Assignment Report

CSC 4005 Parallel Odd-Even Transposition Sort

Wei Wu (吴畏)

118010335

October 6, 2021

The School of Data Science

香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

I. Introduction

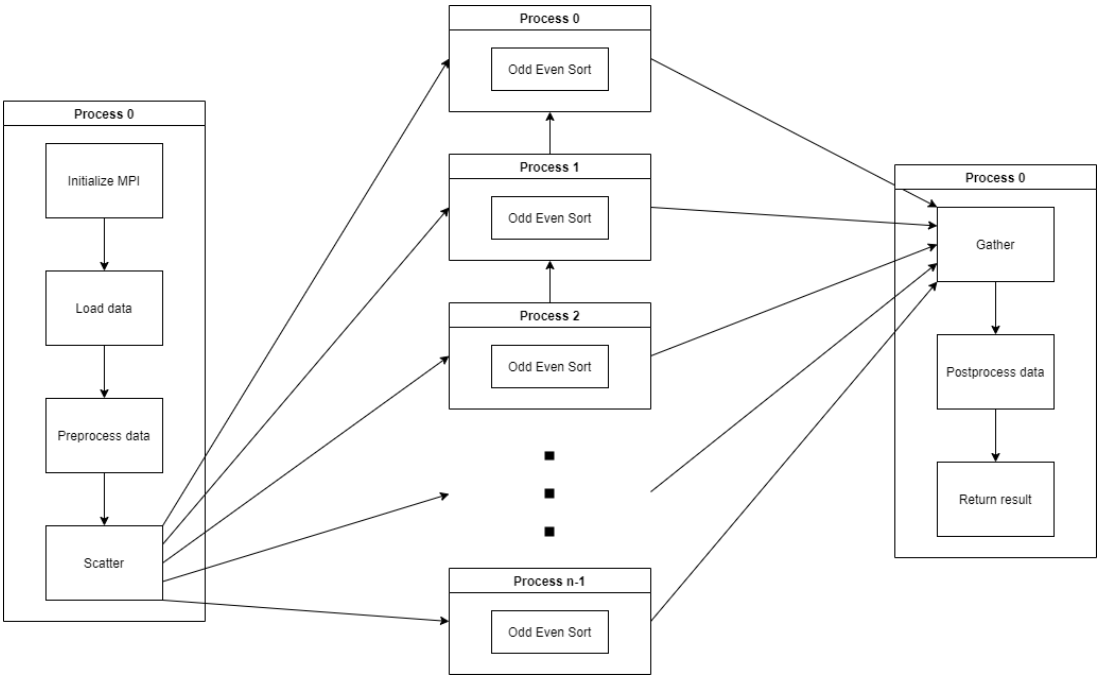
This assignment is implementing a parallel odd-even transposition sort by using MPI. A parallel odd-even transposition sort is performed as follows:

Initially, m numbers are distributed to n processes, respectively.

1. For each process with odd rank P, send its number to the process with rank P-1.
2. 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.
3. For each process with even rank Q, send its number to the process with rank Q-1.
4. 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.
5. Repeat 1-4 until the numbers are sorted.

II. Design Approaches

1. Architecture



Generally, the program is divided into three stages. In the first stage, the root process (rank 0) initializes MPI, loads data from an external source, and preprocesses the data. The most important part in the data preprocessing is padding. The program inserts LONG_MAX into the array of data to ensure that each process is assigned the same number of elements. Then, the root process scatters the array of data to all the processes, including itself.

In the second stage, all the processes receive and store the assigned subarray of data. Then, the processes perform a coordinated Odd-Even Transposition Sort. Each element in each process is handled according to its index in the complete array. In this way, the algorithm behaves exactly the same no matter how many processes are used. In other words, the

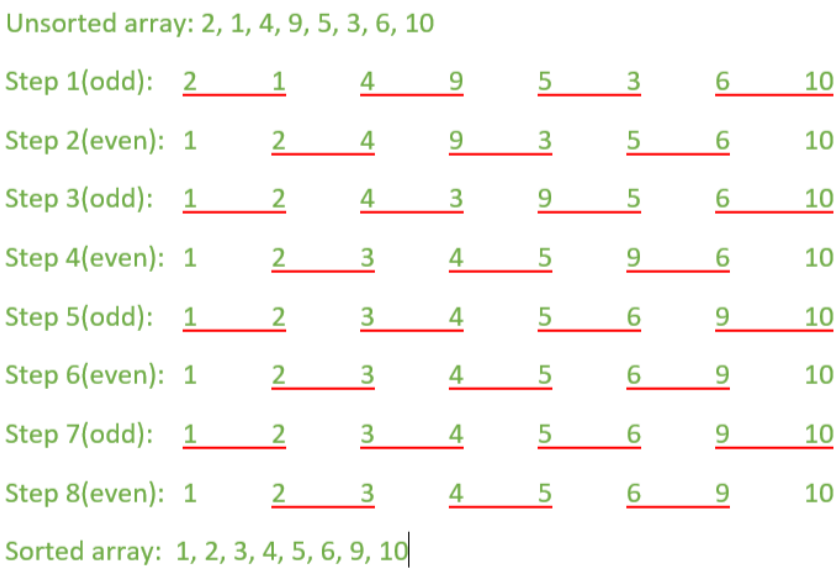
behavior of the algorithm is independent of the number of processes. The Odd-Even Transposition Sort terminates when all the subarrays do not change any more.

In the third stage, the root process gathers the sorted subarrays of data from all the processes. Furthermore, the root process does some postprocessing and returns the result.

