NTHU CS542200 Parallel Programming
Homework 1: Odd-Even Sort

111062625 蔡哲平

1. Implementation

i. How do you handle an arbitrary number of input items and processes?

Given N input data and P processes, it can be divided into 3 cases below:

- N < P

  In this case, it means that the data size is smaller than the process number. In other words, some processes may be redundant and hence I exclude them from the original group using *MPI_Group_range_excl*. Then, each left process gets one piece of data and runs the Odd-Even-Sort algorithm.

- N = P

  Each process gets exactly one piece of data and runs the Odd-Even-Sort algorithm.

- N > P

  In this case, it means that the data size is bigger than the process number, which means a process will get multiple data. In my implementation, all processes will get N/P data at first, and those processes with ranks smaller than N%P will get an additional piece of data. By doing this, the difference in the number of data obtained by all processes is less than or equal to one, which is more load balancing compared to distributing all N%P data into one single process.

ii. How do you sort in your program?

- Local Sort

  Local sort refers to the sorting of each process before the Odd-Even-Sort begins. After trying multiple sorting libraries such as *qsort*, *std::sort*, and *boost::float_sort*, I concluded that *boost::float_sort* has the shortest execution time compared to others.

- Odd-Even-Sort (original version)

  During the odd phase, processes with odd ranks R, denoted by P_R, can only communicate with their neighbor processes with ranks R+1, denoted by P_{R+1}, and vice versa.

  At first, P_R will send its last element of local data, denoted as *tail*, to P_{R+1}, and P_{R+1} will also send its first element of local data, denoted as *head*, to P_R, using one *MPI_Send* and *MPI_Recv* call. If *tail* is smaller than *head*, then it means the data between the two processes are already sorted. Otherwise, perform Odd-Even-Sort.

When performing Odd-Even-Sort, P_{R+1} will send its local data size and local data array to P_R by calling two times of *MPI_Send* and P_R will also call 2 times of *MPI_Recv* to wait for the data to arrive. Next, P_R will merge the received data with its local data by a self-implemented function *mergeData*, which can merge two arrays in ascending order with O(n) time complexity and copy the sorted array into the array passed as an argument. After merging, P_R will send the bigger half of the sorted array to P_{R+1} using *MPI_Send* and P_{R+1} will call *MPI_Recv* to obtain it. Finally, all processes call *MPI_Allreduce* to collect the result of this phase to determine whether the whole data is sorted or not.

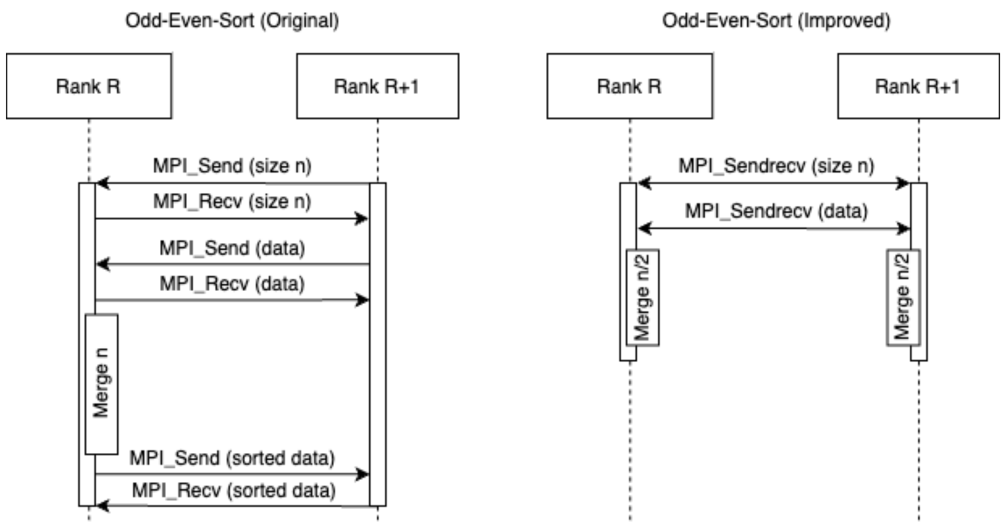
The even phase will be almost the same manner but denotes processes with even ranks R as P_R and their neighbor processes with ranks R+1 as P_{R+1}.

