Parallel Programming
Homework 1: Odd-Even Sort

王鈺鎔 106062600

Implementation

Basic

1 Split input data
  - 1.1 Using `MPI_File_read_all` to read data.
  - 1.2 Divide input data by Process number and put the remainder to the last process to split data.

2 Main algorithm: classify odd-even sort into 4 cases:
  - 2.1 Even phase with even batch size
    - 2.1.1 Group element k and k+1, where k belongs {0, 2, 4,…..}.
    - 2.1.2 Swap element k and k+1 if element k bigger than element k+1.
  - 2.2 Even phase with odd batch size
    - 2.2.1 Group element k and k+1, where k belongs {0, 2, 4,…..}.
    - 2.2.2 The even rank processes (except the last) need to transfer the last element to its next process, and receive the first element of its next process.
    - 2.2.3 The odd rank processes need to transfer the first element to its previous process, and receive the last element of its previous process.
    - 2.2.4 Swap element k and k+1 if element k bigger than element k+1.
  - 2.3 Odd phase with even batch size
    - 2.3.1 Similar to the case even phase with even batch size. However, k belongs {1, 3, 5,…..}.
    - 2.3.2 However, every process exchange the first element and the last element with the previous and the next process.
  - 2.4 Odd phase with odd batch size
    - 2.4.1 Similar to the case even phase with odd batch size. However, k belongs {1, 3, 5,…..}, and exchange the task of odd and even processes.

3 Repeat step 2 until the stop criteria matches.

4 Stop Criteria
  - 4.1 After per even/odd phase, record whether swap or not. If there are no any swaps in each process, then `odd_/even_is_swap` is false. Otherwise it is true.
  - 4.2 When odd_is_swap and even_is_swap are both false, the sorting is finish.

5 Using `MPI_File_write_at`, so that each rank write to the output file individually.

Advanced

Except the main algorithm part, other parts are similar to those of the basic version.

Main algorithm:

1 Each process sort its elements by the built-in quick sort.

2 Even phase (shown in the fig. 1)

- 2.1 The even rank processes (except the last) need to transfer all its elements to its next process, and receive all the elements of its next process.
- 2.2 The odd rank processes need to transfer all its elements to its previous process, and receive all the elements of its previous process.
- 2.3 Merge the process's own elements with the receive elements. If the rank is even/odd replace its elements with the former/latter half part.

3 Odd phase

- 3.1 Similar to the even phase, but exchange the task of the odd and even processes.

