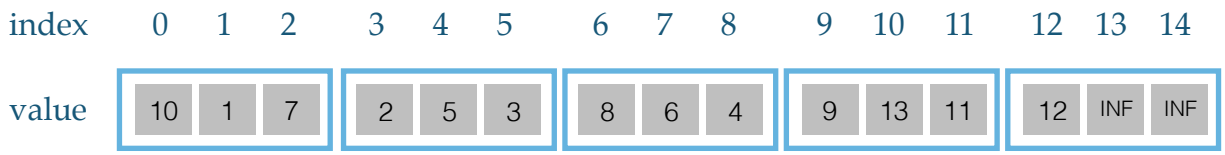
資工系100062236林修安

Parallel Programming HW1

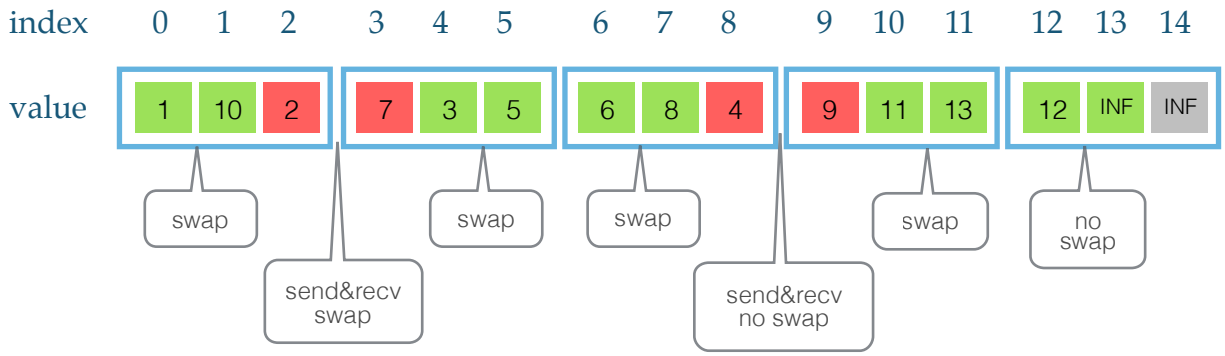
Odd-Even Sort

Implementation

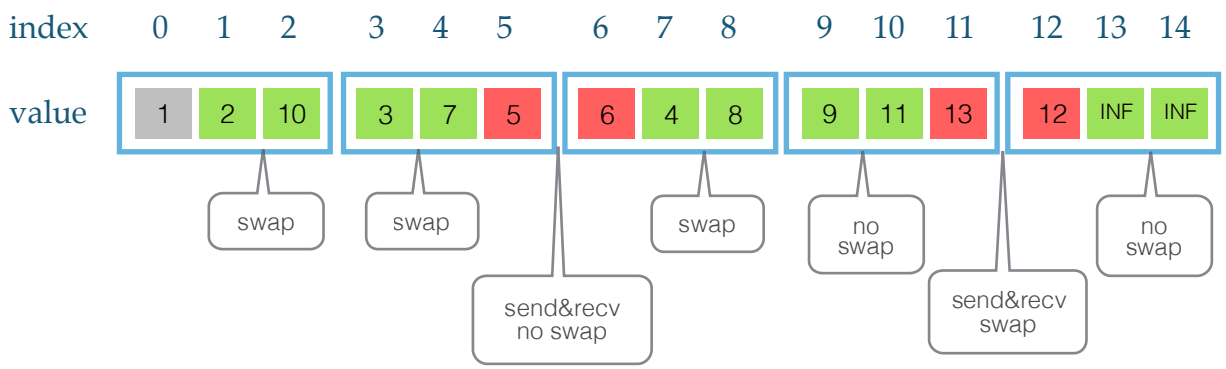
1. Basic Version
   - For any input that the length is not dividable by number of process, I append INF (2147483647) to the array up to the minimum dividable length after reading the input file, so every process can be distributed with segment of numbers of equal length.



