# Parallel Programming – Odd-Even Sort in MPI

101062337 Hung Jin, Lin

## Implementation

**Basic Version:**

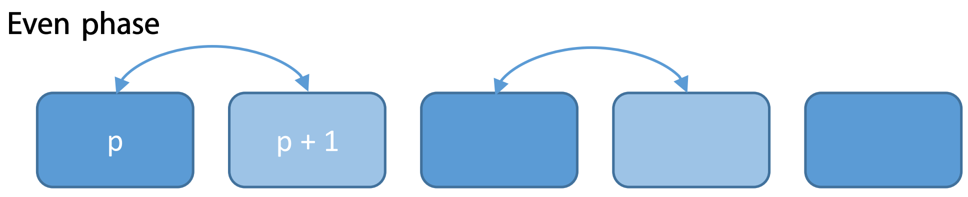
- Initialize the program environment by input arguments.
- Calculate the elements' number for each process.
  - Simply by dividing data size and process size.
  - Number for each = Data Size/Processes; if not divisible, then add one on it.
  - This will let the last process has the fewest numbers.
- Follow odd-even sort pattern,
  - Take odd-phase and even-phase one after the other.
  - Inside process, it act as what normal version do on single process.
  - The first/last element in process may not be the 'real' first/last number of entire sequence which means the element has something to do with next or previous process.
  - So, it need to communicate between processes when needed.
- Use MPI-IO for file operations.
  1. MPI-IO functions will make numbers 'distribute' into each process.
  2. With *MPI_File_set_view* to offset each process' file handler cursor.
  3. Use collective I/O for better performance.

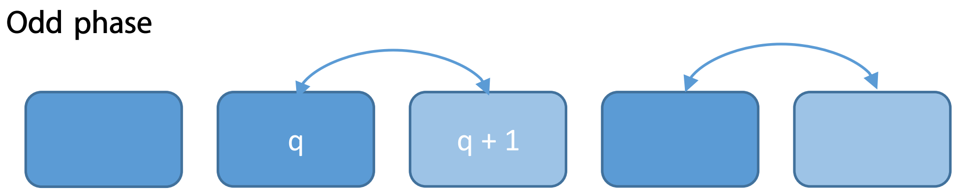
**Main algorithm:**

Inside odd-even phases, the method to decide which process should be a sender/receiver is important. And, there's a simple way by checking its 'global index' of original retire sequence, for example: in even-phase, if the last element of even process has even global index, then it should send its last one to next neighboring process; and the situation in next process is that the first element has odd global index and process itself has odd process-rank, so it will exchange its first one with previous process, finally, two processes are matched for the communication, and compare elements and swap if needed. By this way, each odd-even phase and each odd-even ranking process will has its own work to do and matches perfectly.

**<u>Advanced Version:</u>**

- Similar to the basic version, only some little changes.
- Each process gets its own number sequence, and use standard C *qsort()* for local sorting first which should be done once before entering odd-even phases.
- In odd-even phases loops,
  - Even phase will make even-ranked process **p** exchange entire list with its neighboring process **p + 1**.
  - One process will has two sequences, original one and another one from neighboring process. Merge them as what *merge* function do in merge-sort, and finally get a sorted sequence.
  - Take only needed part, for instance, the small-ranked process will remain smaller numbers in first half of merged sequence.
  - Odd phase will do the similar thing in even phase, odd-ranked process **q** and **q + 1** exchange their sequence and then remain the needed part.
  - qInside merge procedure, check that whether the entire sequence is sorted and decide when to stop.





## Experiment & Analysis

### System Spec

8 nodes, each has:

- Intel Xeon CPU L5640@2.27GHz (2x6 cores)
- 24 GB memory
- 2TB HDD Storage

## Strong Scalability & Time Distribution

**Timer function choice:**

I choose *clock_gettime(CLOCK_REALTIME)* to measure the elapsed time in IO, communication and computation. There are many functions to choose,

- MPI_Wtime()
- gettimeofday()
- clock_gettime(CLOCK_REALTIME)
- clock_gettime(CLOCK_MONOTONIC)

Though *MPI_Wtime* is a built-in function in MPI programming and it return a *double* type of time in second, there may be precision problem in floating point calculations. Finally, I choose standard GNU C functions *clock_gettime*, it stores all its value in separate integer counters, and it provides nanosecond resolution for higher precision than *gettimeofday*.

**How to measure:**

Wrap the concerned part of code with two timer record, and calculate the elapsed time immediately and add it to the according counter (e.g. io_time counter, comp_counter, comm_counter).

