1. Introduction

In this project, we are going to extend the odd-even transposition sort from sequential version to parallel version with MPI. This report will give the details of the design of each version and compare their performance.

2. Methods

2.1 Sequential version:

The idea of sequential odd-even transposition sort is relatively easy. First, we check each numbers in the odd position (says i-th position) of the array and ask them to compare its value for the next value which is the value of (i+1)-th number and swap with it if the number on the odd position has a smaller value. Then, we need to do the same thing on the numbers on the even position. We will repeat these two steps until the value is sorted.

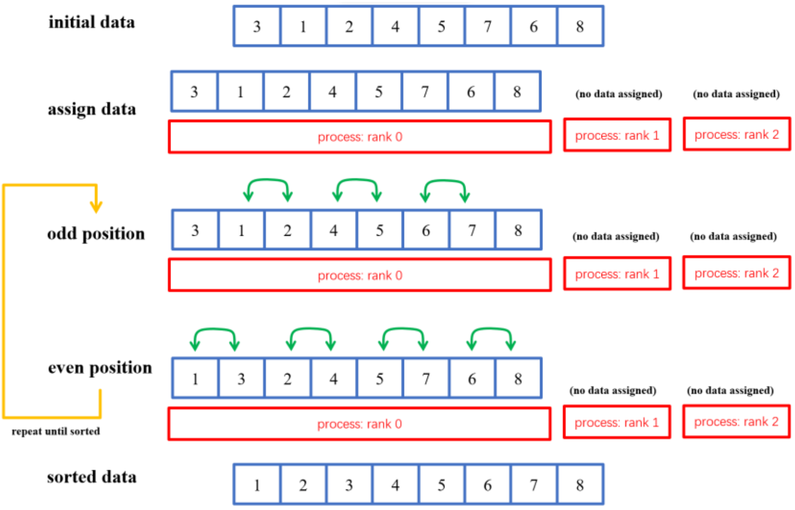


Figure 1: sequential version workflow

Correctness: The sort won't terminal until there is no swap occur, which means any the number of no matter even or odd positions at the array, will larger than its previous value. Thus, the whole array will be sorted.

2.2 Parallel version

Since we can use multiple nodes, we actually can separate the data array into all nodes and conduct the sorting in parallel way.

Step 1: The process with rank 0 would pad the array to the size which is multiple of the size of total processes. It will pad with the maximum number of the data type of the array. After that, rank 0 process would first broadcast the size of the data would be sent to each process. All processes will then allocate a space to hold that size of data and used it to hold the data scattered by the rank 0 processes next.

Step 2: This step will be the sorting step. (1) Firstly, for all processes, they will compare the data at odd position in their received array with successive data and swap then if data at odd position is bigger. Then all the process with odd rank (say N) would send its first number to the process with rank one smaller than it (N-1). After receiving data, the process (rank N-1) would compare the data with its last data and resend the larger one back to the sending process (rank N). (2) Second, all processes will compare the data at even position with its successive data, and make sure that the bigger one will be put in latter place. Then all the process with even rank (say N) would send its first data to the process one smaller than its rank (N-1). The process with smaller rank (N-1) will compare the received data with its last number and resend back the larger one. After that, the process with rank 0 would use reduce to check all the processes whether data swap has occurred. If data swap does happen, then another sorting iteration will be needed (go through sub-steps (1) and (2)), otherwise, it will jump to step 3 since all data will be sorted.

Step 3: After that, all the sorted data will be gathered into a space in process with rank 0. Then process with rank 0 would strip the padded data and put the rest data into the original data array.

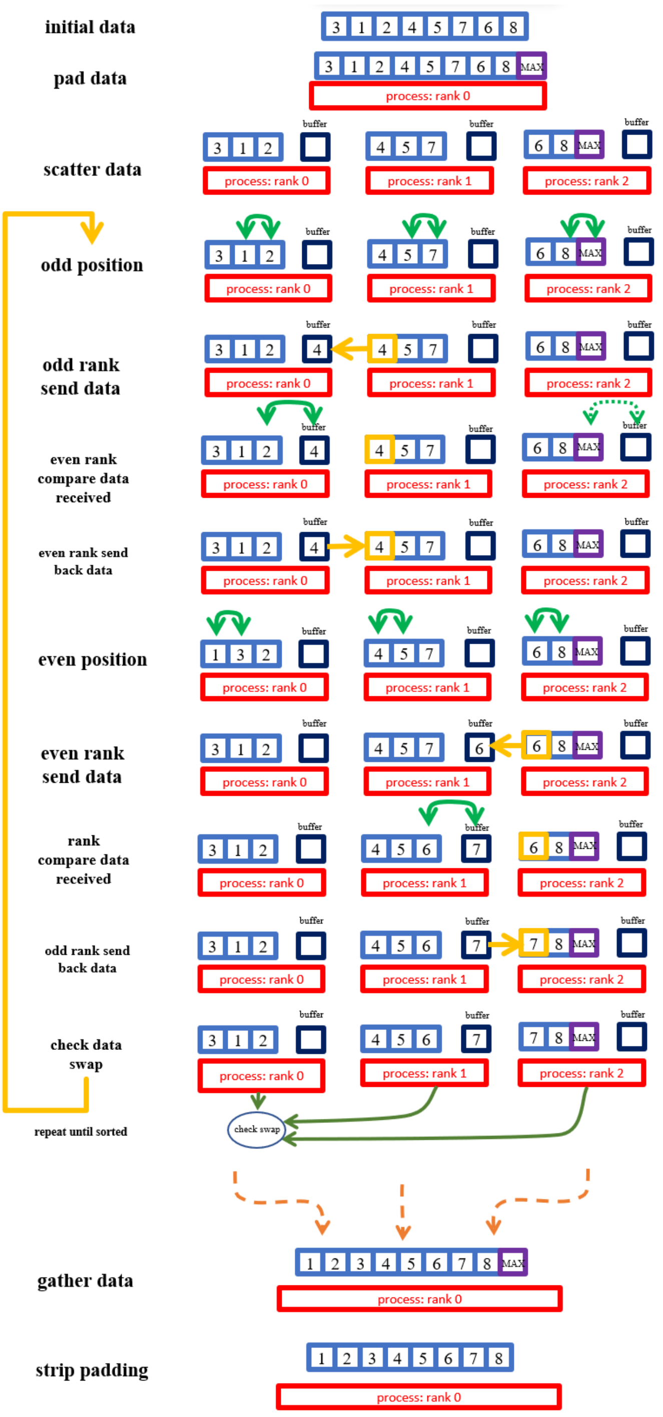


Figure 2: parallel version workflow

Correctness: the terminal condition is also no swap occurring. Since all the processes will perform odd-even sort on their local data array, which means all the local data array will be sorted when no swap occurs. We also consider the swaps between the processes. Thus, no swap means that the last number in one process local array will be no larger than the first number of the next process. Thus we make sure that all the number in process will no larger than all the numbers of its successive process. In general, the data as a whole will be sorted.

2.3 Extended parallel version

Instead of just separating the data into different nodes and do odd-even transposition sort, we can actually abstract the idea of the sorting algorithm in a higher level and combine it with other sorting algorithm. In this case, multiple data will be transferred between processes and the rule of data transporting will follow the idea of odd-even sort.

Step 1: Process with rank 0 would pad data array to a size which is the multiple of the number of processes. After that, rank 0 process would broadcast the size of number sent to each process. Then processes will allocate a buffer twice of the size of data scattered from rank 0 processes and put the data received from the beginning of the allocating position.