   - Even Phase
     - Calculate the indices of the first and last values of each process.
     - If the value with even index is at the end of the segment and it's not at the last process, it sends the value to the next process.
     - If the value with odd index is at the front of the segment, it should wait to receive value from previous process.
     - After receiving number, if the sender is bigger, then swap the values and send the smaller one back to the sender; if not, just send the original value back.
     - Otherwise swap if the subsequent value is smaller.



   - Odd Phase
     - Calculate the indices of the first and last values of each process.
     - If the value with odd index is at the end of the segment and it's not at the last process, it sends the value to the next process.
     - If the value with even index is at the front of the segment and it's not indexed 0, it should wait to receive value from previous process.
     - After receiving number, if the sender is bigger, then swap the values and send the smaller one back to the sender; if not, just send the original value back.
     - Otherwise swap if the subsequent value is smaller.



- Asychronous Communication
  - While a value is waiting to receive, the remaining values can simultaneously do swapping. I sectioned `MPI_Wait()`,`MPI_Irecv()` and `MPI_Isend()` into different branches using flags roughly as below.

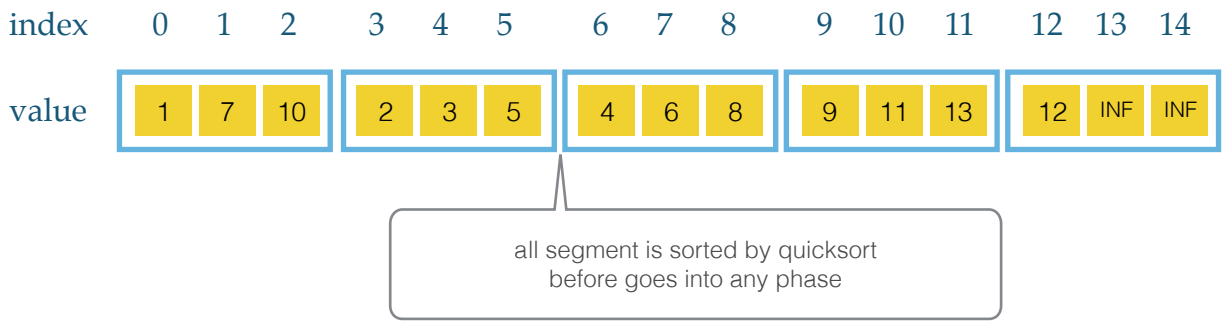
```
if(need to recv){
      flag1 = 1;
      MPI_Irecv();
}
if(need to send){
      flag2 = 1;
      MPI_Isend();
}

for(do swap-work){
      …
}

if(flag1){
      MPI_Wait();
      swap() or not;
      MPI_Isend();
}
if(flag2){
      MPI_Irecv();
      MPI_Wait();
}
```

    In such way, the swapping won't be stuck because of waiting.