4 Repeat step 2 until the stop criteria matches.

5 Stop Criteria is similar to that of basic version.

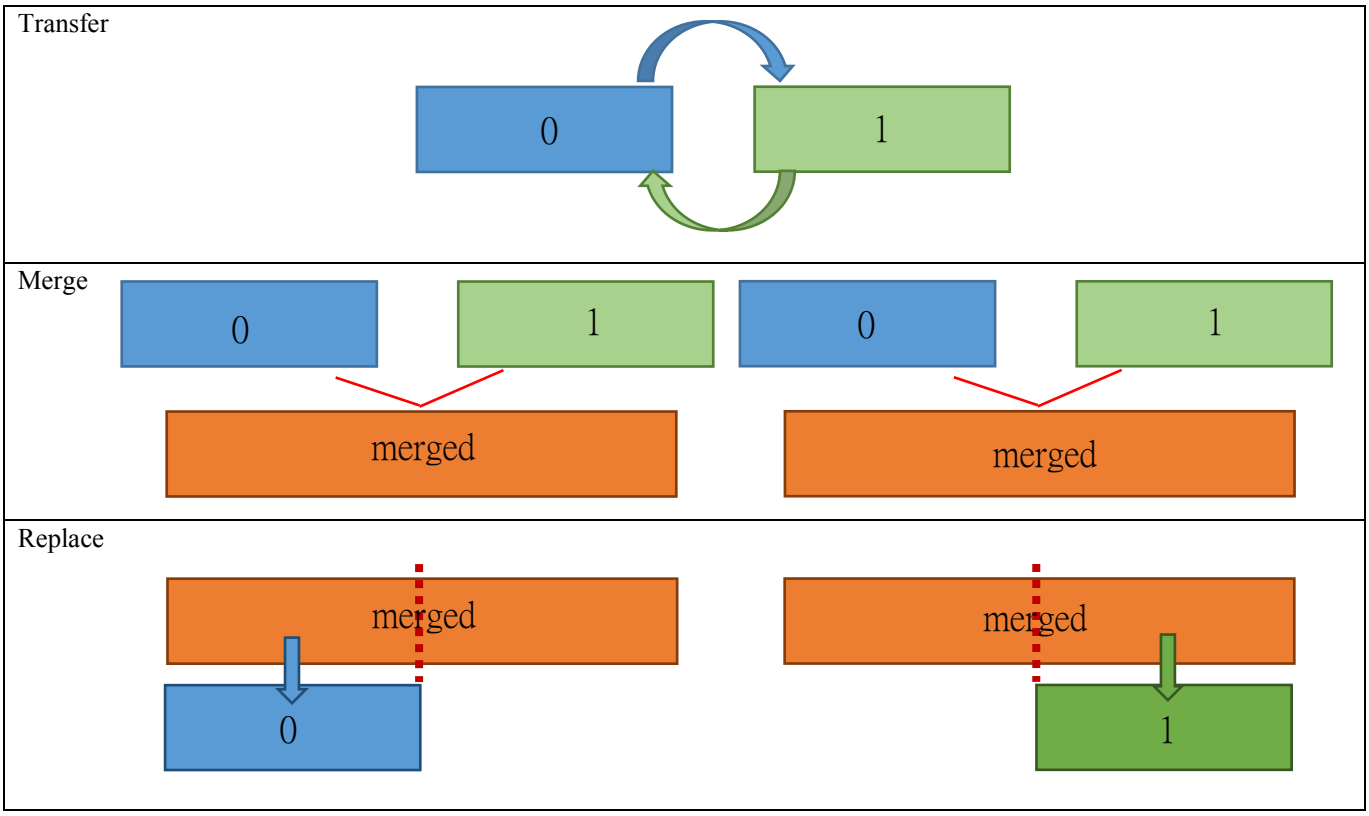


Figure 1. Even phase

Experiment & Analysis

Methodology

- **System Spec**

  Using the provided cluster to measure.

- **Performance Metrics**

  I use clock_gettime(CLOCK_REALTIME) to measure my experiment performance.

  Sample code like:

```
long compute_time;
struct timespec begin, end;
clock_gettime(begin);
// Computing code
clock_gettime(end);
calculate_time(&compute_time, begin, end);
```

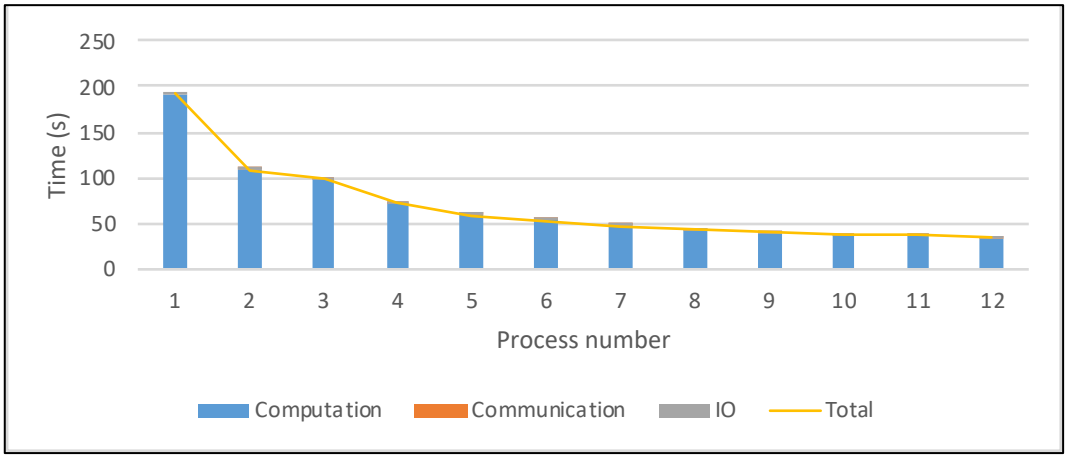
Plots: Speedup Factor & Time Profile and Discussion