2. Algorithm

Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if the order is not desired. Each iteration of the algorithm can be divided into 2 phases – the odd phase and even phase:

Odd phase: Every odd indexed element is compared with the next even indexed element(considering 1-based indexing).



Even phase: Every even indexed element is compared with the next odd indexed element.

The algorithm terminates when the array does not change anymore. It is guaranteed to terminate in finite iterations since the number of unordered pairs must reduce in each iteration.

The time complexity of this algorithm in sequential execution (only one process) is

$$O(n^2) \text{ (worst case)}$$

$$O(n) \text{ (best case)}$$

$$O(n^2) \text{ (average case)}$$

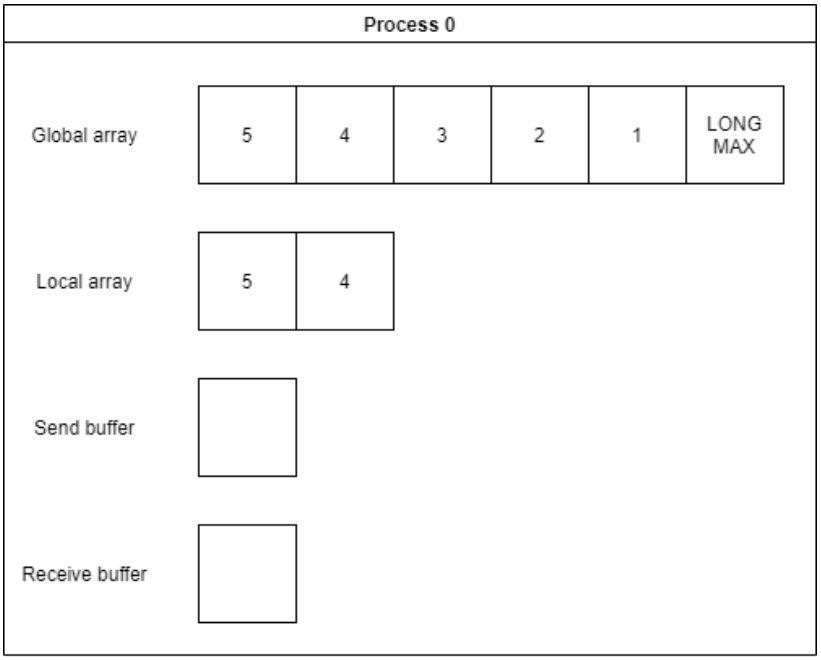
The time complexity of this algorithm in parallel execution (n processes) is

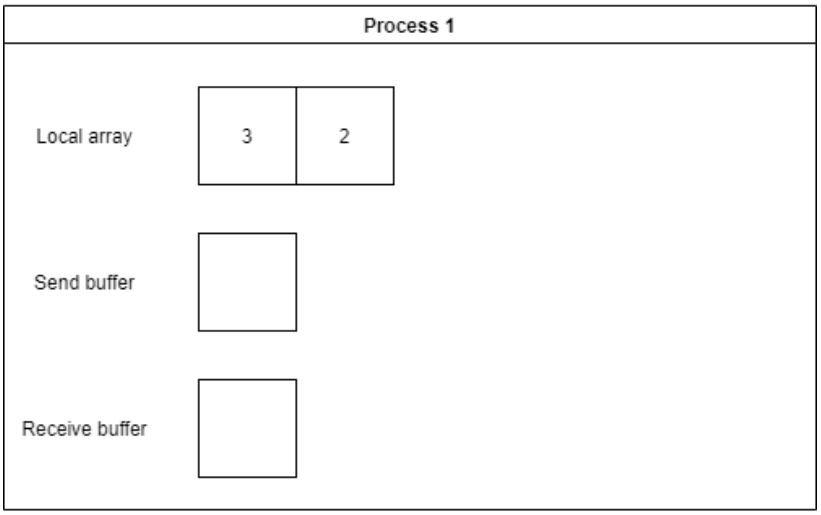
$$O(n)$$

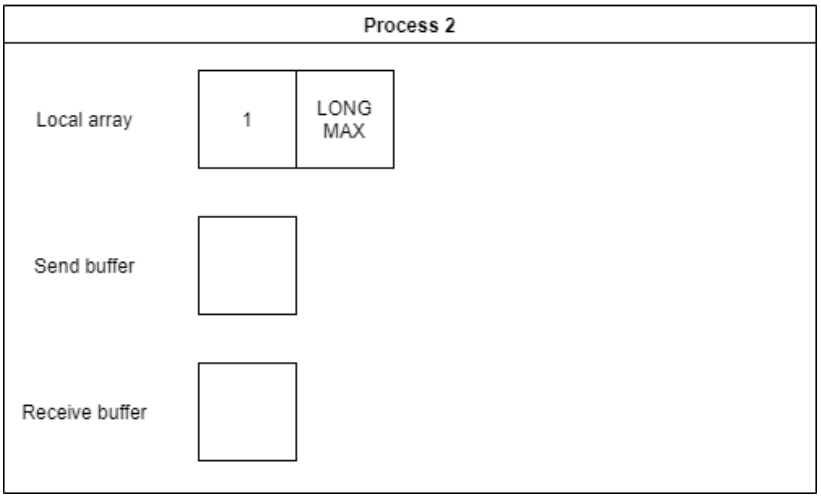
The space used by the algorithm is constant. Thus, the space complexity is

$$O(1)$$

3. Data Structure







In the root process, a so-called global array is used to store all the elements. If the total number of elements is not integer multiples of the number of processes, the global array will be padded with LONG_MAX so that each process will be assigned the same number of elements. This padding mechanism simplifies the computation in each process.