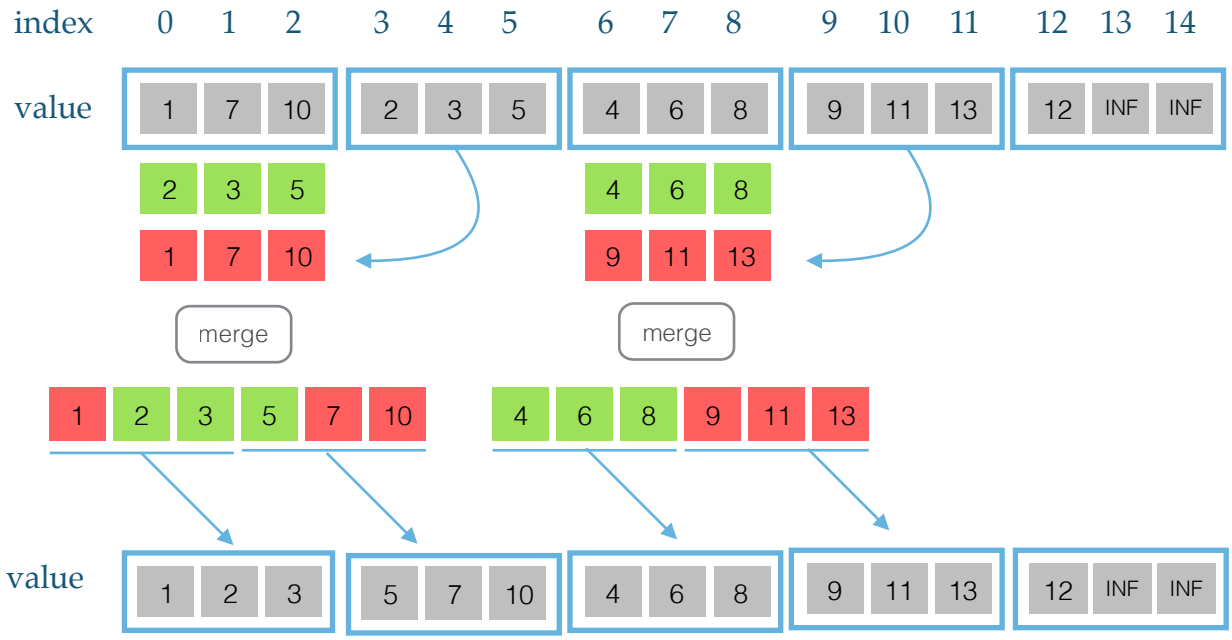
- Detection of Sorted or not
  - In each phase of sorting, the function returns 0 if swapping still occurs, and 1 otherwise.
  - After an even-phase and an odd-phase, every process holds two values indicating whether either phases is sorted. By multiplying all resultant values together using `MPI_Allreduce()`, if the product is 0 then the operation of both phase should continue, otherwise the list is sorted.

2. Advanced Version

- The same preprocess is performed by appending INF to the input array making it dividable by the number of process.
- Before any sorting phases, I use quicksort to sort the segment of each process in advance.