- **Strong Scalability**

Test data: 17.in

Data size = 400000

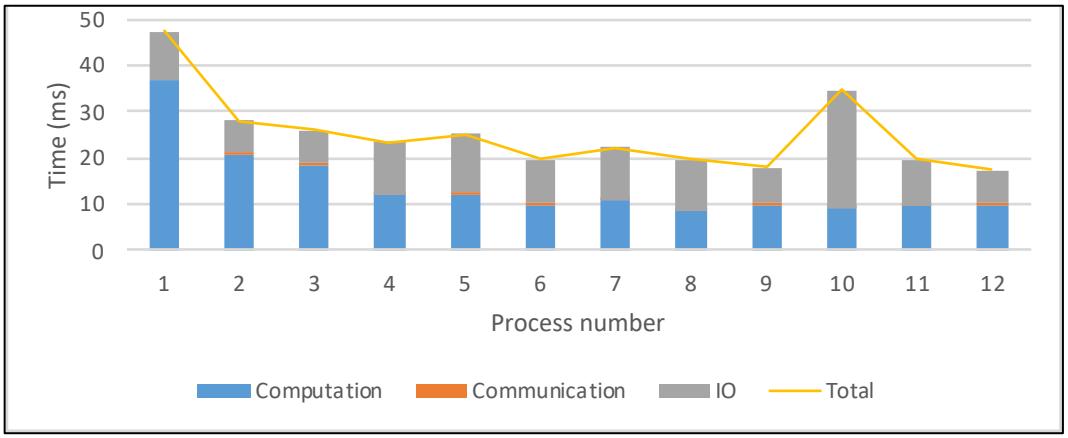
Figure 2. Basic: Strong Scalability in Single Node



Average execution time: 68.43225 seconds

First 4 average execution time: 117.6461 seconds

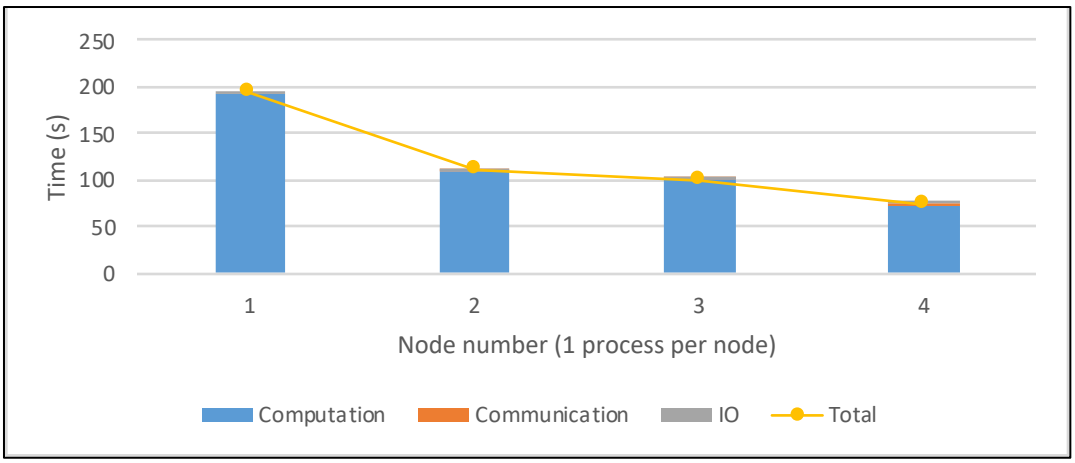
Figure 3. Advanced: Strong Scalability in Single Node



Average execution time: 25.22451 milliseconds

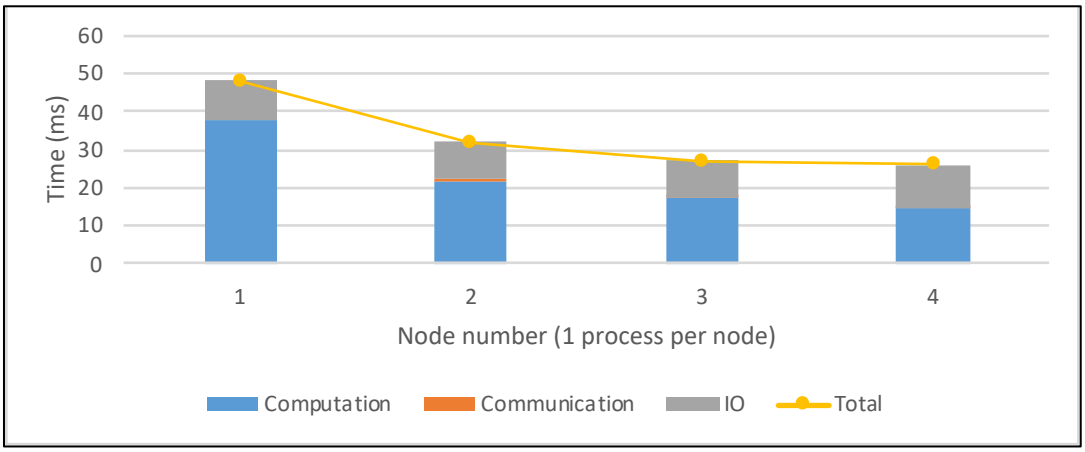
First 4 average execution time: 31.31914 milliseconds