Step 2: This step is the sorting step. (1) Firstly, the process with odd rank number (say N) would send data to the process with 1 smaller rank (N-1). The processes receiving data will then put the received data on the latter half of the buffer it allocated (the first half will be the data scattered to it). Then it (process rank N-1) will perform sorting on the whole data array and send back the latter half of the sorting result to its successive process (rank N). (2) Next, the even rank process (say N) will instead send its data to process with one smaller rank (rank N-1). The process of rank N-1 will put the received data on the latter half of the data buffer it allocated together with the first half occupied with its data. Then the process with rank N-1 will perform sorting on the whole array and send back latter half of sorted data to its successive one (rank N). After (1) and (2), the process will use reduce to check whether data swap was appended in any process and broadcast the result to all the processes. If no data swap happened, then processes will jump to step 3, otherwise, another sorting iteration of (1) and (2) will be needed.

Step 3: The rank 0 process would gather the local data from all processes and strip the data padded and copy the rest of data to the original array.

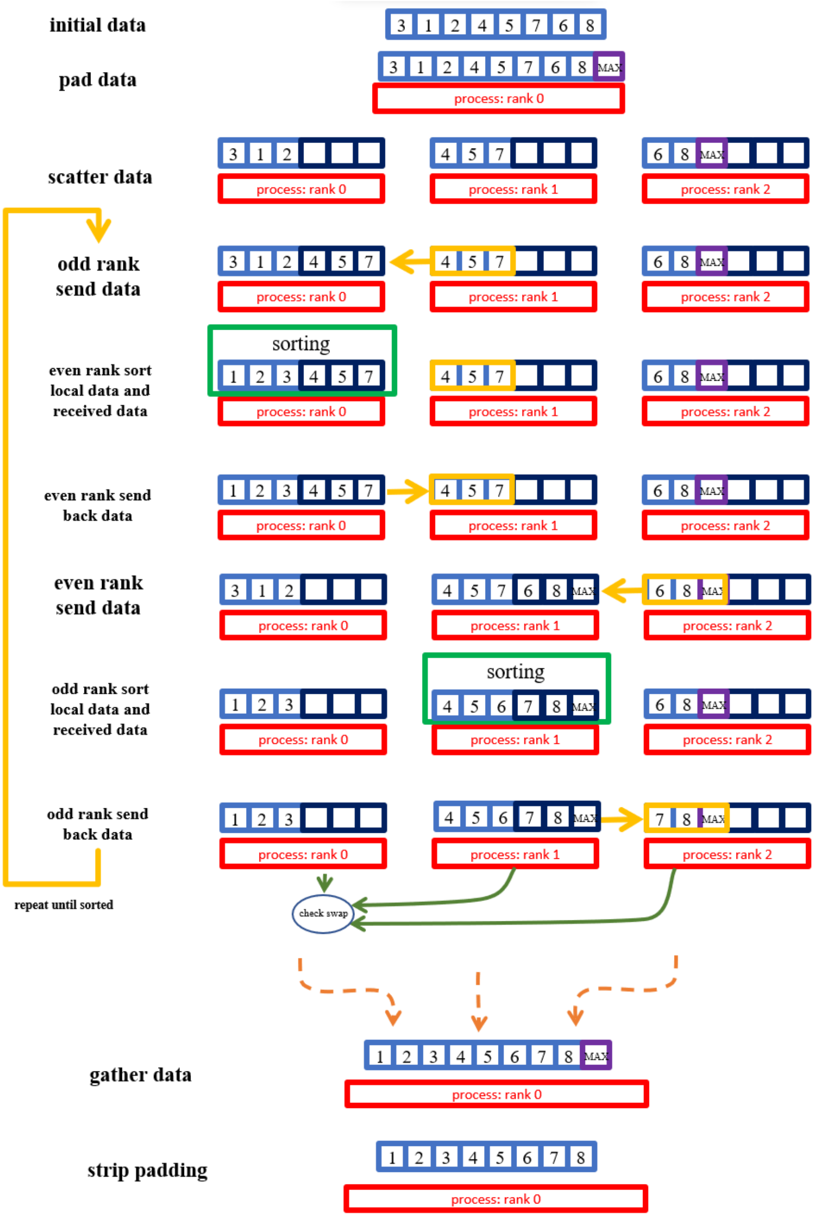


Figure 3: extended parallel version workflow

Correctness: the terminal condition for this program is still no swap occur. Since we basically abstract the odd-even sort at higher level, we can actually treat every local array as a single element as a whole, which then reduce the whole program as simple odd even sort just like parallel version except that all process only contains one element. Also, all the number at one local array will be sorted automatically after received the sorted sent data from its previous process or when it sorted its number with the numbers from its successive process all together. Thus, the number will be sorted when no swap occurrs.

3. Program execution

Compile and build:

Put whole folder under /pvfsmnt/(student_id) and enter one of folder of the project like '/pvfsmnt/118010224/parallel_version'. Type 'mkdir build' to create a folder called 'build' and type 'cd build' to enter it. Type 'cmake .. -DCMAKE_BUILD_TYPE=Release' and 'cmake --build . -j4' to build and compile the program.

Execution:

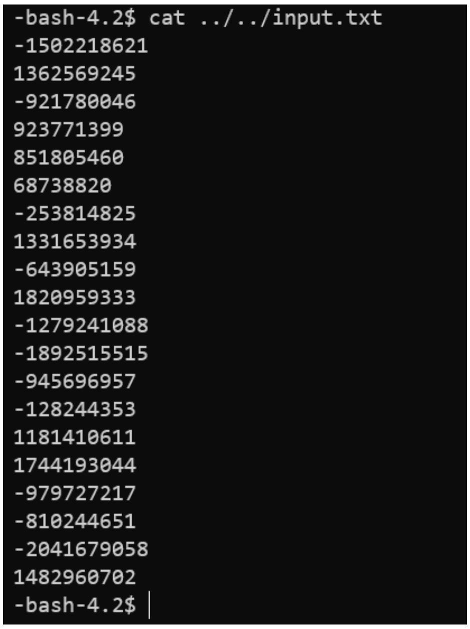
You need to first allocate some node resources using salloc to execute the program in parallel way. After that, you need to type 'srun hostname'.

1) main:

To run the 'main' program, you need to type 'mpirun ./main <input-file> <output-file>' on the build folder.

Results (1 nodes with totally 3 cores):

