Project1: Odd-Even Transposition Sort

Name: Xinyu Xie
Student ID: 119020059

Introduction

The Odd-Even Transposition Sort is a variation of bubble sort by dividing its algorithm into the Odd Phase and the Even Phase. In each iteration, the two phases - Odd Phase and Even Phase occur, and the algorithm runs until all of the array elements are sorted.
In this project, I implemented the seqential Odd-Even Transposition Sort and the parallel Odd-Even Transposition Sort. The detailed implementation methods, results and conclusions are provided in the following sections.

Method

Sequential Odd-Even Transposition Sort

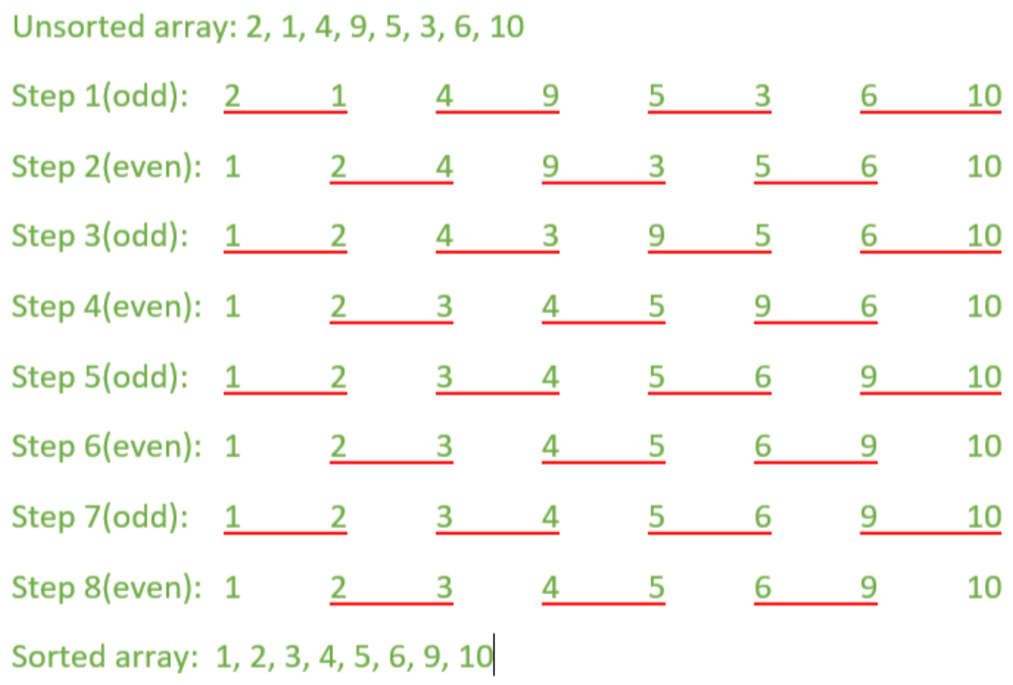
The following are the steps to perform sequential odd-even transposition sort:

1. For each number in odd position (i-th), compare its value with the number on (i+1)-th position, and swap these two numbers if the number on i-th position is larger than number on (i+1)-th position.

2. For each number in even position (i-th), compare its value with the number on (i+1)-th position, and swap these two numbers if the number on i-th position is larger than number on (i+1)-th position.

3. Repeat these two steps above until all the elements in array are sorted.

This sorting process is illustrated in the following figure:


Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10
Step 1(odd): 2 1 4 9 5 3 6 10
Step 2(even): 1 2 4 9 3 5 6 10
Step 3(odd): 1 2 4 3 9 5 6 10
Step 4(even): 1 2 3 4 5 9 6 10
Step 5(odd): 1 2 3 4 5 6 9 10
Step 6(even): 1 2 3 4 5 6 9 10
Step 7(odd): 1 2 3 4 5 6 9 10
Step 8(even): 1 2 3 4 5 6 9 10
Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

The complexity of Sequential Odd-Even Sort algorithm is O(N^2), where N = Number of elements in the array. Since to sort the array, we will have N iterations and do O(N) times comparison in each iteration.

