Parallel Programming HW1 Report

Name: 周聖諺
ID: 106033233

1. **Implementation**

(1) How do you handle an arbitrary number of input items and processes?

Assume that there are n processes, k arbitrary number and, k % n = r. I divide the arbitrary number into n segments. The residual numbers will be distributed into the first r process.

(2) How do you sort in your program?

I use the Spread Sort of Boost library for local sorting. The Spread Sort is a novel hybrid radix sort algorithm. The time complexity given by the Boost official documentation is min(N logN, N key_length)(Note: N is the length of the sequence to be sorted).

As for Odd-Even Sort, I use Baudet-Stevenson odd-even sort. The Baudet-Stevenson sort transfers the whole segment that the processes have to another one each time which reduces the transfer time extremely. It guarantees that it can finish within n phase for n parallel processes.

(3) Other efforts you've made in your program.

(a) Early Stop: The algorithm is designed to be stopped when there is no swapping between the processes. However, it needs an extra AllReduce operation to sync every process and check whether it can be stopped or not. Finally, since it increases the communication time extremely, I cancel this functionality after some experiments.

(b) Parallel Read/Write: For each process, just read/write the segment that it needs to handle.

(c) Non-Blocking Message Passing: I use Non-Blocking message pass operation to pass the segment to another. It means that for each process it would send the segment it has to the target process without waiting.

2. **Experiment & Analysis**

(1) Methodology

(a) System Spec

Only Appolo

CPU: Intel(R) Xeon(R) X5670 @ 2.93GHz 12 cores

Memory: 94.4G

(b) Performance Metrics

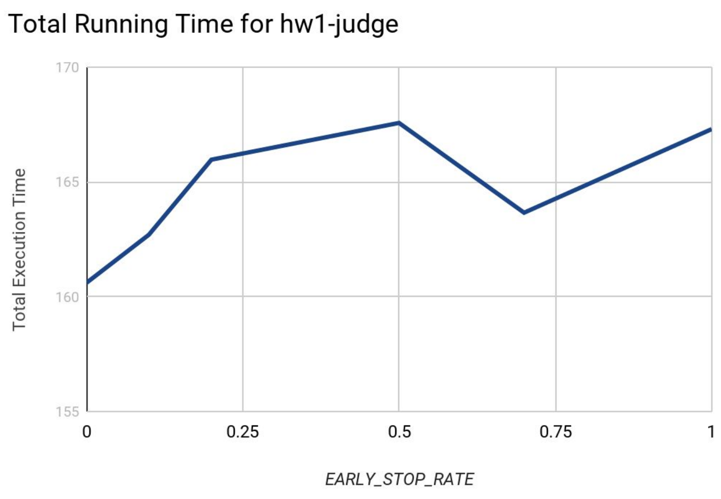
I write a new class Timer to record the clock time with clock(). Subtract the ending and starting time and divided by the constant CLOCL_PER_SEC. Then, we get the seconds of execution. However, it needs additional code like putting the start/pause record call at the front/back of the section that I want to measure. It would cause some extra time to execute the operation.

(2) Plots

(a) Early-Stop:

The following plot is the result of running time on hw1-judge versus different EARLY_STOP_RATE. The EARLY_STOP_RATE is that the program will do AllReduce to check whether it can stop or not in the last few phases(more accurate, MPI world communicae size * EARLY_STOP_RATE , i.e. EARLY_STOP_RATE=0, there is no early stop check and when =1, it checks every phase).

As the following plot, generally, the running time reaches minimum when EARLY_STOP_RATE is 0. As a result, I canceled this functionality finally.



