CSC 4005 Assignment 1

Parallel Odd-Even Transposition Sort

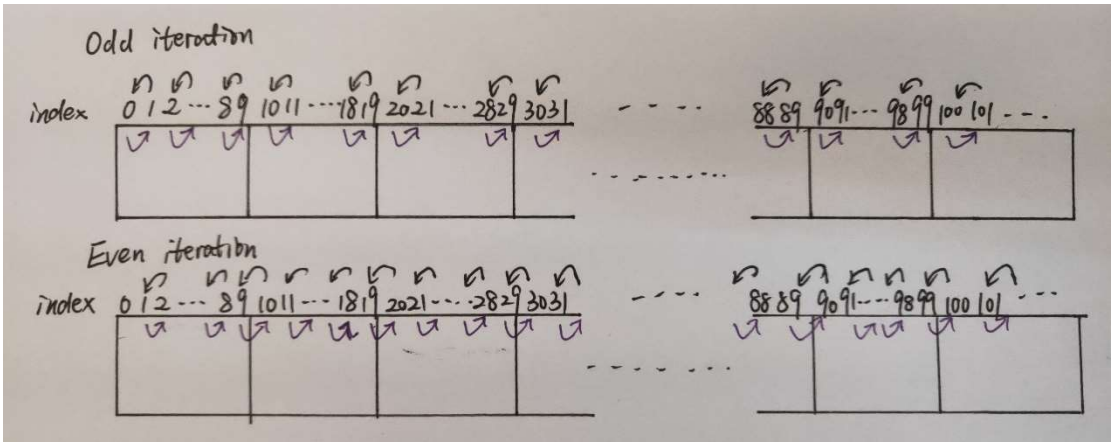
Name: Zixuan Yao

Student ID: 115010267

Methods & Program Design

The program is required to implement Parallel Odd-Even Transposition Sort, which is to separate the original array of m elements into n processors. MPI library is used to communicate between processors.

To avoid complicated discussions of situations, I adopt a method to ensure that there's always a even number of elements in the first n-1 processors. In this case, processors only need to communicate with their neighbors at even iteration, and do not need to communicate with each other at odd iteration, as illustrated in the figure below.



Based on this idea, the implementation of the program is intuitive. First, generate a random array of given size m. Second, scatter the array to n processors with even number of elements in first n-1 processors. Third, odd-even sort the array m times. At last, the gather all the sorted local_array to the master processor and print the result and execution time.

After m steps, the array is guaranteed to be sorted. The detailed source code is at the end of the report.

Instructions

Please compile the source code (attached at the end of the report) under Windows desktop C++ environment with Microsoft MPI environment. Please close the debug function when compiling, otherwise, there will be some fault preventing you from successful compile.

To run the program, please go to the direction where the compiled exe file locates and type the following command.

```
C:\Users\12941\source\repos\Project1\x64\Debug>mpiexec -n 4 Project1.exe 40
```

It means to run the program with four processors to sort a random array of size 40.

mpiexec -n $number of processors Project1.exe $size of array

Results

The demo result is shown below to sort the random array of size 50 by 4 processors.

