CSC4005 Project 1 Report

Parallel Odd-Even Transportation Sort

Xi Mao (119020038)

October 11, 2022

1 Abstract

Odd-even sort is a variation of bubble-sort that sorts an array with alternating odd and even phases. It is a simple algorithm developed originally for use on parallel processors with local interconnections. In this project, I implemented both sequential and parallel odd-even sort in C++, and then analyzed their performances on different configurations (number of processors and array size).

2 Introduction to Odd-Even Sort

i. Sequential Version

The sequential odd-even sort functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second), the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

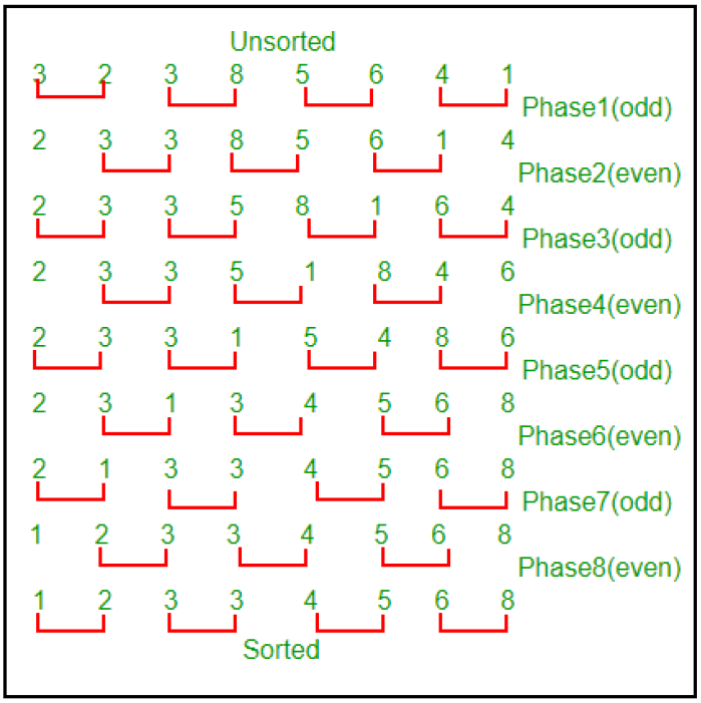


Figure 1: Sequential Odd-Even Sort

The implementation in C++ is quite simple, which consists of an outer for loop to alternate

odd and even phases and an inner loop to compare and swap odd/even indexed pairs:

```
bool sorted = false;
while (!sorted) {
    sorted = true;
    // alternate odd and even passes
    for (int offset = 0; offset < 2; ++offset) {
        // do the actual odd/even sort
        for (int j = offset; j < num_elements - 1; j += 2) {
            if (sorted_elements[j] > sorted_elements[j + 1]) {
                std::swap(sorted_elements[j], sorted_elements[j + 1]);
                sorted = false;
            }
        }
    }
}
```

The worst time complexity is $O(N^2)$, where N is the size of the array. This is because, similar to bubble sort, the outer loop needs to run at most N times to sort the array, and the complexity of each iteration is O(N). The space complexity is O(1), obviously.

ii. Parallel Version

The parallel version is similar to the sequential one. Firstly, the array is evenly distributed into N processes. Next, the algorithm runs N iterations. In iteration i, each node runs an odd(even) pass of the serial odd-even sort locally, depending on the parity of i. Then, if i is even, each process compares its first and last number with the previous and next process, respectively, and switches if necessary.

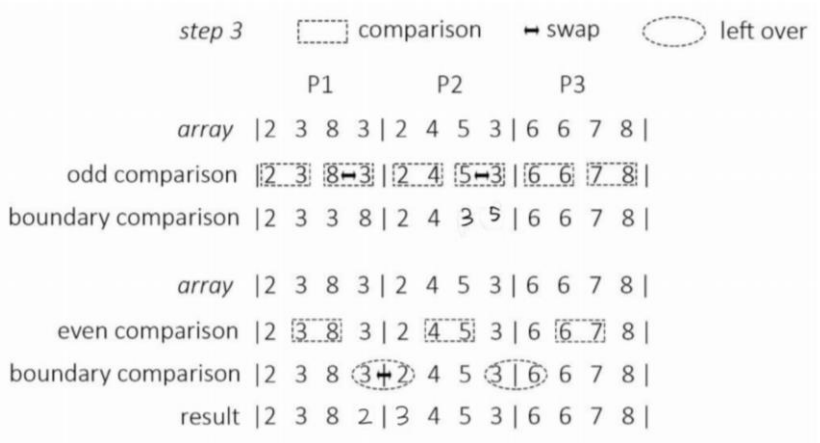


Figure 2: Run local odd/even pass and compare boundary elements

Finally, all processes gather their local sorted array to the master process to get the output.