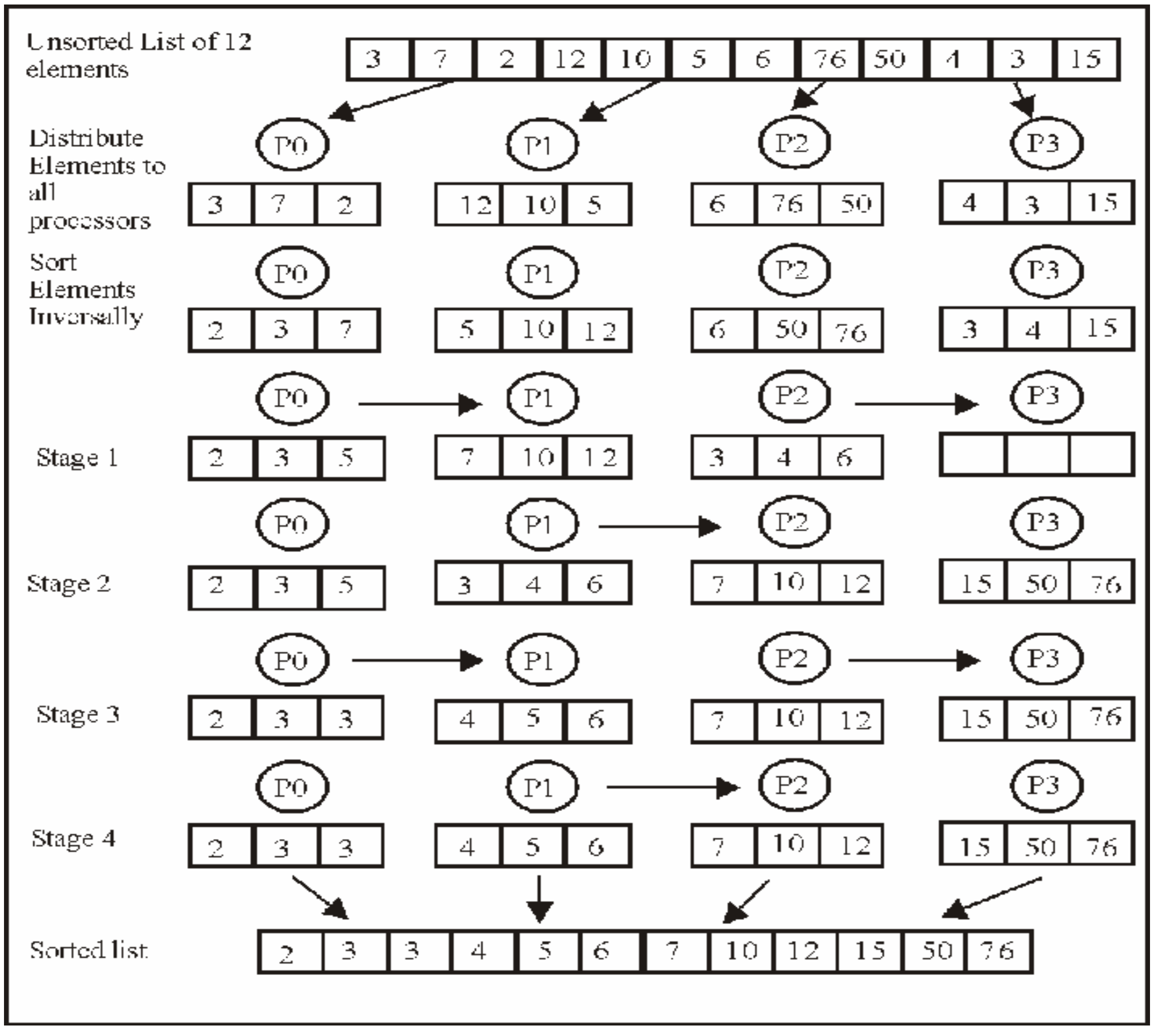
Parallel Odd-Even Transposition Sort

The following are the steps to perform sequential odd-even transposition sort:

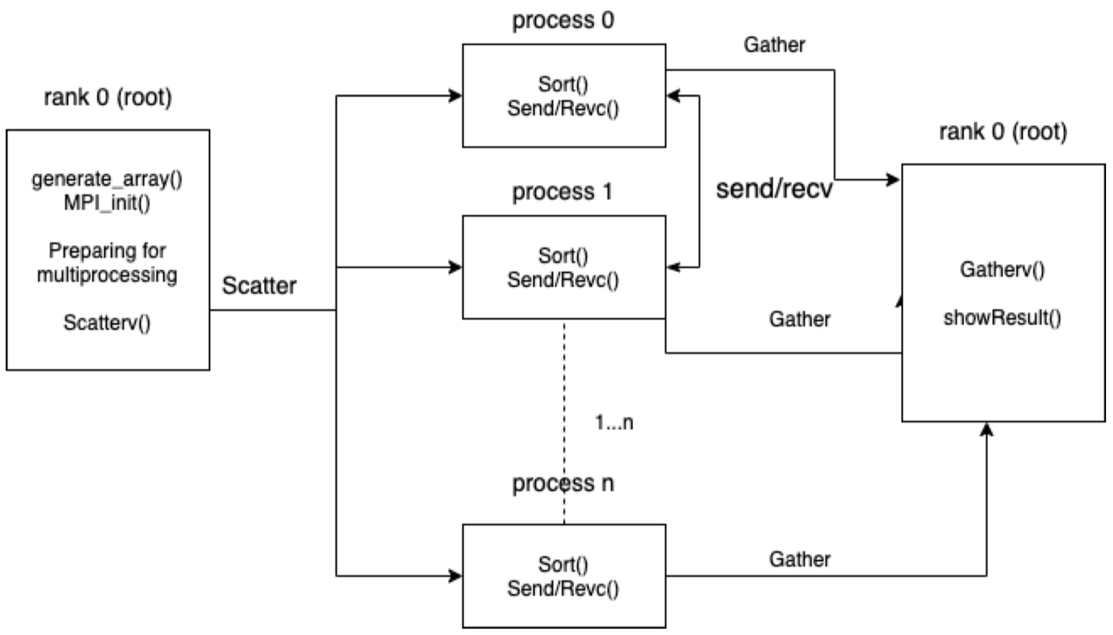
1. Initially, m numbers are distributed to n processes, respectively.
2. For each process with odd rank P, send its number to the process with rank P-1.
3. For each process with rank P-1, compare its number with the number sent by the process with rank P and send the larger one back to the process with rank P.
4. For each process with even rank Q, send its number to the process with rank Q-1.
5. For each process with rank Q-1, compare its number with the number sent by the process with rank Q and send the larger one back to the process with rank Q.
6. Repeat 2-5 until the numbers are sorted.

The main idea is as follows. Firstly, the input array of numbers is equally distributed to different processes. And in each process, the sequential odd-even sort is performed. However, we need to do the boundary check in each iteration of the parallel odd-even sort. For processes with rank 0 or rank N-1, they only need to do one send/receive for communication between processes. For other processes, they need to do both send and receive to communicate with previous and next processes and determine whether the leftover elements need to be swapped or not.