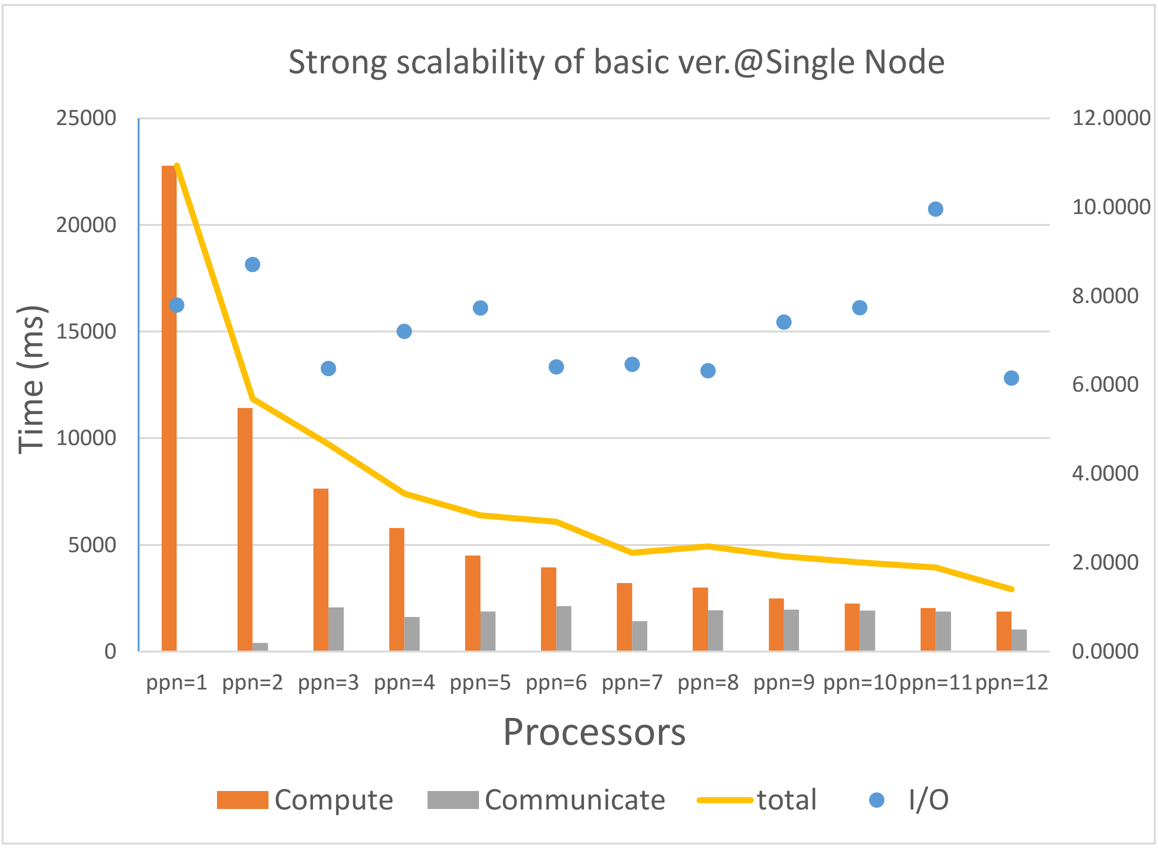
Sample code:

```
struct timespec s, e;
clock_gettime(CLOCK_REALTIME, &s);
// Some computation here...
clock_gettime(CLOCK_REALTIME, &e);
calc_time(&comp_time, e, s);
```

Before the program finish, make each process print out each timing counts' value, and write another program to parse and analysis the timing log. In parsing program, I make it generate a report of many log files from different submit result and its result will be the overall average time. Below are the results, and the time unit in graph are all microsecond (ms).

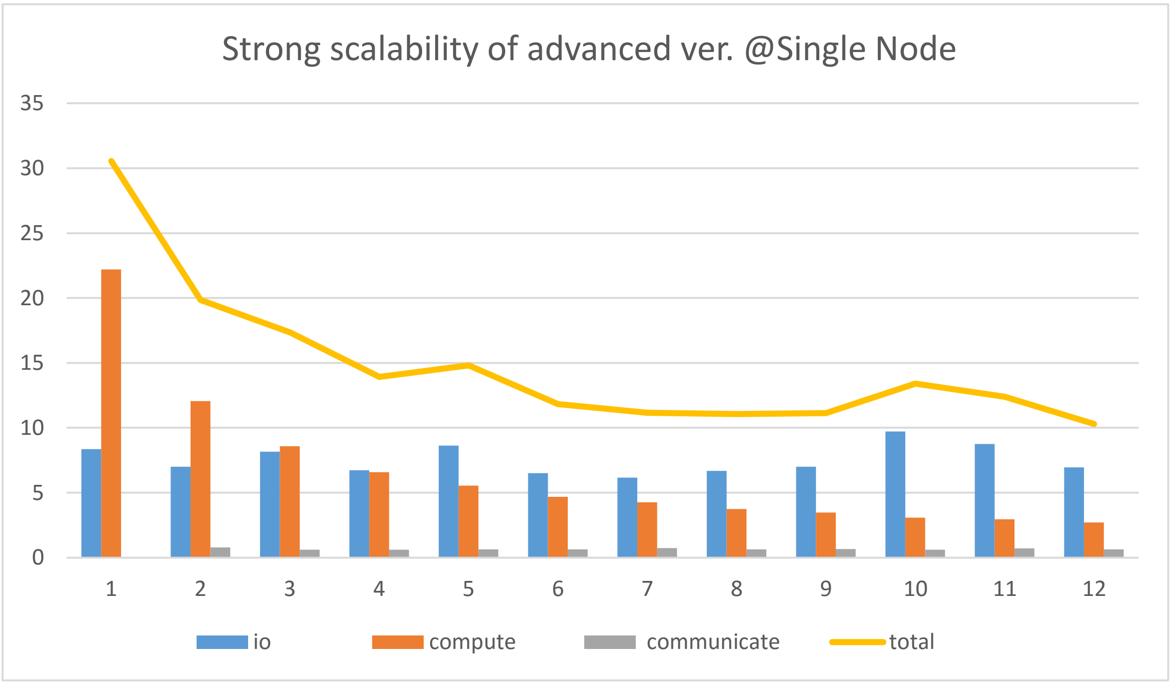
- Basic version **@Single Node**, data size = 100,000

Use single node, and test for process number from 1 to 12.