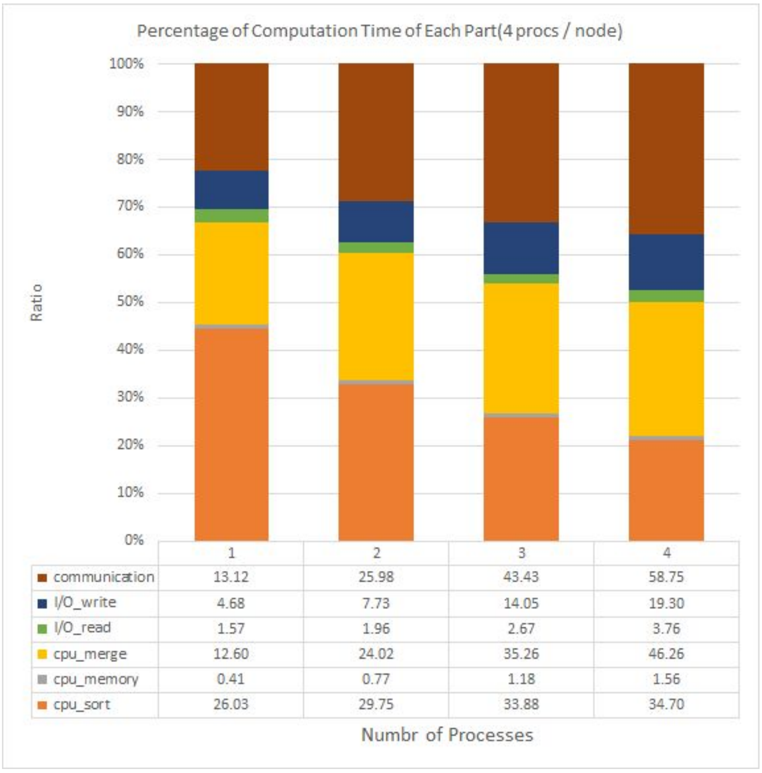
(b) Speedup Factor & Time Profile

For more sophisticated analysis, I divide the original 3 classes into several subclasses.