Each process has a local array to store the assigned elements. All swaps happen within the local array so that no extra memory space is needed. Besides, each process has a send buffer and a receive buffer for inter-process communication.

III. Build And Run

To build the program, use the following commands on the server:

```
cd /path/to/project
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
cmake --build . -j
```

To run the program, use the following commands on the server:

```
mpirun main input output
```

or

```
mpirun gtest_sort
```

IV. Performance Analysis

1. Test Results

Input size	Number of processes: 1	2	4	8	16	32	64
10k	577	292	156	85	49	35	117
20k	2348	1185	649	347	182	108	125
40k	9460	4821	2622	1449	749	392	312
80k	37820	19306	10546	5876	3104	1586	964
160k	151789	77298	42196	23587	12608	6515	3501
320k	610191	309867	168618	94308	50645	26339	13918

Table 1. Execution time (ms) of the program with respect to different input sizes and number of processes

Note:

a. All tests were conducted on 10.26.1.30.

b. Each test was repeated for three times and averaged.

c. All input data were generated randomly.

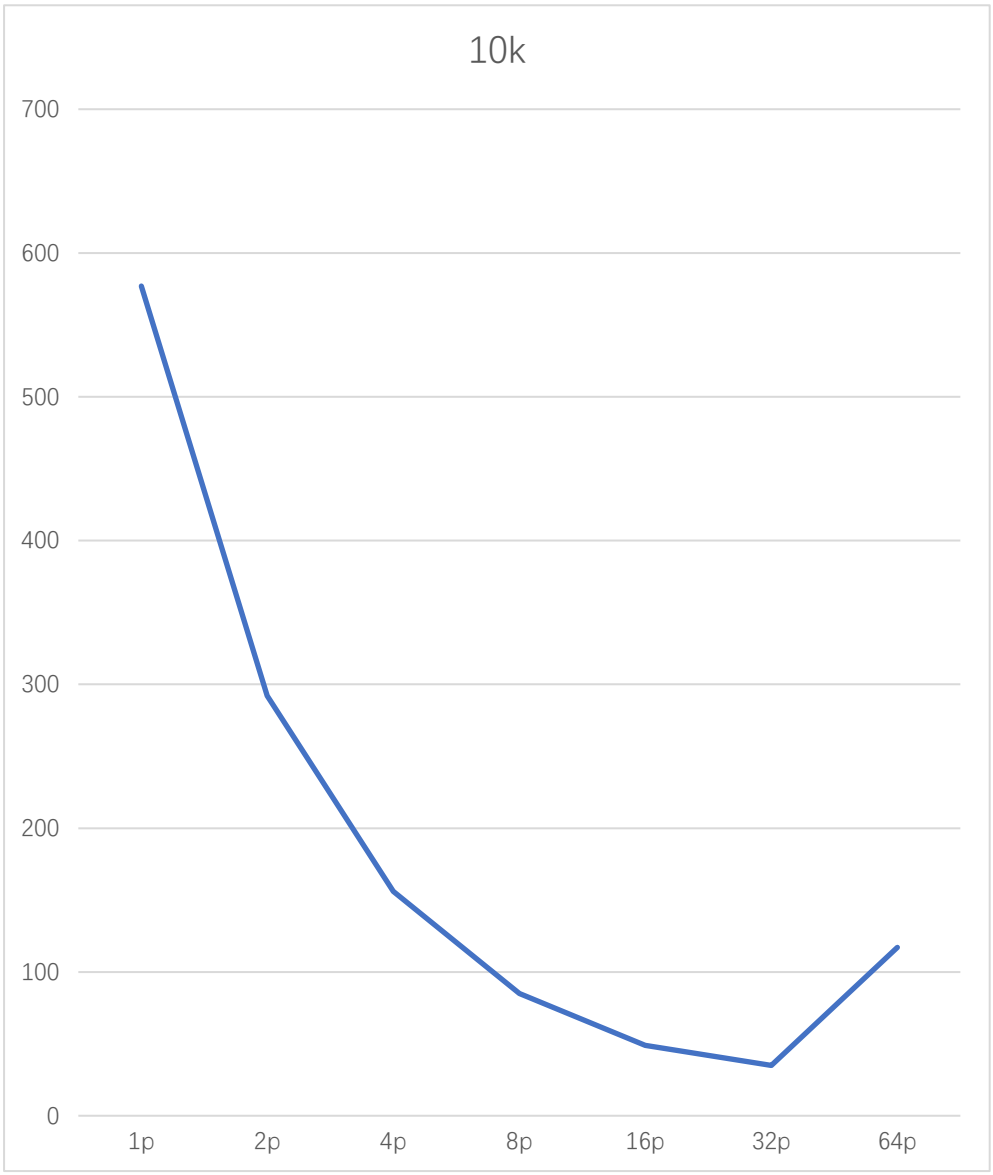


Figure 1. Execution time (ms) of the program with respect to a small input size (10k) and different number of processes