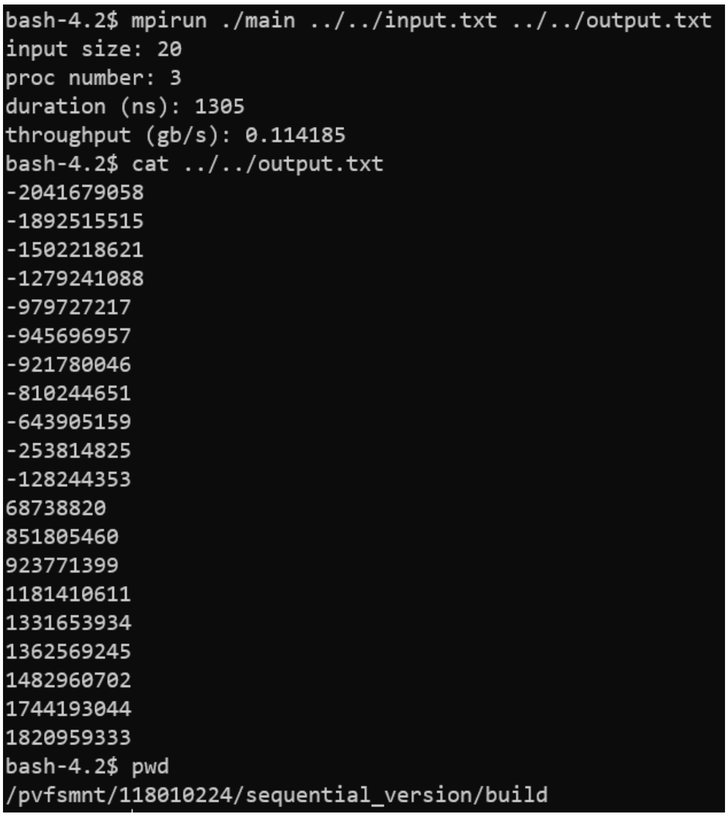
The input file will be same for all the three program, which is showing in the following:



```
-bash-4.2$ cat ../../input.txt
-1502218621
1362569245
-921780046
923771399
851805460
68738820
-253814825
1331653934
-643905159
1820959333
-1279241088
-1892515515
-945696957
-128244353
1181410611
1744193044
-979727217
-810244651
-2041679058
1482960702
-bash-4.2$ 
```

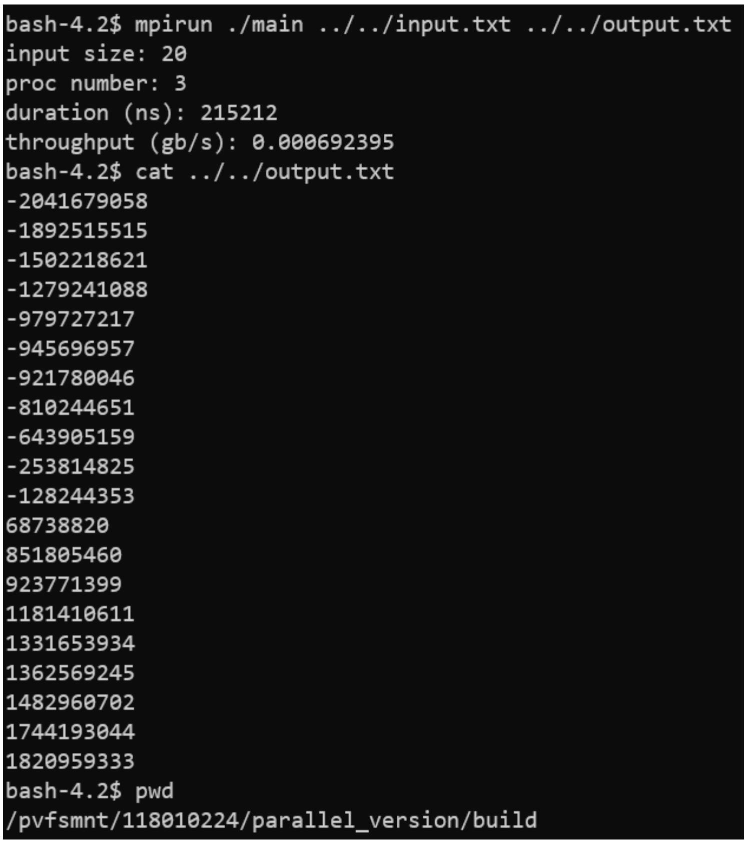
Notice that at the beginning of every tests, the output file is cleared, although it is not shown in the screenshots.

a. sequential sorting



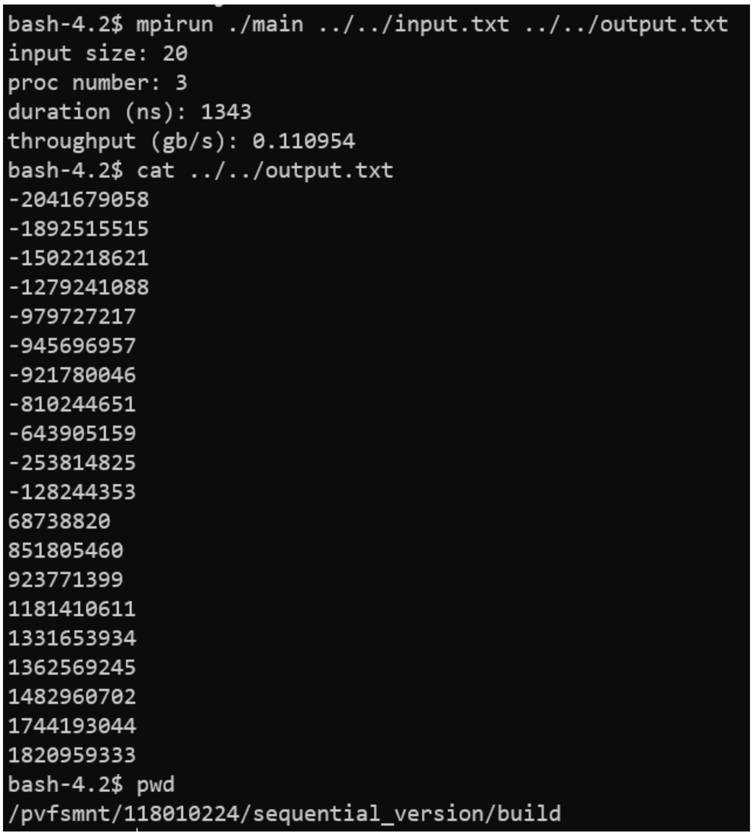
```
bash-4.2$ mpirun ./main ../../input.txt ../../output.txt
input size: 20
proc number: 3
duration (ns): 1305
throughput (gb/s): 0.114185
bash-4.2$ cat ../../output.txt
-2041679058
-1892515515
-1502218621
-1279241088
-979727217
-945696957
-921780046
-810244651
-643905159
-253814825
-128244353
68738820
851805460
923771399
1181410611
1331653934
1362569245
1482960702
1744193044
1820959333
bash-4.2$ pwd
/pvfsmnt/118010224/sequential_version/build
```

b. parallel sorting



```
bash-4.2$ mpirun ./main ../../input.txt ../../output.txt
input size: 20
proc number: 3
duration (ns): 215212
throughput (gb/s): 0.000692395
bash-4.2$ cat ../../output.txt
-2041679058
-1892515515
-1502218621
-1279241088
-979727217
-945696957
-921780046
-810244651
-643905159
-253814825
-128244353
68738820
851805460
923771399
1181410611
1331653934
1362569245
1482960702
1744193044
1820959333
bash-4.2$ pwd
/pvfsmnt/118010224/parallel_version/build
```