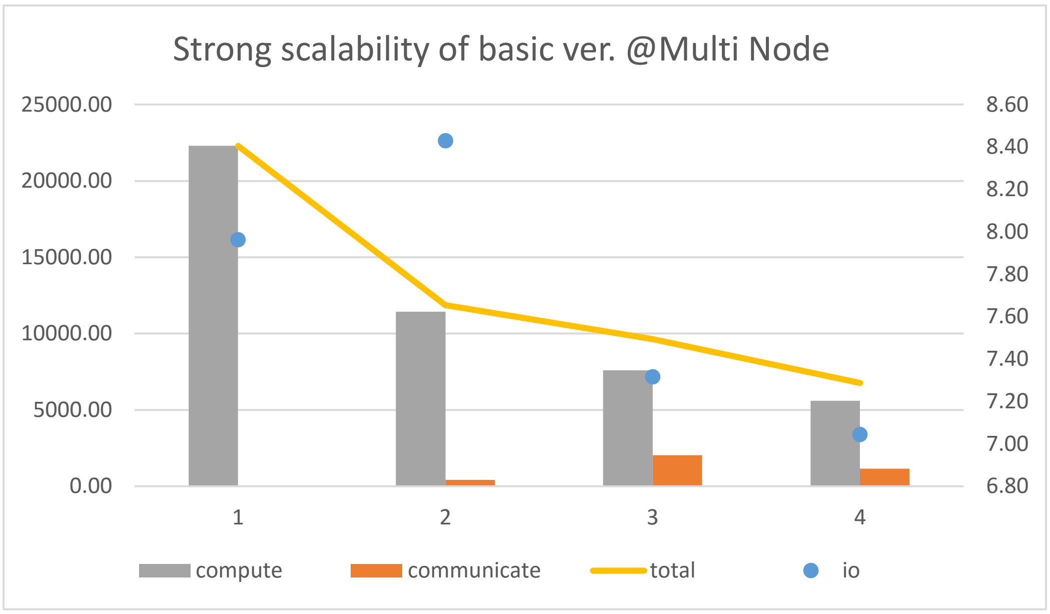
- Advanced version **@Single Node**, data size = 100,000

Use single node, and test for process number from 1 to 12.