- Odd-Even-Sort (improved version)

In the original version, when P_R is merging data, P_{R+1} is waiting for P_R to complete its merging, in other words, idling. To improve, I change one-way data sending to two-way data exchanging, which means P_R and P_{R+1} exchange their local data simultaneously. After data is exchanged, both P_R and P_{R+1} start merging their parts of data. P_R merges data from the *head (min)*, in contrast, P_{R+1} merges data from the *rear (max)*. By doing this, we can not only overlap computations between the two processes but also reduce the merged size by a factor of two, thus speeding up the program.

Another optimization is to prune the sending data. It's inefficient to send all local data to another process because not all of them will appear in another process's local data after merging. Instead, P_R should only send data that are bigger than P_{R+1}'s *head*. Similarly, P_{R+1} should only send data that are smaller than P_R's *tail*. To achieve this, I've implemented a function using binary search to find the first bigger-than-target index in the data array with O(logN) time complexity. After getting this index, we can easily determine how much data is necessary to be sent and hence shorten the communication time. However, after applying the above method, I've found that it isn't effective enough, the amount of sending data it can reduce is very limited. Hence, I did a lot of experiments and found that the best approach is to send only half of the local data, more discussion will be shown in the further part of this report.

Below is a sequence diagram of the original and improved versions of Odd-Even-Sort, showing the difference between them.



iii. Other efforts you've made in your program.

- Parallel IO

  When reading (writing) files, we can first determine the offset of each process by its rank and use *MPI_File_read_at (MPI_File_write_at)* to parallel the IO operation by reading or writing their part of the file instead of one process reading (writing) the whole file while other processes idling.

- *MPI_Sendrecv*

  Instead of calling *MPI_Send* and *MPI_Recv* separately, we can use the combined version, *MPI_Sendrecv*, to shorten the time of initiating an extra MPI call.

- Coding Styles

  Other optimization techniques such as using pointers to access array elements and using bitwise operators to perform division can slightly speed up the program.

2. Experiment & Analysis

i. Methodology

- System spec

  All tests were run on the clusters provided by this course.

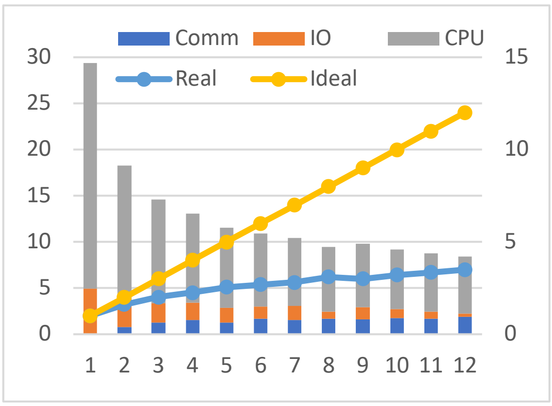
- Performance Metrics

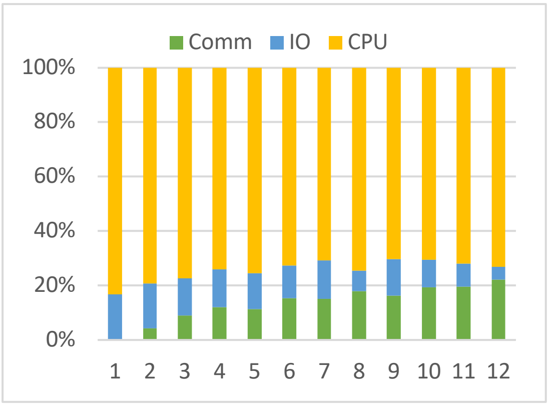
  Use *MPI_Wtime* for all time measurements. Calculate the time difference between *MPI_Init* and *MPI_Finalize* to get the total execution time, denoted as *total_tim*e. Calculate the time difference of all *MPI_Send*, *MPI_Recv,* and *MPI_Sendrecv* and summarize all of them to get the communication time, denoted as *comm_time*. Calculate the time difference of *MPI_File_read_at* and *MPI_File_write_at* to get the I/O time, denoted as *IO_time*. Finally, use *total_time* to subtract *comm_time* and *IO_time* to get the calculation time, denoted as *CPU_time*. As for the test case, I used 33.in provided by TAs.