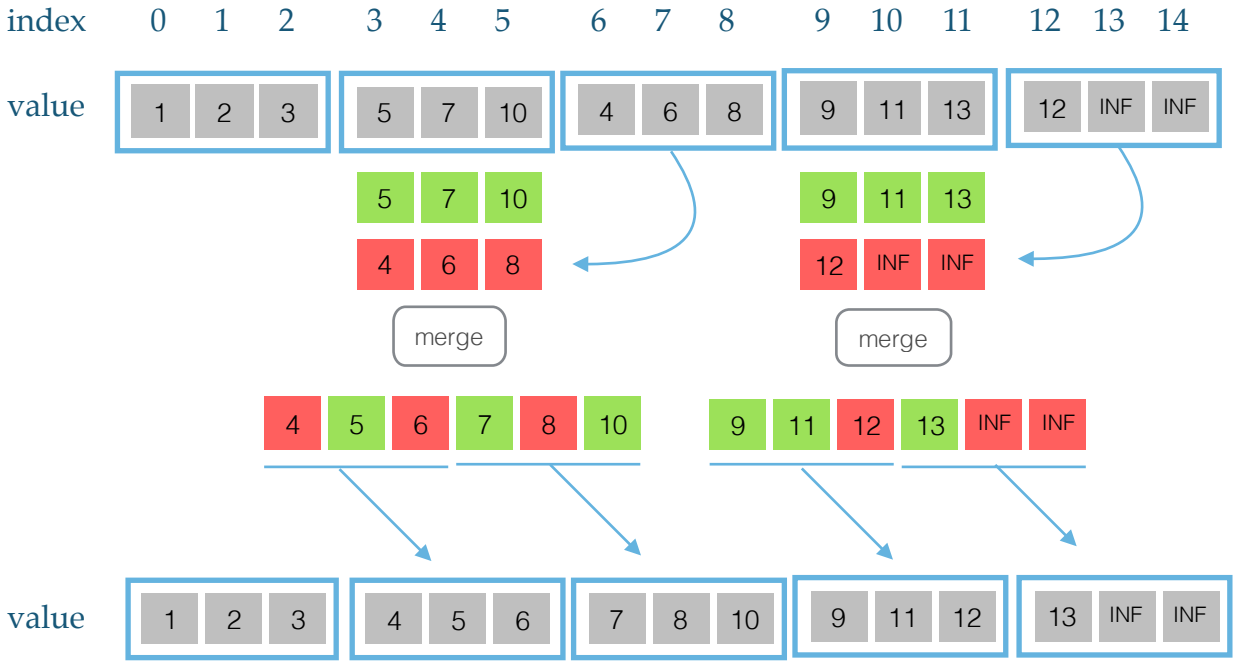


- Even Phase
  - Process with odd rank send the entire segment to the previous process.
  - In process with even rank, a new array is created to merge its own segment and the one sent to it.
  - After merging, the values will be sorted in order, and the latter half is sent back to the process with odd rank. The former half remains as the new segment of the even rank process.



- Odd Phase
  - Process with even rank send the entire segment to the previous process, except process ranking 0.
  - In process with odd rank, a new array is created to merge its own segment and the one sent to it.

- After merging, the values will be sorted in order, and the latter half is sent back to the process with odd rank. The former half remains as the new segment of the even rank process.



• Synchronous Communication

- A recipient process must wait for the whole segment to be sent, and after merging, the sender process must wait for the sorted segment to be brought back. During the operation, unlike the basic version, nothing else is done while waiting, so synchronous communication is applied here.

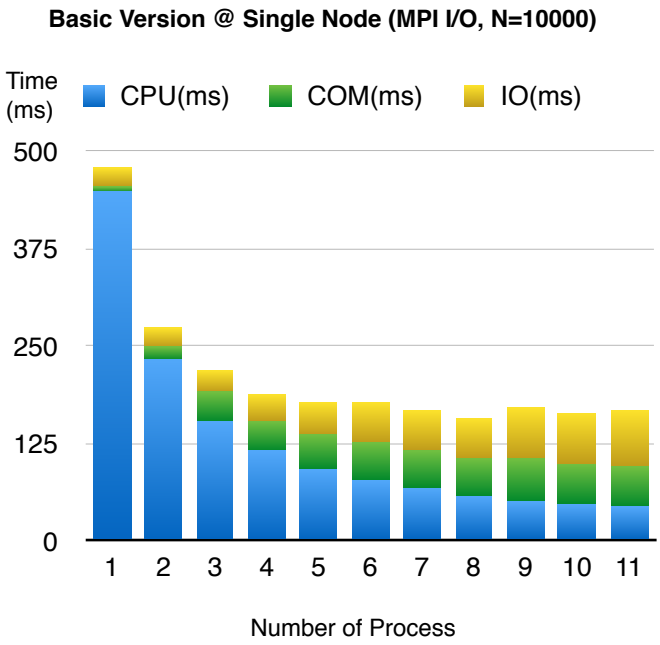
• Detection of Sorted or not

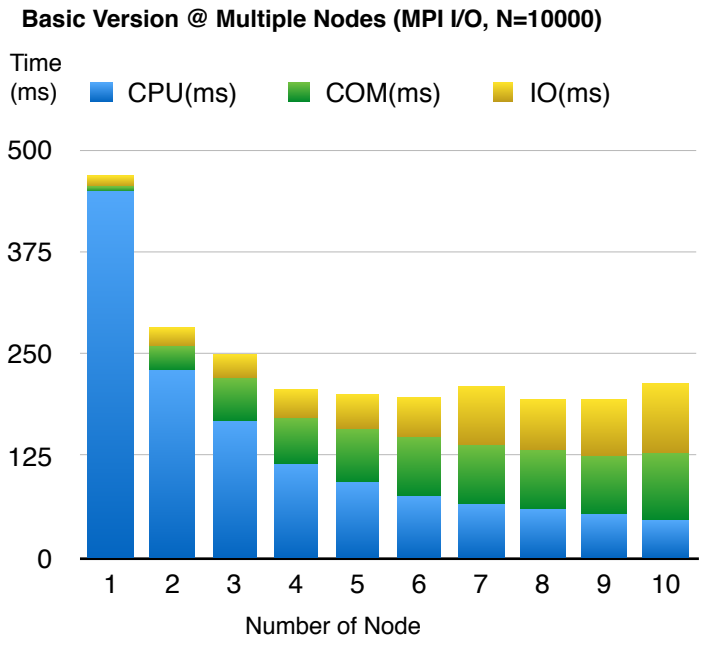
- Implementation is as same as the basic version.