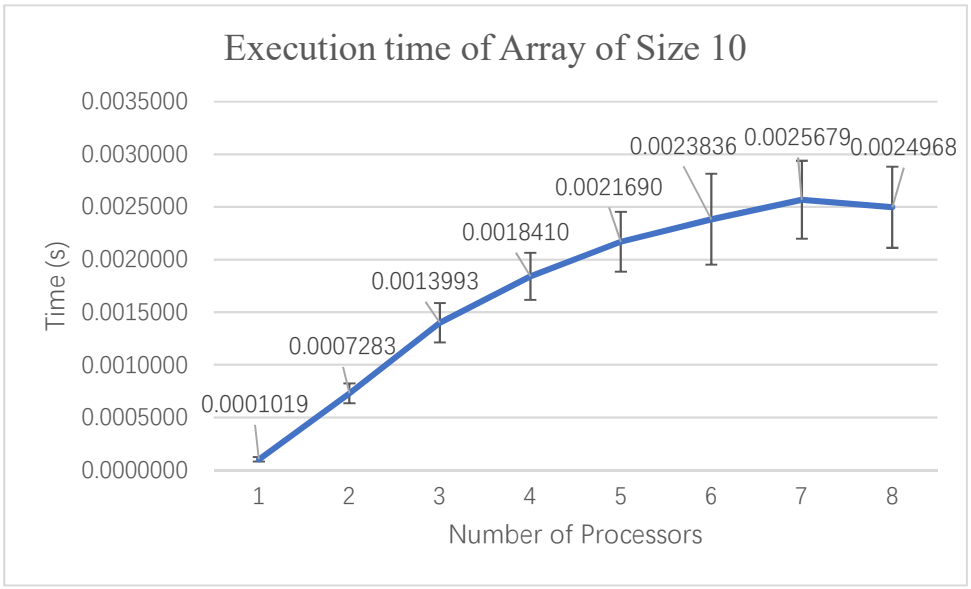
```
C:\Users\12941\source\repos\Project1\x64\Debug>mpiexec -n 4 Project1.exe 50

number of tasks= 4 my rank= 3 local_size= 14 running on DESKTOP-TQHSBN4 Local array: 667 299 35 894 703 81
1 322 333 673 664 141 711 253 868

number of tasks= 4 my rank= 2 local_size= 12 running on DESKTOP-TQHSBN4 Local array: 292 382 421 716 718 8
95 447 726 771 538 869 912

number of tasks= 4 my rank= 1 local_size= 12 running on DESKTOP-TQHSBN4 Local array: 281 827 961 491 995 9
42 827 436 391 604 902 153

number of tasks= 4 my rank= 0 local_size= 12 running on DESKTOP-TQHSBN4 Local array: 41 467 334 500 169 72
4 478 358 962 464 705 145

Name : Yao ZIuxan
ID : 115010267
Sorting 50 random numbers using 4 processors
Random Numbers Generating...

Orignal array: 41 467 334 500 169 724 478 358 962 464 705 145 281 827 961 491 995 942 827 436 391 604 902
153 292 382 421 716 718 895 447 726 771 538 869 912 667 299 35 894 703 811 322 333 673 664 141 711 253 868


Sorting...

Sorted array: 35 41 141 145 153 169 253 281 292 299 322 333 334 358 382 391 421 436 447 464 467 478 491 50
0 538 604 664 667 673 703 705 711 716 718 724 726 771 811 827 827 868 869 894 895 902 912 942 961 962 995
total time : 0.001855

C:\Users\12941\source\repos\Project1\x64\Debug>
```