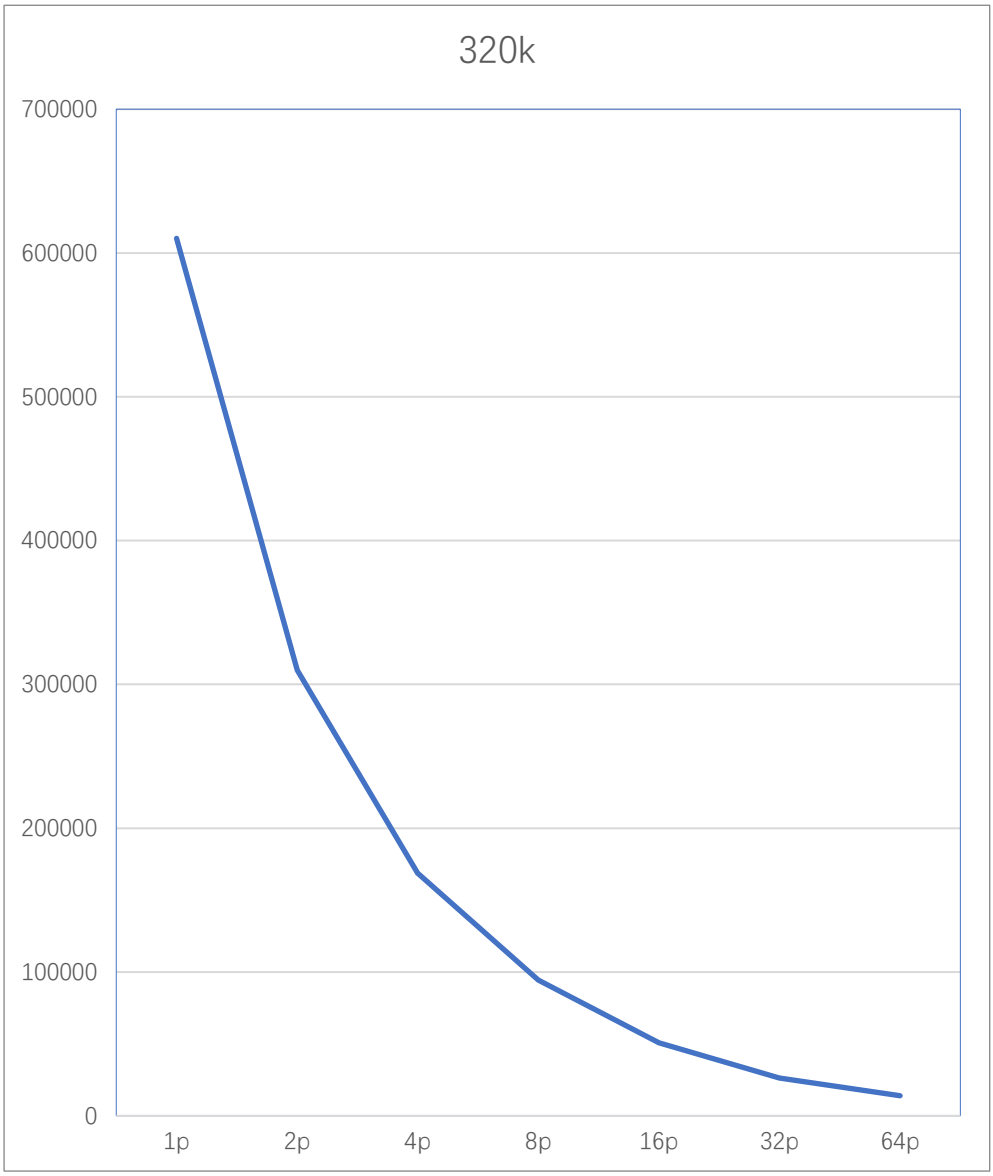


Figure 2. Execution time (ms) of the program with respect to a large input size (320k) and different number of processes