Since the provided clusters aren't very stable, the program's runtime can vary by several seconds. To minimize this effect, within each test command, I ran 10 tests and collected the one with the shortest execution time as my test result.

ii. Plots: speedup factor & time profile

- Experiment 1: Under a single-node environment (N=1), observe the results of running different numbers of processes.



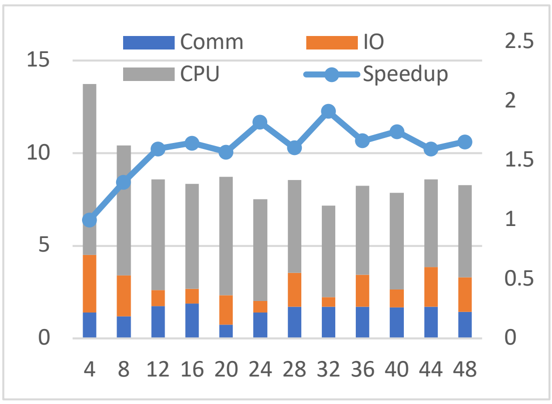


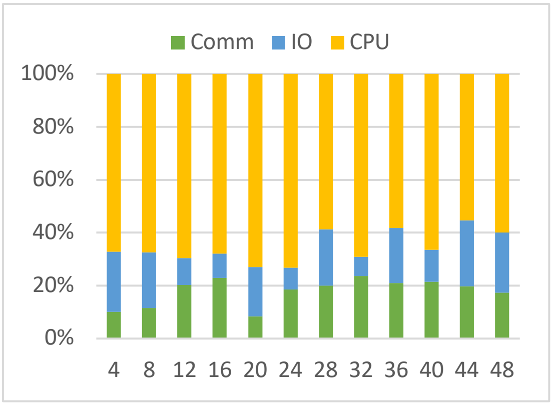
Time Profile and Speedup (1 Node)
[X: process num, LY: runtime(sec), RY: speedup]

Time Profile (1 Node)
[X: process num, Y: percentage]

Observation 1: According to the above figure, as the number of processes increases, *CPU_time* decreases significantly and *IO_time* decreases slightly because the amount of data allocated to each process decreases, and most importantly, I parallel the computation and disk I/O within the processes. Moreover, *comm_time* increases a little bit, and the proportion of it also increases as the number of processes grows, which makes sense because the more processes, the more communications are needed. Finally, the ideal speedup when using 12 processes is 12, however, this program can only achieve about a 4 times speedup. This is because after sorting their local data, processes still need to communicate and merge until the entire input data is sorted, which takes more time.

- Experiment 2: Under a multiple-node environment (N=4), observe the results of running different numbers of processes.