This sorting process is illustrated in the following figure:



This flow chart is illustrated in the following figure:

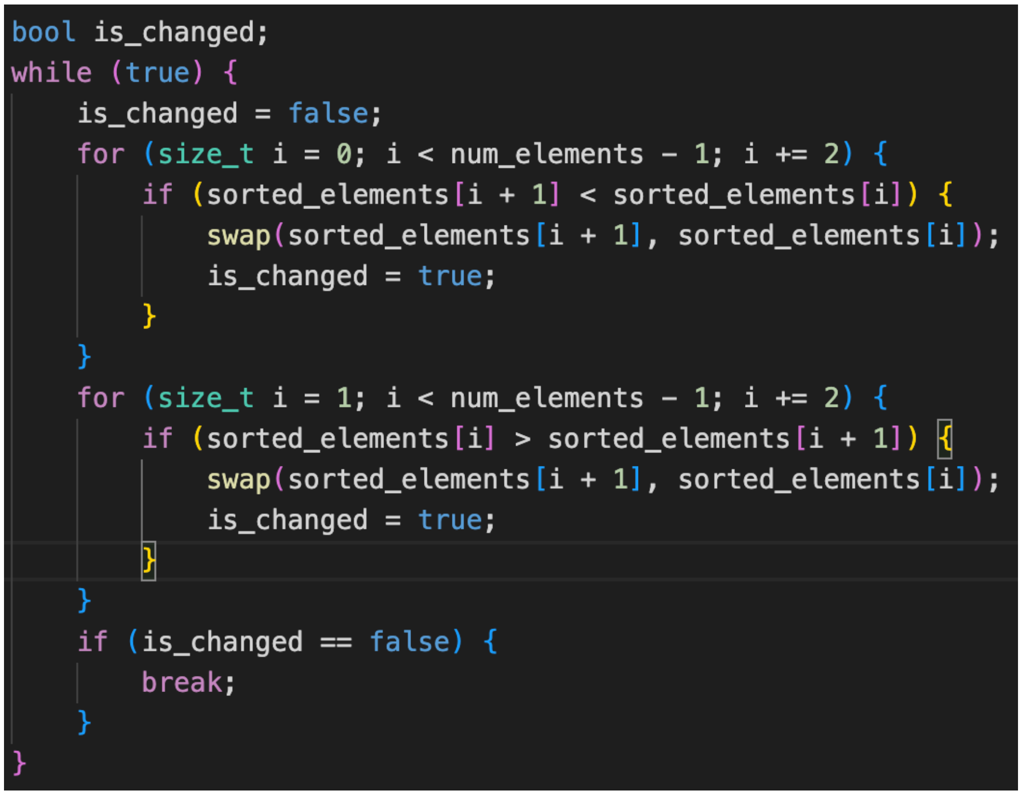


The complexity of Parallel Odd-Even Sort algorithm is O(N^2/m), where N = Number of elements in the array, m = Number of processes. Since the parallel odd-even sort is the variation of bubble sort by distributing the algorithm into multiple processes and executing them simultaneously. Since the complexity of bubble sort is O(N^2), and there are m processes in total, the complexity of parallel odd-even sort is O(N^2/m).

Implementation

Sequential Odd-Even Transposition Sort

The implementation of sequential odd-even sort directly follows the method mentioned above. The following is the core part of detailed implementation code:



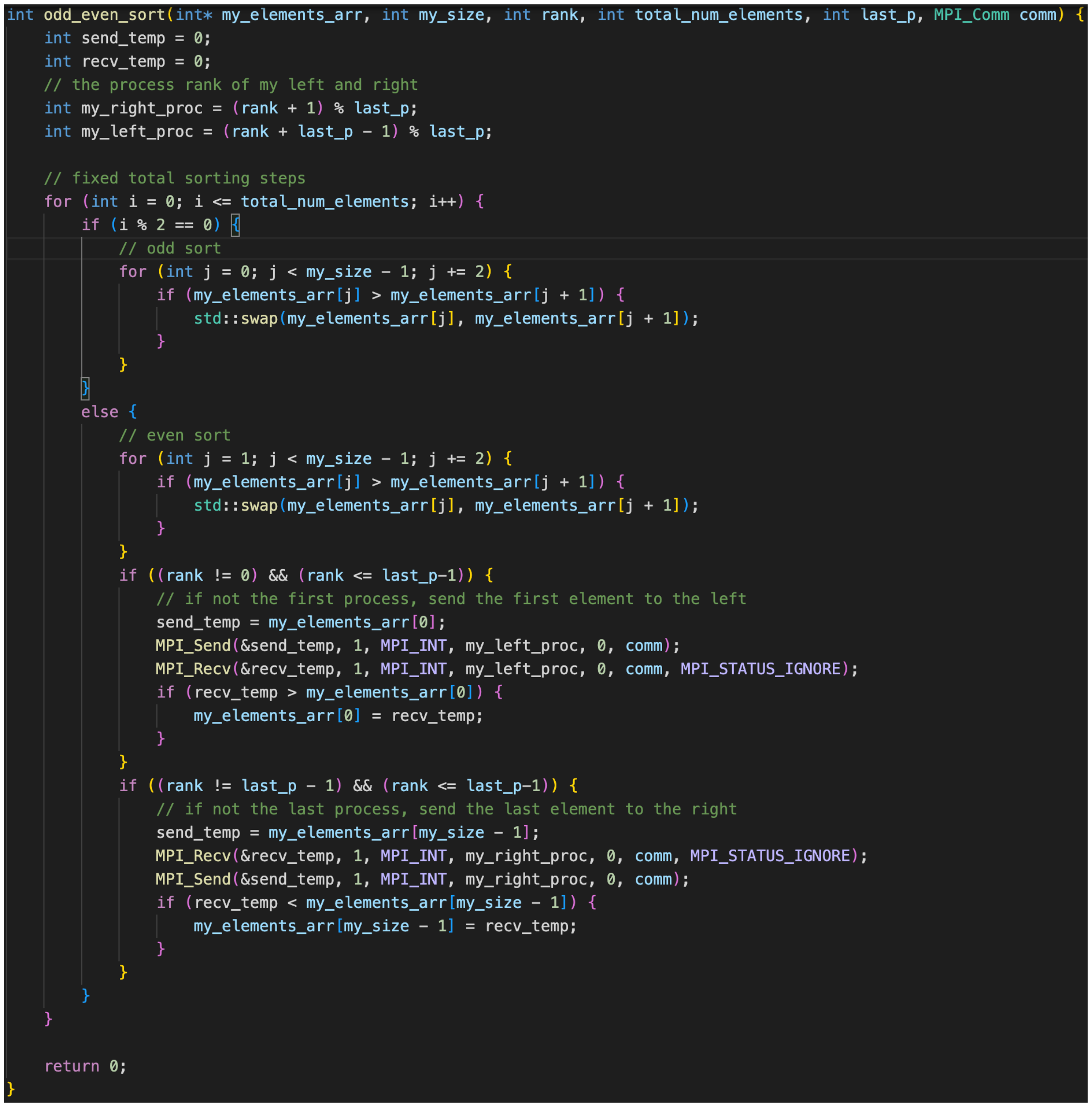
```
bool is_changed;
while (true) {
    is_changed = false;
    for (size_t i = 0; i < num_elements - 1; i += 2) {
        if (sorted_elements[i + 1] < sorted_elements[i]) {
            swap(sorted_elements[i + 1], sorted_elements[i]);
            is_changed = true;
        }
    }
    for (size_t i = 1; i < num_elements - 1; i += 2) {
        if (sorted_elements[i] > sorted_elements[i + 1]) {
            swap(sorted_elements[i + 1], sorted_elements[i]);
            is_changed = true;
        }
    }
    if (is_changed == false) {
        break;
    }
}
```