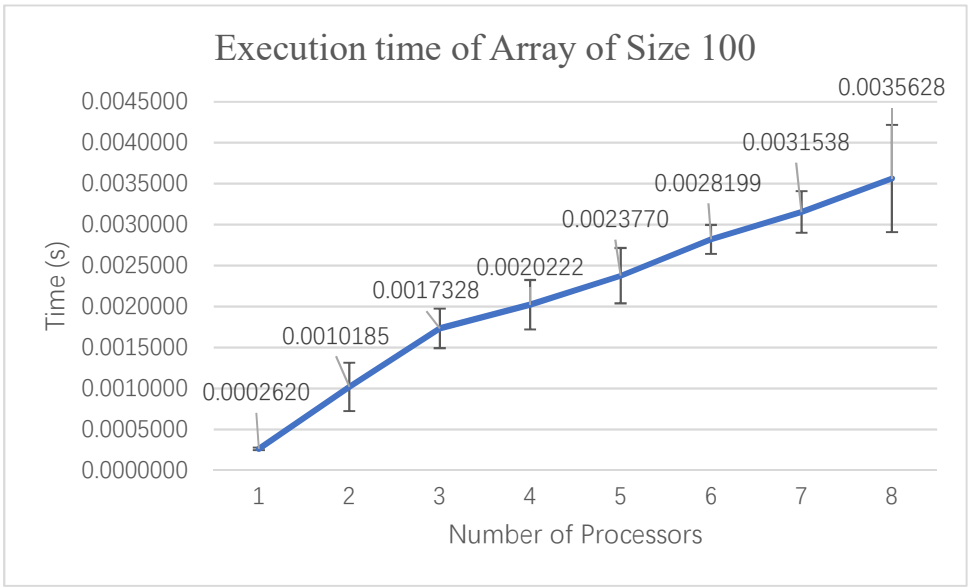
Performance Analysis

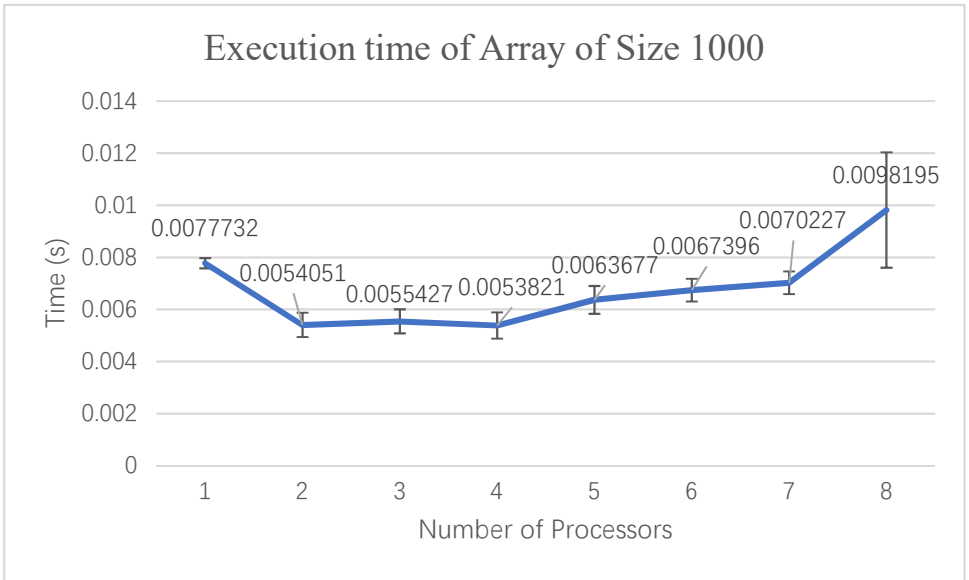
To do the performance analysis, we need to first need to collect all the performance data and try to eliminate the variance of the data, because the time of sorting the array using MPI contains randomness. Generally, the more processors are used and the smaller the array is, the larger the relative variance is. So, I did various experience to calculate the average of the execution time and variance. The variance is shown by the black bar on the following 5 graphs.

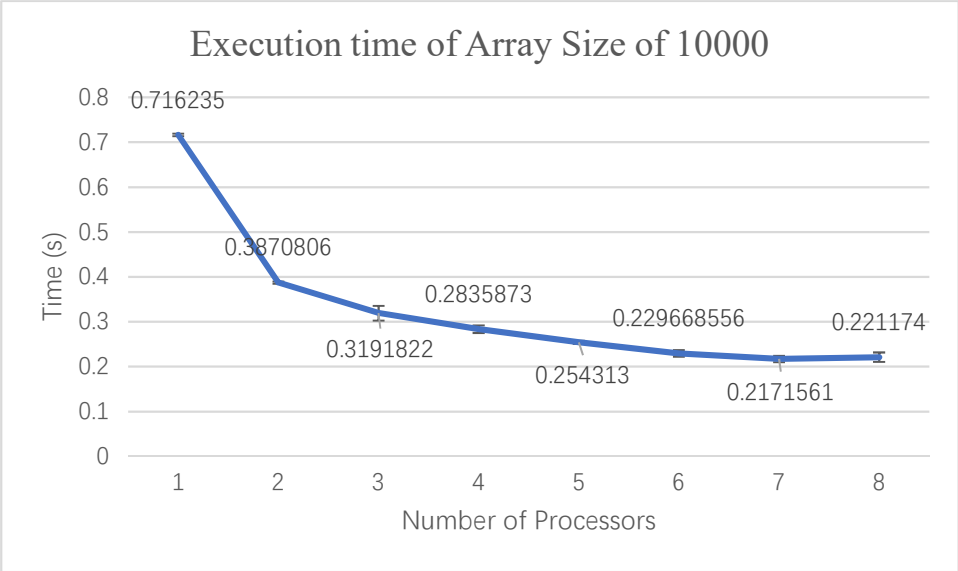
Average execution time

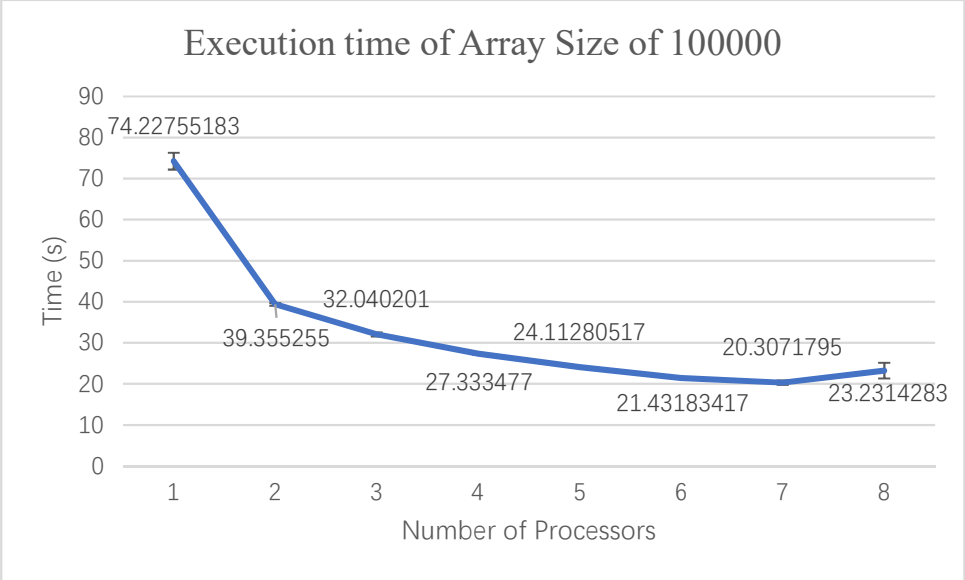
Processors numbers / Size of array	1	2	3	4	5	6	7	8
10	0.000102	0.000728	0.001399	0.001841	0.002169	0.002384	0.002568	0.002497
100	0.000262	0.001019	0.001733	0.002022	0.002377	0.002820	0.003154	0.003563
1000	0.007773	0.005405	0.005543	0.005382	0.006368	0.006740	0.007023	0.009820
10000	0.716235	0.387081	0.319182	0.283587	0.254313	0.229669	0.217156	0.221174
100000	74.227552	39.355255	32.040201	27.333477	24.112805	21.431834	20.307180	23.231428