Computation complexity

Similar to the sequential version, the total workload of the parallel odd-even sort is still $O(N^2)$, but now we have M processes running concurrently to solve the problem (i.e., do odd-even passes). So, the computation complexity is $O(\frac{N^2}{M})$.

Communication complexity

The time complexity of data scattering and gathering is O(N).

The time complexity of sending messages is O(MN), since every process sends O(1) integers per iteration, and there are N iterations.

Therefore, the total time complexity of parallel odd-even sort is $O(\frac{N^2}{M})$ + O(MN).

3 Method

i. Implementation Details

Data Generation

The test array contains randomly generated integers ranging from 1 to 99999999 using the array generator in the code template. The core code is displayed below:

```
srand((int)time(0));
    for (int i = 0; i < num_elements; i++){
        out << RANDOM(1, 99999999) << std::endl;
    }
```

Data Scattering

Numbers in the test array are evenly distributed to every process. To decide the size of the sub-array of each process, I divide the array size by the number of processes and split the remainder

K evenly to the first K processes.

```
        int quotient = num_elements / world_size;
        int remainder = num_elements % world_size;

        std::vector<int> send_counts(world_size);
        for (int i = 0; i < world_size; i++) {
            send_counts[i] = quotient;
            if (i < remainder) {
                send_counts[i]++;
            }
        }
```

Next, use MPI_Scatterv together with a displacement array to scatter the test array into all processes:

```
        std::vector<int> displs(world_size);
        displs[0] = 0;
        for (int i = 1; i < world_size; i++) {
            displs[i] = displs[i - 1] + send_counts[i - 1];
        }
        int* my_elements = new int[quotient + 1];
```

```
MPI_Scatterv(
    elements, send_counts.data(), displs.data(), MPI_INT,
    my_elements, send_counts[rank], MPI_INT, 0, MPI_COMM_WORLD
);
```

Sorting

As mentioned before, there are N iterations. Let i denote the current iteration number.

Firstly, do local odd-even sort pass:

```
for (int j = i % 2; j < my_size - 1; j += 2) {
    if (my_elements[j] > my_elements[j + 1]) {
        std::swap(my_elements[j], my_elements[j + 1]);
    }
}
```

Then, if i is even, adjacent processes compare and swap their boundary elements using MPI_Send and MPI_Recv:

```
        if (i % 2 != 0) {
            int send_num, recv_num;
```

```
        // if not the first process, send the first element to the left
        if (my_rank != 0) {
            send_num = my_elements[0];
            MPI_Send(&send_num, 1, MPI_INT, my_rank - 1, 0, comm);
            MPI_Recv(&recv_num, 1, MPI_INT, my_rank - 1, 0, comm,
                     MPI_STATUS_IGNORE);
            if (recv_num > my_elements[0]) {
                my_elements[0] = recv_num;
            }
        }
        // if not the last process, send the last element to the right
        if (my_rank != world_size - 1) {
            send_num = my_elements[my_size - 1];
            MPI_Recv(&recv_num, 1, MPI_INT, my_rank + 1, 0, comm,
                     MPI_STATUS_IGNORE);
            MPI_Send(&send_num, 1, MPI_INT, my_rank + 1, 0, comm);
            if (recv_num < my_elements[my_size - 1]) {
                my_elements[my_size - 1] = recv_num;
            }
        }
    }
```

Data Gathering

This can be achieved by calling MPI_Gatherv:

```
MPI_Gatherv(
    my_elements, send_counts[rank], MPI_INT,
    sorted_elements, send_counts.data(), displs.data(), MPI_INT, 0,
            MPI_COMM_WORLD
);
```

After that, print out important information like array size, running time, and number of processes, and save the sorted array to a file.

