Student ID: 101062319

Name: 巫承威

Homework 1 - Odd-Even Sort

Report

- ## Part 1: Implementation
  - ### Basic

In my basic version, I followed the the restriction by working items. So I applied the classic odd-even sort algorithm directly in parallel to my code.

First, I handled the exception like the lab1, and assigned the arguments to my own declared variables. Then I use the following function to open the input file by MPI I/O:

```
//  Part 1: Use MPI I/O to open the input file
MPI_File ifh;
MPI_Status istatus;
MPI_File_open(MPI_COMM_WORLD, input, MPI_MODE_RDONLY, MPI_INFO_NULL, &ifh);
```

Also, I declared some variables and use them to store and record different input items such like ***"N < # process"***, ***"N > #process but N can't be divided by #process"***, etc. And I decided to **let each process has same # numbers except the last process. Therefore, the last process may handle more numbers**. The following segments are my implementation of MPI I/O:

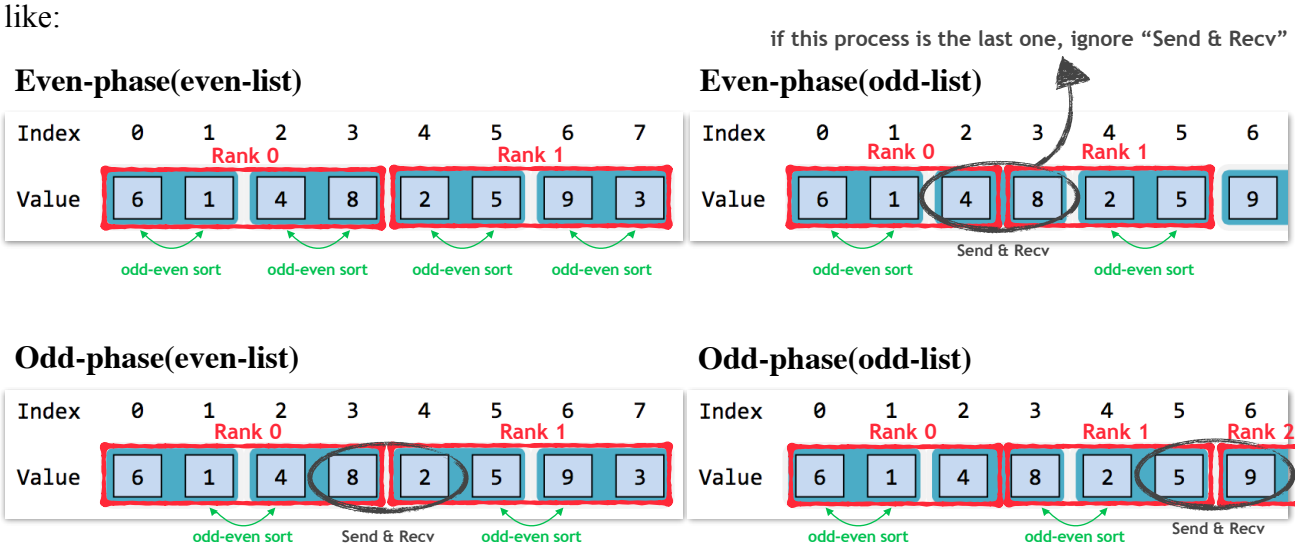
```
node_arr = (int*) calloc(num_per_node, sizeof(int)); // store the N/P numbers in each node
if(N>=size && rank==rank_last-1)
    MPI_File_set_view(ifh, rank*(N/size)*sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
else
    MPI_File_set_view(ifh, rank*num_per_node*sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_read_all(ifh, node_arr, num_per_node, MPI_INT, &istatus);

MPI_File_close(&ifh);
MPI_Barrier(MPI_COMM_WORLD);
```

I used the function "MPI_File_set_view" to indicate every independent file pointer in each process to tell the process where to start read the input file. And then use "MPI_File_read_all" to read the input file collectively. Finally, I closed the input file pointer and set "MPI_Barrier" to insure every process are all reach here.

Now I will briefly introduce my implementation of odd-even sort algorithm. First, I set two variables "node_sorted" and "all_sorted" to specify whether the sorting process is done or not. The following segments are the pseudo codes of my sorting algorithm:

```
while(not all process sorted){
        initial the value of node_sorted & all_sorted to be 1
        if(now is even-phase){
                // specify the length of number array in each process
                // and sort them first
                if(even-length){
                        // only sort (by odd-even sort)
                }
                else{
                        if(rank of process==even){
                                // sort local array (by odd-even sort)
                                // send the last number to the next process
                                // send first, then receive
                        }
                        else{
                                // sort local array (by odd-even sort)
                                // receive the number sent from previous process
                                // compare it with the first number in local array
                                // sent back the result of swapping
                        }
                }
        }
        else{
                // do almost the same with even-phase
                // only change the thing that different rank of process will do
        }
        // checkout if every process is sorted or not
        // if all processes sorted, break the while loop
}
```

In above codes, I used "MPI_Send" and "MPI_Recv" to perform the communication between two contiguous processes. And used "MPI_Allreduce" to logically AND the result of every process whether it is sorted or not, then send it to all process to inform them of whether stopping the iteration or not. After performing sorting algorithm, I used "MPI_File_set_view" again to indicate the location to start to write the result into the output file, and used "MPI_File_write_all" to perform the writing operation. Then, call "MPI_Finalize" to stop the program. The diagram will be looked like:



- Advanced

At first, in my advanced version, I changed the partition assignment of each array. Since I make the last process to store more numbers, I decided to let every process store the same numbers. Therefore, I **add additional number** (***MAX_INT=2147483647***) to the processes which have spaces to store this max number. For example, if N = 37, # process = 3, then I let each process store 3 numbers, and from rank 12 process to the last process, every process has empty space to store the MAX_INT. After filling up all the processes which have empty space, **every process has the same number of element**.