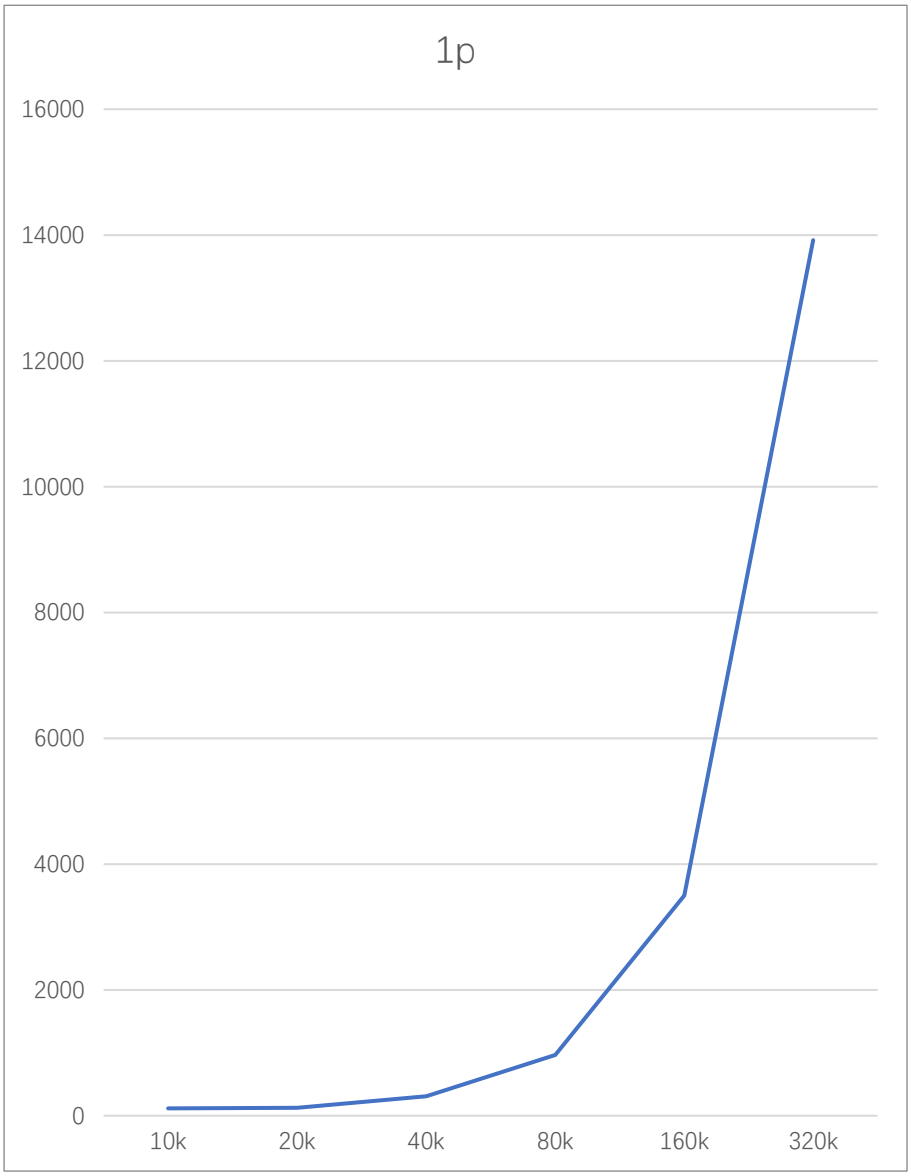


Figure 3. Execution time (ms) of the program with respect to a small number of processes (1) and different input sizes

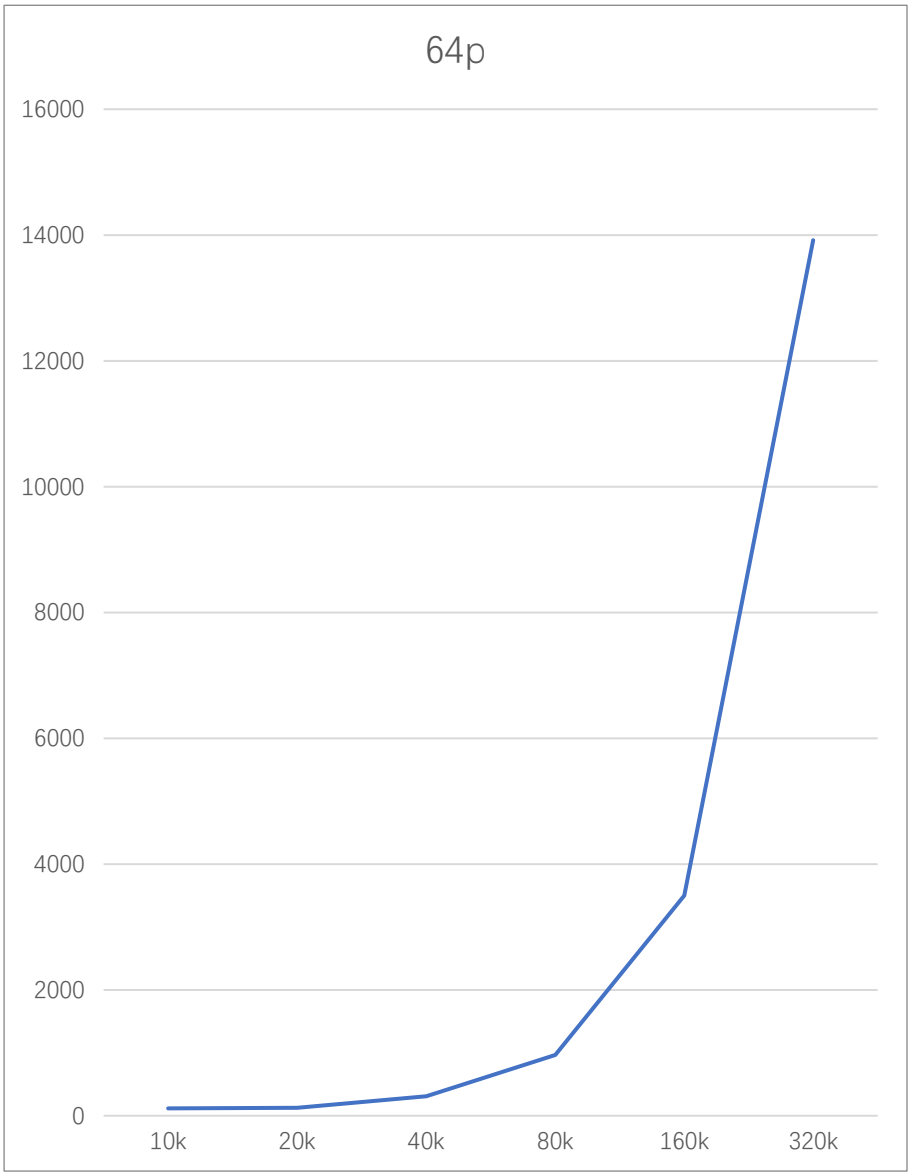


Figure 4. Execution time (ms) of the program with respect to a large number of processes (64) and different input sizes

2. Analysis

As shown in Figure 1, when the size of input data is small (10k) and the number of processes is also small (1-4), the execution time roughly halves as the number of processes doubles. However, when the number of processes grows, the reduction in execution time becomes smaller. The reason is that there is an overhead of inter-process communication. As the number of processes increases, this overhead gets larger, which makes the speedup smaller. Finally, from 32 processes to 64 processes, the execution time becomes even larger because the extra inter-process communication overhead outweighs the speedup from multiprocessing.

However, in Figure 2, when the input data size is relatively large (320k), the reduction in the overall speedup is smaller. This is because when the input data size is larger, each process has a larger computation work. For a certain number of processes, in each iteration, the amount of communication is fixed, while the amount of computation becomes larger. Therefore, the effect of the overhead is smaller.

To further illustrate the interaction between multiprocessing speedup and inter-process communication overhead, the speedup and efficiency are shown in the following figures.