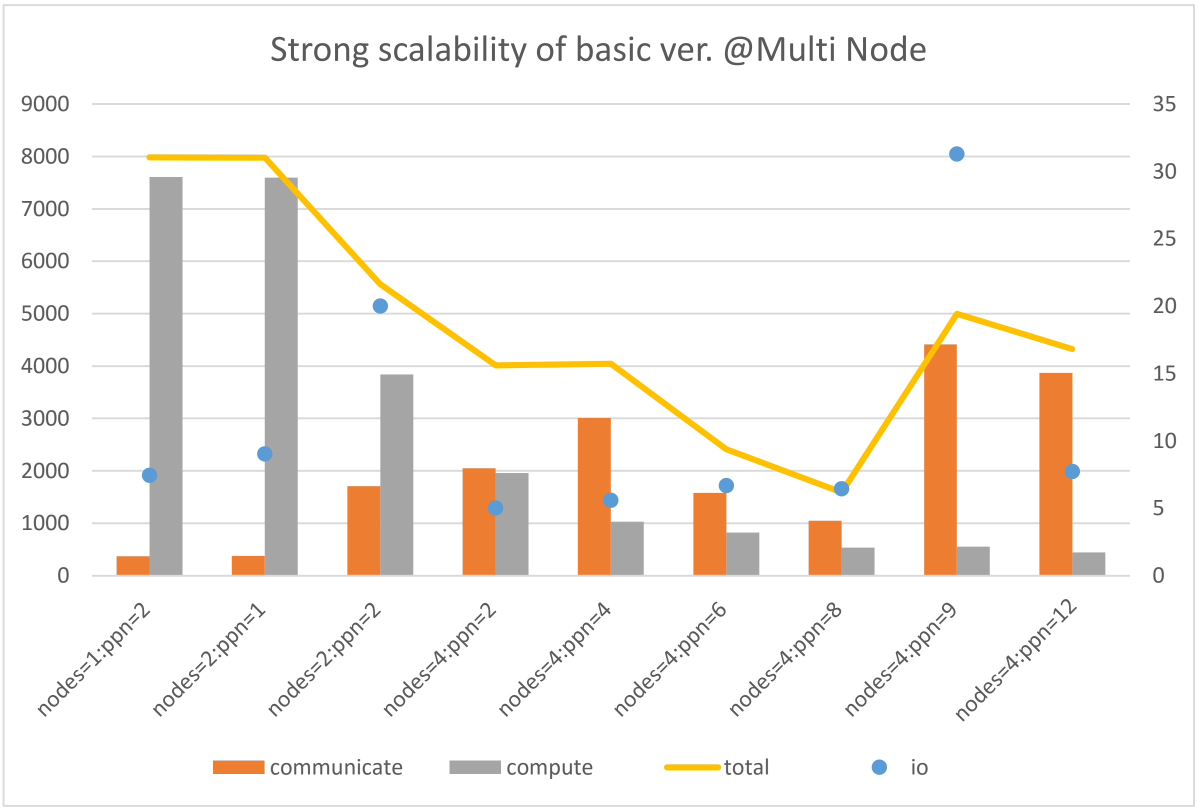


Basic version **@Multi Node**, data size = 100,000

- Use only one process in every nodes (there are 4 nodes on testing machine).

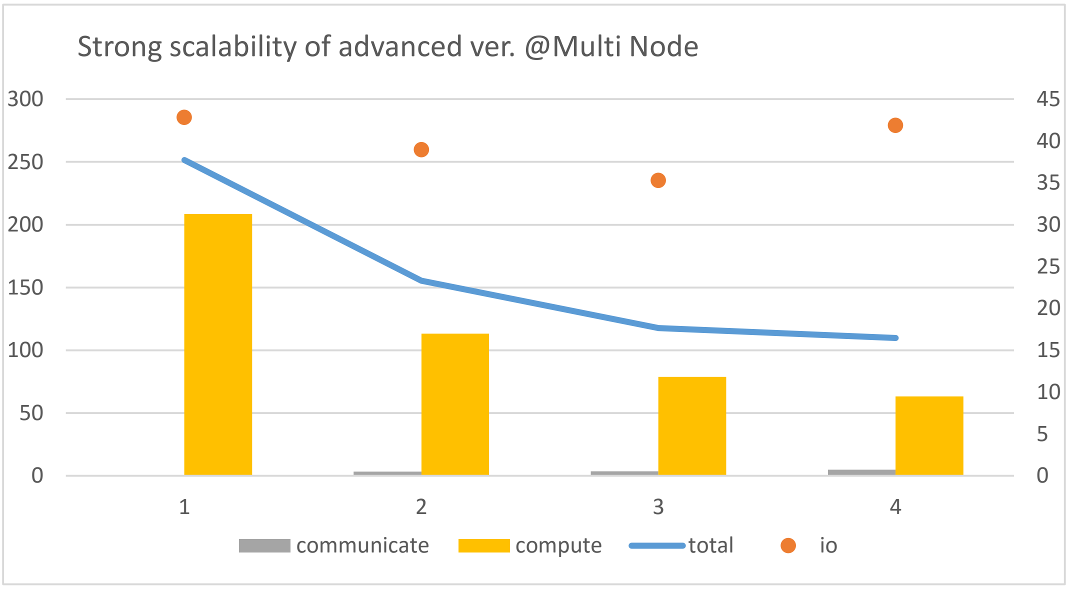


- Use multi nodes and multi process per nodes.

