Parallel Programming HW1: Odd-Even Sort

9961244 EE14 吳德霖

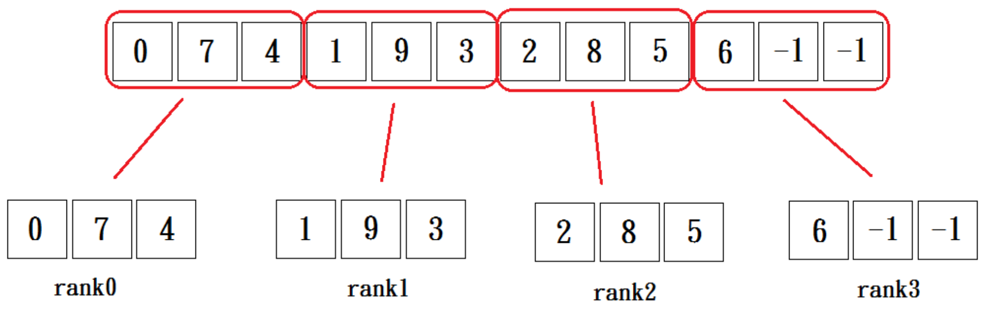
Compile: mpicc odd_even_bsc.c –o odd_even_bsc
Run: mpirun ./odd_even_bsc test.txt output_bsc.txt time.txt

◆ Implementation and Design

Basic version:

The algorithm of my basic version’s implementation will be detail explain with the example shown below. Here, we take 10 numbers accompanying with 4 processors (4 nodes, 1 processor each) as the exhibition test case.

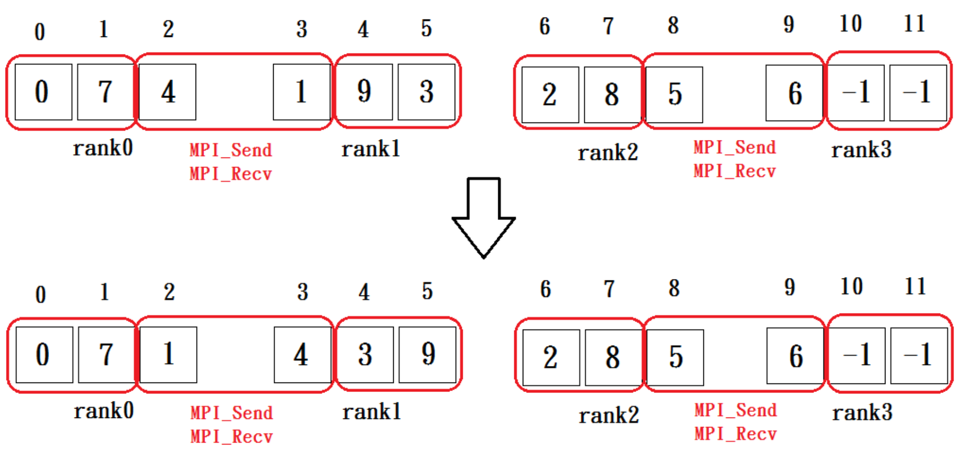
1. Scatter



First, detecting the numbers % available processors would give out how many -1 should be added to make the total number of the numbers to be sorted divisible by the number of processors. In this case, 10 % 4 = 2, and thus (10 -2) / 4 = 2, therefore I conclude that each processor should contain 3 numbers, which makes additional -1 needed be 4 * 3 – 10 = 2, as indicated at the last of the pre-sorted queue.