c. extended parallel sorting



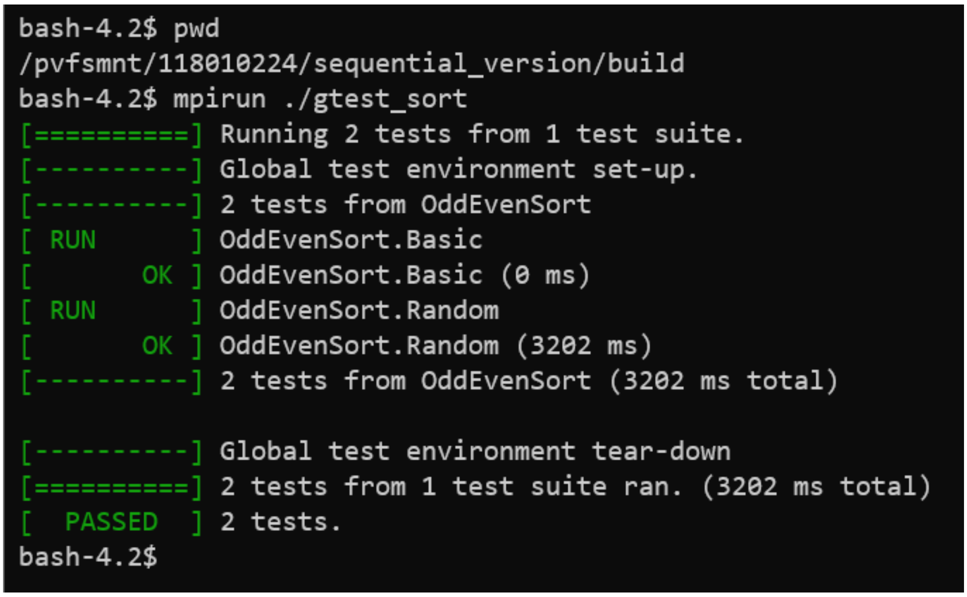
```
bash-4.2$ mpirun ./main ../../input.txt ../../output.txt
input size: 20
proc number: 3
duration (ns): 1343
throughput (gb/s): 0.110954
bash-4.2$ cat ../../output.txt
-2041679058
-1892515515
-1502218621
-1279241088
-979727217
-945696957
-921780046
-810244651
-643905159
-253814825
-128244353
68738820
851805460
923771399
1181410611
1331653934
1362569245
1482960702
1744193044
1820959333
bash-4.2$ pwd
/pvfsmnt/118010224/sequential_version/build
```

2) gtest_sort:

Type ‘mpirun ./gtest_sort’ directly in the build folder.

Results:

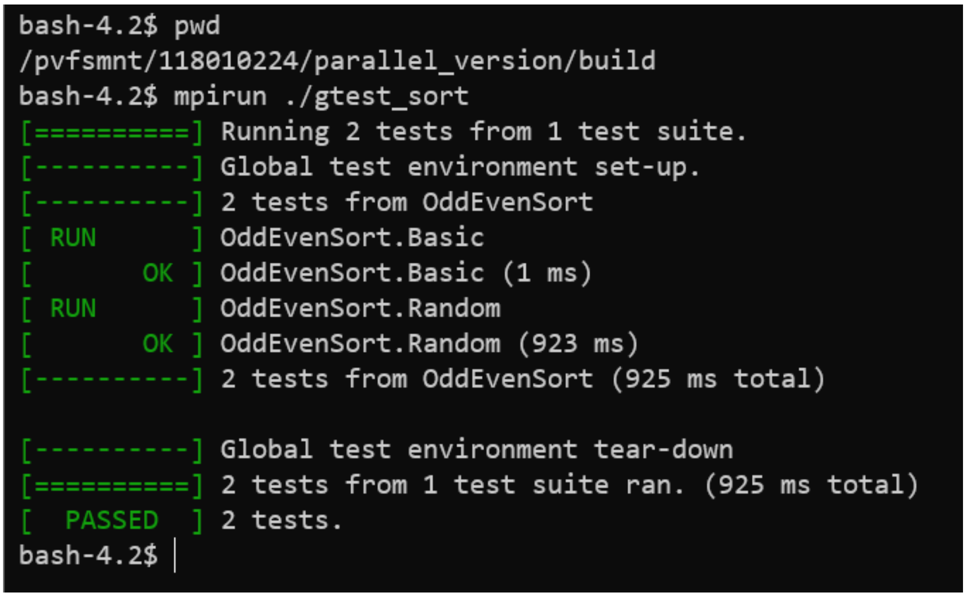
a. sequential sorting:



```
bash-4.2$ pwd
/pvfsmnt/118010224/sequential_version/build
bash-4.2$ mpirun ./gtest_sort
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from OddEvenSort
[ RUN      ] OddEvenSort.Basic
[       OK ] OddEvenSort.Basic (0 ms)
[ RUN      ] OddEvenSort.Random
[       OK ] OddEvenSort.Random (3202 ms)
[----------] 2 tests from OddEvenSort (3202 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (3202 ms total)
[  PASSED  ] 2 tests.
bash-4.2$
```

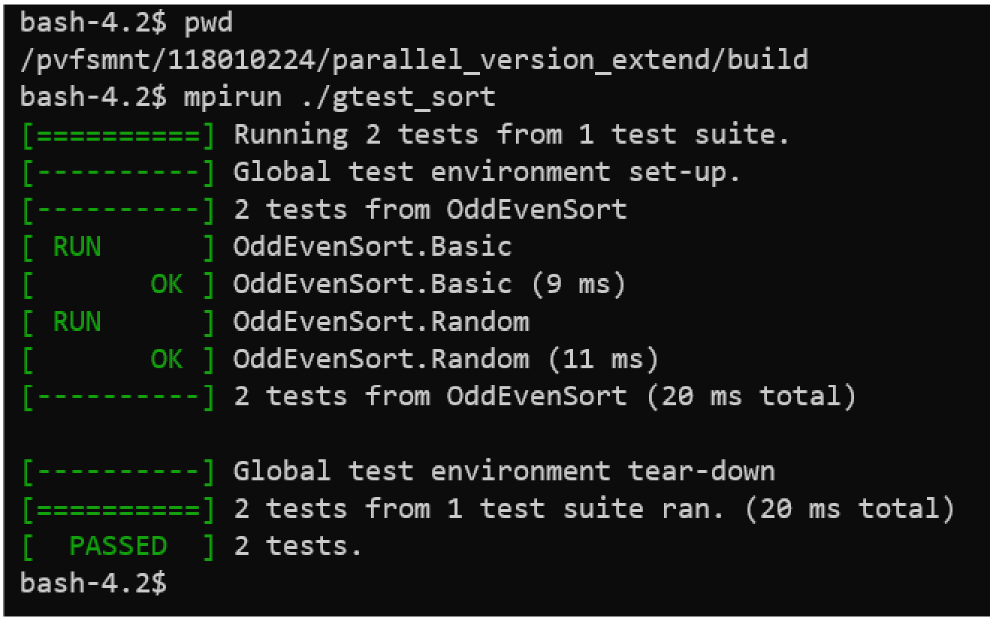
b. parallel sorting:



```
bash-4.2$ pwd
/pvfsmnt/118010224/parallel_version/build
bash-4.2$ mpirun ./gtest_sort
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from OddEvenSort
[ RUN      ] OddEvenSort.Basic
[       OK ] OddEvenSort.Basic (1 ms)
[ RUN      ] OddEvenSort.Random
[       OK ] OddEvenSort.Random (923 ms)
[----------] 2 tests from OddEvenSort (925 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (925 ms total)
[  PASSED  ] 2 tests.
bash-4.2$
```

c. extended sorting:



```
bash-4.2$ pwd
/pvfsmnt/118010224/parallel_version_extend/build
bash-4.2$ mpirun ./gtest_sort
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from OddEvenSort
[ RUN      ] OddEvenSort.Basic
[       OK ] OddEvenSort.Basic (9 ms)
[ RUN      ] OddEvenSort.Random
[       OK ] OddEvenSort.Random (11 ms)
[----------] 2 tests from OddEvenSort (20 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (20 ms total)
[  PASSED  ] 2 tests.
bash-4.2$
```

4. Performance analysis

1) Time complexity:

a. Sequential case:

Assume the size of data array is N. At the worst case, if the biggest data is set in the first place, it needs N swaps to swap it to the last position. And we have totally N number which means we may at most N*N swap to put every data in right place. Therefore, the time complexity is $O(N^2)$.