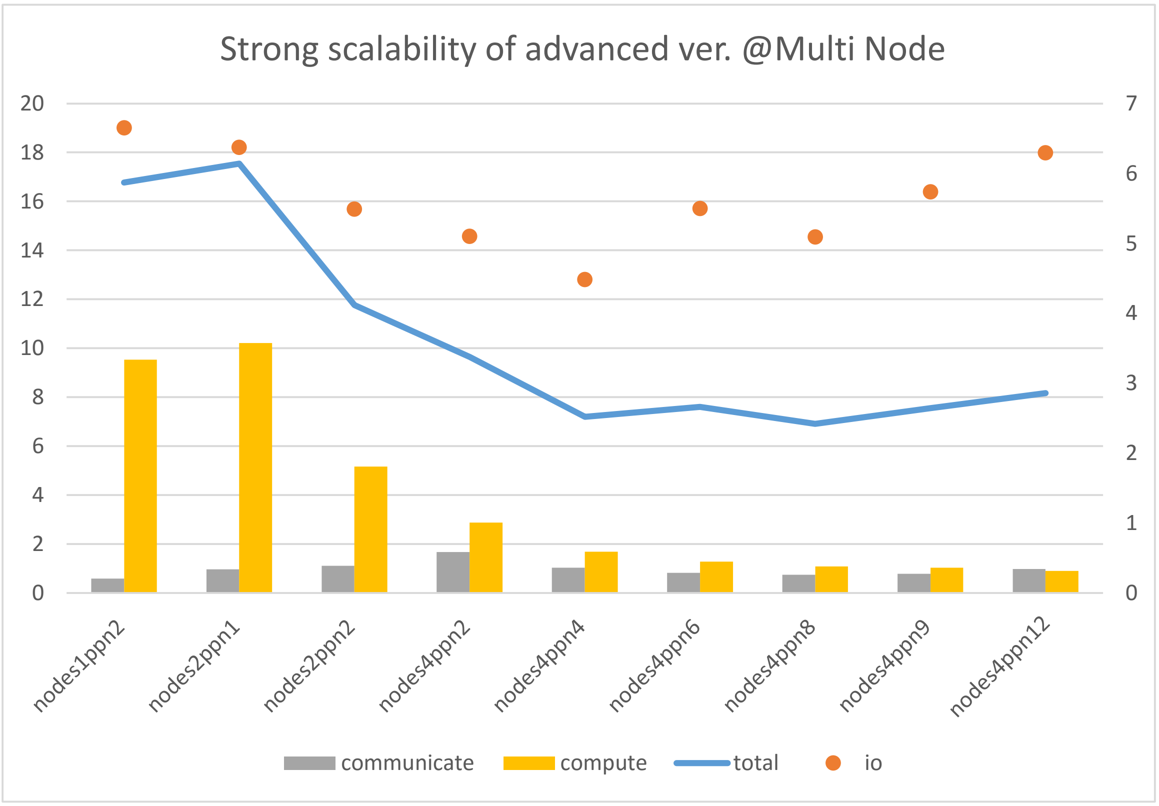


● Advanced version **@Multi Node**

◆ Use only one process on each node, data size = 100,000.



◆ Use multi processes on multi nodes, data size=100,000.



## Comparison

In both basic and advanced version, there are apparent reduces in multi-process environment, but the curves in two versions result from different reasons:

- The one in basic is that there are a **great amount of compare and swap actions in each odd-even phase' for-loop procedure** inside each process, and such a simple work is easy to distribute the work into many other processes. So, it has a nearly exponential curve's decay in graph which can be seemed as:
  *(numbers in each process) x (number of process) = sequence size, and it make a x*y = N curve.*
- Inside advanced one, the main computations are in two parts, first one is in sorting the subsequence numbers in each process, which is related to the size and its time complexity is O(nlogn); another part is in **merge procedure and it's also an linear complexity procedure**, so they will both have improvements in multiprocessing.
- But from algorithm of **advanced version, it exchange a list of sorted elements which means it may ends much earlier than basic one**. So, there will be more loops in basic one and will magnify the speedup effect in the basic version; while advanced one may terminate in at most the number of process times.

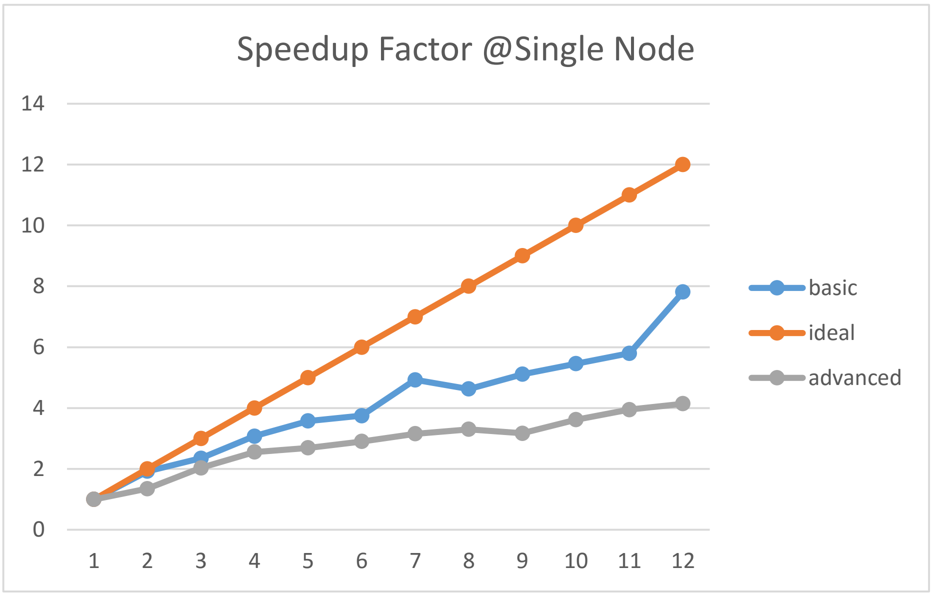
Another important point is that, we can observe in both cases that the communication **inside same nodes performs much better than it does between nodes**. Compare to two implementations, exchange times between processes will be the most significant variable that effect the time efficiency.

**More processes for basic version don't mean better communication time consuming, it has an up-going curve and then down which be shown in basic version graph (page four and five).** I think it has something to do with the random group particle distributions and the entropy. While in advanced, exchanging sorted list is an achievement to speed up the whole 'movement' procedure, but the cost is the operations of memory to exchange big size data.