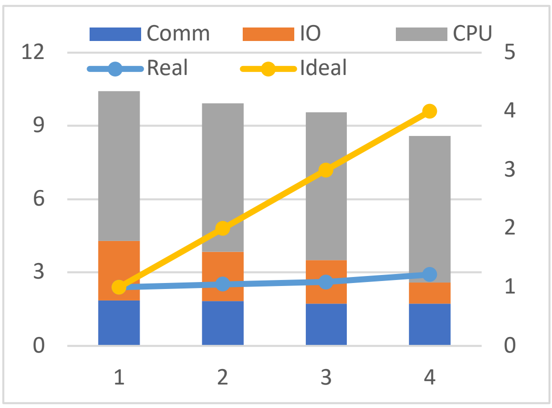


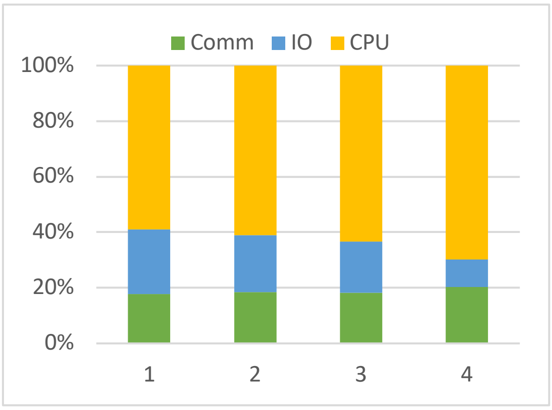
Time Profile and Speedup (N=4)
[X: node num, LY: runtime(sec), RY: speedup]

Time Profile (N=4)
[X: node num, Y: percentage]

Observation 2: According to the above figure, we can see that *CPU_time* generally decreases as the number of processes grows. In addition, the proportion of *comm_time* generally grows too. However, the actual speedup is only at most 2. Since this is a strong scaling test, the speedup factor will eventually converge. In this case, the "sweet spot" is about using 12 processes to complete in a reasonable amount of time, yet not wasting too many cycles due to parallel overhead.

- Experiment 3: Under a fixed process number environment (n=12), observe the results of running on different numbers of nodes.



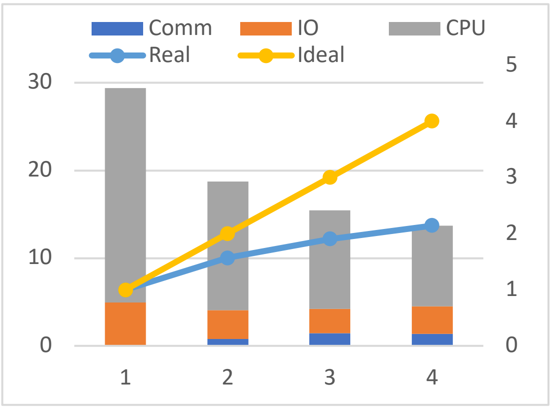


Time Profile and Speedup (12 Processes)
[X: node num, LY: runtime(sec), RY: speedup]

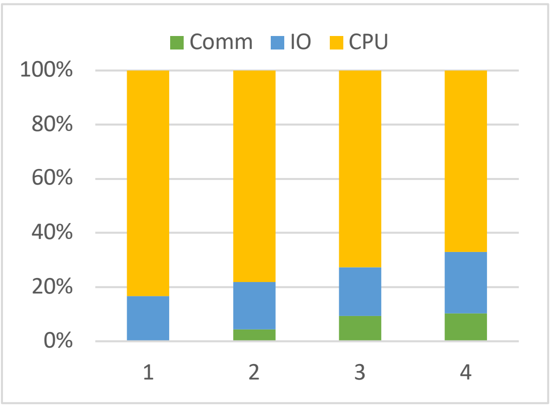
Time Profile (12 Processes)
[X: node num, Y: percentage]

Observation 3: According to the above figure, we can see that the total runtime slightly decreases as the number of nodes grows. In my opinion, when processes are distributed to multiple nodes, resource contention isn't as severe as in a single-node environment, resulting in improved execution time. However, the actual speedup is only slightly higher than 1, because, for each process, it is still running on a single node rather than multiple nodes at the same time.

- Experiment 4: Under a fixed process per node environment (ppn=1), observe the results of running on different numbers of nodes.