Execution time of Array of Size 10
Time (s)
0.0035000
0.0030000
0.0025000
0.0020000
0.0015000
0.0010000
0.0005000
0.0000000
0.0001019
0.0007283
0.0013993
0.0018410
0.0021690
0.0023836
0.0025679
0.0024968
1
2
3
4
5
6
7
8
Number of Processors


Execution time of Array of Size 100
Time (s)
0.0045000
0.0040000
0.0035000
0.0030000
0.0025000
0.0020000
0.0015000
0.0010000
0.0005000
0.0000000
0.0002620
0.0010185
0.0017328
0.0020222
0.0023770
0.0028199
0.0031538
0.0035628
1
2
3
4
5
6
7
8
Number of Processors


Execution time of Array of Size 1000
Time (s)
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0.0077732
0.0054051
0.0055427
0.0053821
0.0063677
0.0067396
0.0070227
0.0098195
1
2
3
4
5
6
7
8
Number of Processors





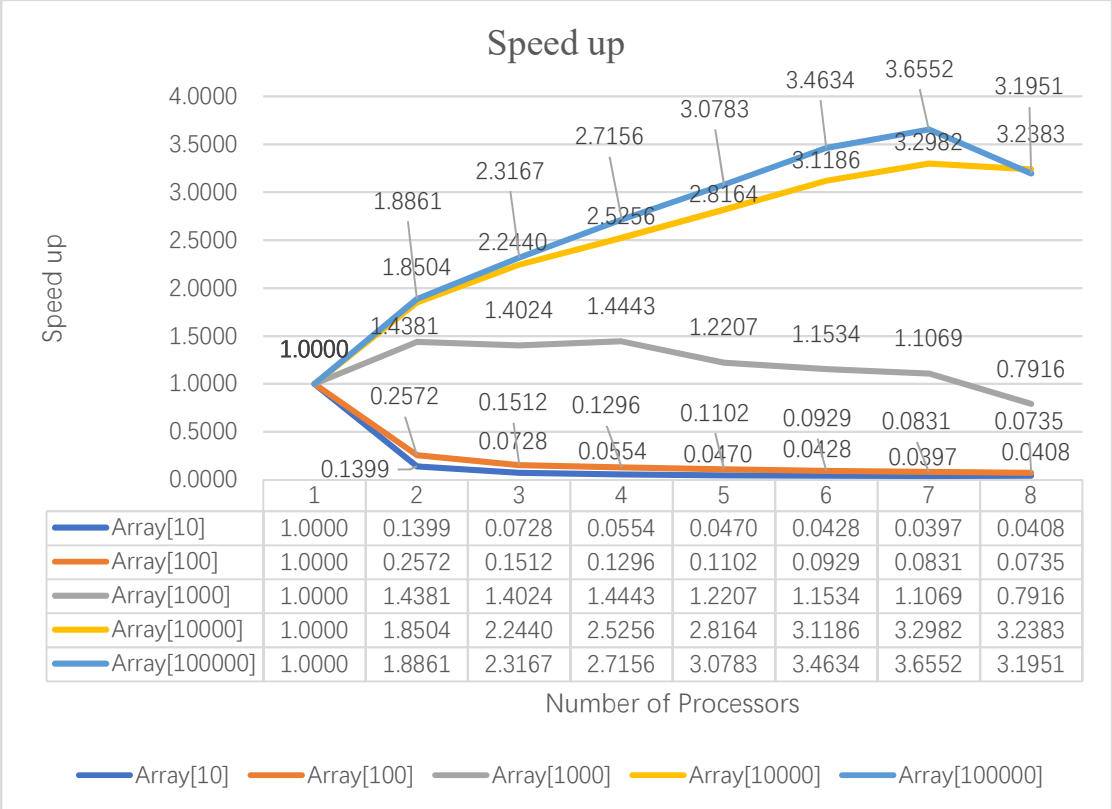
From the graphs, it can be easily observed that parallel computing on multi-processors won't speed up at all when array size is small, such as 10 and 100, because the communication between processors is generally slow and takes a lot of execution time. In this case, it is faster to run the sorting on a single processor.