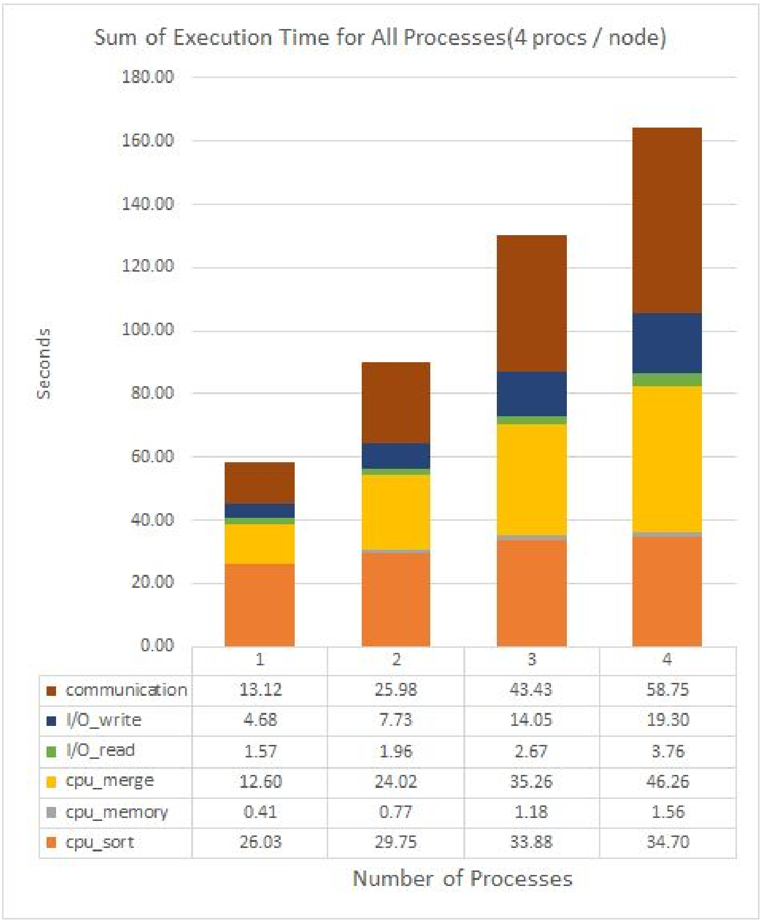
CPU = cpu_sort + cpu_merge + cpu_memory

I/O = I/O_read + I/O_write

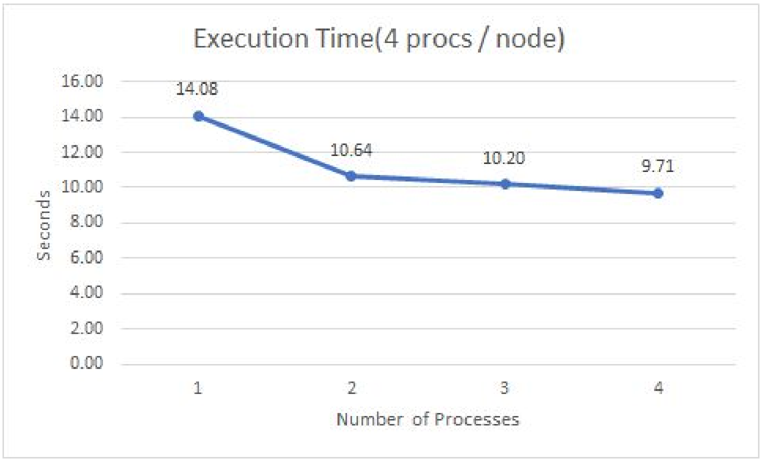
Communication remain the same

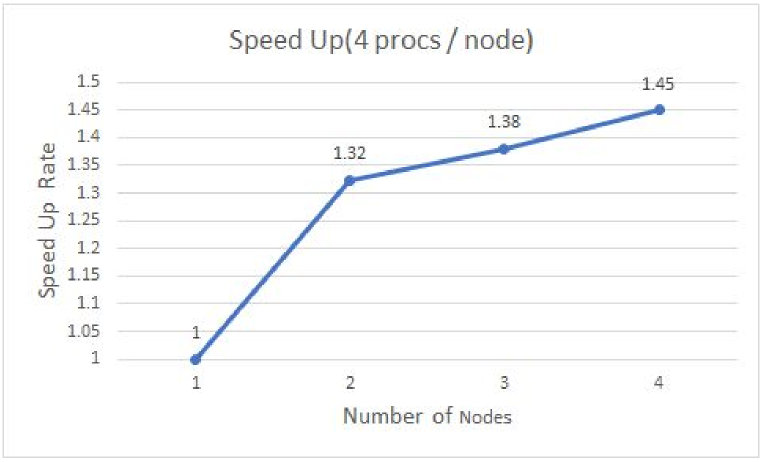


	1	2	3	4
communication	13.12	25.98	43.43	58.75
I/O_write	4.68	7.73	14.05	19.30
I/O_read	1.57	1.96	2.67	3.76
cpu_merge	12.60	24.02	35.26	46.26
cpu_memory	0.41	0.77	1.18	1.56
cpu_sort	26.03	29.75	33.88	34.70



	1	2	3	4
communication	13.12	25.98	43.43	58.75
I/O_write	4.68	7.73	14.05	19.30
I/O_read	1.57	1.96	2.67	3.76
cpu_merge	12.60	24.02	35.26	46.26
cpu_memory	0.41	0.77	1.18	1.56
cpu_sort	26.03	29.75	33.88	34.70





(3) Discussion

(a) Performance Comparison

For less processes, the local sorting is the bottleneck. However, the more processes, the communication cost more execution time. Thus, in a highly parallelized environment, communication is the bottleneck. In addition, because of parallelization, it needs more time to merge different segments.

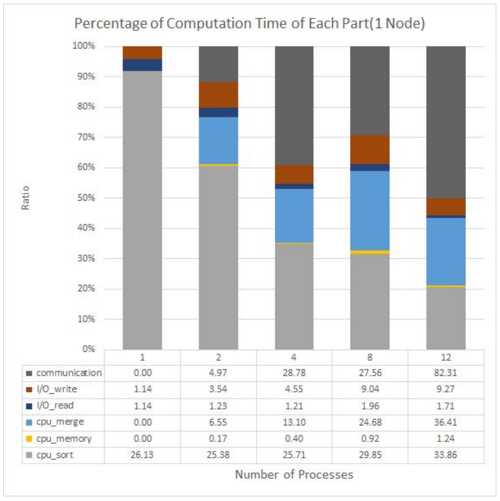
To improve the performance, we can sort the segments of different processes in the same node because the messages passing through local memory are faster than communicating through Ethernet cable. After the segments of the same node are sorted, it can be transferred to another node and merge. Through this, it can reduce communication between nodes massively.