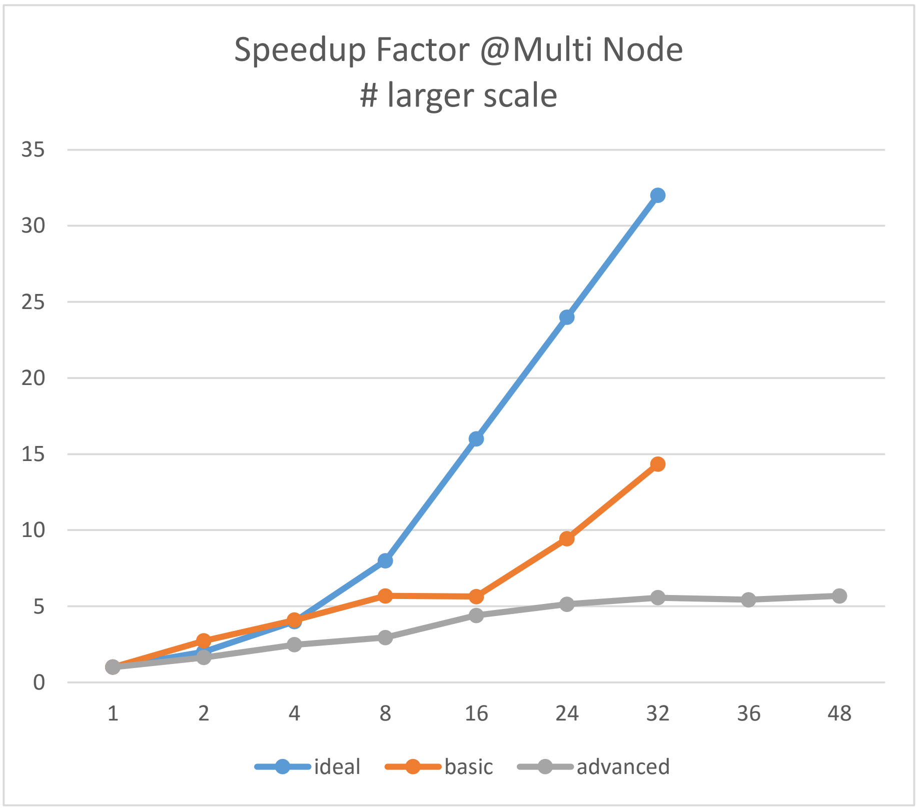
So, we can conclude that IO part thought has some variant in each test, its change is small enough. The main factors are communications and computations, and **communications doesn't match the easy intuition rule: more processes make better performance**. We can conclude that from *Strong scalability of basic ver. @Multi Node* in page five.