Second, I used "**Merge-Sort Algorithm**" to sort the local number array of each process. And then, instead of exchanging only one number in the basic version, I send and receive the whole array of each process to its neighbor process just like the script of "Chap2_MPI.pdf" said. Before starting the algorithm, I sort the local array first. While any process receiving the array sent from its neighbor process, it will merger it with its own local array and then dynamically generate the merged larger array. Then apply "**Merge-Sort Algorithm**" to it. After sorting this larger array, receiving process will **send back the half of the bigger numbers to its neighbor process whose rank is bigger than itself, and hold on the left smaller numbers**. By **doing this iteration after # process times**, all the number will be sorted correctly. Finally, output the result just like the basic version. The following segments show how the algorithm works:

Doing the following algorithm # process times

- *Even-phase*

  *For each process with **odd rank P**, **send** its number array to the process with **rank P-1**. For each process with **rank P-1**, **merge** its own number array with the number array sent by the process with **rank P** and apply **merge-sort** to it. Finally **send** the **larger** one **back** to the process with **rank P** and **hold** on the **smaller** one.*

- *Odd-phase*

  *For each process with **even rank Q**, **send** its number to the process with **rank Q-1**. For each process with **rank Q-1**, **merge** its own number array with the number array sent by the process with **rank Q** and apply **merge-sort** to it. Finally **send** the **larger** one **back** to the process with **rank Q** and **hold** on the **smaller** one.*

- # Part 2: Experiment & Analysis
  - ## Strong Scalability & Time Distribution (Plot)

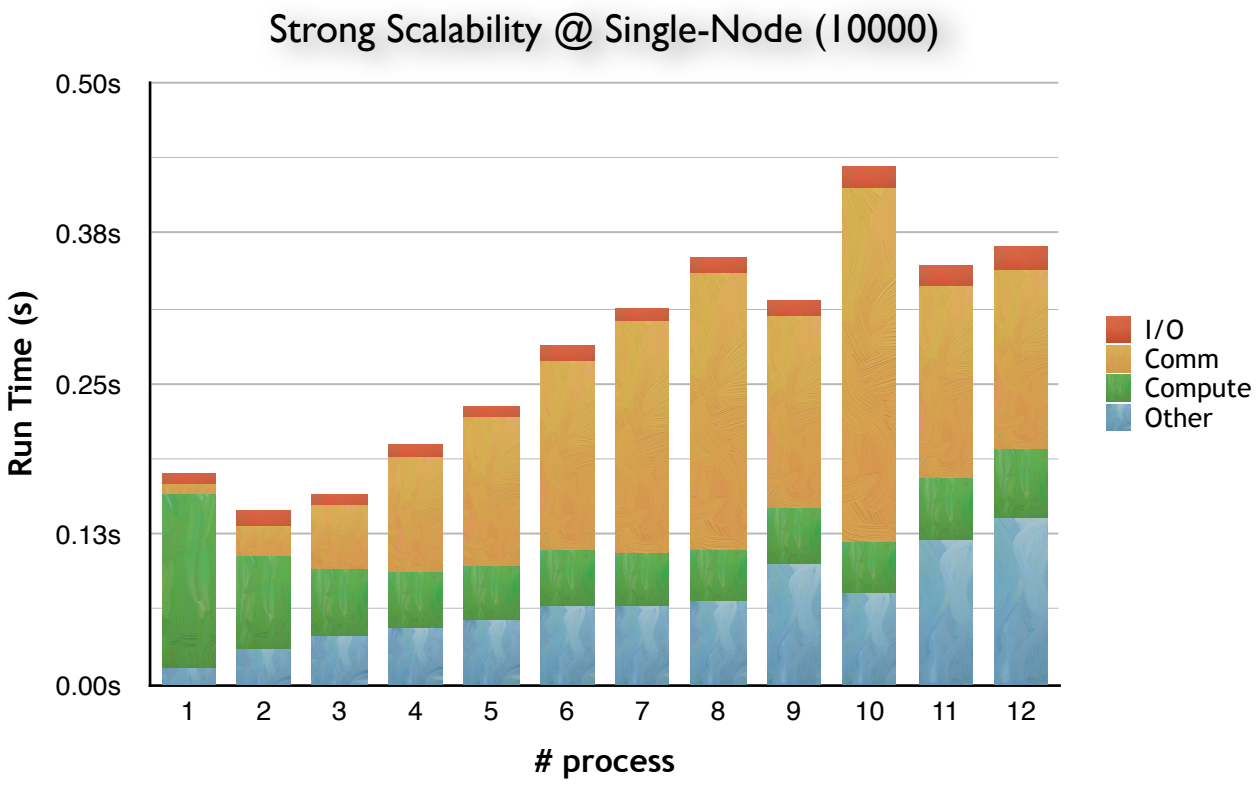
In order to test the correctness and the performance of my basic and advanced version, I tried different combination of the # numbers, # nodes and # process. And the method I used for my profiling is that:

- Use the function "MPI_Wtime" to record the **time stamp**
- **Read** time **starts before opening the file** and **ends after closing the file**
- **Write** time **starts before opening the file** and **ends after closing the file**
- **I/O time sums up** the **read time** and the **write time**
- **Communication time sums up every elapsed time** when call "MPI_Send" and "MPI_Recv"
- **Computation time sums up every computation part** such like **computing the location of the first empty process**, **swapping elements**, **running loops**, **merging** arrays and **sort ing** arrays