Parallel Odd-Even Transposition Sort

The implementation of parallel odd-even sort is divided into following steps:

1. Initialization and do distribution of input array elements.
2. Perform local sorting in each process and do communication between different processes. We use MPI_send and MPI_recv function for the comparision of boundary elements.
3. Gather the results from each process and generate the result sorted array.

The following is the core part of detailed implementation code:



```
int odd_even_sort(int* my_elements_arr, int my_size, int rank, int total_num_elements, int last_p, MPI_Comm comm) {
    int send_temp = 0;
    int recv_temp = 0;
    // the process rank of my left and right
    int my_right_proc = (rank + 1) % last_p;
    int my_left_proc = (rank + last_p - 1) % last_p;

    // fixed total sorting steps
    for (int i = 0; i <= total_num_elements; i++) {
        if (i % 2 == 0) {
            // odd sort
            for (int j = 0; j < my_size - 1; j += 2) {
                if (my_elements_arr[j] > my_elements_arr[j + 1]) {
                    std::swap(my_elements_arr[j], my_elements_arr[j + 1]);
                }
            }
        }
        else {
            // even sort
            for (int j = 1; j < my_size - 1; j += 2) {
                if (my_elements_arr[j] > my_elements_arr[j + 1]) {
                    std::swap(my_elements_arr[j], my_elements_arr[j + 1]);
                }
            }
            if ((rank != 0) && (rank <= last_p-1)) {
                // if not the first process, send the first element to the left
                send_temp = my_elements_arr[0];
                MPI_Send(&send_temp, 1, MPI_INT, my_left_proc, 0, comm);
                MPI_Recv(&recv_temp, 1, MPI_INT, my_left_proc, 0, comm, MPI_STATUS_IGNORE);
                if (recv_temp > my_elements_arr[0]) {
                    my_elements_arr[0] = recv_temp;
                }
            }
            if ((rank != last_p - 1) && (rank <= last_p-1)) {
                // if not the last process, send the last element to the right
                send_temp = my_elements_arr[my_size - 1];
                MPI_Recv(&recv_temp, 1, MPI_INT, my_right_proc, 0, comm, MPI_STATUS_IGNORE);
                MPI_Send(&send_temp, 1, MPI_INT, my_right_proc, 0, comm);
                if (recv_temp < my_elements_arr[my_size - 1]) {
                    my_elements_arr[my_size - 1] = recv_temp;
                }
            }
        }
    }

    return 0;
}
```

Execution

The following are the steps for execution:

1. Go to project1 folder, compile all files using “make all”.
2. Use “gen [array size] [input array file path]” to generate unsorted input array.
3. Use “ssort [array size] [input array file path]” to run sequential odd-even transposition sort.
4. Use “mpirun -np=[number of processes] psort [array size] [input array file path]” to run parallel odd-even transposition sort.
5. Use “check [array size] [output array file path]” to check whether the output array is correctly sorted or not.

The following are the steps to do experiments in hpc:

1. Go to project1 folder, compile all files using “make all”.
2. Config the task template batch file by specifying job name, nodes number, process number, number of CPU cores per process, total memory limit, time limit, and partition name shown as the following example.

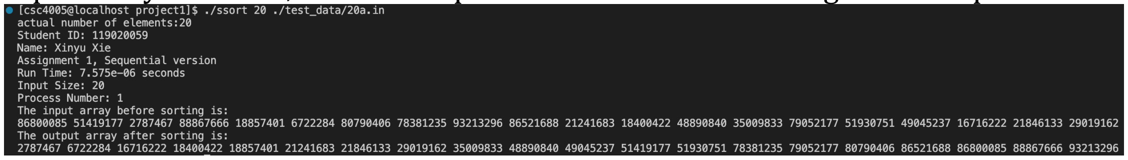
```
#!/bin/bash
#SBATCH --job-name=119020059_seq_{size}    # Job name
#SBATCH --nodes=1                 # Run all processes on a single node
#SBATCH --ntasks=1                # number of processes = 4
#SBATCH --cpus-per-task=1         # Number of CPU cores per process
#SBATCH --mem=1000mb              # Total memory limit
#SBATCH --time=00:60:00           # Time limit hrs:min:sec
#SBATCH --partition=Debug         # Partition name: Project or Debug (Debug is default)
```

3. SSH to hpc and run “sbatch [batch file path]” to submit the job.
4. Collect running output file “slurm-[num].out”.

Sample Outputs

Sequential Odd-Even Transposition Sort

Input array size = 20, Number of processes = 1. The following is the output:

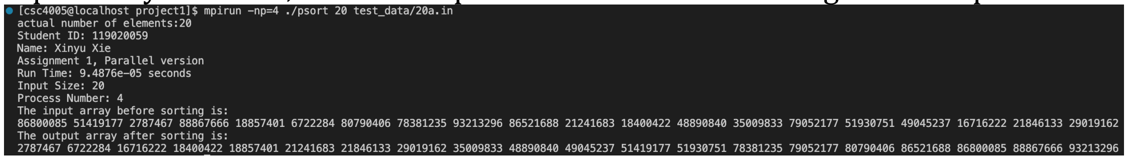


```
● [csc4005@localhost project1]$ ./ssort 20 ./test_data/20a.in
  actual number of elements:20
  Student ID: 119020059
  Name: Xinyu Xie
  Assignment 1, Sequential version
  Run Time: 7.575e-06 seconds
  Input Size: 20
  Process Number: 1
  The input array before sorting is:
  86800085 51419177 2787467 88867666 18857401 6722284 80790406 78381235 93213296 86521688 21241683 18400422 48890840 35009833 79052177 51930751 49045237 16716222 21846133 29019162
  The output array after sorting is:
  2787467 6722284 16716222 18400422 18857401 21241683 21846133 29019162 35009833 48890840 49045237 51419177 51930751 78381235 79052177 80790406 86521688 86800085 88867666 93213296
```

Parallel Odd-Even Transposition Sort

Normal Case

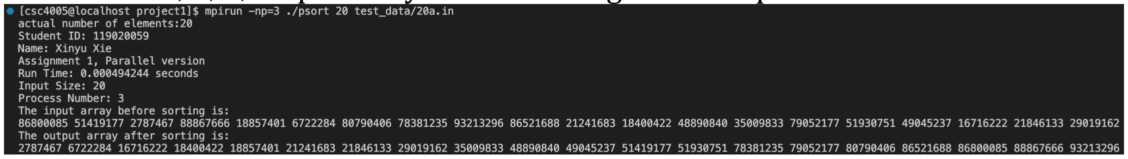
Input array size = 20, Number of processes = 4. The following is the output:



```
● [csc4005@localhost project1]$ mpirun -np=4 ./psort 20 test_data/20a.in
  actual number of elements:20
  Student ID: 119020059
  Name: Xinyu Xie
  Assignment 1, Parallel version
  Run Time: 9.4876e-05 seconds
  Input Size: 20
  Process Number: 4
  The input array before sorting is:
  86800085 51419177 2787467 88867666 18857401 6722284 80790406 78381235 93213296 86521688 21241683 18400422 48890840 35009833 79052177 51930751 49045237 16716222 21846133 29019162
  The output array after sorting is:
  2787467 6722284 16716222 18400422 18857401 21241683 21846133 29019162 35009833 48890840 49045237 51419177 51930751 78381235 79052177 80790406 86521688 86800085 88867666 93213296
```

Case with unequally distributed elements

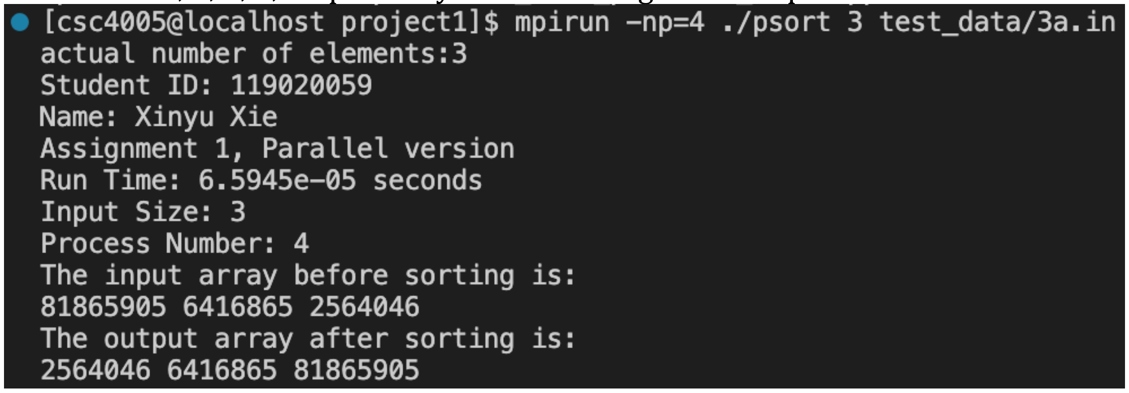
Input array size = 20, Number of processes = 3. The 3 processes have number of elements = 7, 7, 6, respectively. The following is the output:



```
● [csc4005@localhost project1]$ mpirun -np=3 ./psort 20 test_data/20a.in
  actual number of elements:20
  Student ID: 119020059
  Name: Xinyu Xie
  Assignment 1, Parallel version
  Run Time: 0.000494244 seconds
  Input Size: 20
  Process Number: 3
  The input array before sorting is:
  86800085 51419177 2787467 88867666 18857401 6722284 80790406 78381235 93213296 86521688 21241683 18400422 48890840 35009833 79052177 51930751 49045237 16716222 21846133 29019162
  The output array after sorting is:
  2787467 6722284 16716222 18400422 18857401 21241683 21846133 29019162 35009833 48890840 49045237 51419177 51930751 78381235 79052177 80790406 86521688 86800085 88867666 93213296
```

Case with elements number < processes number

Input array size = 3, Number of processes = 4. The 4 processes have number of elements = 1, 1, 1, 0, respectively. The following is the output:



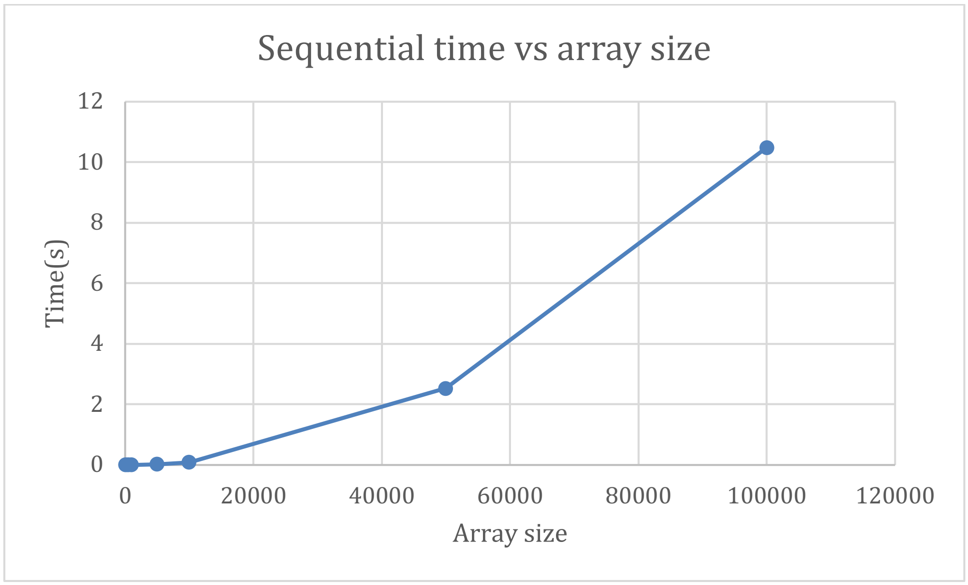
```
● [csc4005@localhost project1]$ mpirun -np=4 ./psort 3 test_data/3a.in
  actual number of elements:3
  Student ID: 119020059
  Name: Xinyu Xie
  Assignment 1, Parallel version
  Run Time: 6.5945e-05 seconds
  Input Size: 3
  Process Number: 4
  The input array before sorting is:
  81865905 6416865 2564046
  The output array after sorting is:
  2564046 6416865 81865905
```

Analysis and Results

Sequential: Running time vs. Input array size

For input array size = 100, 500, 1000, 5000, 10000, 50000, 100000, the running time of sequential odd-even sort is as follows:

type	cores	array size	time(s)
seq	1	100	4.10E-05
seq	1	500	0.0006022
seq	1	1000	0.0015733
seq	1	5000	0.0236769
seq	1	10000	0.0875864
seq	1	50000	2.53727
seq	1	100000	10.4895

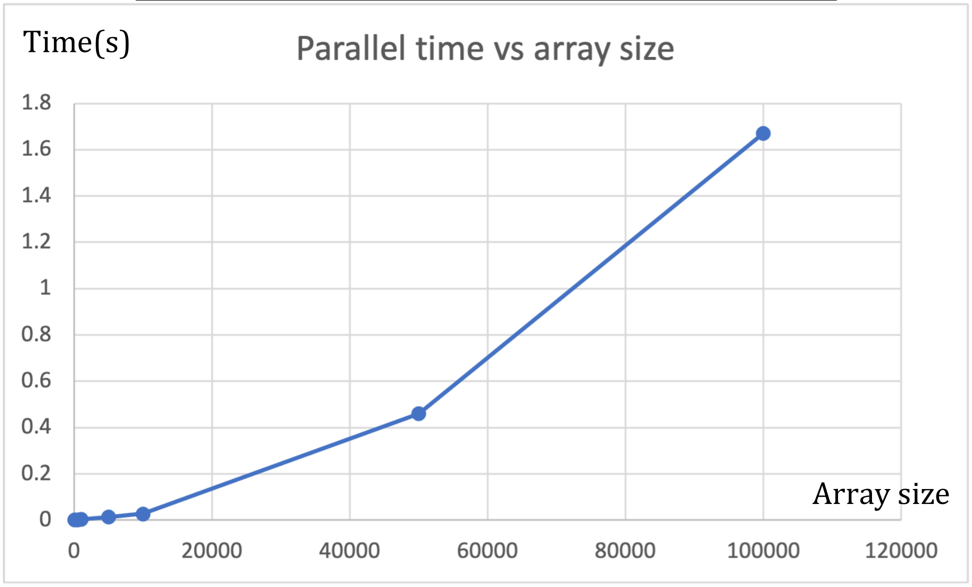


From the figure above, we can see that the running time of sequential odd-even sort increases quadratically with the input array size. Since the complexity of sequential odd-even sort is O(N^2), the running time of sequential odd-even sort increases quadratically with the input array size, especially when array size is large so that other time costs are negligible.

Parallel: Running time vs. Input array size

With fixed number of processes = 10, for input array size = 100, 500, 1000, 5000, 10000, 50000, 100000, the running time of sequential odd-even sort is as follows:

type	cores	array size	time(s)
par	10	100	0.00123769
par	10	500	0.00148145
par	10	1000	0.00235995
par	10	5000	0.0124839
par	10	10000	0.0271363
par	10	50000	0.458867
par	10	100000	1.6701