b. Parallel case:

Assume there are N numbers and p processes. Since the sorting could be divided into p processes, the total time for sorting would become $O(\frac{N^2}{p})$. The time for communication for one iteration of sorting is $O(1)$ and we need at most N iterations to complete sorting, then totally is $O(N)$. Also, the rank 0 process needs to make another copy of the data and put back the new data into original array which would be $O(2N) = O(N)$. Therefore, the total time complexity would be $O\left(\frac{N^2}{p}\right) + O(3N) = O\left(\frac{N^2}{p}\right) + O(N)$. In this case, the parallel time will be the sorting part which is $O\left(\frac{N^2}{p}\right)$ while sequential time including communication cost and data array copy and restore is $O(N)$.

c. Extended parallel case:

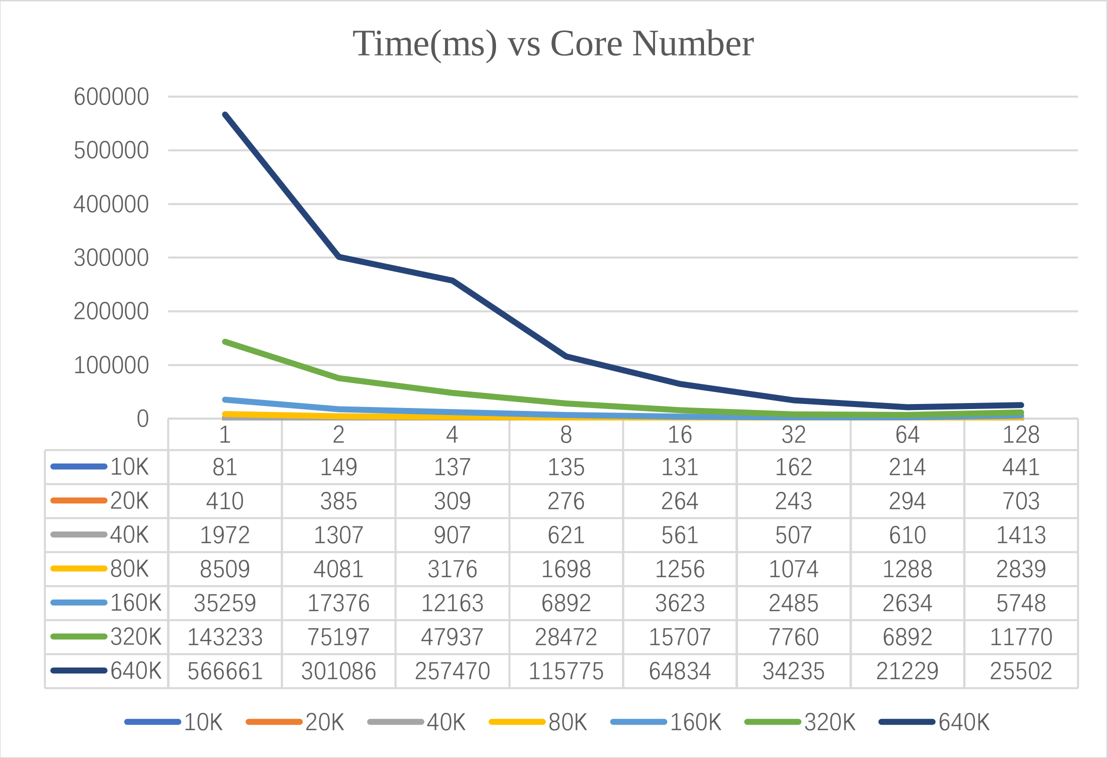
Assume there are N numbers and p processes. In this case, since the program directly use the sorting function in C++, the time complexity for the sorting in each process would be $O(\frac{N}{p}\log\frac{N}{p})$. If we abstract the process in a higher level, we can consider all the numbers in one process as a single number as a whole. In this case, we only need p iterations of sorting. Since the time for one iteration of data transportation is $O\left(\frac{N}{p} * 2\right)$, the total time for communication would be $O\left(\frac{N}{p} * 2 * p\right) = O(N)$. And the total time used on sorting would be $O\left(\frac{N}{p}\log\frac{N}{p} * p\right) = O(N\log\frac{N}{p})$. Also, rank 0 process still need $O(2 * N)$ time to copy and restore data array. Thus, the time complexity would be $O\left(N\log\frac{N}{p}\right) + O(N)$. The parallel part would be $O\left(N\log\frac{N}{p}\right)$ (sorting time) and sequential para would be $O(N)$.

2) Test:

The report will mainly focus on the parallel version sorting which does not include the extended one.

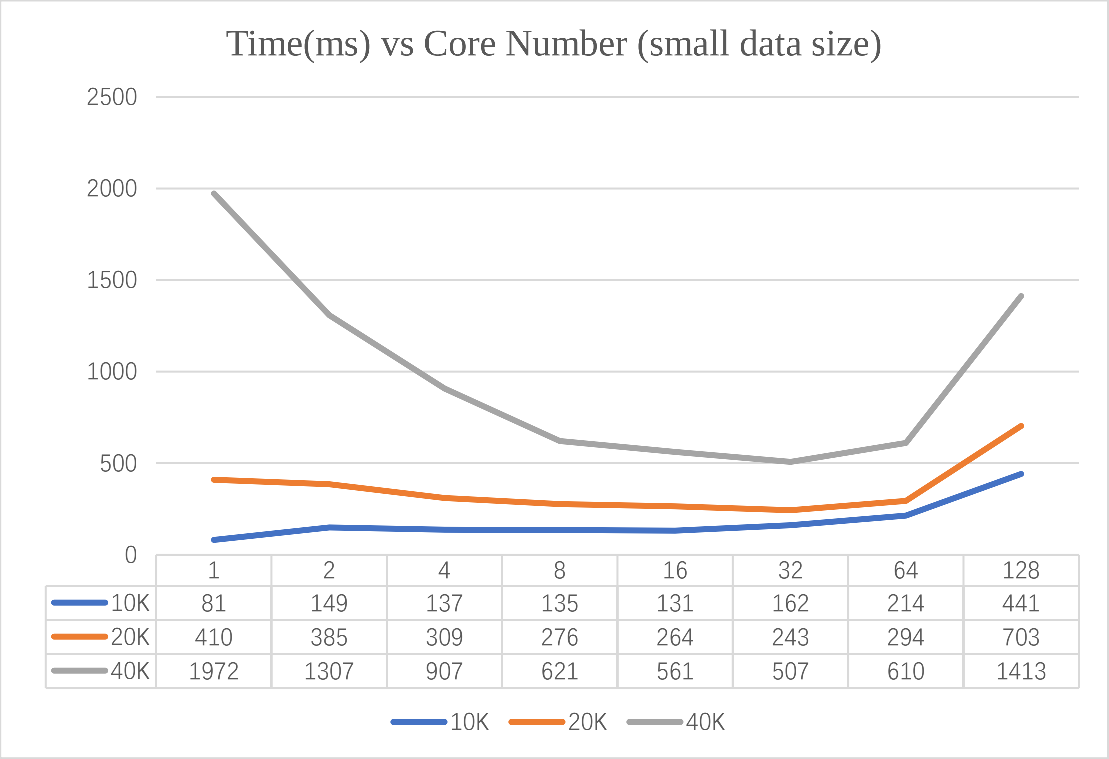
a. Time vs Core Number:

The configuration of the cluster constraints that one CPU does not contain so many cores. To make the test result more reliable, the test will be conducted within small nodes. For sort with one process, it will be conducted within one node. For sort with number of cores from 2 to 64, it will be conducted on two nodes. For the sort with 128 nodes, it will be tested with 4 nodes. All the test results are based on the random test part gtest_sort program provided with data size modified.