Experiment & Analysis

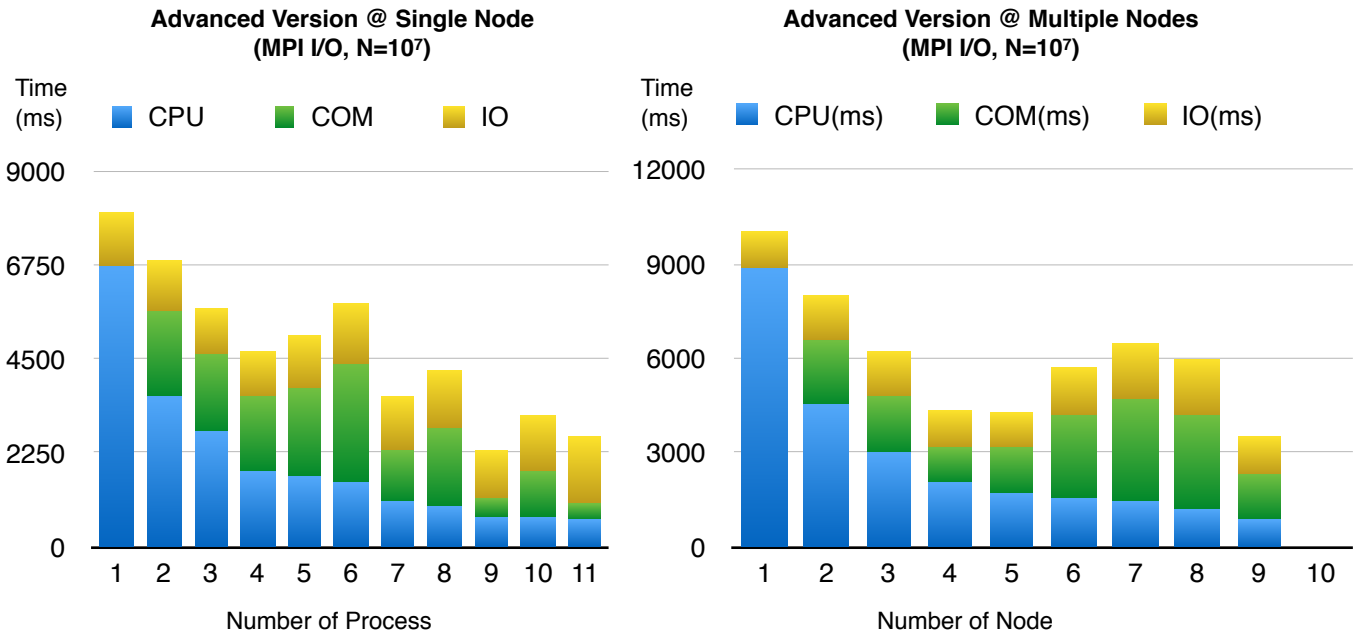
• Time Measurement:
- I/O time is measured when file reading and writing occur.
- Communication time is measured when Send(), Recv(), and Wait() occur, and where Bcast(), Gather(), Allreduce() take place.
- Computation time (CPU time) is the difference of total time deducting I/O time and communication time.

• Single Node vs Multiple Nodes
- Purpose : to determine the factor causing difference in communication time.
- The following figures are data of basic version and advanced version run respectively on single node and multiple nodes.
- The communication time increases as the number or process increases. But the communication time gets longer on multiple nodes. It is because on multiple nodes, the processes are across nodes, which require more time to transfer between network, while processes on single do not need to communicate across computers, making the increment smaller than that of multiple nodes.