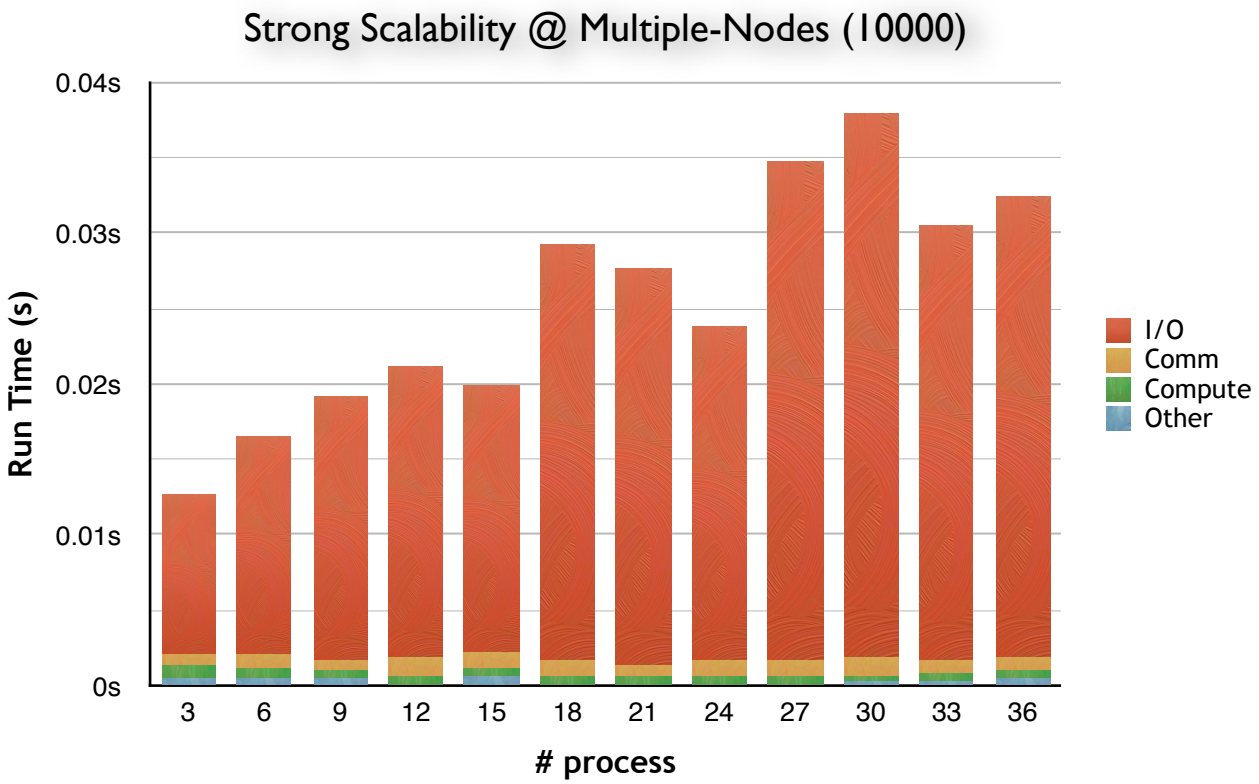
Thus I set the following combination to be my testing input:

(N: # numbers to be sorted, Node: # nodes, Proc: # process per node)

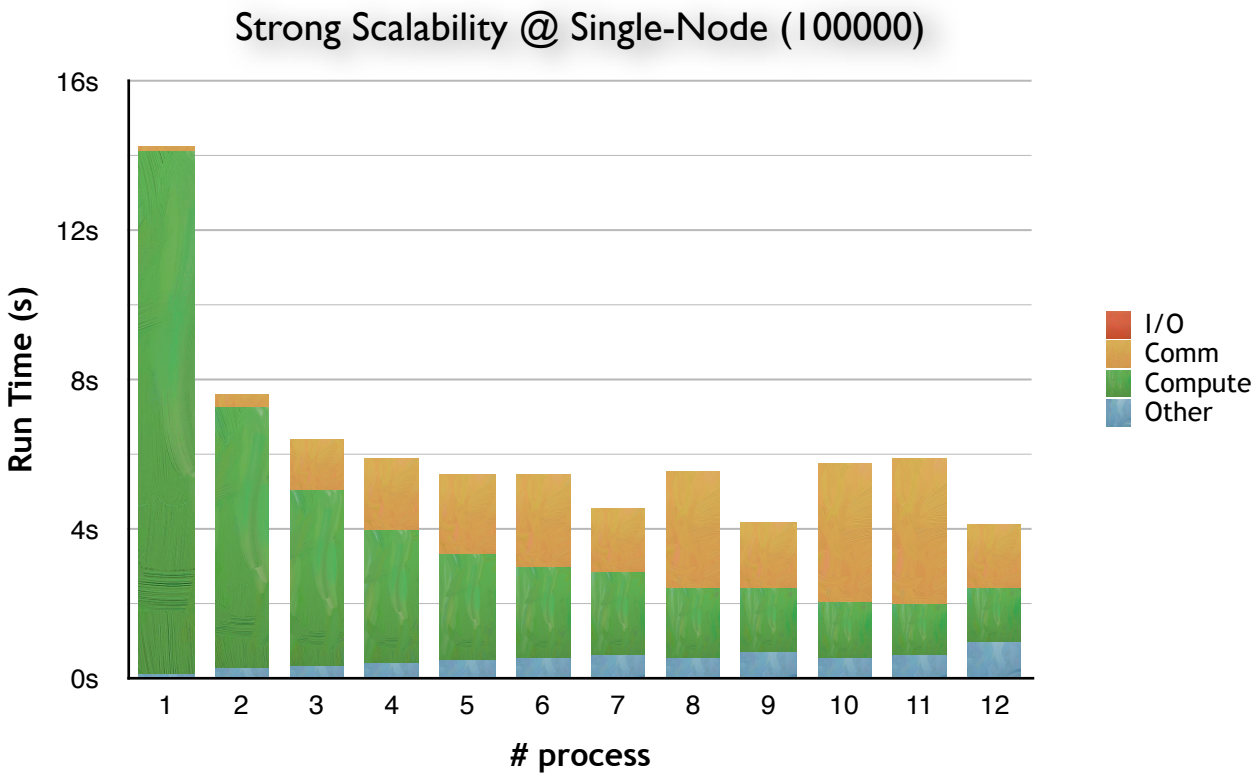
1. N: 10000 (10^4)
   - Single-node (Node: 1) - Basic
     - Proc: 1~12