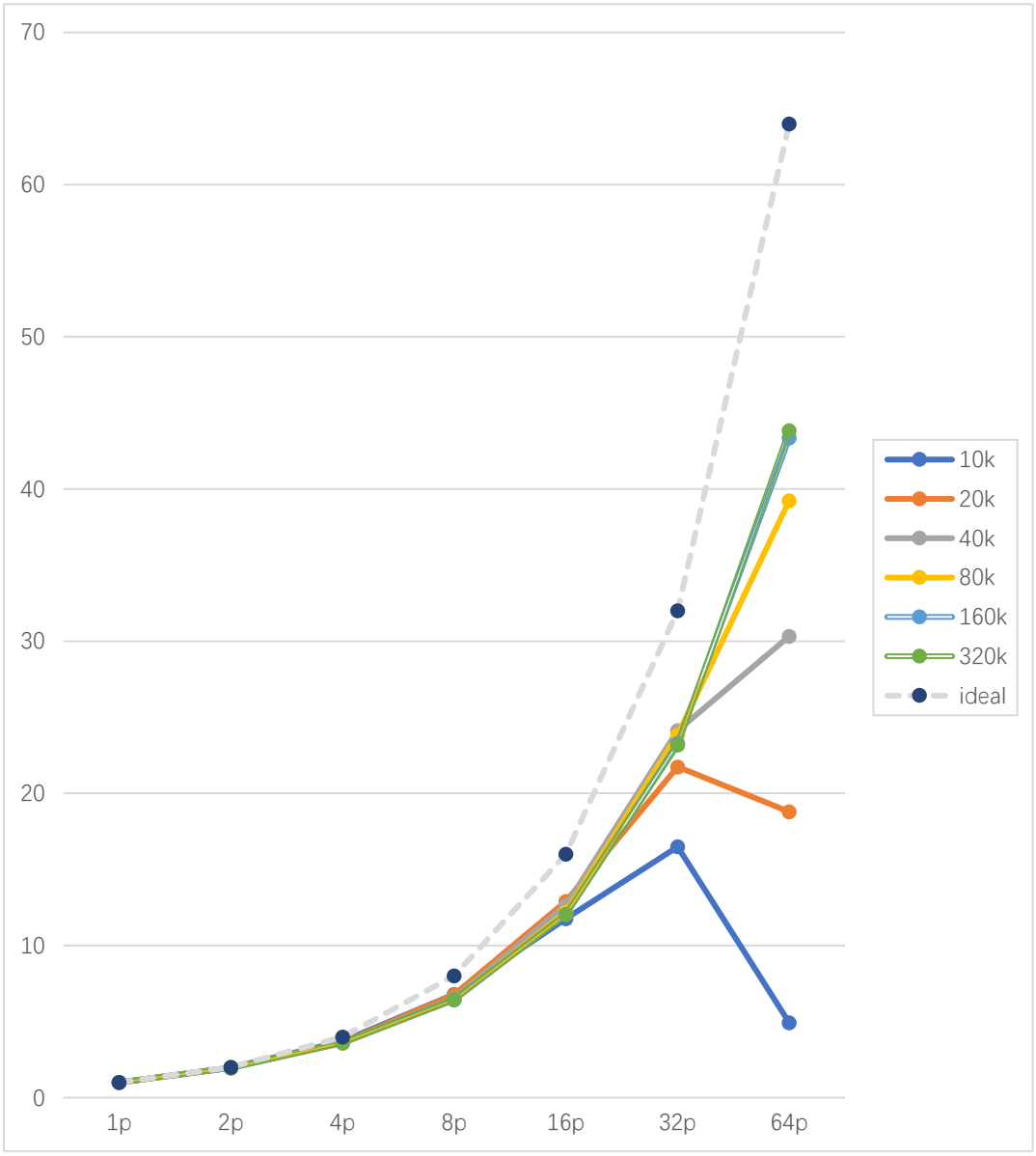


Figure 5. Actual speedup factor of the program with respect to different number of processes and input sizes