When the array size is increased to 1000, the execution time will decrease as the number of processors increase from 1 to 4 but will increase as the number of processors increase from 5 to 8. I think it is because my computer is physically 4-cores 8-threads by Intel hyper-thread technology.

When the array size becomes even larger, increasing to 10000 and 100000, the execution time generally decrease as the number of processors increase from 1 to 7. Adding one more processor from 7 to 8 won't improve the performance. My explanation may be windows is still running the system and other background jobs except the sorting program, assigning task to all the 8 processors will take a little more time due to the running of windows operation system.

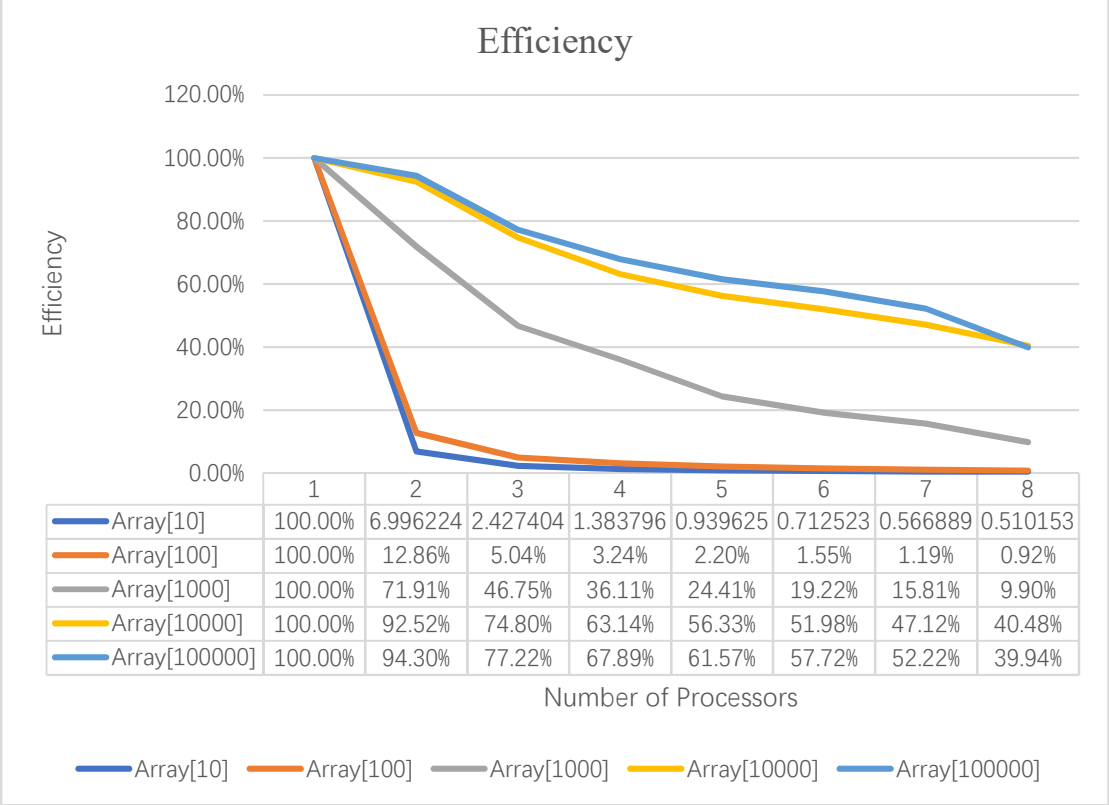
Furthermore, “Speed up”, “Efficiency” and “Cost” is calculated and compared.