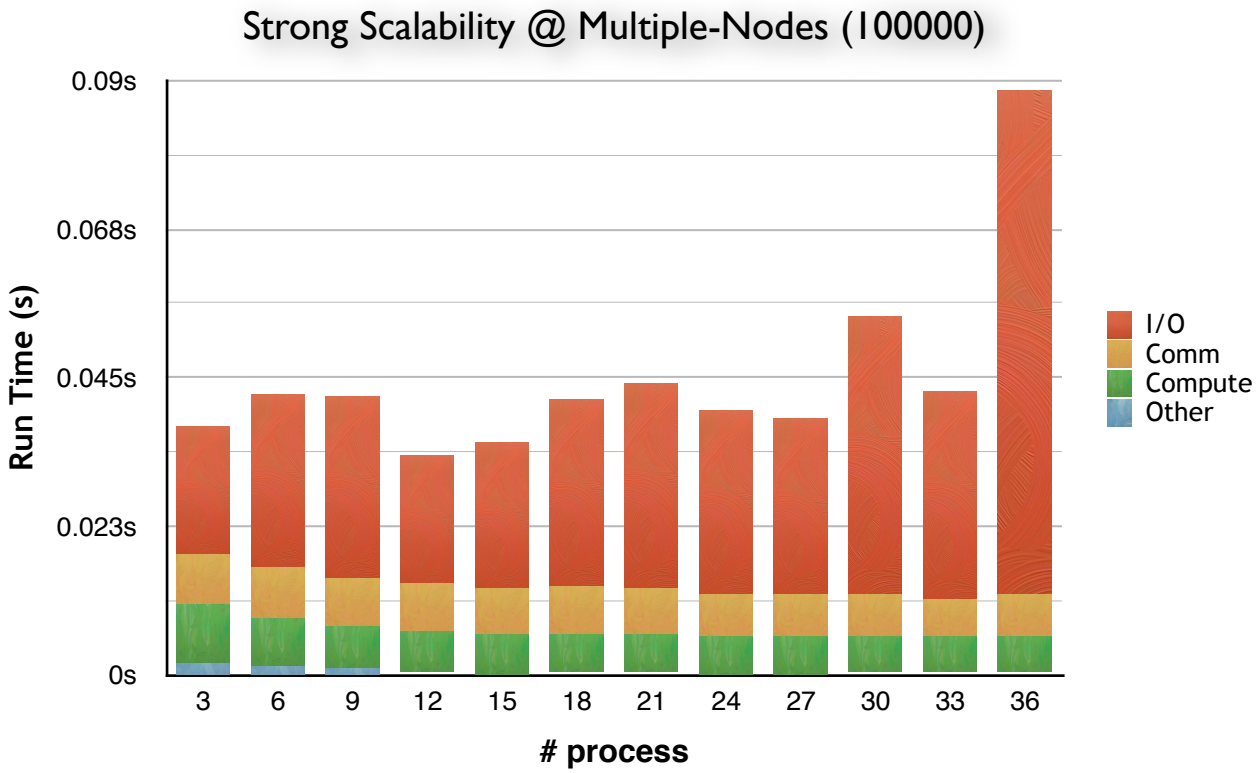
- Multiple-nodes (Node: 3) - Advanced
  - Proc: 1~12



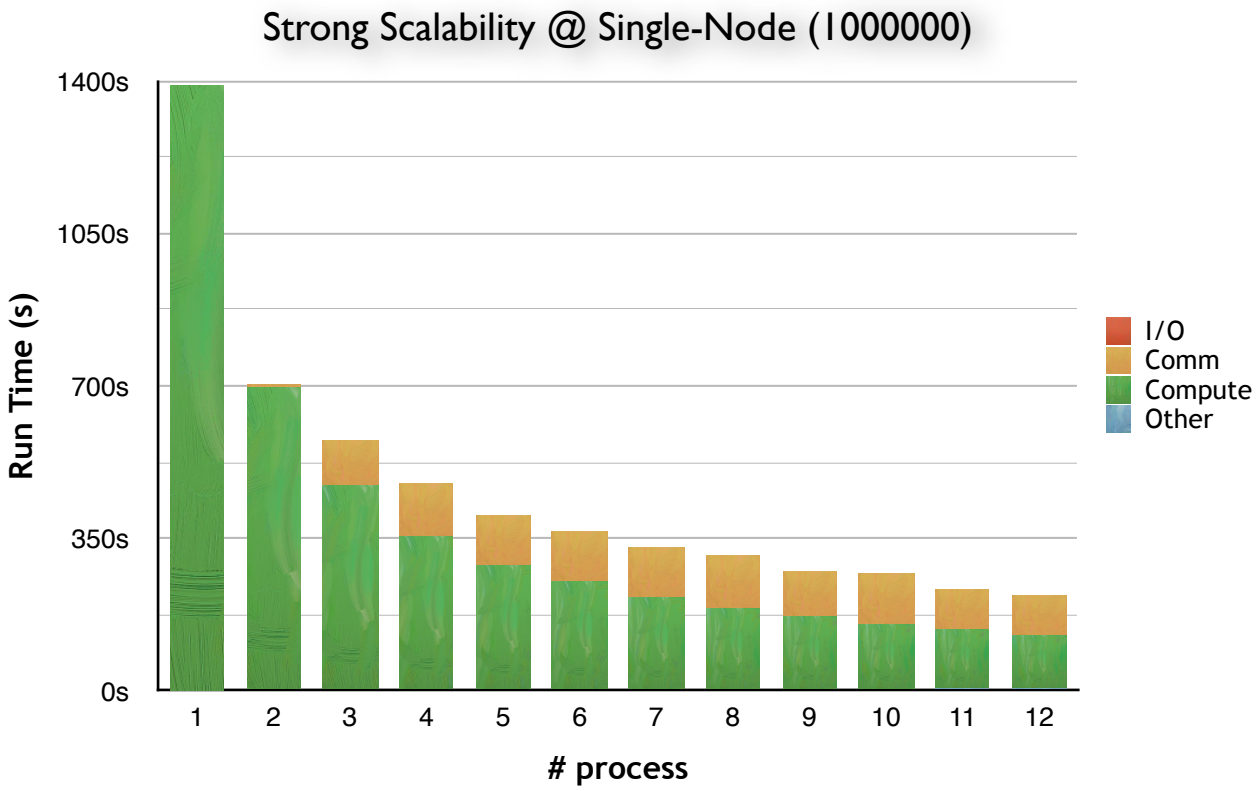
2. N: 100000 (10^5)
   - Single-node (Node: 1) - Basic
     - Proc: 1~12