Figure 4. Basic: Strong Scalability in Multiple Nodes



Average execution time: 119.5636 seconds

Figure 5. Advanced: Strong Scalability in Multiple Nodes



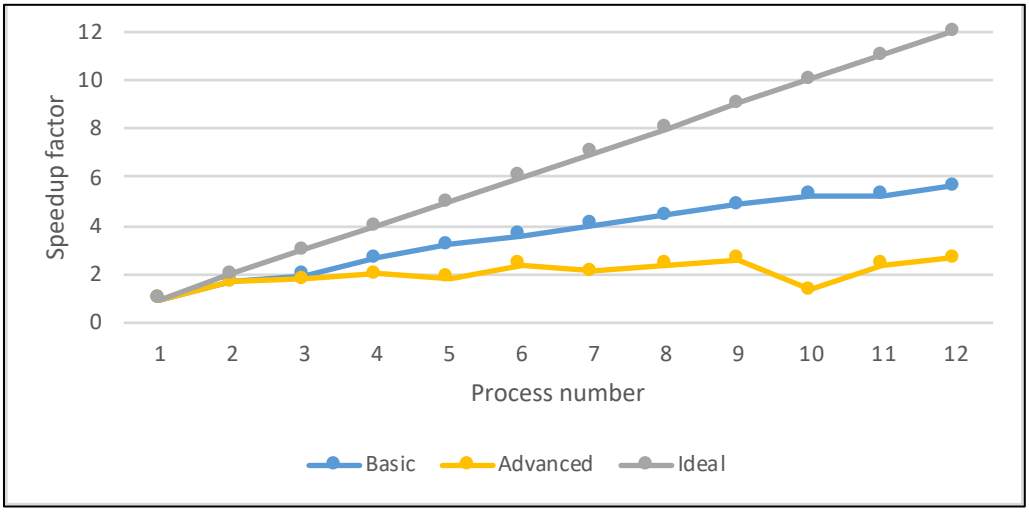
Average execution time: 33.30866 milliseconds