Speed up



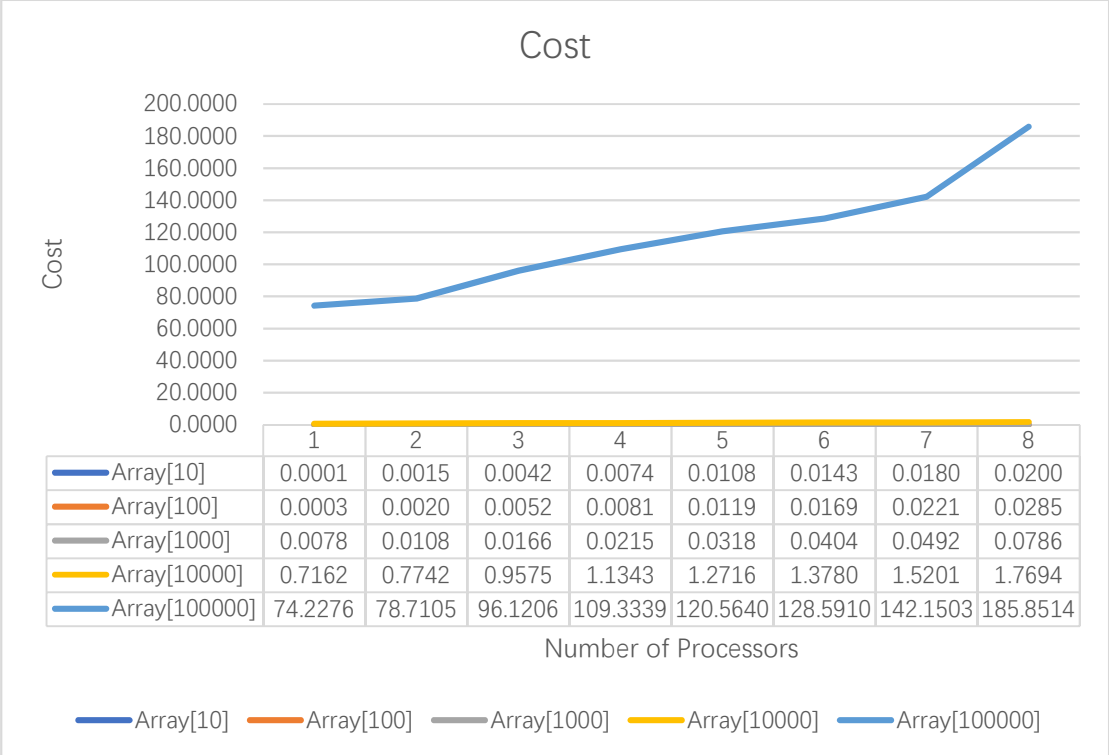
	1	2	3	4	5	6	7	8
Array[10]	1.0000	0.1399	0.0728	0.0554	0.0470	0.0428	0.0397	0.0408
Array[100]	1.0000	0.2572	0.1512	0.1296	0.1102	0.0929	0.0831	0.0735
Array[1000]	1.0000	1.4381	1.4024	1.4443	1.2207	1.1534	1.1069	0.7916
Array[10000]	1.0000	1.8504	2.2440	2.5256	2.8164	3.1186	3.2982	3.2383
Array[100000]	1.0000	1.8861	2.3167	2.7156	3.0783	3.4634	3.6552	3.1951

Efficiency



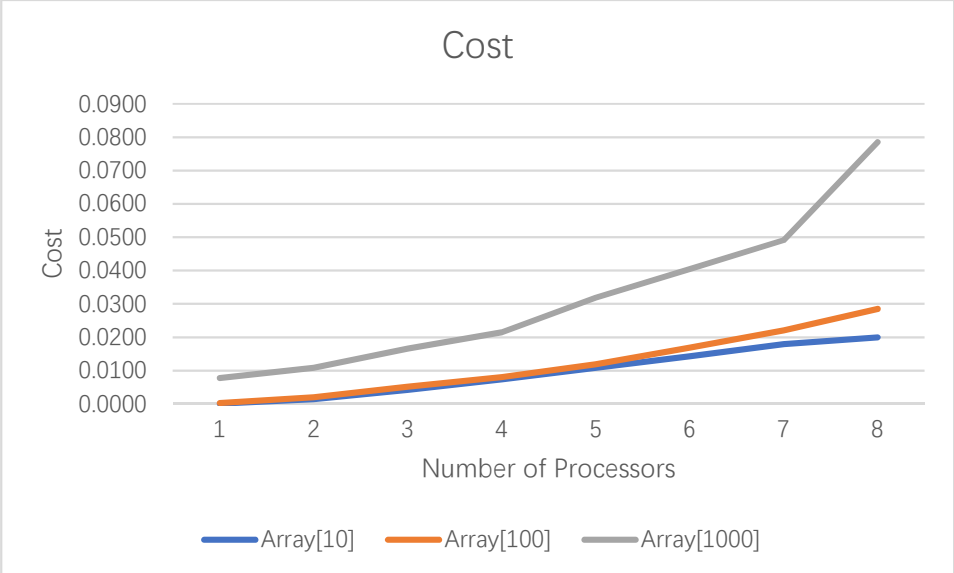
	1	2	3	4	5	6	7	8
Array[10]	100.00%	6.996224	2.427404	1.383796	0.939625	0.712523	0.566889	0.510153
Array[100]	100.00%	12.86%	5.04%	3.24%	2.20%	1.55%	1.19%	0.92%
Array[1000]	100.00%	71.91%	46.75%	36.11%	24.41%	19.22%	15.81%	9.90%
Array[10000]	100.00%	92.52%	74.80%	63.14%	56.33%	51.98%	47.12%	40.48%
Array[100000]	100.00%	94.30%	77.22%	67.89%	61.57%	57.72%	52.22%	39.94%