2. Odd-even sort, even phases

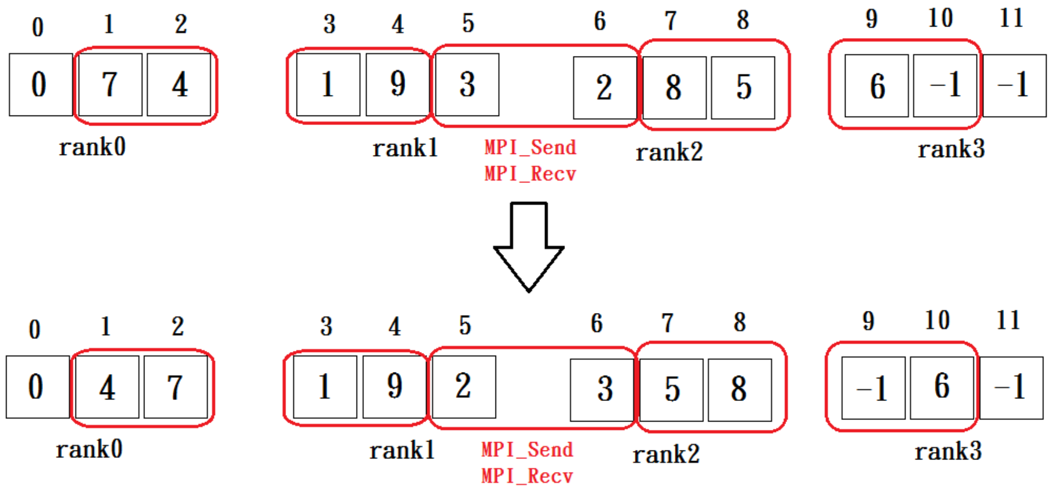
After the preparation is done in the previous step, we can start conducting conventional odd-even sort now.



As shown above in the graph, the small numbers above the queue is the order, or the ID of each number, indicating whether the number is of ”even order” or “odd order”. We can obviously see that order 2 and order 3 are not in the same rank of processor, leading to the requirement of sending and receiving API provided by the MPI library. The upper part of the graph thus will be interchanged into the lower part, since it’s in the even phase; orders that are even numbers dominate this step.

3. Odd-even sort, odd phases

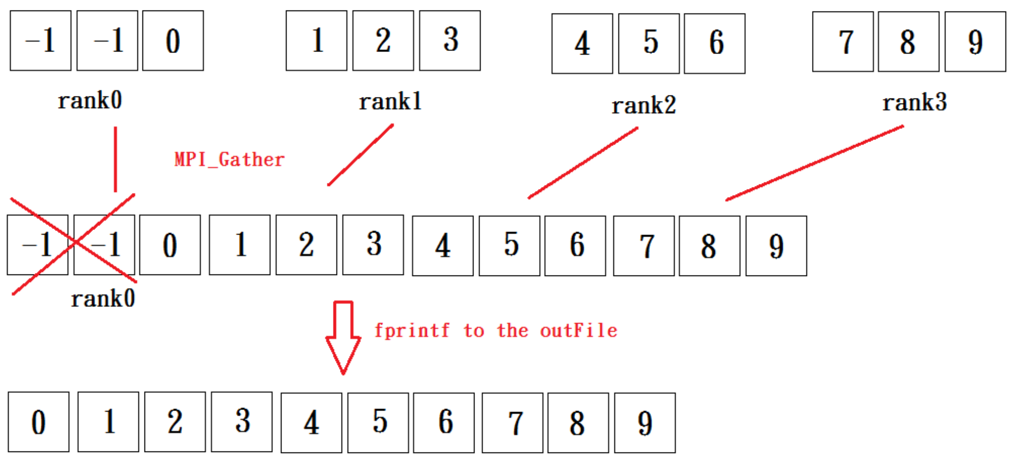
The following graph displays the odd phases, where the place requiring sending and receiving changed, so that we need another case of function code to deal with such situation.



Obviously, the orders labeled with odd numbers dominate this phase.

4. Gather

After at most 3 * 4 times of recursively odd-even sorting, the final queue would be in a sorted increasing order, with several numbers of -1 leading the entire array. Now, we can simply apply MPI_Gather to gather the sorted array to processor ranking 0, and eliminate those -1’s while storing into the output file.