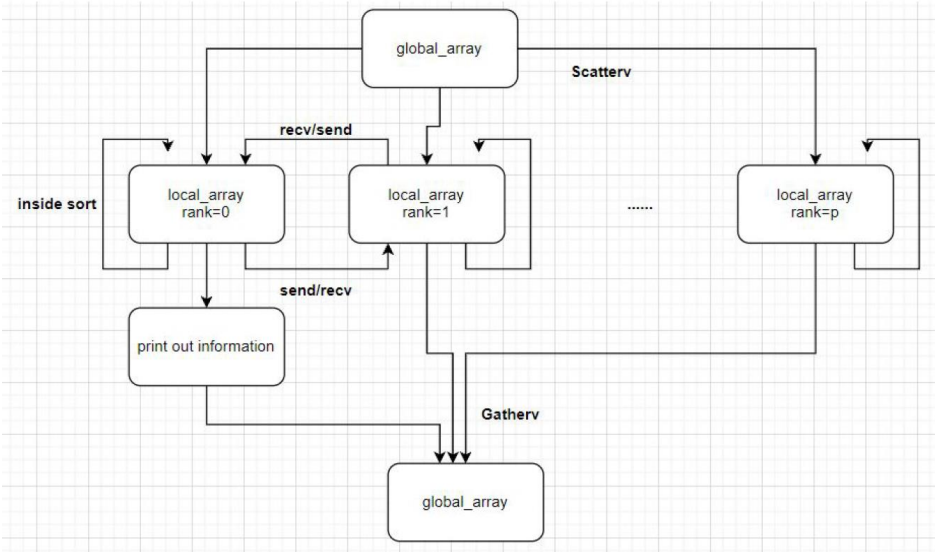


Figure 3: The Flow Chart of the MPI program (source: the Internet)

ii. Experiment Designing

I selected the following array sizes:

- Small: 100, 200, 300, 400, 500
- Medium: 10000, 20000, 30000, 40000, 50000
- Large: 100000, 200000, 300000, 400000, 500000

Moreover, the following numbers of processors for parallel odd-even sort: 1, 2, 4, 8, 16 (1 process config were also used for sequential odd-even sort).

Then, I wrote a python script "run_tests.py" to enumerate all combinations of array size and number of processes, generate a sbatch script for each configuration, and submit the task to the HPC. I also wrote a python script "parse_output.py" to collect the running time of each task and put the statistics into a csv file.

iii. Procedures to Run Code and Reproduce Results

Firstly, enter the project root folder and build the project using "make all".

- To generate test array, run "build/gen <array size> <output file path>"
- To run sequential odd-even sort, run "build/ssort <array size> <input file path>"

  Sample output:

```
[119020038@node21 CSC4005-Assignment-1]$ build/ssort 20 test_data/20.in
seq
actual number of elements:20
Student ID: 119020038
Name: Xi Mao
Assignment 1
Run Time: 9.077e-06 seconds
Input Size: 20
Process Number: 1

Original Array: 8098425 19016636 86841600 92563177 83234079 62805266 87703491 6691
1760 77173468 68702358 3979750 21483554 28433736 29811789 56146531 70623169 612016
52 62517615 27475204 67374028

Sorted Array: 3979750 8098425 19016636 21483554 27475204 28433736 29811789 5614653
1 61201652 62517615 62805266 66911760 67374028 68702358 70623169 77173468 83234079
 86841600 87703491 92563177
```

- To run parallel odd-even sort, run "mpirun -np=<process number> build/psort <array size> <input file path>"

  Sample output:

```
[119020038@node21 CSC4005-Assignment-1]$ mpirun -np=3  build/psort 20 test_data/20.in
mpi
actual number of elements:20
Student ID: 119020038
Name: Xi Mao
Assignment 1
Run Time: 0.000707833 seconds
Input Size: 20
Process Number: 3

Original Array: 8098425 19016636 86841600 92563177 83234079 62805266 87703491 66911760 77173468 6
8702358 3979750 21483554 28433736 29811789 56146531 70623169 61201652 62517615 27475204 67374028

Sorted Array: 3979750 8098425 19016636 21483554 27475204 28433736 29811789 56146531 61201652 6251
7615 62805266 66911760 67374028 68702358 70623169 77173468 83234079 86841600 87703491 92563177
```

- To check the correctness of an output file, run "build/check <array size> <output file path>"

To run experiments in the HPC, configure the first few lines of "run_tests.py" to tweak the number of cores and array size (optionally), and then run "python3 run_tests.py" to submit all jobs to the HPC. Wait for tasks to finish and run "python3 parse_output.py" to get statistics written in a csv file.