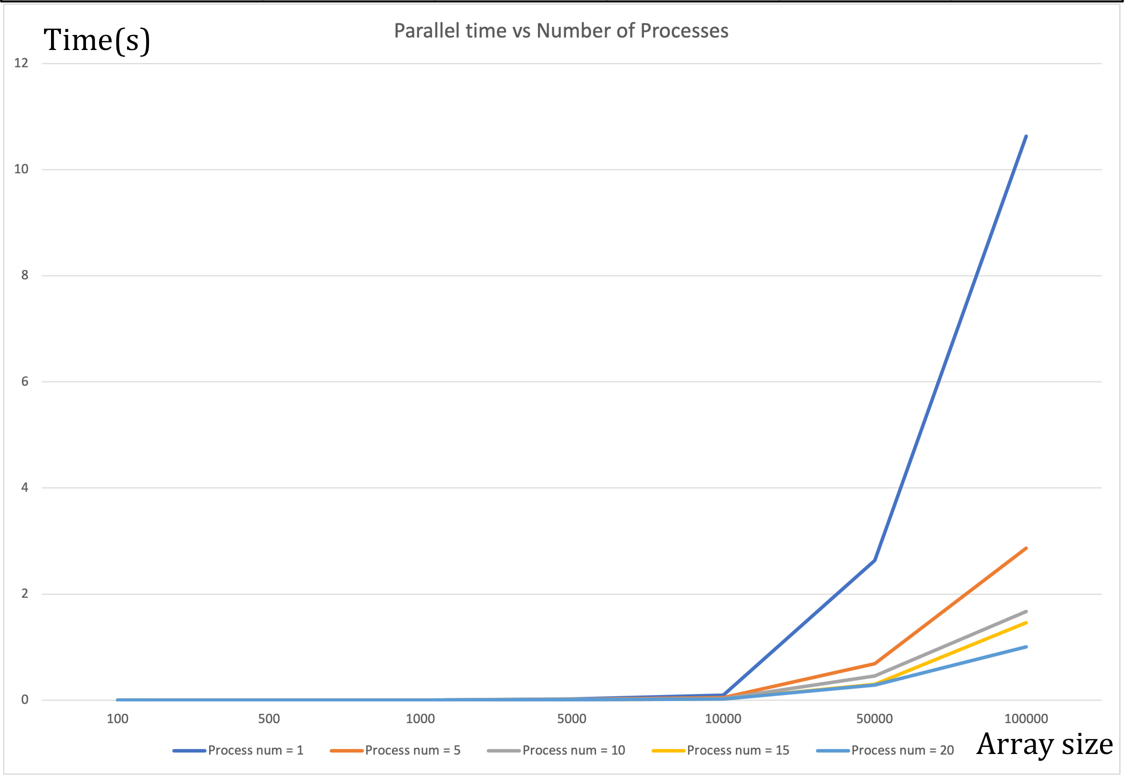


From the figure above, we can see that the running time of parallel odd-even sort also increases quadratically with the input array size. Since the complexity of sequential odd-even sort is O(N^2/m).

Parallel: Running time vs. Number of processes

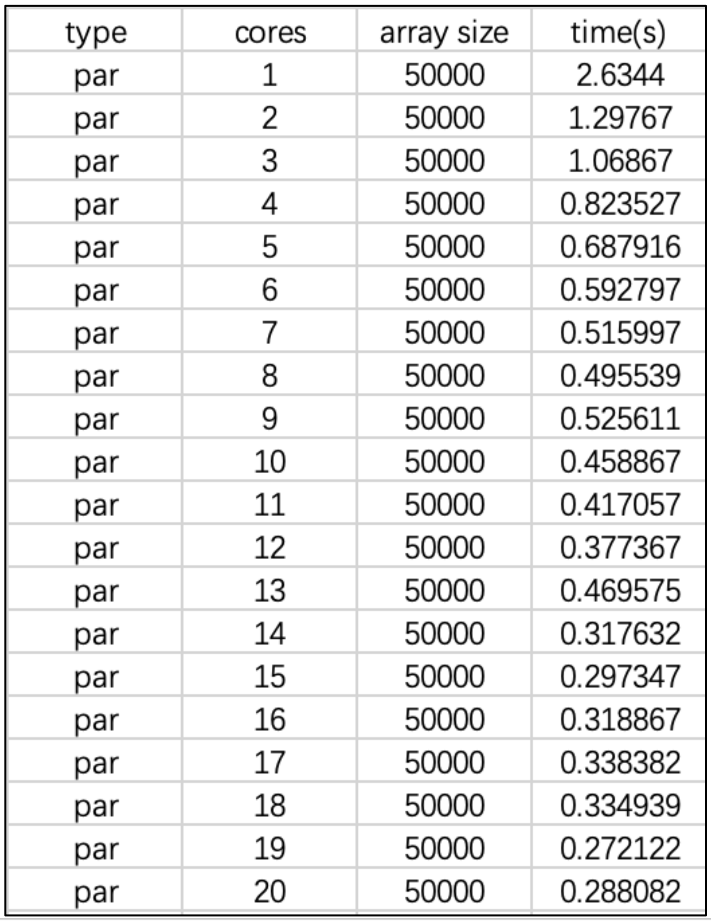
We compare parallel version with number of processes = 1, 5, 10, 15, 20 and input array size = 100, 500, 1000, 5000, 10000, 50000, 100000. The comparison is shown as follows:

size/process num	1	5	10	15	20
100	5.3563E-05	0.00029644	0.00123769	0.00063279	0.00044587
500	0.00031123	0.00166388	0.00148145	0.00106631	0.00128569
1000	0.00140845	0.00174114	0.00235995	0.00264934	0.00188533
5000	0.0230407	0.0117786	0.0124839	0.00990714	0.0103471
10000	0.095733	0.0476638	0.0271363	0.0224112	0.0210131
50000	2.6344	0.687916	0.458867	0.297347	0.288082
100000	10.629	2.86221	1.6701	1.45958	1.00715