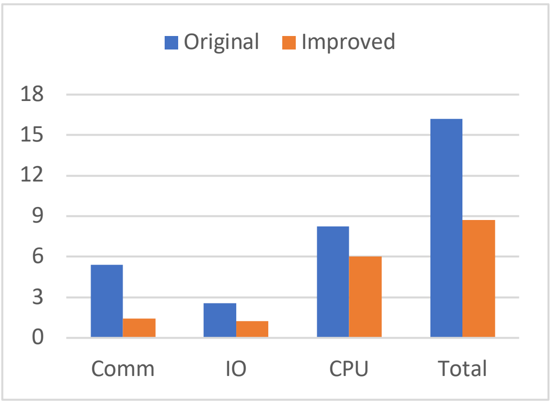
Time Profile and Speedup (ppn=1)
[X: node num, LY: runtime(sec), RY: speedup]



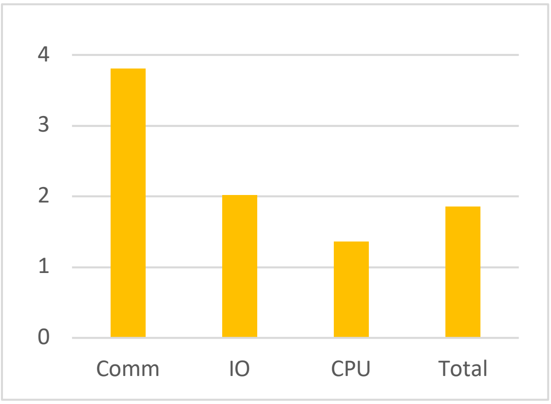
Time Profile (ppn=1)
[X: node num, Y: percentage]

Observation 4: According to the above figure, we can see that with a fixed number of processes per node, as the number of nodes increases, the total number of processes increases too, resulting in shortened execution time. However, it's only 2 times faster when using 4 nodes since we also need to consider the communication overhead between processes and nodes.

- Experiment 5: Under a fixed node and fixed process number environment (N=1, n=12), observe the results of running the original and improved version of Odd-Even-Sort.



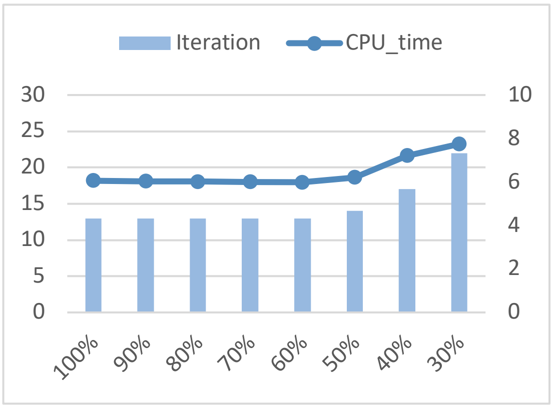
Time Profile (N=1, n=12)
[X: partition, Y: runtime(sec)]



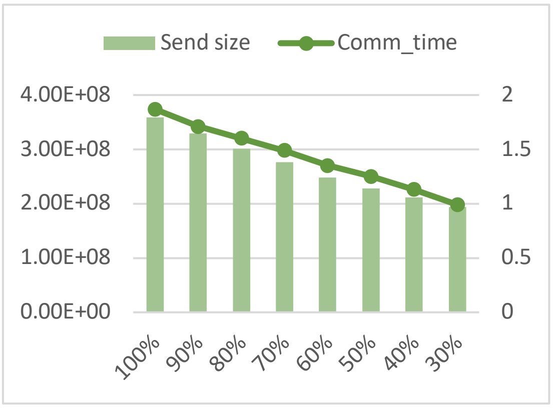
Speedup (N=1, n=12)
[X: partition, Y: speedup]

Observation 5: According to the above figure, the improved version of Odd-Even-Sort has almost 2 times the speedup compared to the original version. It's because, in the improved version, there are fewer MPI calls and fewer data to send, which reduces communication time by 4 times. Furthermore, each process merging only their part of the data shortens *CPU_time* by almost 1.5 times. All in all, the improved version is 2 times better than the original one, which is a tremendous optimization.

- Experiment 6: Under a fixed node and fixed process number environment (N=1, n=12), observe the results of sending different proportions of local data to the neighbor process.