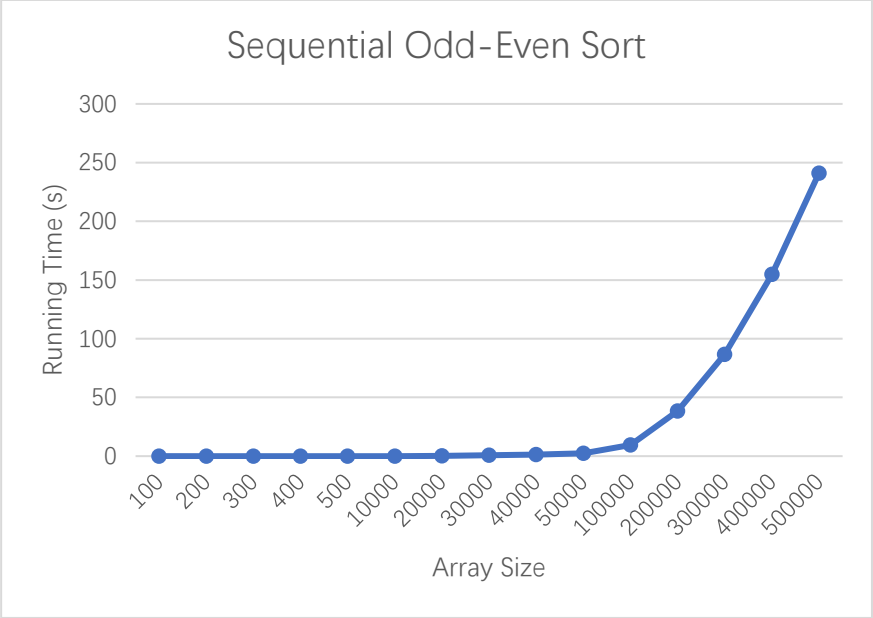
4 Result and Discussion

i. Sequential and Parallel Comparison

- **Sequential Results**

The following are array size - running time figures for the sequential program:

ArraySize	Time(s)
100	0.000134
200	0.000274
300	0.001
400	0.00172
500	0.00276
10000	0.409783
20000	1.6583
30000	3.74197
40000	6.68171
50000	10.4756
100000	42.1625
200000	169.548
300000	383.664
400000	679.18
500000	1064.38

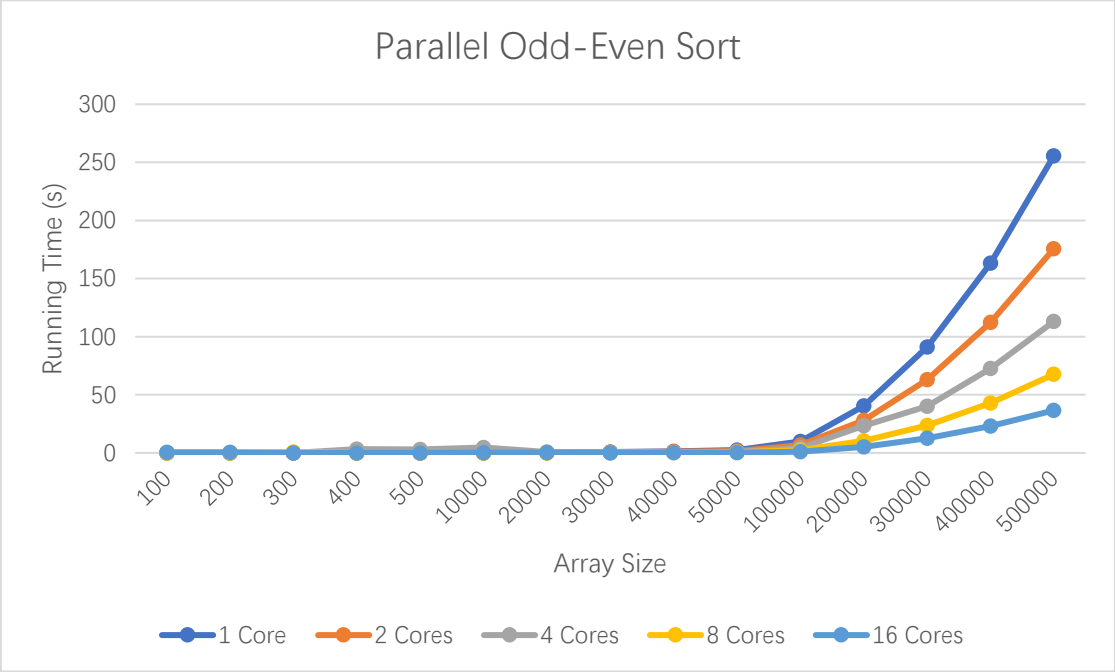


The data shows that, as array size multiples, running time quadruples. Therefore, we can roughly regard this curve as a quadratic function, which is consistent with its time complexity $O(N^2)$.