(b) Scalability

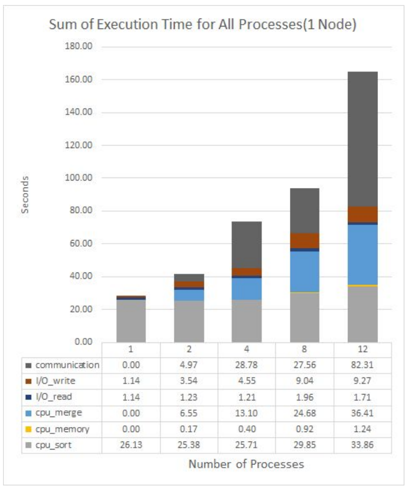
Not very well. Although the percentage of each part is similar in general, the communication time grows. In addition, there is only a little improvement from 3 to 4 nodes. It becomes more and more costly to speed up.

For different implementations, I've designed an Early-Stop mechanism to reduce the sorting phase. However, it failed, since the communication is too costly to sync all processes. For more detail, please read part (a).

(4) Others

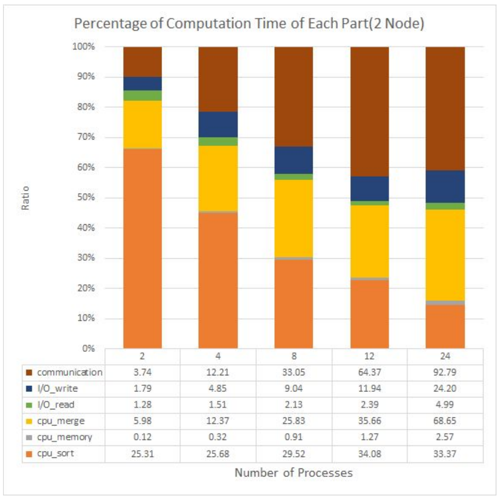


	1	2	4	8	12
■ communication	0.00	4.97	28.78	27.56	82.31
■ I/O_write	1.14	3.54	4.55	9.04	9.27
■ I/O_read	1.14	1.23	1.21	1.96	1.71
■ cpu_merge	0.00	6.55	13.10	24.68	36.41
■ cpu_memory	0.00	0.17	0.40	0.92	1.24
■ cpu_sort	26.13	25.38	25.71	29.85	33.86

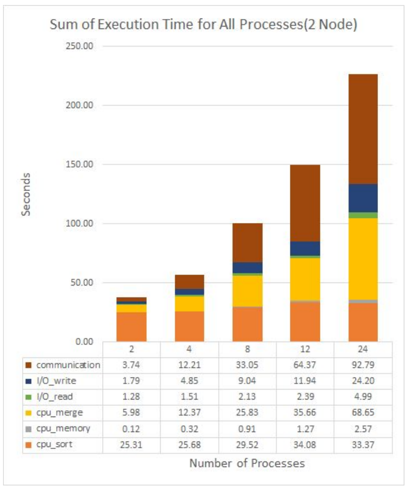


	1	2	4	8	12
■ communication	0.00	4.97	28.78	27.56	82.31
■ I/O_write	1.14	3.54	4.55	9.04	9.27
■ I/O_read	1.14	1.23	1.21	1.96	1.71
■ cpu_merge	0.00	6.55	13.10	24.68	36.41
■ cpu_memory	0.00	0.17	0.40	0.92	1.24
■ cpu_sort	26.13	25.38	25.71	29.85	33.86