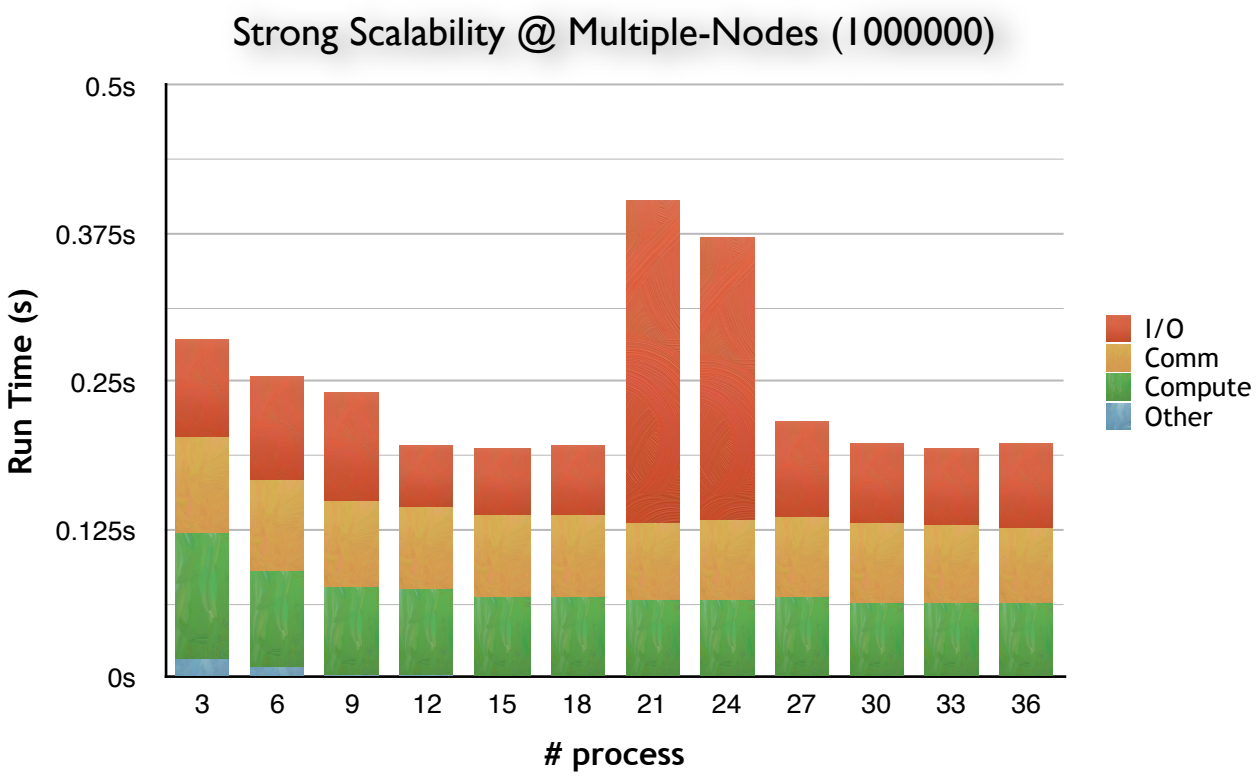
- Multiple-nodes (Node: 3) - Advanced
  - Proc: 1~12



3. N: 1000000 (10^6)
   - Single-node (Node: 1) - Basic
     - Proc: 1~12



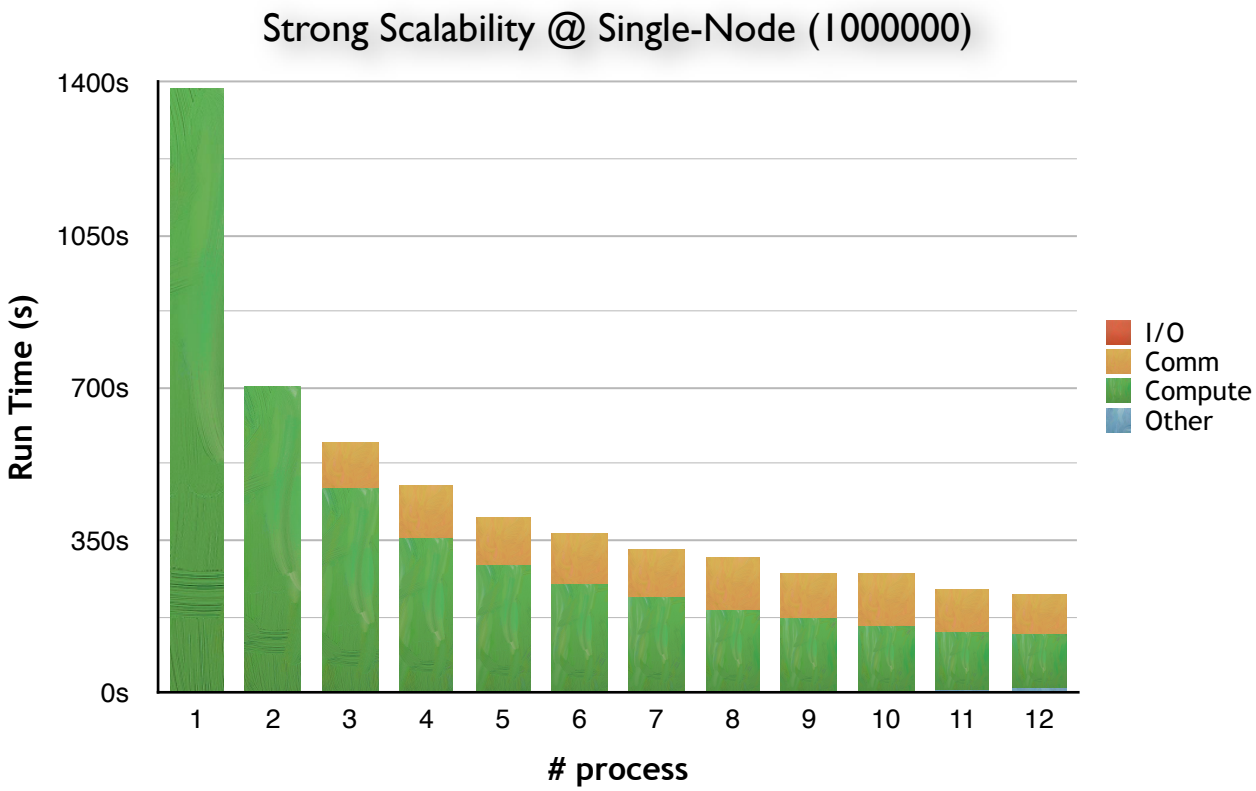
- Multiple-nodes (Node: 3) - Advanced
  - Proc: 1~12