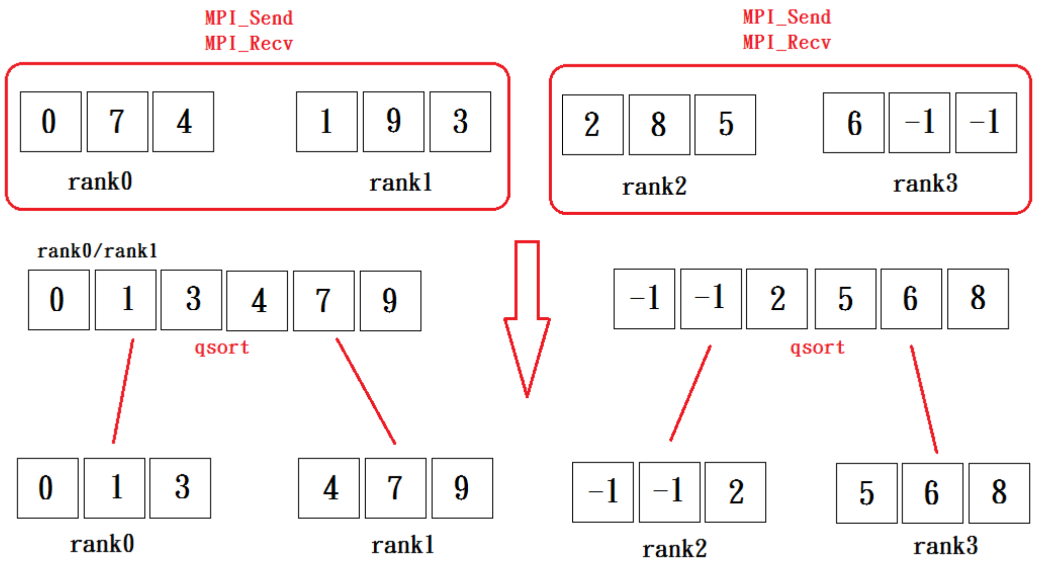
Advance Version:

The algorithm here is quite different comparing to the basic version since the restriction now is that the communicators can only interact with their neighbors, thus a "communicator-based" odd-even sort. We take the same example to demonstrate the idea of this version.

1. Scatter

The detail is the very same thing done in the basic version, so I will skip this part.

2. Odd-even sort, even phases



As shown above, in the even phase, processors ranking with even numbers dominate the situation, all numbers in rank 0 and rank 2 have been sent to rank 1 and rank 3 respectively. After a quick sort, the intermediate sorted arrays stored in all processors will be redistributed to their corresponding processors. That is, the former half part will be stored back to the even ranking processors, while the latter half part is stored in the odd ranking processors.

3. Odd-even sort, odd phases

Since the restriction is simple and only associate with communicators, we can view the whole process as the odd-even sort of ranks, thus no specific case is required here, simply conduct the original algorithm will do.

4. Gather

This step is just the same as what we've done in the basic version.

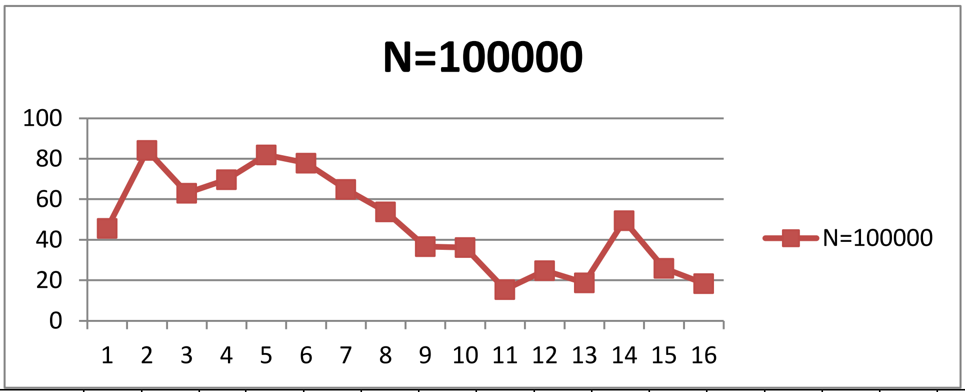
Note: In additional to the required input and output file parameters, I added an extra output file to store the information of the running time classified with some environmental configurations. (For demo, these lines are commented out)

◆ Experiment & Analysis:

1. Scalability

To conduct the experiment, I simply add a library <time.h> to take advantage of its clock() function to set up the start and end points, determining the difference between those two check points to obtain the running time for processing.