Plot 1: Different number of processes in 1 node

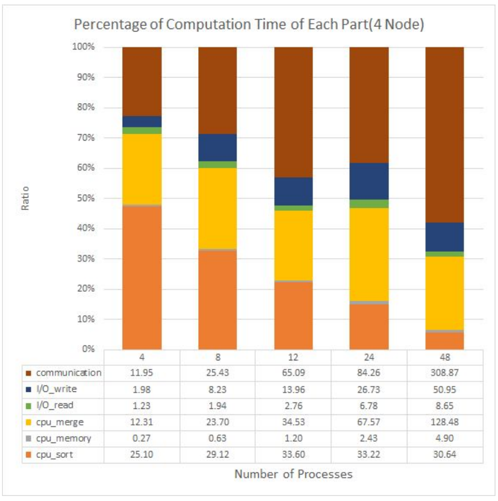


	2	4	8	12	24
■ communication	3.74	12.21	33.05	64.37	92.79
■ I/O_write	1.79	4.85	9.04	11.94	24.20
■ I/O_read	1.28	1.51	2.13	2.39	4.99
■ cpu_merge	5.98	12.37	25.83	35.66	68.65
■ cpu_memory	0.12	0.32	0.91	1.27	2.57
■ cpu_sort	25.31	25.68	29.52	34.08	33.37

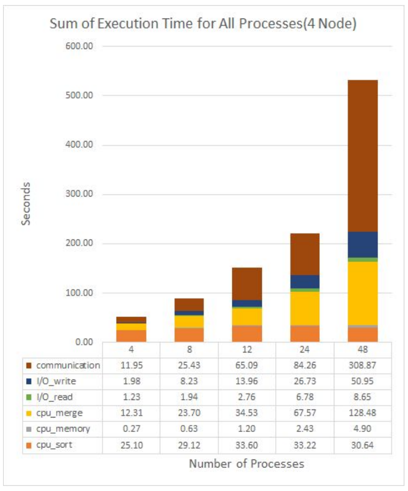


	2	4	8	12	24
■ communication	3.74	12.21	33.05	64.37	92.79
■ I/O_write	1.79	4.85	9.04	11.94	24.20
■ I/O_read	1.28	1.51	2.13	2.39	4.99
■ cpu_merge	5.98	12.37	25.83	35.66	68.65
■ cpu_memory	0.12	0.32	0.91	1.27	2.57
■ cpu_sort	25.31	25.68	29.52	34.08	33.37

Plot 2: Different number of processes in 2 node



	4	8	12	24	48
🟫 communication	11.95	25.43	65.09	84.26	308.87
🟦 I/O_write	1.98	8.23	13.96	26.73	50.95
🟩 I/O_read	1.23	1.94	2.76	6.78	8.65
🟨 cpu_merge	12.31	23.70	34.53	67.57	128.48
⬜ cpu_memory	0.27	0.63	1.20	2.43	4.90
🟧 cpu_sort	25.10	29.12	33.60	33.22	30.64



	4	8	12	24	48
🟫 communication	11.95	25.43	65.09	84.26	308.87
🟦 I/O_write	1.98	8.23	13.96	26.73	50.95
🟩 I/O_read	1.23	1.94	2.76	6.78	8.65
🟨 cpu_merge	12.31	23.70	34.53	67.57	128.48
⬜ cpu_memory	0.27	0.63	1.20	2.43	4.90
🟧 cpu_sort	25.10	29.12	33.60	33.22	30.64

Plot 3: Different number of processes in 4 node

In general, once we increase the number of processes, the execution time would decrease. However, here are some interesting things.

1. 8 process takes less portion of time in communication:

   It is weird and I think it may result from the load balancing. However, the execution time still grows but slightly.

2. I/O takes more time when processes get more:

   It is also weird. Since I've used MPI to read/write files asynchronously in parallel. Each process only reads/writes a part of the sequence that it needs to handle. The time should remain the same.

3.

3. Experiences / Conclusion

It seems that communication is a very big issue when parallelizing the program. It can harm the performance of the program extremely. Because of the penalty of the communication, early-stop is too costly to implement. I think the most difficult part of this assignment is to understand the Baudet-Stevenson odd-even sort. It takes me a half day. In addition, there is a weird bug when I add a timer to count the execution time with merge. It causes the whole program to hang in there. Finally, I think it's a good assignment to understand MPI for C from scratch.