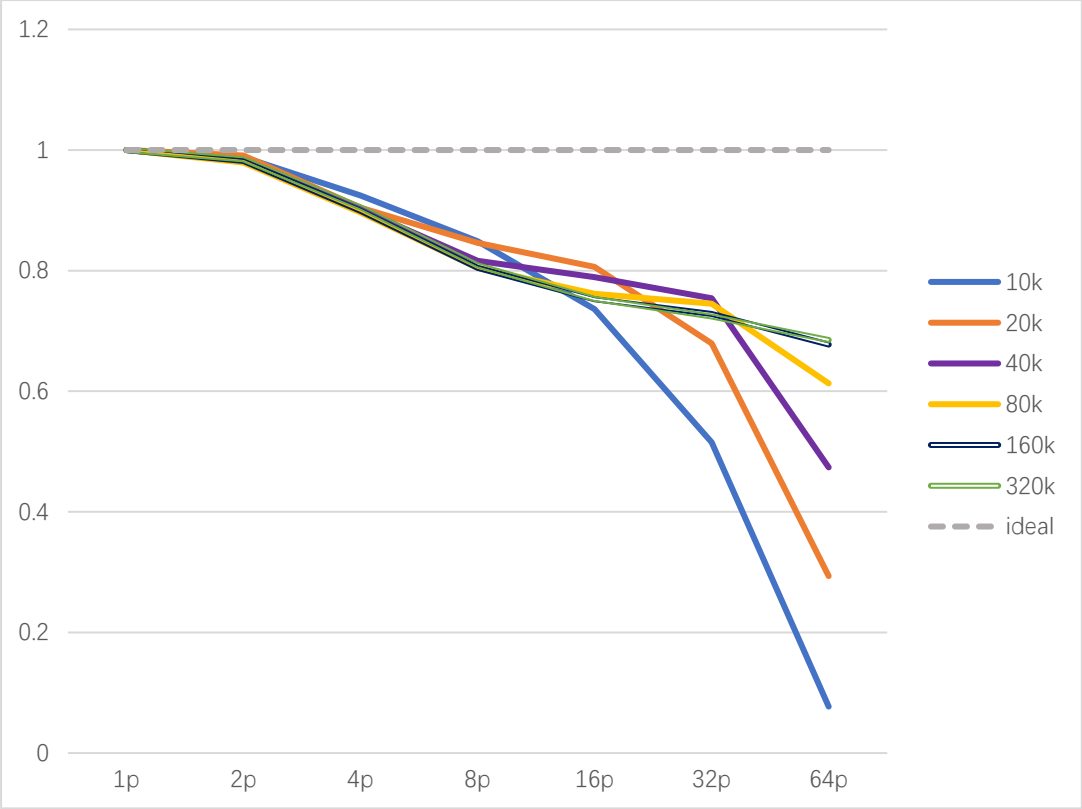


Figure 6. Efficiency of the program with respect to different number of processes and input sizes

From Figure 5 and 6, it is obvious that as the number of processes increases, the speedup factor increases slower and even decreases for small input sizes, while the efficiency decreases. Also, as the input size increases, the speedup factor becomes relatively larger, while the efficiency becomes higher. In particular, when the number of processes reaches 32, the efficiency drops dramatically. This may be related to the number of physical cores within a single node. When the number of processes is larger than 32, the server allocates 2 different nodes to execute the program. The communication overhead between different nodes is much higher than that within a single node.

V. Conclusion

In summary, this assignment explores the Parallel Odd-Even Transposition Sort algorithm, the concepts and implementation of parallel computing, the MPI framework, and the speedup and overhead of multiprocessing. Due to the inter-process communication overhead, when the number of processes becomes larger, the efficiency of multiprocessing gets lower. For small input sizes, this overhead may even outweigh the speedup from parallelism when the number of processes is large. However, for large input sizes, the impact of the communication overhead is relatively smaller since the portion of computation is larger. Therefore, parallel computing is best suitable for large input sizes.

There exist some limitations in this assignment. First, since the maximum time of a single session of the cloud server is 10 minutes, it is not feasible to test larger input sizes. Second, since the maximum processor cores of a single session of the cloud server is 64, it is not feasible to test larger number of cores. Nevertheless, from the existing experiment results, we can infer that when the input size is large enough, the efficiency of multiprocessing will be close to 1. Also, we can infer that the multiprocessing overhead will become larger and larger as the number of processors increases, and finally slows down the execution.

VI. Source Code

1. Sequential Version

```
    void Context::sequential_sort(Element *begin, Element *end) const {
        size_t size = end - begin;
        if (size < 2)
            return;
        bool is_sorted = false;
        while (!is_sorted) {
            is_sorted = true;

            // odd round
            for (int i = 1; i < size; i = i + 2) {
                if (begin[i - 1] > begin[i]){
                    swap(begin + i - 1, begin + i);
                    is_sorted = false;
                }
            }
            // even round
            for (int i = 2; i < size; i = i + 2) {
                if (begin[i - 1] > begin[i]){
                    swap(begin + i - 1, begin + i);
                    is_sorted = false;
                }
            }
        }
    }
```