	1	2	4	8	16	32	64	128
10K	81	149	137	135	131	162	214	441
20K	410	385	309	276	264	243	294	703
40K	1972	1307	907	621	561	507	610	1413
80K	8509	4081	3176	1698	1256	1074	1288	2839
160K	35259	17376	12163	6892	3623	2485	2634	5748
320K	143233	75197	47937	28472	15707	7760	6892	11770
640K	566661	301086	257470	115775	64834	34235	21229	25502

It is obvious that the running time will generally decrease with more cores added and it generally follows the pattern that double the number if cores will result in half reduce in the running time.



	1	2	4	8	16	32	64	128
10K	81	149	137	135	131	162	214	441
20K	410	385	309	276	264	243	294	703
40K	1972	1307	907	621	561	507	610	1413

However, the result at the sort with data in small size shows a different trend. When sorting with 10k, 20k and 40k, the performance of the program would first improved then fall which is resulted from the communication overhead between processes. Although increased number of cores do help in reduce the time of sorting, the increasing communication cost offset that improved sorting time.

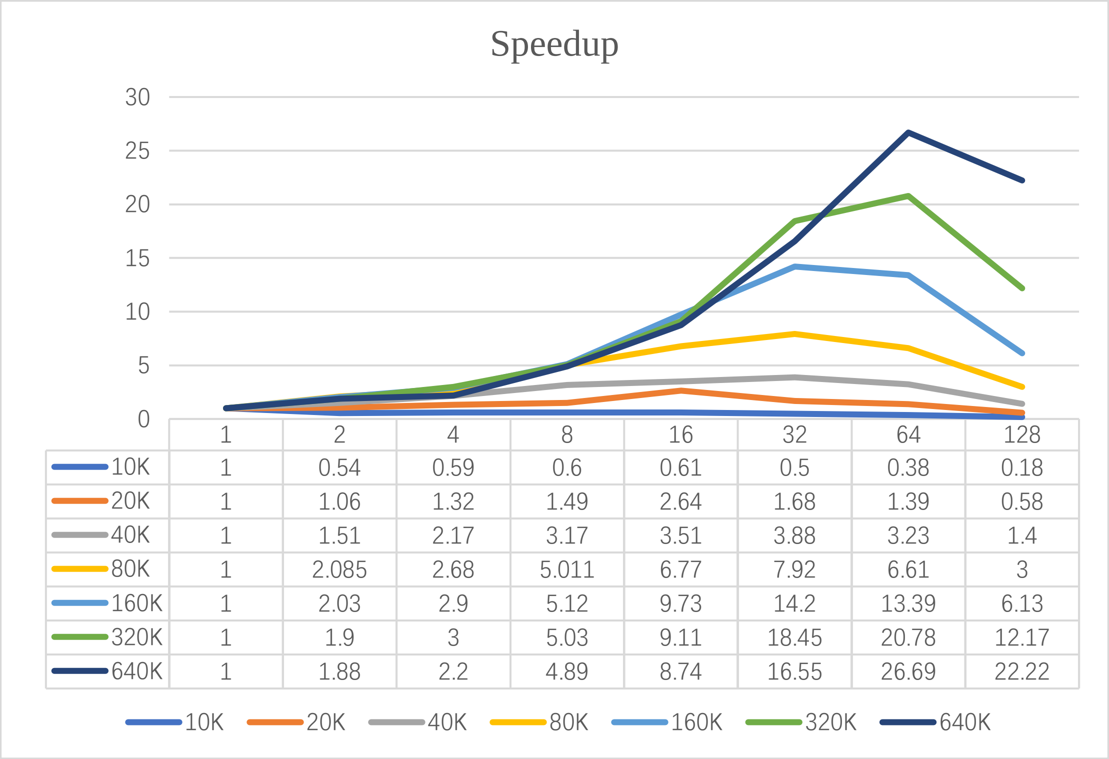
b. Speedup

We can also calculate the Speedup of the program.

The formula of the Speedup is given as:

$$Speedup = \frac{running\ time\ on\ one\ server}{running\ time\ on\ parallel\ server}$$

And we can actually calculate it from our obtained data.



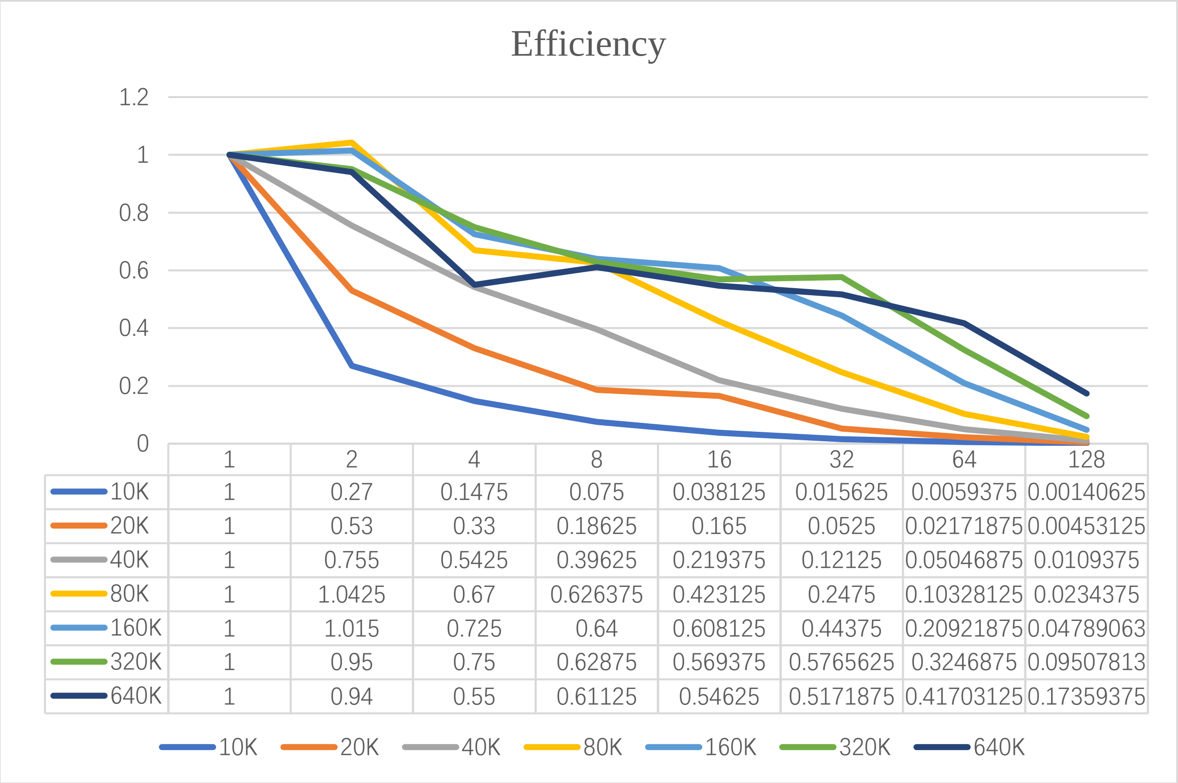
	1	2	4	8	16	32	64	128
10K	1	0.54	0.59	0.6	0.61	0.5	0.38	0.18
20K	1	1.06	1.32	1.49	2.64	1.68	1.39	0.58
40K	1	1.51	2.17	3.17	3.51	3.88	3.23	1.4
80K	1	2.085	2.68	5.011	6.77	7.92	6.61	3
160K	1	2.03	2.9	5.12	9.73	14.2	13.39	6.13
320K	1	1.9	3	5.03	9.11	18.45	20.78	12.17
640K	1	1.88	2.2	4.89	8.74	16.55	26.69	22.22