## Speedup Factor

- Speedup factor in single node, data size=1,000,000

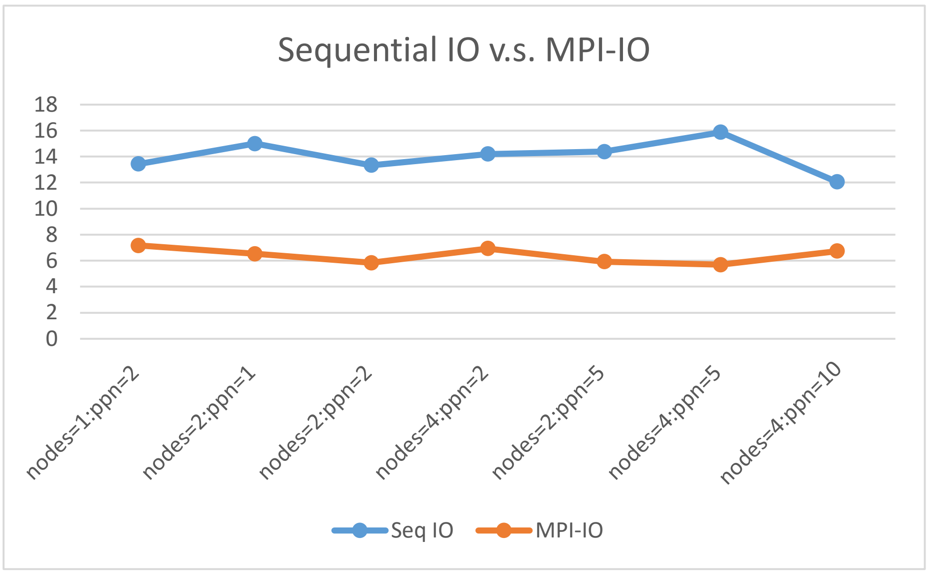


- Speedup factor in multi node, data size=1,000,000



## Performance of different I/O ways

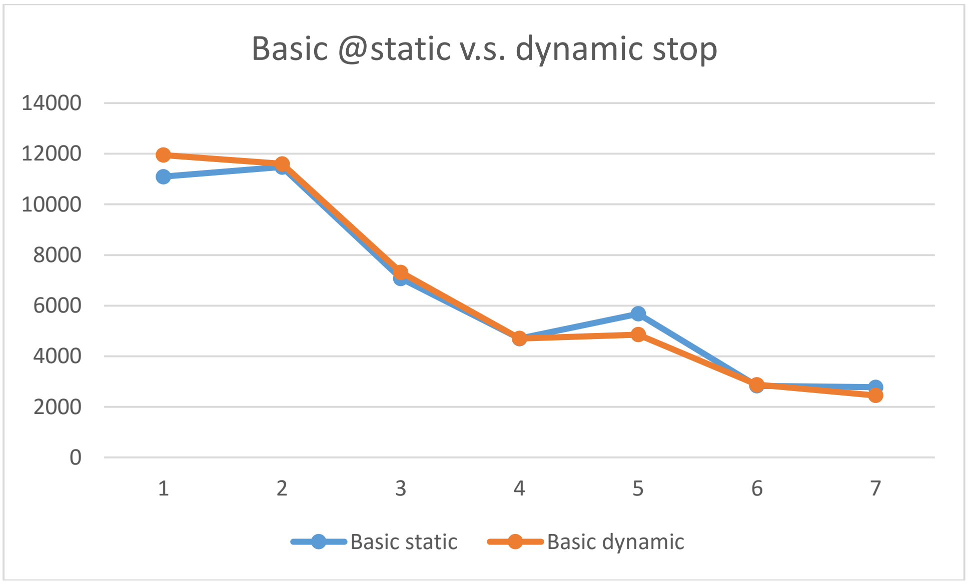
Data size = 100,000

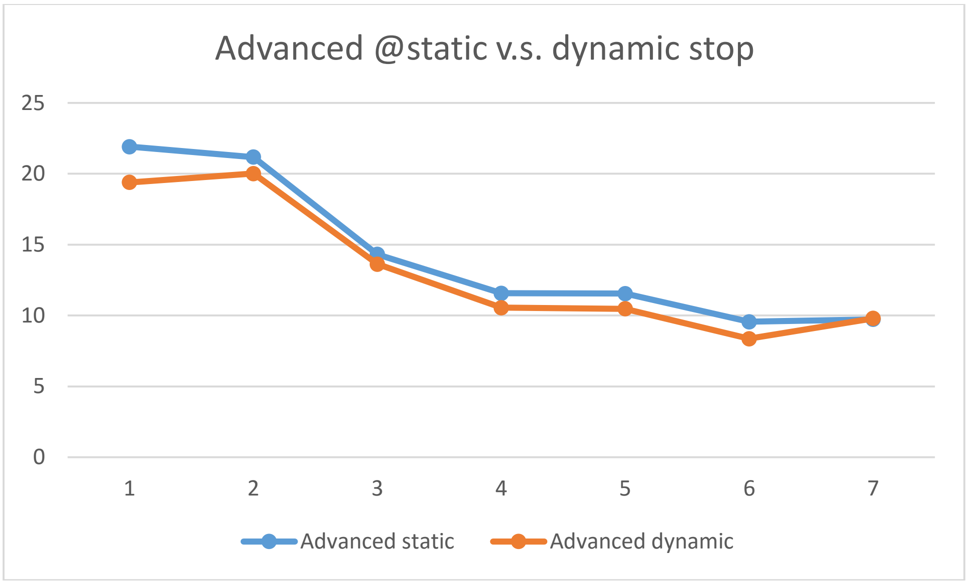


Performance between sequential and MPI IO are not clear enough to claim which one is much better. Maybe the testing machines system doesn't have well-built distributed file system, so there's no obvious speed up by MPI-IO API.

## Static and Dynamic Stop

Data size = 100,000





- Basic version can stop by dynamic detect with help of some flags inside the odd-even sort procedure, or it will be bounded by sequence size **N**.
- Advanced version can also stop by flags in merge and compare procedure, or it can stop by the process number size **p**.

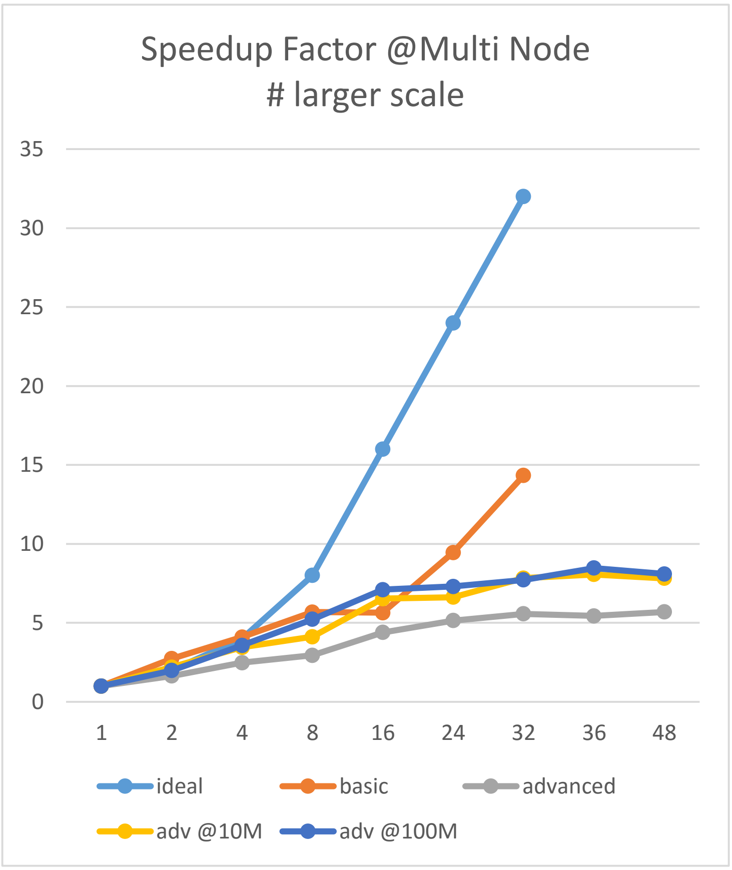
We can observe that, the dynamic versions both have better performance than static stop one. In the graph, the data size is 10,000.