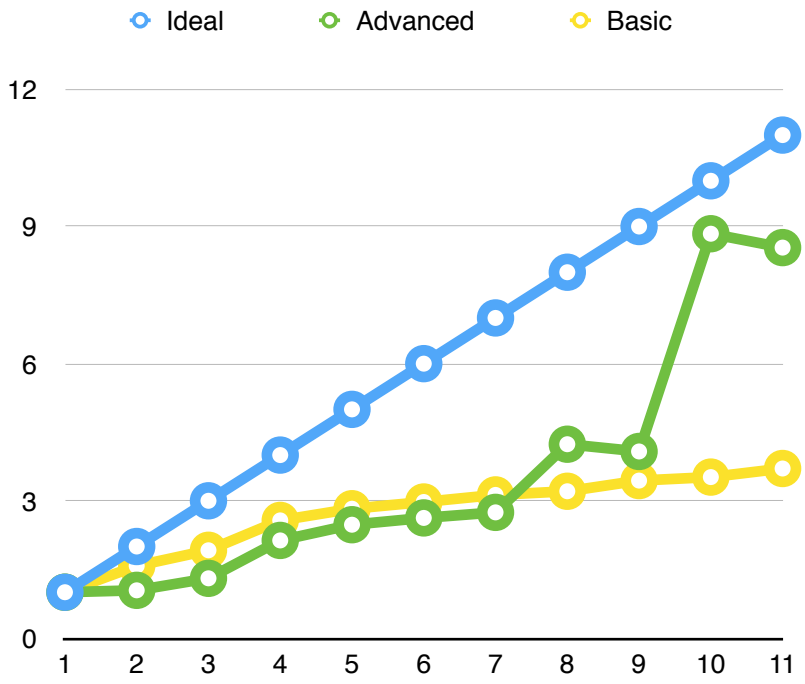
- For the figures of advanced version not the next page, it's apparent that the communication time fluctuates. This happens maybe because the experiment was performed during rush hours that many people were also doing tests. But the communication time of multiple nodes are less likely to be affected, and acts as expected to some extent.



- The computation time is constantly decreasing since time is distributed to more processes.

• Speedup Factor

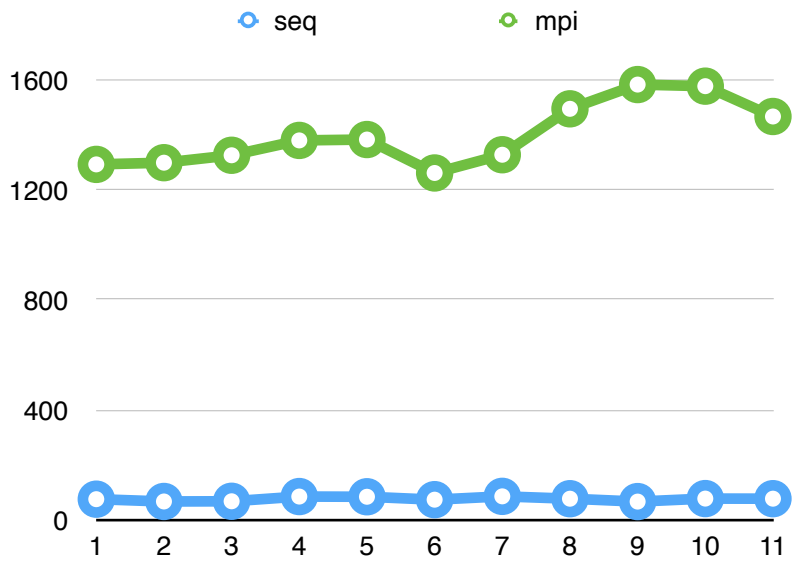
- The speedup factor is not as good as ideal because I/O time does not decreases as the number of process increases, and communication time also becomes larger.



Basic@Single Node N=10000
Advanced@Single Node N=10^7

• Performance of sequential and MPI I/O

- I used MPI_File_read() and MPI_File_write() to implement MPI I/O. Since the function causes process to wait for previous one to finish reading or writing, the time increases as the number of process increases.
- Sequential I/O spends pretty much the same amount of time, because there is only one process reading or writing the file, so only a small amount of time is needed.



Basic@Single Node N=10000
Advanced@Single Node N=10000