Cost



	1	2	3	4	5	6	7	8
Array[10]	0.0001	0.0015	0.0042	0.0074	0.0108	0.0143	0.0180	0.0200
Array[100]	0.0003	0.0020	0.0052	0.0081	0.0119	0.0169	0.0221	0.0285
Array[1000]	0.0078	0.0108	0.0166	0.0215	0.0318	0.0404	0.0492	0.0786
Array[10000]	0.7162	0.7742	0.9575	1.1343	1.2716	1.3780	1.5201	1.7694
Array[100000]	74.2276	78.7105	96.1206	109.3339	120.5640	128.5910	142.1503	185.8514

Array[10], Array [100] and Array[1000] cannot be seen from the big figure, which is shown in the following small figure.



The results of the three measures agree with the previous analysis.

The efficiency is always decreasing as the number of processors increases, because more processors consumes more sources and it is unable to superliner speedup the sorting.

The cost is always increasing as the number of processors increases, because this program is unable to superliner speedup the sorting.

Experience

1. MPI_Send & MPI_Recv should be written in pair and sequentially.
   If the MPI_Send & MPI_Recv are not written sequentially in pair, the compiler cannot correctly compile the code and the program will enter into deadlock, which this the most difficult part of writing the parallel program because you don't know where the error is.

2. Inside MPI_Gater & MPI_Scatter, sendcount and recvcount should be consistent between processors.

   MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
   void* recvbuf, int recvcount, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
   is equilibrium to MPI_Send by every process (including master process) to the master processor
   MPI_Send(sendbuf, sendcount, sendtype, root, ...),

   At the same time, master process executes MPI_Recv by n times (n is the number of process).
   MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount, recvtype, i,...),

   So, sendcount and recvcount should be consistent between processors. If the number of elements you need to gather is different processors, you need to take care of that specially, or use MPI_Gatherv function. It is the same for MPI_Scatter.

3. When generating a random array, you need to initialize a random seed first, otherwise the random number you generate is always the same.