## More in Speedup

In previous speedup section, the advanced one doesn't perform well in small data size. So, here add two lines to the chart.

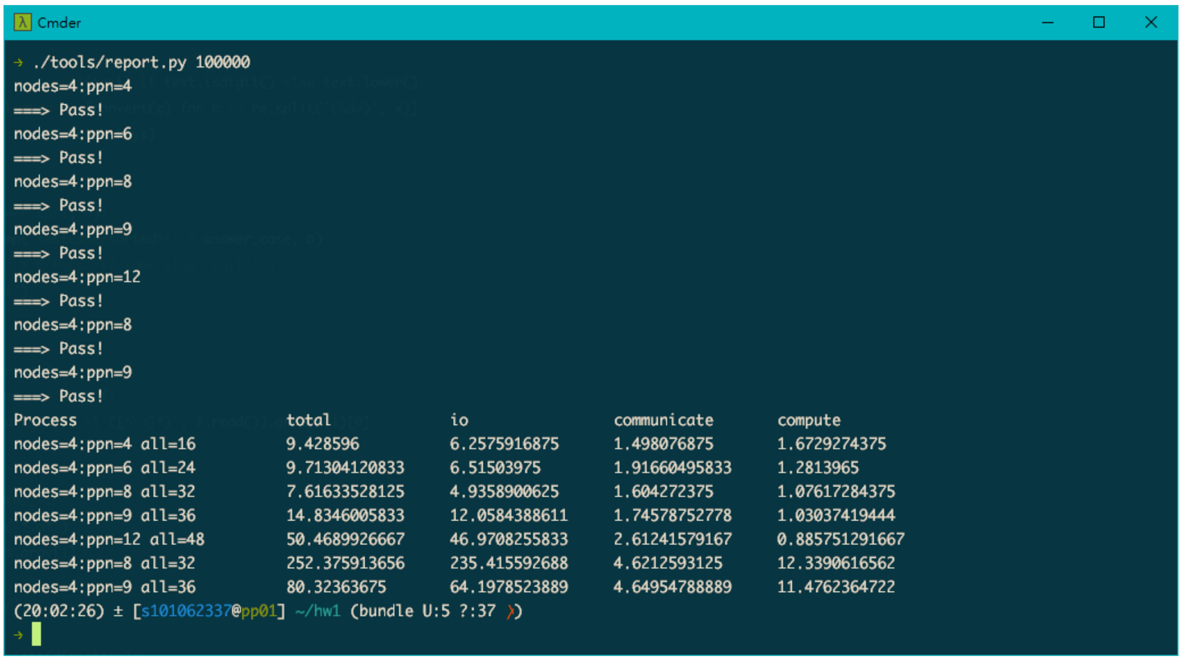
- Advanced with 10M size (10,000,000)
- Advanced with 100M size (100,000,000)

We can now see that, the larger the data size is, the better the speedup factor will be shown. The deep blue advanced one (adv@100M) got the best result compare to other advanced lines.



## Experience / Conclusion

During programming, when I go search for the MPI usages from documents and find that the document doesn't explain each arguments' details, so I face a hard time when using these functions at the beginning. Another difficulty is that when do experiment with my program, the procedure of making each testing profile is too complicated, so I build up my own testing tool by shell script, Makefile and some python code.



Cmder

```
→ ./tools/report.py 100000
nodes=4:ppn=4
===> Pass!
nodes=4:ppn=6
===> Pass!
nodes=4:ppn=8
===> Pass!
nodes=4:ppn=9
===> Pass!
nodes=4:ppn=12
===> Pass!
nodes=4:ppn=8
===> Pass!
nodes=4:ppn=9
===> Pass!
Process                 total           io              communicate     compute
nodes=4:ppn=4 all=16    9.428596        6.2575916875    1.498076875     1.6729274375
nodes=4:ppn=6 all=24    9.71304120833   6.51503975      1.91660495833   1.2813965
nodes=4:ppn=8 all=32    7.61633528125   4.9358900625    1.604272375     1.07617284375
nodes=4:ppn=9 all=36    14.8346005833   12.0584388611   1.74578752778   1.03037419444
nodes=4:ppn=12 all=48   50.4689926667   46.9708255833   2.61241579167   0.885751291667
nodes=4:ppn=8 all=32    252.375913656   235.415592688   4.6212593125    12.3390616562
nodes=4:ppn=9 all=36    80.32363675     64.1978523889   4.64954788889   11.4762364722
(20:02:26) ± [s101062337@pp01] ~/hw1 (bundle U:5 ?:37 ✗)
→
```

And this is my first time to program a code and make such an accurate time profiling and apply so many analysis on and finally generate the graph to illustrate the effort of parallel program. It's very exciting to see the program run and do speed up on multi-process.

In this project, I have learned that the program is essential part but the experiment is another vital part in which you can fully prove the work you make is good enough in parallel computing.