- **Parallel Results**

The following are (array size & number of cores) – running time figures for the parallel program:

ArraySize\Cores	1	2	4	8	16
100	0.000088347	0.000125976	0.000337879	0.00023665	0.4195
200	0.000112086	0.000150922	0.00282443	0.10027	0.509391
300	0.000110317	0.000246474	0.000543527	0.83218	0.001911
400	0.000195984	0.000335244	3.22168	0.00059766	0.003429
500	0.00025553	0.000404464	2.99955	0.00346626	0.001456
10000	0.0729734	0.0563297	4.71228	0.0244493	0.342521
20000	0.333138	0.247303	0.872617	0.0643717	0.156058
30000	0.803436	0.583328	0.299096	0.167437	0.084719
40000	1.47015	1.06685	0.574168	0.299049	0.145492
50000	2.37413	1.6952	1.07286	0.517833	1.91317
100000	9.88193	6.90762	4.25245	2.38327	2.80351
200000	40.4148	27.9697	23.51	10.3037	5.27612
300000	91.2566	63.0433	40.2298	23.6784	12.7108
400000	163.209	112.233	72.7641	42.8084	23.0517
500000	255.552	175.646	113.126	67.602	36.5565



The curves above each are roughly in the form of a quadratic function. If we compare different curves, we can find that the running time is roughly inversely proportional to the number of processes, which agrees with the $O(\frac{N^2}{M})$ + O(MN) complexity mentioned above.

- **Sequential vs. MPI**

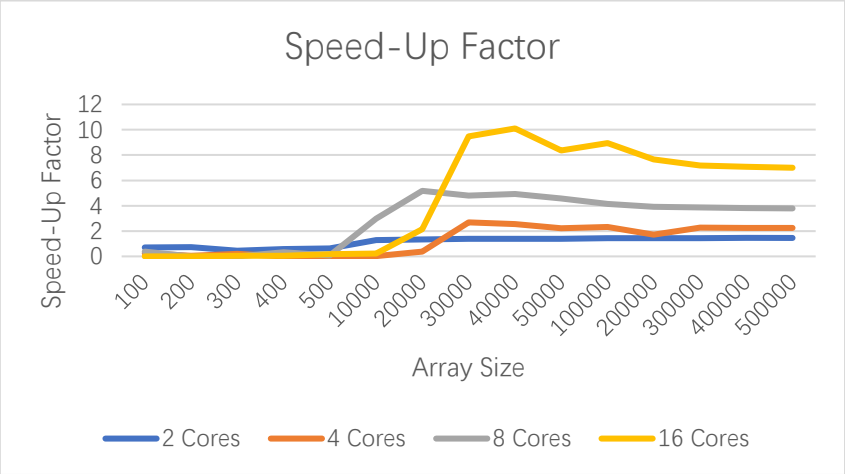
The figures in this section show that the 1-Core curve in the parallel output figure matches the sequential one. Moreover, as expected, running time decreases significantly as the number of processes increases.

ii. Speed-Up Factor and System Efficiency of the Parallel Program

The following are a speed-up factor table and the corresponding chart:

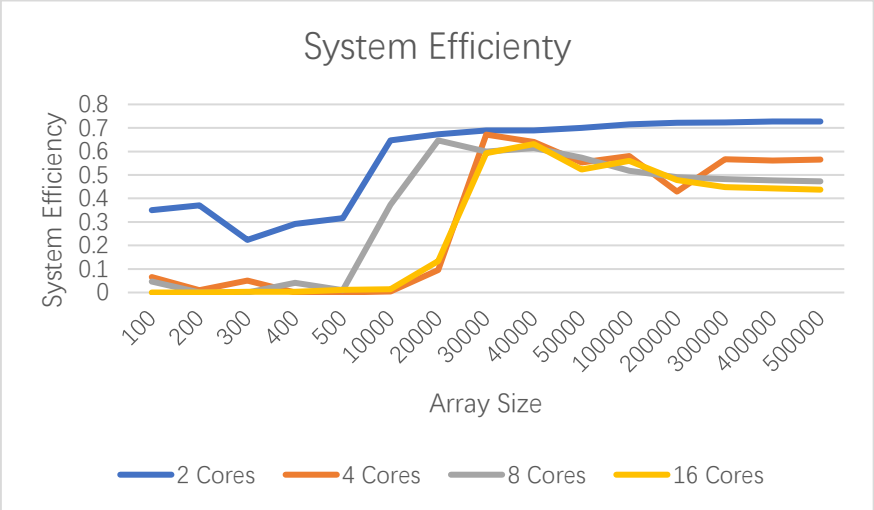
ArrSize\Cores	2	4	8	16
100	0.701300248	0.261475262	0.37332032	0.000211
200	0.742675024	0.039684467	0.00111784	0.00022
300	0.447580678	0.202965078	0.00013256	0.057736
400	0.584601067	6.08329E-05	0.32791669	0.057153
500	0.631774398	8.51894E-05	0.07371922	0.175501