Source Code

```
#include "mpi.h"
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string>
#include <string.h>
#include <time.h>

using namespace std;

int swap(int *num, int i, int j);

int main(int argc, char * argv[]) {
```

```
    string argvi(argv[1]);
    int size = atoi(argv[1]);
    int *array_in = (int *)malloc(sizeof(int)*size); // array in
    int *array_out = (int *)malloc(sizeof(int)*size); // array out
    int local_size, local_size_max, rem;
    int recv_right, send_left, send_right, recv_left;
    int numtasks, rank, len;
    double start, finish;
    double totaltime;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv); // initialize MPI
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // get number of tasks
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my rank
    MPI_Get_processor_name(hostname, &len); // this one is obvious

    //Generate random array
    for (int i = 0; i < size; i++) {
        array_in[i] = ((int)rand()) % 100;
        //printf("%d ", array_in[i]);
    }

    start = MPI_Wtime();

    rem = size % numtasks;
    local_size = size / numtasks;
    local_size_max = local_size;
    if (local_size % 2 == 1) { // odd numbers in each processor
        local_size += 1;
        local_size_max = local_size;
    }
    else { // even numbers in each processor
        if (rank == numtasks - 1) {
            local_size_max = local_size + rem;
        }
    }

    printf("\nnumber of tasks= %d my rank= %d local_size= %d running
on %s \nLocal array: ", numtasks, rank, local_size_max, hostname);

    int *local_array = (int *)malloc(sizeof(int)*local_size);  //
local array
```

```
    for (int i = 0; i < local_size_max; i++)
{ //MPI_Scatter(&array_in, local_size_max, MPI_INT, local_array,
local_size_max, MPI_INT, 0, MPI_COMM_WORLD);
        if (local_size*rank + i < size) {
            local_array[i] = array_in[local_size*rank + i];
        }
        else{ // padding
            local_array[i] = 100; //1000 actually should be the max of
array
        }
        printf("%d ", local_array[i]);
    }

    if (rank == 0) {
        printf("\n\nName : Yao Zixuan\n");
        printf("ID : 115010267\n");
        printf("Sorting %d random numbers using %d processors\n",
size, numtasks);

        srand((int)time(NULL)); // random seed
        printf("Random Numbers Generating... \n\nOrignal array: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", array_in[i]);
        }
        printf("\n\nSorting...\n");
    }

    for (int i = 1; i < size+1; i++) {
        //printf("\n Iteration%d  ", i);
        if (i % 2 == 1) { // odd iteration
            for (int k = 0; k < local_size_max / 2; k++) { //
local_size_max odd, but no action
                if (local_array[2 * k] > local_array[2 * k + 1]){
                    swap(local_array, 2 * k, 2 * k + 1);
                }
            }
        }
        else { // even iteration
            if (rank == numtasks-1) {
                send_left = local_array[0];
                MPI_Send(&send_left, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);
                //printf("send_left: %d  ", send_left);
            }
```

```
            else if (rank != 0){
                MPI_Recv(&recv_right, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                send_left = local_array[0];
                MPI_Send(&send_left, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);
                //printf("recv_right: %d, send_left: %d  ", recv_right,
send_left);
            }
            else if (rank == 0){
                if (numtasks != 1) {
                    MPI_Recv(&recv_right, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    //printf("recv_right: %d  ", recv_right);
                }
            }

            for (int k = 0; k < local_size_max / 2; k++) { //
local_size_max odd
                if (local_size_max % 2 != 0) { // rank == numtasks - 1
                    if (local_array[2 * k + 1] > local_array[2 * k + 2])
{
                        swap(local_array, 2 * k + 1, 2 * k + 2);
                    }
                }
                else {
                    if (k < (local_size_max / 2) - 1) {
                        if (local_array[2 * k + 1] > local_array[2 * k +
2]) {
                            swap(local_array, 2 * k + 1, 2 * k + 2);
                        }
                    }
                    else {
                        if (rank != numtasks - 1) {
                            if (local_array[2 * k + 1] > recv_right) {
                                send_right = local_array[2 * k + 1];
                                local_array[2 * k + 1] = recv_right;
                            }
                            else {
                                send_right = recv_right;
                            }
                        }
                    }
                }
```

```
            }

            if (rank == 0) {
                if (numtasks != 1) { // only one processor
                    MPI_Send(&send_right, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
                    //printf("send_right: %d  ", send_right);
                }
            }
            else if (rank != numtasks - 1) {
                MPI_Recv(&recv_left, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                local_array[0] = recv_left;
                MPI_Send(&send_right, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
                //printf("recv_left: %d, send_right: %d  ", recv_left,
send_right);
            }
            else if (rank == numtasks - 1) {
                MPI_Recv(&recv_left, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                local_array[0] = recv_left;
                //printf("recv_left: %d  ", recv_left);
            }
        }

        if (i == size - 1) {
            printf("\nOutput array: ");
            for (int i = 0; i < local_size_max; i++) {
                printf("%d ", local_array[i]);
            }
        }
    }

    int *send_rem = (int *)malloc(sizeof(int)*rem);
    int *recv_rem = (int *)malloc(sizeof(int)*rem);
    MPI_Gather(local_array, local_size, MPI_INT, array_out,
local_size, MPI_INT, 0, MPI_COMM_WORLD);
    if (((size / numtasks) % 2 == 0) && (rem > 0)) {  //rem still need
to send
        if (rank == numtasks - 1) {
            for (int i = 0; i < rem; i++) {
                send_rem[i] = local_array[local_size + i];
            }
```

```
            MPI_Send(send_rem, rem, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        else if (rank == 0) {
            MPI_Recv(recv_rem, rem, MPI_INT, numtasks - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = 0; i < rem; i++) {
                array_out[size - rem + i] = recv_rem[i];
            }
        }
    }

    finish = MPI_Wtime();

    MPI_Finalize();// done with MPI
    if (rank == 0) {
        printf("\nSorted array: ");
        for (int j = 0; j < size; j++) {
            printf("%d ", array_out[j]);
        }
        totaltime = (double)(finish - start);
        printf("\ntotal time : %f\n", totaltime);
    }
    return 0;
}

int swap(int *num, int i, int j) {
    int tmp = num[i];
    num[i] = num[j];
    num[j] = tmp;

    return 0;
}
```