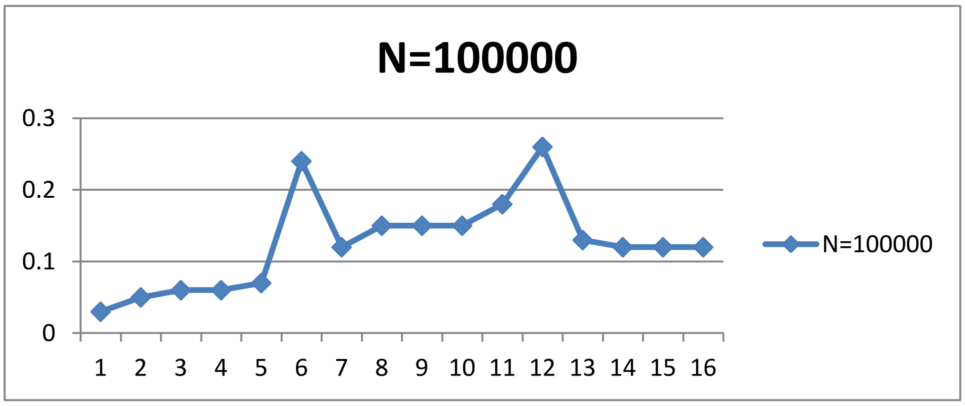
✧ Basic



Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time(s)	45.6	84.1	63	69.7	81.9	77.8	64.9	53.8	36.6	36.2	15.4	24.8	18.8	49.5	25.9	18.4

Without considering the result of one and fourteen processors, this graph shows a reasonable scalability, that is, as the number of processors involved increases, the total required time is shrinked. As for the first exception, I suggest that because the scale of the numbers aren't large enough, so maybe it's sufficient to use just one processor, and some time would be wasted due to communications when the numbers of processors aren't ample to surpass such issue. For the second exception, I guess it's caused by some interference and complexity to deal with the extended array containing those -1 remainders (100000 % 14 != 0).

✧ Advance



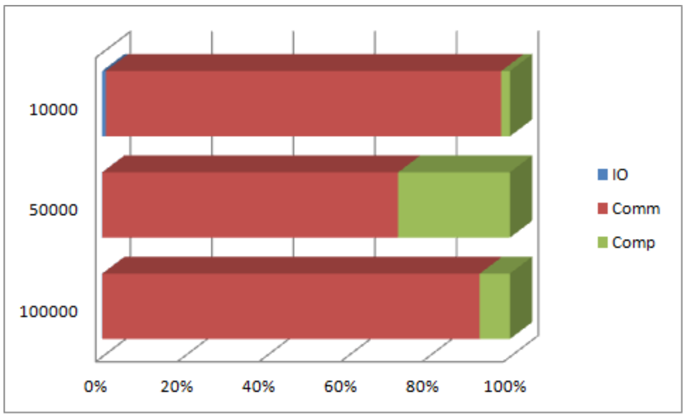
Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time(s)	0.03	0.05	0.06	0.06	0.07	0.24	0.12	0.15	0.15	0.15	0.18	0.26	0.13	0.12	0.12	0.12

The result at first look might be quite weird, since it's not reasonable to have the fastest times when the processors used are less. But if we take the algorithm of this version into consideration, we can figure out the reason is relatively simple. The sorting algorithm I used when the values are sent from one rank to another is quick sort, which makes the sorting time surpass the communication time significantly. To clarify, there's no need for consuming communication time when using only one processor, while it's needed when the processors increases. Eliminating the result of 12 ranks, beginning from 6 or 7, the graph again shows reasonable scalability, I guess here the computing time is more compatible with the communication time. The reason for the exception is considered the same as previously discussed in the basic version.

2. Timing intervals analysis

It's pretty much the same methods what I applied in the previous section, despite the fact that the check points here is relatively more complicated and requiring more. I defined the communication time if the process involves MPI's sending, receiving and gathering APIs. Other local sorting are all related to the computing period, while the IO only determine the input and output formatting or importing/exporting.