The result shows that generally the sorting performance will first reach a peak then falls down because of the increasing communication overhead. To obtain a better performance on sorting, it is quite necessary to balance the data size and core numbers to achieve the best result.

c. Efficiency

The efficiency formula is given as:

$$E = \frac{Speedup}{number\ of\ processes}$$



	1	2	4	8	16	32	64	128
10K	1	0.27	0.1475	0.075	0.038125	0.015625	0.0059375	0.00140625
20K	1	0.53	0.33	0.18625	0.165	0.0525	0.02171875	0.00453125
40K	1	0.755	0.5425	0.39625	0.219375	0.12125	0.05046875	0.0109375
80K	1	1.0425	0.67	0.626375	0.423125	0.2475	0.10328125	0.0234375
160K	1	1.015	0.725	0.64	0.608125	0.44375	0.20921875	0.04789063
320K	1	0.95	0.75	0.62875	0.569375	0.5765625	0.3246875	0.09507813
640K	1	0.94	0.55	0.61125	0.54625	0.5171875	0.41703125	0.17359375

In general, the efficiency will generally decrease with the increase of number of cores. One possible reason is that with more cores running parallel, the data size assigned to each process will decrease a lot, which result in the ratio of sorting time to communication time for each process increase. Since each core will spend generally more time for waiting, their efficiency decrease.

5. Conclusion and limitations

The report discuss the parallel design of the odd-even transposition sort and it then further apply its idea in a much abstract level. Tested with different cores and data size, the report shows the improvement of parallel sorting and also discuss the communication overhead may offset the performance improvement brough by parallel sorting. There are some limitations of the program: since the processes are only allowed to send and receive one data, it may require the process multiple iterations to pass data from one to another completely. Although the extended version of sorting solves this problem by allowing multiple data transposition and the performance do improve (from the result of the execution result in 3), it still has some resource wasted since during one process sorting, the paired process need to wait and do nothing. To improve the efficiency of the extended version, the program can be designed to sort two data arrays simultaneously, which may not waste too much time on waiting.

Codes (odd-even-sort.cpp for different versions):

Parallel version:

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

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, Element *end) const {
        int res;
        int rank;
        int size;
        int data_package_size;
        std::vector<Element> padded_data_array;

        std::unique_ptr<Information> information{};

        res = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);


        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_proc);
            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();
        }

        {
            /// now starts the main sorting procedure

            // local buffer holding data sent from other process
            Element data_from_other_process;

            // step 1
            // rank 0 scatter data to all process
            if (rank == 0) {
                int original_total_data_size = end - begin;
                int padded_total_data_size = original_total_data_size;

                // allocate a new array for padded array
                padded_data_array.assign(begin, end);

                // pad positive infinity to the data array
                int size_remainder = original_total_data_size % size;
                if(size_remainder != 0){
                    int padded_size = size - size_remainder;
                    padded_total_data_size += padded_size;

                    for(int i=0 ; i<padded_size; i++){
                        padded_data_array.push_back(std::numeric_limits<Element
>::max());
                    }
                }
                data_package_size = padded_total_data_size / size;
            }

            // boardcast the size of data received for each process
            MPI_Bcast(&data_package_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
            // allocate buffer holding data
            std::vector<Element> local_data_buffer(data_package_size);

            // scatter data
            MPI_Scatter(padded_data_array.data(), data_package_size, MPI_LONG, local
_data_buffer.data(), data_package_size, MPI_LONG, 0, MPI_COMM_WORLD);

            // step 2
            // sort data until sorted
            bool sorted_flag_all = false;
            bool sorted_flag_local = false;

            while(!sorted_flag_all){
                sorted_flag_local = true;

                // sort the odd position
                for(int i=1;i<data_package_size-1;i+=2){
                    if(local_data_buffer[i] > local_data_buffer[i+1]){
                        Element temp = local_data_buffer[i];
                        local_data_buffer[i] = local_data_buffer[i+1];
                        local_data_buffer[i+1] = temp;
                        sorted_flag_local = false;
                    }
                }

                // odd rank send data
                if(size % 2 != 0 && rank == size-
1){} // last process would do nothing if the total size is odd
                else if(rank % 2 != 0){
                    // send first number to previous process and wait for result
                    MPI_Sendrecv(local_data_buffer.data(), 1, MPI_LONG, (rank-
1+size)%size, 0,
                                local_data_buffer.data(), 1, MPI_LONG, (rank-
1+size)%size, 0,
                                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
                else{
                    // receive data from other process and compare the value with th
e last number
                    MPI_Recv(&data_from_other_process, 1, MPI_LONG, (rank+1+size)%si
ze, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    if(data_from_other_process < local_data_buffer[data_package_size
-1]){
```

```
                        Element temp = data_from_other_process;
                        data_from_other_process = local_data_buffer[data_package_siz
e-1];
                        local_data_buffer[data_package_size-1] = temp;
                        sorted_flag_local = false;
                    }
                    MPI_Send(&data_from_other_process, 1, MPI_LONG, (rank+1+size)%si
ze, 0, MPI_COMM_WORLD);
                }

                // sort the even position
                for(int i=0;i<data_package_size-1;i+=2){
                    if(local_data_buffer[i] > local_data_buffer[i+1]){
                        Element temp = local_data_buffer[i];
                        local_data_buffer[i] = local_data_buffer[i+1];
                        local_data_buffer[i+1] = temp;
                        sorted_flag_local = false;
                    }
                }

                // even rank send data
                if(rank != 0){
                    if(size%2 == 0 && rank == size-
1){} // last process would do nothing if the total size is even
                    else if(rank % 2 != 1){
                        // send first number to previous process and wait for result
                        MPI_Sendrecv(local_data_buffer.data(), 1, MPI_LONG, (rank-
1+size)%size, 0,
                                    local_data_buffer.data(), 1, MPI_LONG, (rank-
1+size)%size, 0,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    }
                    else{
                        // receive data from other process and compare the value wit
h the last number
                        MPI_Recv(&data_from_other_process, 1, MPI_LONG, (rank+1+size
)%size, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if(data_from_other_process < local_data_buffer[data_package_
size-1]){
                            Element temp = data_from_other_process;
                            data_from_other_process = local_data_buffer[data_package
_size-1];
                            local_data_buffer[data_package_size-1] = temp;
                            sorted_flag_local = false;
                        }
                        MPI_Send(&data_from_other_process, 1, MPI_LONG, (rank+1+size
)%size, 0, MPI_COMM_WORLD);
                    }
                }

                // check whether no swap has happened in all processes
                MPI_Reduce(&sorted_flag_local, &sorted_flag_all, 1, MPI_C_BOOL, MPI_
LAND, 0, MPI_COMM_WORLD);

                // broadcast the signal whether another sort iteration is necessary
                MPI_Bcast(&sorted_flag_all, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
            }