2. Parallel Version

```
#include "odd-even-sort.hpp"
#include <mpi.h>
#include <iostream>
#include <vector>
#include <limits.h>

namespace sort {
    using namespace std::chrono;


    Context::Context(int &argc, char **&argv) : argc(argc), argv(argv)
{
```

```
        MPI_Init(&argc, &argv);
    }

    Context::~Context() {
        MPI_Finalize();
    }

    std::unique_ptr<Information> Context::mpi_sort(Element *begin, Elem
ent *end) const {
        int res; // result
        int rank; // rank of process
        int proc; // number of processes
        int size; // actual total size
        int global_size; // total size after padding
        int local_size; // local size (per process)
        Element send_buffer; // send buffer
        Element recv_buffer; // receive buffer
        Element* global_buffer; // global buffer containing all numbers
 (only available in process 0)
        Element* local_buffer; // local buffer containing part of numbe
rs

        std::unique_ptr<Information> information{};

        res = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if (MPI_SUCCESS != res) {
            throw std::runtime_error("failed to get MPI world rank");
        }

        if (0 == rank) {
            information = std::make_unique<Information>();
            information->length = end - begin;
            res = MPI_Comm_size(MPI_COMM_WORLD, &information->num_of_pr
oc);
            if (MPI_SUCCESS != res) {
                throw std::runtime_error("failed to get MPI world size"
);
            };
            information->argc = argc;
            for (auto i = 0; i < argc; ++i) {
                information->argv.push_back(argv[i]);
            }
            information->start = high_resolution_clock::now();
            size = end - begin;
```

```
        }
        // Broadcast the total data size
        MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Comm_size(MPI_COMM_WORLD, &proc);
        local_size = (size + proc - 1) / proc;
        local_buffer = (Element *) malloc(sizeof(Element) * local_size)
;
        global_size = local_size * proc;

        // Build the global buffer
        if (0 == rank) {
            global_buffer = (Element *) malloc(sizeof(Element) * global
_size);
            Element* element = begin;
            int i = 0;
            while (element != end) {
                global_buffer[i] = *element;
                i++;
                element++;
            }
            // do padding if there is empty space
            for (; i < global_size; i++) {
                global_buffer[i] = LONG_MAX;
            }
        }

        // Scatter the data
        MPI_Scatter(global_buffer, local_size, MPI_LONG, local_buffer,
local_size, MPI_LONG, 0, MPI_COMM_WORLD);

        for (int iteration = 0; iteration < global_size; iteration++)
        {
            // odd round
            // if the first element in the local buffer is an odd eleme
nt in the global buffer
            if ((rank * local_size) % 2 == 1) {
                for (int j = 2; j < local_size; j = j + 2) {
                    if (local_buffer[j - 1] > local_buffer[j]) {
                        swap(local_buffer + j - 1, local_buffer + j);
                    }
                }
                // send the first element back to the preceding process
 (rank - 1)
```

```
                if (rank > 0) {
                    send_buffer = local_buffer[0];
                    MPI_Send(&send_buffer, 1, MPI_LONG, rank - 1, 0, MP
I_COMM_WORLD);
                    MPI_Recv(&recv_buffer, 1, MPI_LONG, rank - 1, 0, MP
I_COMM_WORLD, MPI_STATUS_IGNORE);
                    local_buffer[0] = recv_buffer;
                }
            }
            // otherwise, the first element in the local buffer is an e
ven element in the global buffer
            else {
                for (int j = 1; j < local_size; j = j + 2) {
                    if (local_buffer[j - 1] > local_buffer[j]) {
                        swap(local_buffer + j - 1, local_buffer + j);
                    }
                }
            }
            // if the last element in the local buffer is an even eleme
nt in the global buffer
            // receive the element from the succeeding process (rank +
1)
            if ((rank + 1) * local_size % 2 == 0 && rank < proc - 1) {
                MPI_Recv(&recv_buffer, 1, MPI_LONG, rank + 1, 0, MPI_CO
MM_WORLD, MPI_STATUS_IGNORE);
                if (recv_buffer < local_buffer[local_size - 1]) {
                    send_buffer = local_buffer[local_size - 1];
                    local_buffer[local_size - 1] = recv_buffer;
                }
                else {
                    send_buffer = recv_buffer;
                }
                MPI_Send(&send_buffer, 1, MPI_LONG, rank + 1, 0, MPI_CO
MM_WORLD);
            }

            // even round
            // if the first element in the local buffer is an even elem
ent in the global buffer
            if ((rank * local_size) % 2 == 0) {
                for (int j = 2; j < local_size; j = j + 2) {
                    if (local_buffer[j - 1] > local_buffer[j]) {
                        swap(local_buffer + j - 1, local_buffer + j);
                    }
```

```
                }
                // send the first element back to the preceding process
 (rank - 1)
                if (rank > 0) {
                    send_buffer = local_buffer[0];
                    MPI_Send(&send_buffer, 1, MPI_LONG, rank - 1, 0, MP
I_COMM_WORLD);
                    MPI_Recv(&recv_buffer, 1, MPI_LONG, rank - 1, 0, MP
I_COMM_WORLD, MPI_STATUS_IGNORE);
                    local_buffer[0] = recv_buffer;
                }
            }
            // otherwise, the first element in the local buffer is an o
dd element in the global buffer
            else {
                for (int j = 1; j < local_size; j = j + 2) {
                    if (local_buffer[j - 1] > local_buffer[j]) {
                        swap(local_buffer + j - 1, local_buffer + j);
                    }
                }
            }
            // if the last element in the local buffer is an odd elemen
t in the global buffer
            // receive the element from the succeeding process (rank +
1)
            if ((rank + 1) * local_size % 2 == 1 && rank < proc - 1) {
                MPI_Recv(&recv_buffer, 1, MPI_LONG, rank + 1, 0, MPI_CO
MM_WORLD, MPI_STATUS_IGNORE);
                if (recv_buffer < local_buffer[local_size - 1]) {
                    send_buffer = local_buffer[local_size - 1];
                    local_buffer[local_size - 1] = recv_buffer;
                }
                else {
                    send_buffer = recv_buffer;
                }
                MPI_Send(&send_buffer, 1, MPI_LONG, rank + 1, 0, MPI_CO
MM_WORLD);
            }
        }

        // gather the sorted array
        MPI_Gather(local_buffer, local_size, MPI_LONG, global_buffer, l
ocal_size, MPI_LONG, 0, MPI_COMM_WORLD);
```

```
        // copy the sorted array in the global buffer
        if (0 == rank) {
            Element* element = begin;
            for (int i = 0; i < size; i++) {
                *element = global_buffer[i];
                element ++;
            }
            information->end = high_resolution_clock::now();
        }
        return information;
    }

    std::ostream &Context::print_information(const Information &info, s
td::ostream &output) {
        auto duration = info.end - info.start;
        auto duration_count = duration_cast<nanoseconds>(duration).coun
t();
        auto mem_size = static_cast<double>(info.length) * sizeof(Eleme
nt) / 1024.0 / 1024.0 / 1024.0;
        output << "input size: " << info.length << std::endl;
        output << "proc number: " << info.num_of_proc << std::endl;
        // output << "duration (ns): " << duration_count << std::endl;
        output << "duration (ms): " << duration_count / (1000 * 1000) <
< std::endl;
        output << "throughput (gb/s): " << mem_size / static_cast<doubl
e>(duration_count) * 1'000'000'000.0
               << std::endl;
        return output;
    }

    void Context::swap(Element* a, Element *b) const {
        Element temp = *b;
        *b = *a;
        *a = temp;
    }

}
```