++) {
                if (local_size_max % 2 != 0) {
                    swap(local_array, 2*local_array_index+1);
                }
                else {
                    if (local_array_index < local_size_max/2-1) {
                        swap(local_array, 2*local_array_index+1);
                    }
                    else if (rank_of_processor != number_of_processors-1) {
                         if (local_array[local_size-1] > recv_right) {
                             send_right = local_array[local_size-1];  //swap
the boundary
                             local_array[local_size-1] = recv_right;
                         }
                         else send_right = recv_right;
                    }
                }
            }
            if (rank_of_processor == ROOT) {
                MPI_Send(&send_right, 1, MPI_INT, rank_of_processor+1, 0,
MPI_COMM_WORLD);
            } else if (rank_of_processor != number_of_processors-1) {
                MPI_Recv(&recv_left, 1, MPI_INT, rank_of_processor-1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                local_array[0] = recv_left;
                MPI_Send(&send_right, 1, MPI_INT, rank_of_processor+1, 0,
MPI_COMM_WORLD);
            } else if (rank_of_processor == number_of_processors-1) {
                MPI_Recv(&recv_left, 1, MPI_INT, rank_of_processor-1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                local_array[0] = recv_left;
            }
        }
    }

    /* gather all the data */

    MPI_Barrier(MPI_COMM_WORLD);
```

```
    MPI_Gather(local_array, local_size, MPI_INT, sorted_global_array,
local_size, MPI_INT, 0, MPI_COMM_WORLD);

    int *send_remainder = (int *)malloc(sizeof(int)*remainder);
    int *recv_remainder = (int *)malloc(sizeof(int)*remainder);
    if (((global_size/number_of_processors)%2 == 0) && (remainder > 0)) {
        if (rank_of_processor == number_of_processors-1) {
            for (int i = 0; i < remainder; i++) {
                send_remainder[i] = local_array[local_size + i];
            }
            MPI_Send(send_remainder, remainder, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        else if (rank_of_processor == ROOT) {
            MPI_Recv(recv_remainder, remainder, MPI_INT, number_of_processors-
1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = 0; i < remainder; i++) {
                sorted_global_array[global_size-remainder+i] =
recv_remainder[i];
            }
        }
    }
    for (int i = 0; i < global_size; i++) {
        new_sorted_global_array[i] = sorted_global_array[i];
    }

    MPI_Barrier(MPI_COMM_WORLD);
    end_time = MPI_Wtime();

    if (rank_of_processor == ROOT) {
        printf("\n\nSorted array: ");
        for (int i = 0; i < global_size; i++) {
            printf("%d ", new_sorted_global_array[i]);
        }
        printf("\n\nRunning time: %f\n\n", end_time - start_time);
    }

    MPI_Finalize();
    return 0;
}
```