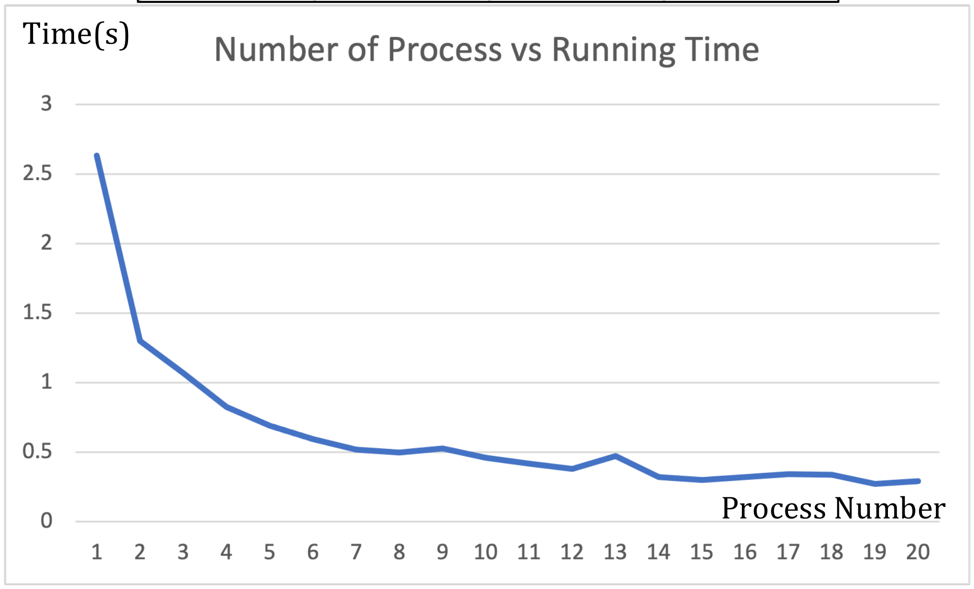


From the figure above, the running time of parallel version is roughly a quadratic function of input data size. When the number of processes is larger, the slope of the quadratic function is smaller. This is consistent with the fact that the complexity of parallel odd even sort is O(N^2/m) with m being number of processes.

Also, if we fix input data size = 50000, for number of processes = 1, 2, 3, ... 20. The comparison is shown as follows:



type	cores	array size	time(s)
par	1	50000	2.6344
par	2	50000	1.29767
par	3	50000	1.06867
par	4	50000	0.823527
par	5	50000	0.687916
par	6	50000	0.592797
par	7	50000	0.515997
par	8	50000	0.495539
par	9	50000	0.525611
par	10	50000	0.458867
par	11	50000	0.417057
par	12	50000	0.377367
par	13	50000	0.469575
par	14	50000	0.317632
par	15	50000	0.297347
par	16	50000	0.318867
par	17	50000	0.338382
par	18	50000	0.334939
par	19	50000	0.272122
par	20	50000	0.288082

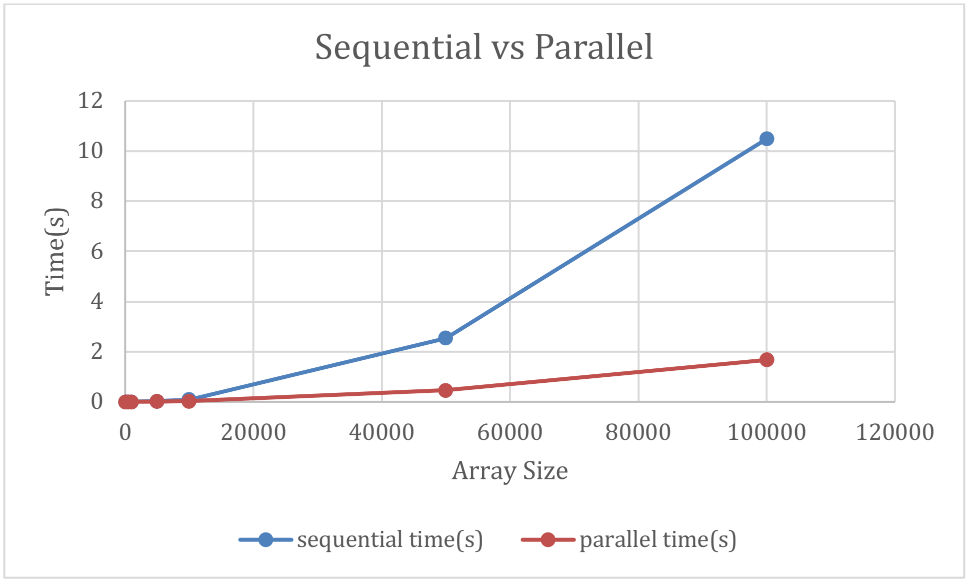


From the figure above, as the number of processes increases, the running time reduces. However, the speed of reduction speed becomes slower. The reason may be that, as the number of processes increases, the cost of process communication becomes larger, since the algorithm needs to exchange elements at the boundary and wait for send and receive actions. The more the processes, the larger the influence of the process communication has. Thus, the running time becomes roughly constant when the number of processes is larger than 10. The slight fluctuation of the running time may be due to network instabilities.

Sequential vs. Parallel

We compare sequential version and parallel version with fixed number of processes = 10, for input array size = 100, 500, 1000, 5000, 10000, 50000, 100000. The comparison is shown as follows:

array size	sequential time(s)	parallel time(s)
100	0.00004098	0.00123769
500	0.000602189	0.00148145
1000	0.00157331	0.00235995
5000	0.0236769	0.0124839
10000	0.0875864	0.0271363
50000	2.53727	0.458867
100000	10.4895	1.6701



For parallel odd even sort, the initialization of multi-processes costs extra time. Thus, when the input array size is relatively small, the running time of parallel version is larger than sequential version. When the input array size is larger than

Array size

10000 with 10000 as the rough boundary, the overhead of communication cost and other additional cost is remedied by lower computation costs offered by multiple processes. Thus, as the input array size still grows, the running time of parallel version is much smaller than sequential version. This is because the time complexity of parallel sort is O(N^2/m) while the time complexity of sequential sort is O(N^2). Moreover, the speed of parallel version is less than 10 times of sequential version when input array size is large enough, this is due to the communication time costs between processes in parallel version.

Conclusion

In this project, the sequential and parallel versions of Odd-Even Transposition Sort are implemented and their proformances are also analyzed. We can conclude that the running time of odd-even sort increases quadratically with the input array size. As the number of processes increases, the running time decreases with slower slope and finally reaches a rough constant. The running time of parallel version is much smaller than sequential version when the input array size is large enough.

Through this project, I learned much knowledge regarding the parallel programming and also how to use the mpi library. I also enjoy a lot when conducting experiments for different array size and exploring the relationship between the running time and the number of processes.