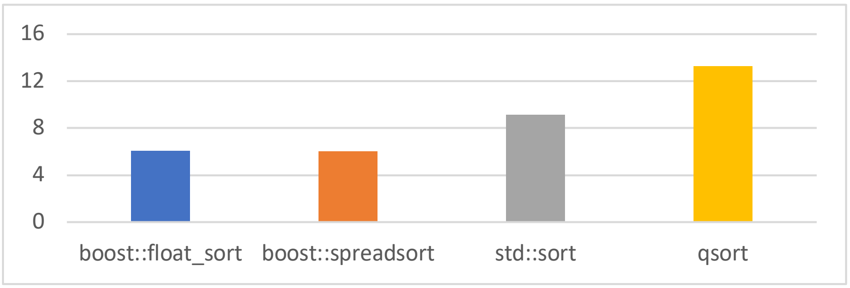
Relationship between iteration and CPU_time (N=1, n=12)
[X: proportion, LY: iteration, RY: runtime(sec)]



Relationship between send size and Comm_time (N=1, n=12)
[X: proportion, LY: send size, RY: runtime(sec)]

Observation 6: According to the above figure, we can see that *CPU_time* grows as the iteration increases and *comm_time* reduces as the sending size decreases. Therefore, our goal is to find the optimal proportion to make both iteration and sending size as low as possible. After tons of experiments, I've found that sending 50% of the local data can get the shortest execution time.

- Experiment 7: Under a fixed node and fixed process number environment (N=1, n=12), observe the results of using different sorting libraries for local sort.



Sort Comparison (N=1, n=12)
[X: sort method, Y: runtime(sec)]

Observation 7: According to the above figure, we can conclude that *boost::float_sort* and *boost::spreadsor*t both have the shortest sorting time compared to others.

iii. Discussion

- Compare I/O, CPU, and network performance. Which is/are the bottleneck(s)? Why? How could it be improved?

According to the above experiments, we can see that no matter how many processes or nodes you use, *CPU_time* always accounts for a significant proportion of execution time. Although *comm_time* increases as the number of processes increases, it's still very small compared to *CPU_time*. Hence, the bottleneck of this program is the CPU. Since the fastest sort is O(nlogn) and the fastest merge is O(n), both of which were implemented in my program, the only solution is to use a faster CPU for computation.

- Compare scalability. Does your program scale well? Why or why not? How can you achieve better scalability? You may discuss the two implementations separately or together.

  According to Expr.1, when using 12 processes running my program, it's only about 4 times faster than using 1 process, which is still far away from the ideal speedup of 12. In other words, my program doesn't scale well. It's because, after the local sort, processes still need to communicate with each other and merge until the input data is sorted, which takes much more *comm_time* and *CPU_time*. To achieve better scalability, we need to parallel the communication and the computation in a process, however, I can't come up with an implementation for this and even think it's impossible in this assignment. In conclusion, although my program doesn't scale well, I still tried my best to keep none of the processes idle for a millisecond. As a result, my program is 3rd on the leaderboard, which is a very fast implementation in my point of view.

3. Experiences / Conclusion

This is my first time writing a parallel program. Although there weren't lots of MPI APIs needed in this homework, the concept of parallel programming still got me confused. Different from sequential programs, parallel programs create many processes to do the same task. Therefore, how to distribute the workload evenly among all processes is an art. In addition, how to overlap all processes in order not to idle them is also quite challenging. However, the most difficult part of this assignment in my point of view is to collect meaningful test results. Due to the unstable clusters, a program's execution time can vary by seconds, which is quite annoying because you can't verify the cause, in other words, it's hard to determine whether it's because of the changes in your code or just too many people running their programs on the clusters at the same time. In conclusion, this assignment is quite challenging, however, I'm willing to accept challenges. Seeing my program keep reaching a higher position on the leaderboard gives me a great sense of accomplishment and more motivation to push my program to the limit.