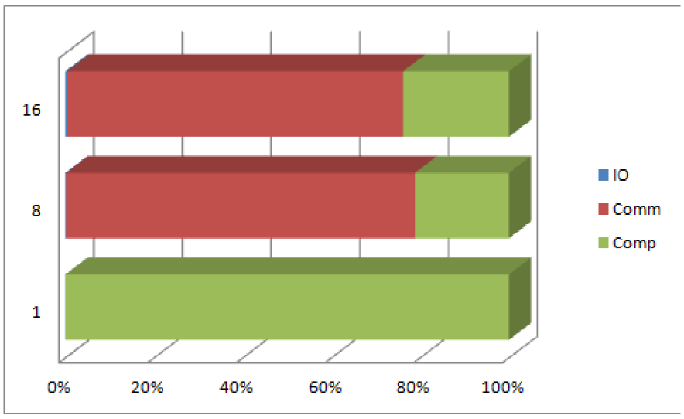
✧ Basic (Problem size, tasks = 8)



N	100000	50000	10000
IO	0.04s / 0.04%	0.04s / 00.04%	0.03s / 1.31%
Comm	85.38s / 92.6%	4.24s / 72.09%	2.175s / 96.0%
Comp	6.81s / 7.38%	1.60s / 27.23%	0.07s / 3.08%

We can obviously see that the communication time dominates the whole processing time throughout three cases. As the problem size increases, the required communication time increases correspondingly due to more requirement of timing consumption among tasks to deal with tougher problems.

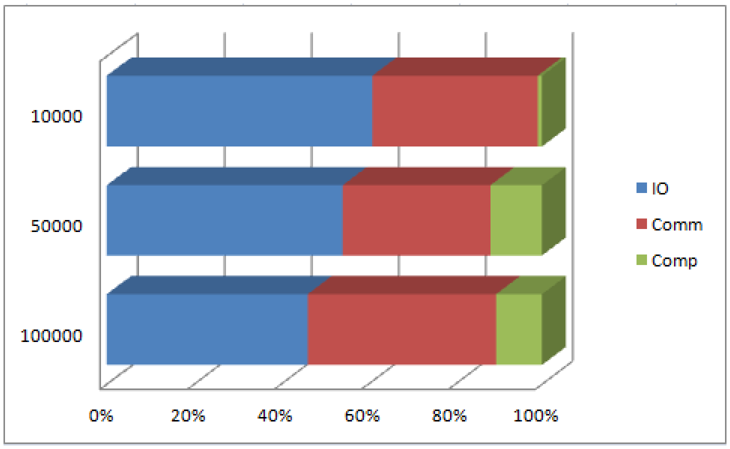
✧ Basic (Numbers of tasks, N = 100000)



Tasks	1	8	16
IO	0.07%	0.17%	0.41%
Comm	0.00%	78.69%	75.77%
Comp	99.93%	21.15%	23.82%

This result is reasonable and predictable, since as the processors increases, although the communications might seem to be made more complicated, but the total require communication times is reduced due to its capability of dealing with tougher problems.

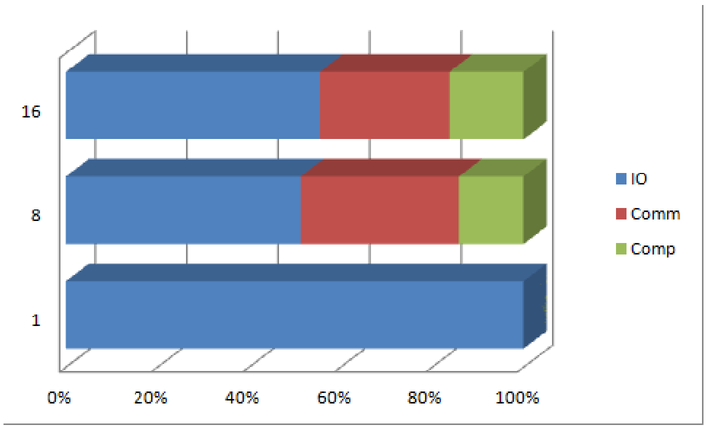
✧ Advance (Problem size, tasks = 8)