10000	1.295469353	0.015485795	2.98468259	0.213048
20000	1.347084346	0.381768863	5.17522452	2.134706
30000	1.37733145	2.68621446	4.79843762	9.48354
40000	1.378028776	2.560487523	4.91608399	10.10468
50000	1.400501416	2.212898235	4.58474064	8.368453
100000	1.430583906	2.323820386	4.14637452	8.954998
200000	1.444949356	1.719047214	3.92235799	7.659947
300000	1.447522576	2.268383139	3.85400196	7.179454
400000	1.454197963	2.242987957	3.81254614	7.080129
500000	1.454926386	2.259003235	3.78024319	6.990604



The following charts display the system efficiency of the parallel program:

ArrSize\Cores	2	4	8	16
100	0.350650124	0.065368815	0.04666504	1.32E-05
200	0.371337512	0.009921117	0.00013973	1.38E-05
300	0.223790339	0.05074127	1.657E-05	0.003609
400	0.292300533	1.52082E-05	0.04098959	0.003572
500	0.315887199	2.12974E-05	0.0092149	0.010969
10000	0.647734676	0.003871449	0.37308532	0.013315
20000	0.673542173	0.095442216	0.64690306	0.133419
30000	0.688665725	0.671553615	0.5998047	0.592721
40000	0.689014388	0.640121881	0.6145105	0.631542
50000	0.700250708	0.553224559	0.57309258	0.523028
100000	0.715291953	0.580955096	0.51829681	0.559687
200000	0.722474678	0.429761803	0.49029475	0.478747
300000	0.723761288	0.567095785	0.48175024	0.448716
400000	0.727098982	0.560746989	0.47656827	0.442508
500000	0.727463193	0.564750809	0.4725304	0.436913



Several phenomena can be explained. The first and most obvious is that the speed-up factor and system efficiency are very low for small arrays. Furthermore, the larger number of cores, the lower the two factors are. This is because, for parallel version programs, initialization of multiple processes and message passing between processes cost time, which outweighs the speed benefit of parallelism. So, the running time is longer than sequential cases in relatively small array sizes. When the test array becomes larger, the speed-up factors become much more reasonable.

Another fact is that the system efficiency is not very high for test cases with more than 2 cores. We may attribute it to message passing overhead and time wasted in blocking message sending / receiving. Moreover, as array size grows, the speed-up factor and system efficiency gradually drop. The phenomenon may be explained by the claim that, as the input size grows, the benefit growth of multi-processing is unable to compensate for the message passing overhead growth. The 4-Cores and 16-Cores curves have fluctuations, possibly because of the unstable server performance influenced by many internal or environmental factors.

iii. Other Discoveries

When I tested the parallel program on my computer and the HPC, I found that when the number of MPI processes exceeded the number of available/allocated processor cores (for example, if I ran “psort” with -np=10 on my laptop with Intel i9-12900H, which had 8 performance cores

and had efficiency cores disabled) the speed up factor dropped. I guess this was caused by hyper-threading, but further investigation needed to be done.

5 Conclusion

In this project, I implemented sequential and parallel odd-even sort in C++ and done experiments to investigate their performance under different configurations. The results indicated parallel programs, although having starting overheads, did outperform sequential ones on large arrays. However, as the number of processes increased, the speed-up factor did not increase at the same scale. Instead, the system efficiency dropped. The problem might be alleviated by improving the parallel algorithm. But it suggests that it is important to find a suitable level of parallelism that provides enough speed up while not wasting too many hardware resources.