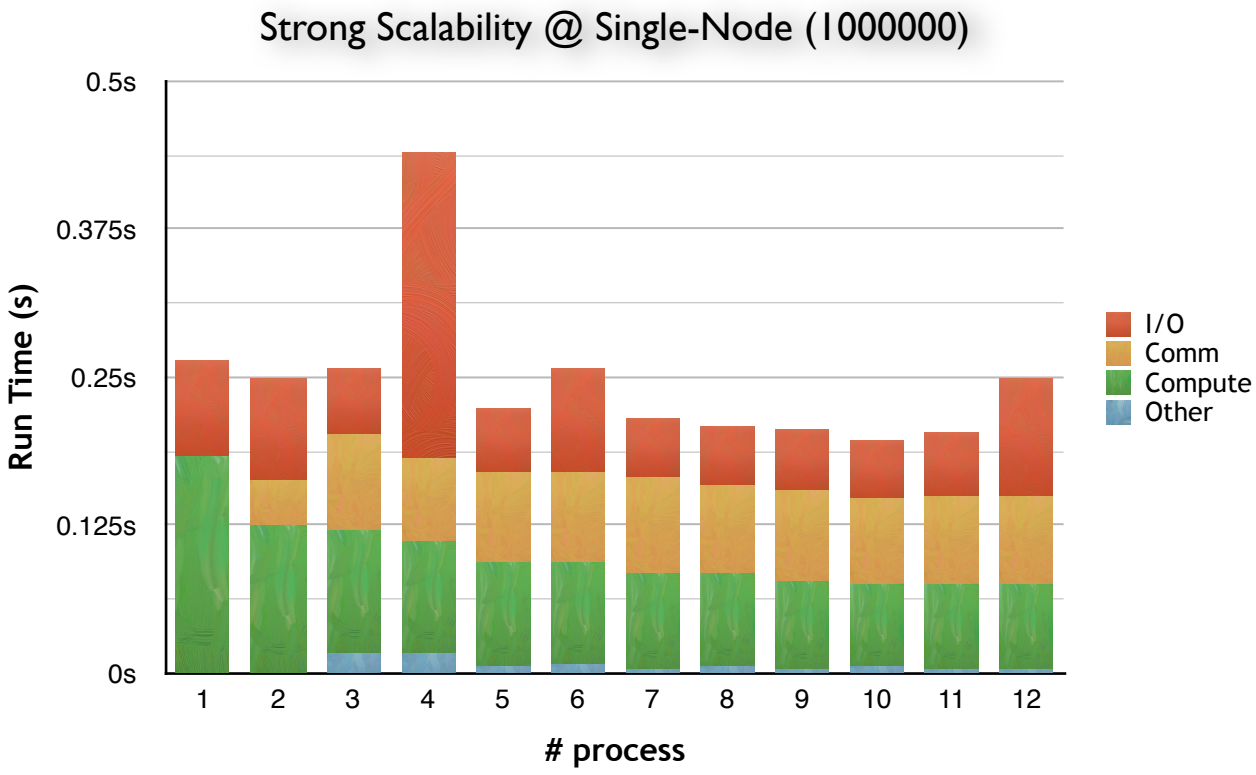
According to the above figures, there are several things I found:

1. When **N is small** (N=10000) and under the **basic version**, the **less processes**, the **less total time**. I think it's because the **communication time will increase** due to more **communication between more and more processes**. However, **advanced version** spend the **most of the running time on I/O**. And I think the reason is that since N is small, **every process** need to **open the file**, **read few numbers** and **close it directly**, that's why the I/O overhead is much more than other operations.
2. While **N is increasing** (now N=100000), the performances of basic and advanced ver sion will be looked more reasonable. In **basic version**, the **time** spent on **computation** part is **dramatically increasing** and also the **communication time in creasing step by step owing to more processes**. And in **advanced version**, the **percentage of I/O overhead decreases apparently**.
3. Finally, if **N is big enough** (N=1000000), the **gap of the performances between basic and advanced version is apparently large**. Basic version need almost about 1400 seconds to run on 1 process. Although the total run time is decreasing amazingly, compared to advanced version, it's totally on different magnitude. So I finally chose the bigger problem size (N=1000000) to profiling the overhead of each part:

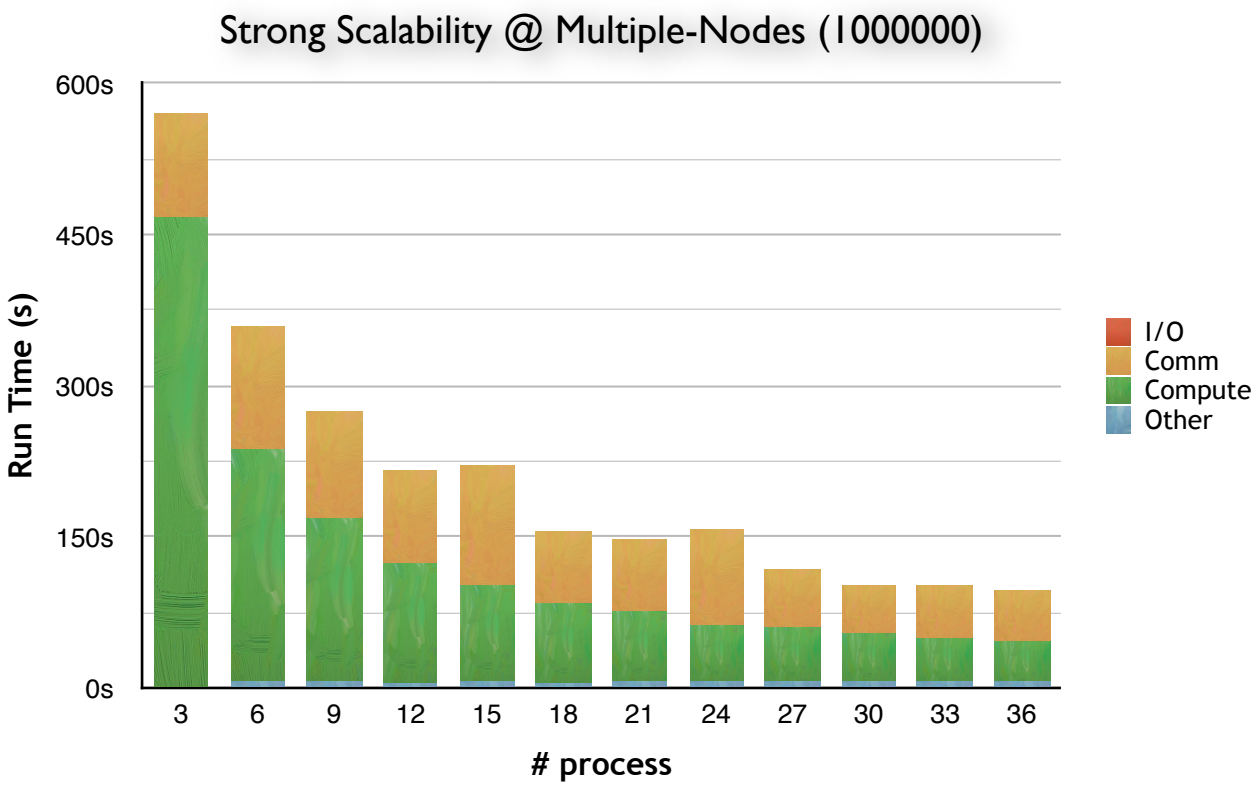
- Single-node (Node: 1)
  - Basic



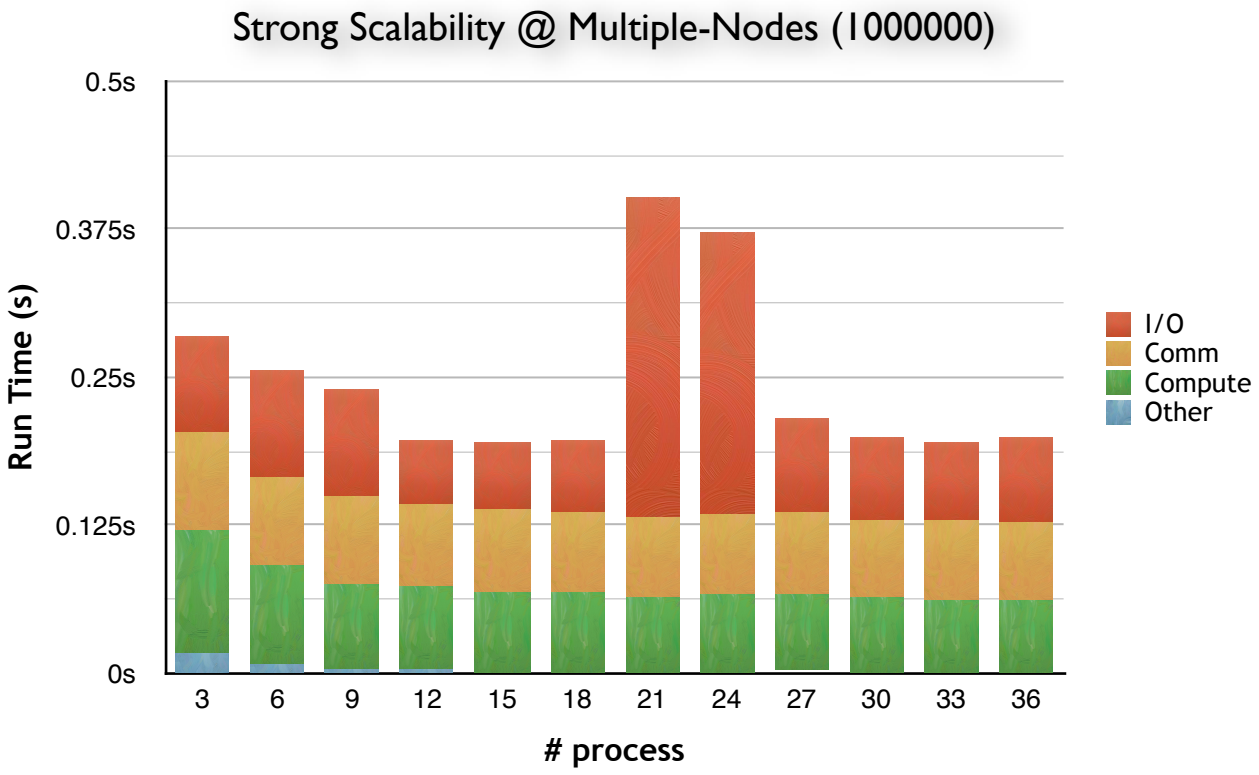
  - Advanced



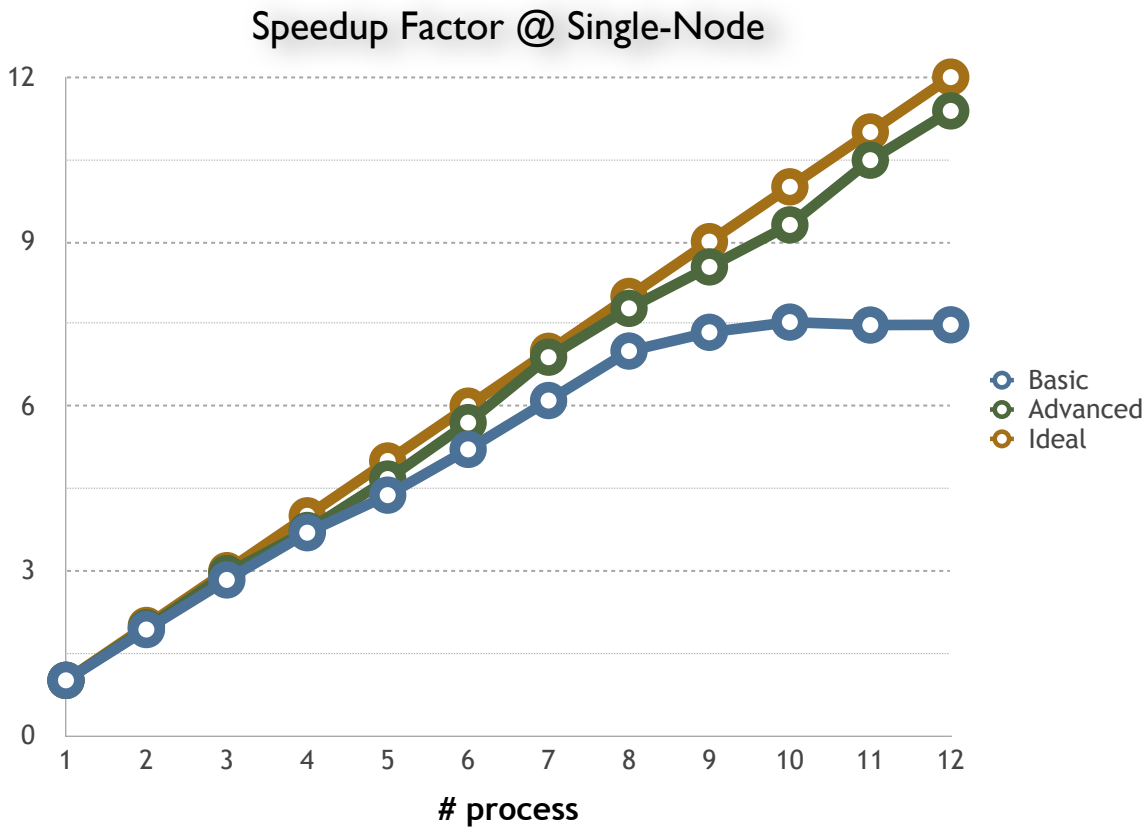
- Multiple-nodes (Node: 3)
  - Basic



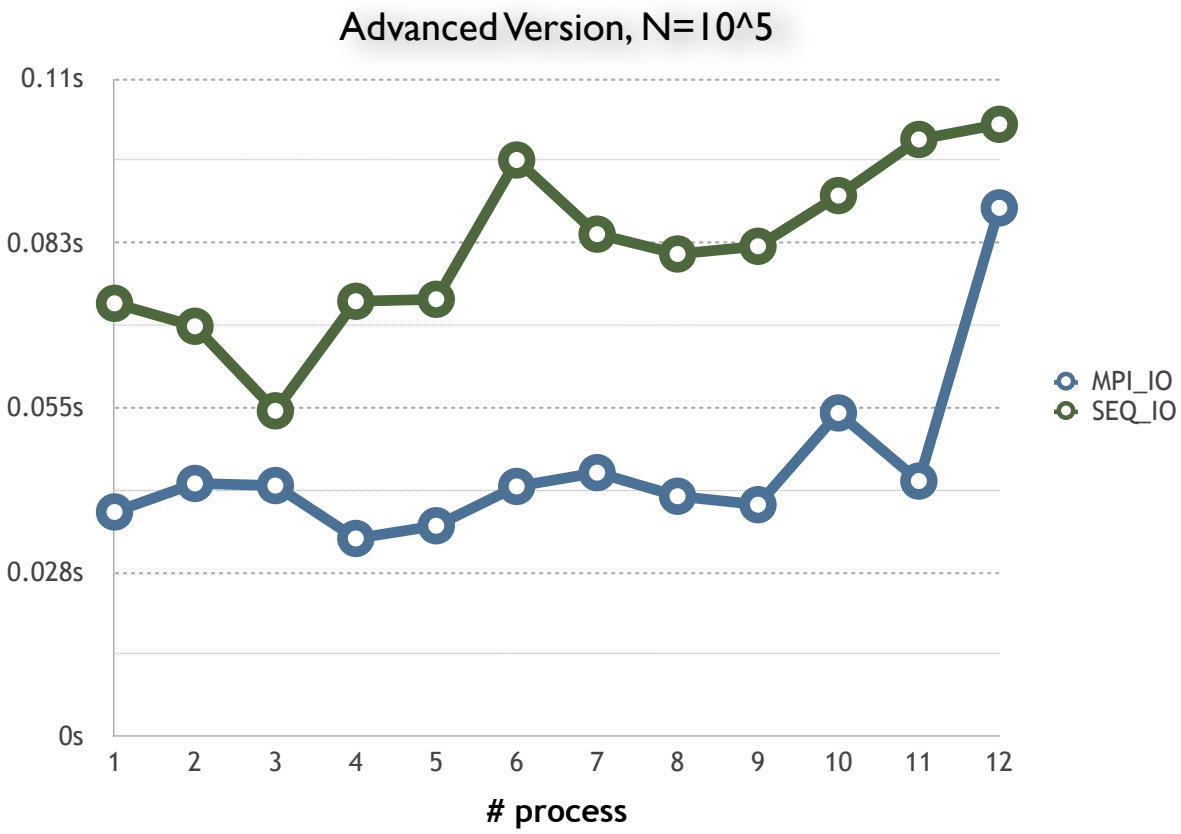
  - Advanced



- Part 3: Speedup Factor



- Part 4: Performance of different I/O ways



• Part 5: Compare two implementations

According to the above figures, it's apparently that the **advanced version is much better than the basic version**. Since the basic version need almost 1400 seconds, advanced version only takes less than 1 second. But these gaps between basic and advanced version are only obvious when the input size is a little bit large enough such like 1,000,000. And the time distribution between these two implementations is quite different. **Basic version** spends **much time** on both **communication** and **computation**, **especially computation part**. Nevertheless, the **advanced version**, compared to the basic version, does **spend specific part of the run time on I/O**. It **spends time on communication and computation almost equally**. I think the reason is that the **algorithm** used in advanced version is **merge-sort**, since its time complexity is (**n*logn**). And also, it can promise that **after the # process iteration time**, **all the elements will be sorted correctly**. So that's why the **time distribution of advanced version** can be **balanced** including **I/O**, **communication** and **computation**.

• Part 6: Experience / Conclusion

Hmm…… All I want to say is "REALLY TIRED……". But……It's very meaningful and interesting, I did really learn much from this homework!!! Thank you all :)