N	100000	50000	10000
IO	0.06s / 46.15%	0.04s / 54.24%	0.02s / 61.53%
Comm	0.056s / 43.27%	0.05s / 33.90%	0.013s / 38.46%
Comp	0.014s / 10.57%	0.009s / 11.86%	~ 0s / 1.000%

In this case, since the requiring time for computing is relatively small, making the communication time and IO time more dominating. So, here the total time consumption relies on exactly how many values are being sent or received, as the problem size increases.

✧ Advance (Numbers of tasks, N = 100000)



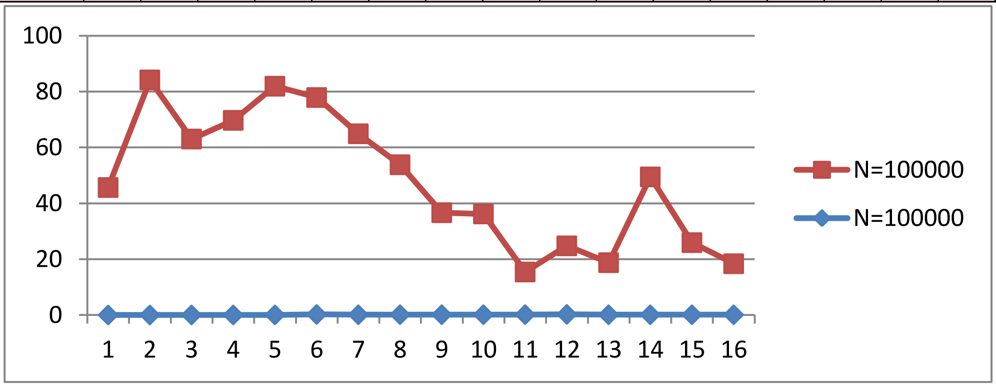
欄 1	1	8	16
IO	100.00%	51.28%	55.49%
Comm	0.00%	34.62%	28.32%
Comp	0.00%	14.12%	16.18%

The similar result that we've seen in the basic version, and the very same reason to explain it. The only difference is the power of computing here is much powerful than that of basic version, so the IO consumes significantly percentage of the total amount of time.

3. Performance measurement

✧ Basic / Advance

Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time(s)	45.6	84.1	63	69.7	81.9	77.8	64.9	53.8	36.6	36.2	15.4	24.8	18.8	49.5	25.9	18.4
Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time(s)	0.03	0.05	0.06	0.06	0.07	0.24	0.12	0.15	0.15	0.15	0.18	0.26	0.13	0.12	0.12	0.12



From the combining graph with the same problem size and correspondence of tasks used, we can easily conclude that the advance version surpass the basic version significantly and consumes relatively less than 1 second for sorting. The reason is very simple, since the only restriction for advance is on the tasks, the sorting method isn't limited. And thus quick sort can be applied to reduce the time required considerably.

Although the communication array is enlarged, but due to its capability of sorting much larger array in each task, advance version require considerably less communications even the problem is complicated, resulting in its better performance.

✧ **Max array in 5 minutes (Advance)**

N = 340000000, and reversed order to ensure this is the critical case.

Processors: 8; **Running time: 169.37s = 2min 49.37s** (I suppose the 5E array can also be sustained in 5 minutes, but due to the long queue, I didn't have enough time to conduct the experiment)

4. Additional analysis or optimization

● **IO formatting for basic version optimization**

Since the algorithm I applied required additional -1 to aid the distribution of the problems into tasks, my original method is to make those compensation -1s at the last of the array, but then I figured out if I put them in the very beginning of the array, it might shortened the required running time due to the simplicity of the order. The following are the experimental results: (N = 100000)