■ **Speedup**

Test data: 17.in

Data size = 400000

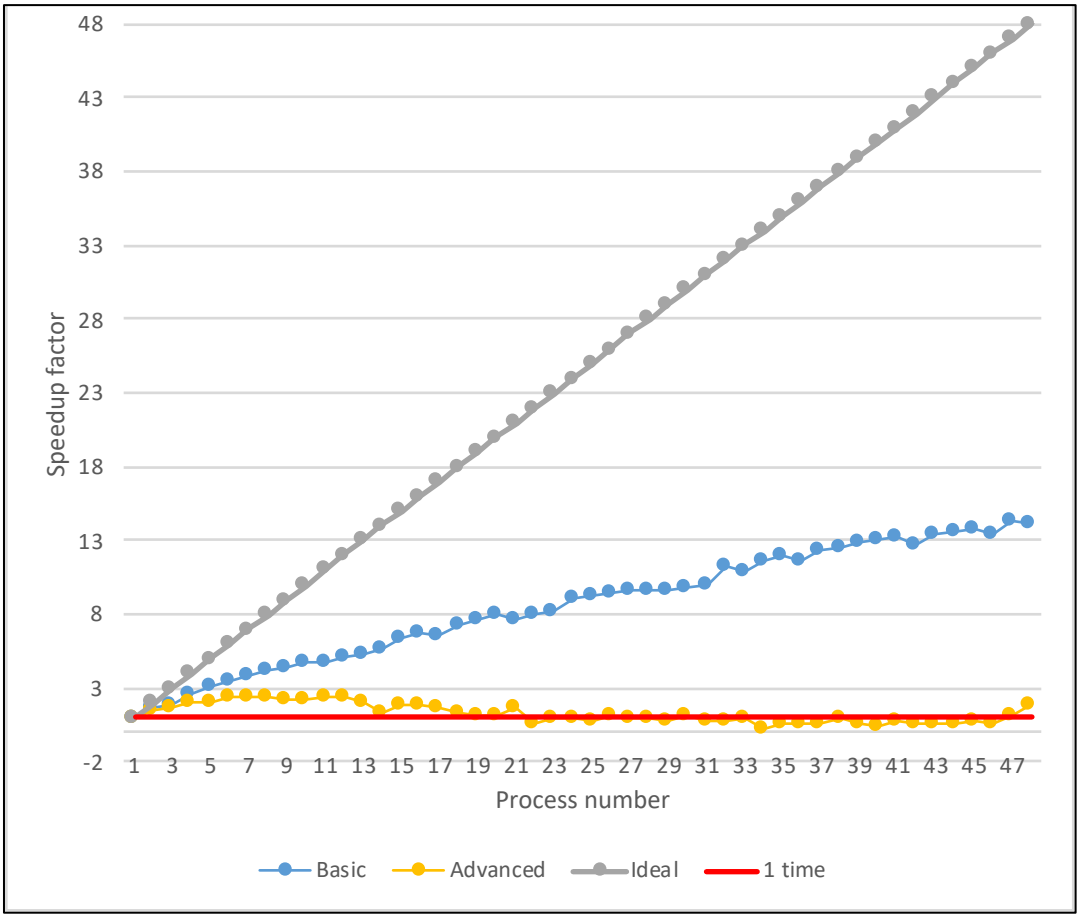
Figure 6. Speedup in multi processes



Average basic speedup factor: 3.65 times

Average advanced speedup factor: 2.03 times

Figure 7. Speedup in multi processes



Average basic speedup factor: 8.592617 times
Average advanced speedup factor: 1.251365 times

- **Discussion**
  - **Basic and Advanced Performance Comparison**

In single node, basic version's average execution time from 1 process to 12 processes is 68.432s, while advanced version's average execution time is 25.225ms , which is shorter and has around 2713 times speedup. The information is shown in the fig. 2 and 4.

In multiple nodes, basic version's average execution time from 1 process to 12 processes is 119.564s, while advanced version's average execution time is 33.309ms , which is shorter and has around 3590 times speedup. The information is shown in the fig. 3 and 5.

Thus, advanced version no matter in single node or multiple nodes is thousands times faster than basic version. However, in single node if we only focus on the first 4 cases which has the same number of processes as the cases in multiple nodes, we can find that advanced version has around 3756 times speedup. That is, in single node or in multiple nodes does not affect a lot on the execution time when the node number is less than 4 nodes. The reason is that the data size is too small and the communication time is not much and not different a lot when the number of nodes increases.

The complexity of the basic version is

$$(\text{data number})\frac{\text{data number}}{\text{process number}} \quad (1),$$

which is $O(\text{data number}^2)$.
and the complexity of advanced version is

$$(\frac{\text{data number}}{\text{process number}}\log\frac{\text{data number}}{\text{process number}}) + 2 * \text{data number} \quad (2),$$

which is $O(\frac{\text{data number}}{\text{process number}}\log\frac{\text{data number}}{\text{process number}})$.

The computing complexity is the main reason that advanced version is thousands times faster than basic version.

◆ **I/O, CPU, Network Performance Comparison**

In my report, the computing time stands for the CPU time, and the communication time stands for the network time.

In basic version, computing time is the bottleneck and the largest part of the whole execution time. Precisely, even in the 12 processes case whose execution time is the last, computing time is still over 99%, which is shown in fig 2. The probable improvement can be that randomizing the input data in advanced to avoid the worst case.

In multiple nodes, computing time is the bottle and the largest part of the whole execution time, when the number of processes is less than 12, which is shown in fig 3. However, we can find that the computing time does not decreases a lot when the number of processes is over 6, while the I/O time increases slightly. Thus, when the processes keep increasing, the I/O will become the bottleneck. The reason is that the data size is too small and my I/O policy is each process read/write individually. Therefore, the I/O time increases as the number of the processes increases. The probable computing time improvement can be that modifying the main algorithm step 1, and replacing the built-in quick sort whose complexity is $O(n^2)$ by radix sort whose complexity is $O(n)$. That is the computing complexity can be $O(\text{data number})$, not $O(\frac{\text{data number}}{\text{process number}}\log\frac{\text{data number}}{\text{process number}})$.

◆ **Basic and Advanced Scalability Comparison**

If we see the figure 6 and 7, we can find that basic version though is not as good as the ideal, its speedup factor is almost linear. Though advanced version' speedup factor increases in the first 12 cases, it decreases slightly afterwards, especially it lower than 1 time sometimes. The reason is that the data size is too small, so that the execution time are almost the I/O time when the processes number is many, that is the speedup factor is close to the error. The improvement is that I need to do experiment with the larger data size to avoid error.

Experiences / Conclusion

- **Conclusion**

  Basic version has great scalability, but the total execution time is large. The reason is that its CPU time takes over 99% time since its computing complexity is $O(\text{data number}^2)$.

Advances version has bad scalability, but the total execution time is less. The reason of bad scalability may be the input data size is too small, so that the I/O time is not stable and the experiment result is closed to error.

- **Experiences**

  In this assignment, I have learned how to write a parallel program, which is also the first time for me to write a parallel program. The main difficulty is that I am not familiar with the MPI and the parallel concepts. Although I have tried many different algorithms to write the advanced version, only one of them is executable.