            // step 3
            // gather sorted array
            MPI_Gather(local_data_buffer.data(), data_package_size, MPI_LONG, padded
_data_array.data(), data_package_size, MPI_LONG, 0, MPI_COMM_WORLD);
            if(rank == 0){
                // strip the padded data and put needed data back to original array
                copy(padded_data_array.begin(), padded_data_array.begin()+(end-
begin), begin);
            }
        }

        if (0 == rank) {
            information->end = high_resolution_clock::now();
        }
        return information;
    }



    std::ostream &Context::print_information(const Information &info, std::ostream &
output) {
        auto duration = info.end - info.start;
        auto duration_count = duration_cast<nanoseconds>(duration).count();
        auto mem_size = static_cast<double>(info.length) * sizeof(Element) / 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 << "throughput (gb/s): " << mem_size / static_cast<double>(duration_c
ount) * 1'000'000'000.0
```

```
               << std::endl;
        return output;
    }
}
```

Sequential version:

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

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, Element *end) con
st {
        int res;
        int rank;

        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_proc);
            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();
    }

    {
        /// now starts the main sorting procedure
        if(rank == 0){
            bool unchanged_flag = false;
            int array_size = end - begin;
            while(!unchanged_flag){
                unchanged_flag = true;
                // odd position
                for(int i=1;i<array_size-1;i+=2){
                    if(begin[i] > begin[i+1]){
                        Element temp = begin[i];
                        begin[i] = begin[i+1];
                        begin[i+1] = temp;
                        unchanged_flag = false;
                    }
                }

                // even position
                for(int i=0;i<array_size-1;i+=2){
                    if(begin[i] > begin[i+1]){
                        Element temp = begin[i];
                        begin[i] = begin[i+1];
                        begin[i+1] = temp;
                        unchanged_flag = false;
                    }
                }
            }
        }
    }

    if (0 == rank) {
        information->end = high_resolution_clock::now();
    }
    return information;
}
```

```
    std::ostream &Context::print_information(const Information &info, std::ostream &output) {
        auto duration = info.end - info.start;
        auto duration_count = duration_cast<nanoseconds>(duration).count();
        auto mem_size = static_cast<double>(info.length) * sizeof(Element) / 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 << "throughput (gb/s): " << mem_size / static_cast<double>(duration_count) * 1'000'000'000.0
               << std::endl;
        return output;
    }
}
```

Extended parallel version:

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

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, Element *end) const {
```

```
    int res;
    int rank;
    int size;
    int data_package_size;
    std::vector<Element> padded_data_array;

    std::unique_ptr<Information> information{};

    res = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);


    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_proc);
        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();
    }

    {
        /// now starts the main sorting procedure

        // step 0
        // check if size is 1
        if(size == 1){
            std::sort(begin, end);
        }
        else{
            // step 1
            // rank 0 scatter data to all process
            if (rank == 0) {
                int original_total_data_size = end - begin;
                int padded_total_data_size = original_total_data_size;
```

```
                    // allocate a new array for padded array
                    padded_data_array.assign(begin, end);

                    // pad positive infinity to the data array
                    int size_remainder = original_total_data_size % size;
                    if(size_remainder != 0){
                        int padded_size = size - size_remainder;
                        padded_total_data_size += padded_size;

                        for(int i=0 ; i<padded_size; i++){
                            padded_data_array.push_back(std::numeric_limits<Element
>::max());
                        }
                    }
                    data_package_size = padded_total_data_size / size;
                }

                // boardcast the size of data received for each process
                MPI_Bcast(&data_package_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

                // allocate buffer holding data and the data from other process
                std::vector<Element> local_data_buffer(data_package_size * 2);

                // scatter data
                MPI_Scatter(padded_data_array.data(), data_package_size, MPI_LONG, l
ocal_data_buffer.data(), data_package_size, MPI_LONG, 0, MPI_COMM_WORLD);

                // step 2
                // sort data until sorted
                bool sorted_flag_all = false;
                bool sorted_flag_local = false;

                while(!sorted_flag_all){
                    // odd rank send data
                    if(size % 2 != 0 && rank == size-1){
                        // last process would do nothing if the total size is odd
                        sorted_flag_local = true;
                    }
                    else if(rank % 2 != 0){
                        // send local data to previous process and wait for result
                        MPI_Sendrecv(local_data_buffer.data(), data_package_size, MP
I_LONG, (rank-1+size)%size, 0,
                                    local_data_buffer.data(), data_package_size, MPI
_LONG, (rank-1+size)%size, 0,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    }
                    else{
                        // receive data from other process
                        MPI_Recv(local_data_buffer.data()+data_package_size, data_pa
ckage_size, MPI_LONG, (rank+1+size)%size, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        sorted_flag_local = std::is_sorted(local_data_buffer.data(),
 local_data_buffer.data()+data_package_size*2);
                        // check whether array is already sort which means no swap w
ill happen
                        if(!sorted_flag_local){
                            std::sort(local_data_buffer.data(), local_data_buffer.da
ta()+data_package_size*2);
                        }
                        // return the latter half of sorted result
                        MPI_Send(local_data_buffer.data()+data_package_size, data_pa
ckage_size, MPI_LONG, (rank+1+size)%size, 0, MPI_COMM_WORLD);
                    }

                    // even rank send data
                    if(rank != 0){
                        if(size%2 == 0 && rank == size-1){
                            // last process would do nothing if the total size is ev
en

                            sorted_flag_local = true;
                        }
                        else if(rank % 2 != 1){
                            // send local data to previous process and wait for resu
lt
                            MPI_Sendrecv(local_data_buffer.data(), data_package_size
, MPI_LONG, (rank-1+size)%size, 0,
                                        local_data_buffer.data(), data_package_size,
 MPI_LONG, (rank-1+size)%size, 0,
                                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        }
                        else{
                            // receive data from other process
                            MPI_Recv(local_data_buffer.data()+data_package_size, dat
a_package_size, MPI_LONG, (rank+1+size)%size, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                            sorted_flag_local = std::is_sorted(local_data_buffer.dat
a(), local_data_buffer.data()+data_package_size*2);
```

```
                            // check whether array is already sort which means no sw
ap will happen
                            if(!sorted_flag_local){
                                std::sort(local_data_buffer.data(), local_data_buffe
r.data()+data_package_size*2);
                            }
                            // return the latter half of sorted result
                            MPI_Send(local_data_buffer.data()+data_package_size, dat
a_package_size, MPI_LONG, (rank+1+size)%size, 0, MPI_COMM_WORLD);
                        }
                    }

                    // check whether no swap has happened in all processes
                    MPI_Reduce(&sorted_flag_local, &sorted_flag_all, 1, MPI_C_BOOL,
MPI_LAND, 0, MPI_COMM_WORLD);

                    // broadcast the signal whether another sort iteration is necess
ary
                    MPI_Bcast(&sorted_flag_all, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
                }

                // step 3
                // gather sorted array
                MPI_Gather(local_data_buffer.data(), data_package_size, MPI_LONG, pa
dded_data_array.data(), data_package_size, MPI_LONG, 0, MPI_COMM_WORLD);
                if(rank == 0){
                    // strip the padded data and put needed data back to original ar
ray
                    copy(padded_data_array.begin(), padded_data_array.begin()+(end-
begin), begin);
                }
            }

        }

        if (0 == rank) {
            information->end = high_resolution_clock::now();
        }
        return information;
    }

    std::ostream &Context::print_information(const Information &info, std::ostream &
output) {
        auto duration = info.end - info.start;
        auto duration_count = duration_cast<nanoseconds>(duration).count();
        auto mem_size = static_cast<double>(info.length) * sizeof(Element) / 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 << "throughput (gb/s): " << mem_size / static_cast<double>(duration_count) * 1'000'000'000.0
               << std::endl;
        return output;
    }
}
```