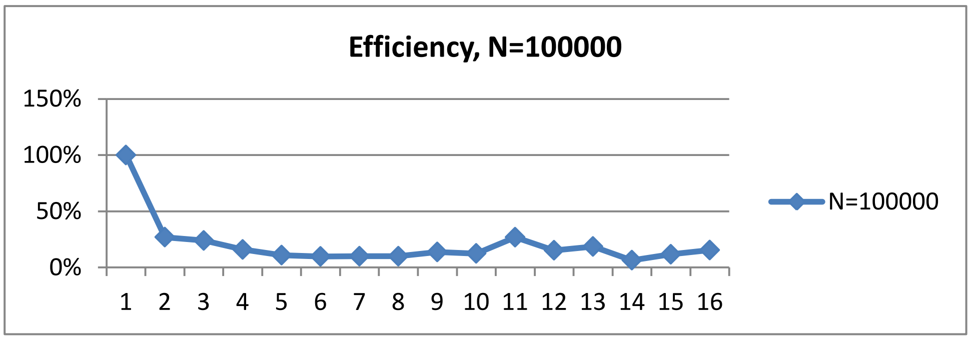
Test/Time	1	2	3
-1 in front	13.82s	13.5s	13.96s
-1 at last	13.91s	13.82s	39.67s

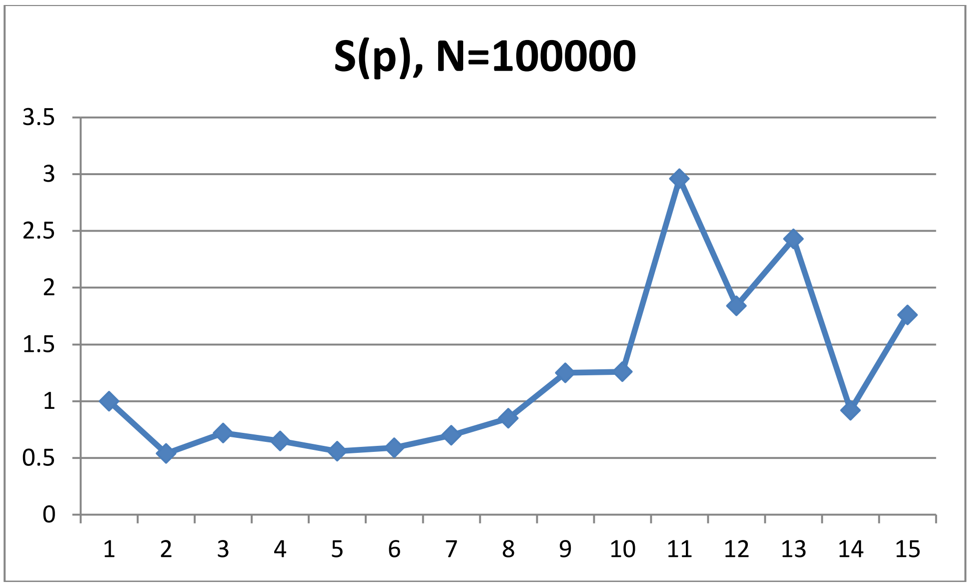
As a result, I can conclude that this optimization really did work, but not considerably empower the efficiency, since if we put -1 in front, we need an additional process for properly formatting, which consumes some amount of IO time, though very little.

● **Performance efficiency**

Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(p)	1.0	0.54	0.72	0.65	0.56	0.59	0.70	0.85	1.25	1.26	2.96	1.84	2.43	0.92	1.76	2.48

The best performance would be using the quick sort, so here we do not consider this case. Take the sequential best record as the reference, and best performance which occurred when there were 11 processors, the system efficiency here is 2.96 / 11 (100%) = 27%. Plotting the efficiency:





(The plot of speed up factor S(p))

On the other hand, it's not reasonable to analyze the advance case since the quick sort is applied. As a result, if the available tasks aren't large enough to surpass the quick sorting time in order to scale down the problem size; it would be not applicable to discuss the efficiency.

5. Source codes

Please refer to the attached files and the instructions of how to execute the executable file at the very beginning of this report, code and executable files labeled with _bsc is the basic version, while labeled with _adv is the advance version.