Homework1: Odd-Even Sort 羅允辰 103061108

Implementation

The method to handle when the number of input item and number of processes are arbitrary is evenly distributed input numbers into N processes, there are three conditions:

A. If the number of input can be divided by the number of processes, we put same number of inputs into N processes.
B. If not, we use floor function to assign same number for N-1 processes. And we leave the rest of number for the last processes.
C. If the number of the process is larger than the number of inputs, then we let some process(rank number bigger than number of input) stop working.

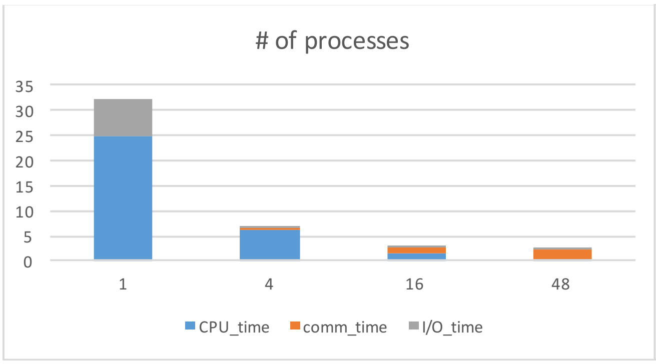
- I researched several sorting method and choose to use “qsort” in C library since the speed and time complexity are better. While researching the proper sorting algorithm, I target only on algorithm with complexity O(N*logN).
- Other effect including implementation “2” version of basic odd-even sort, focusing on following issues:
  - **Size of buffer** to be passed to near processes in each cycle(including worst swapping case scenario analysis).
  - **IO** optimization.
  - **Sorting stop criteria.**
  - **Arbitrary number of input** to **arbitrary number of process** relation.

Experiment & Analysis

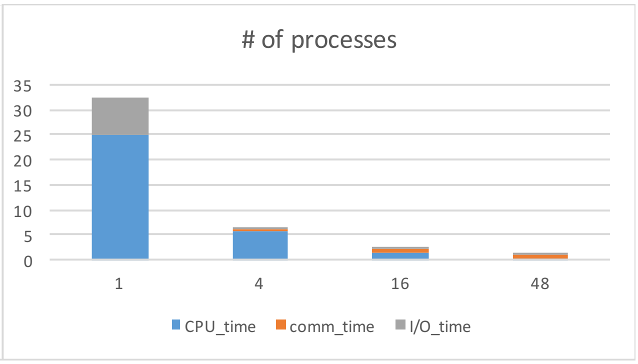
I. Methodology
  (a) System Spec: provided by TAs
  (b) Performance Metrics:
    1. Communication time: Use C library <time.h> to compute the time spending on MPI_Send(), MPI_Receive(), MPI_Barrier(), MPI_ALLReduce().
    2. IO time: Use C library <time.h> to compute the time spending on MPI_Read(), and MPI_Write().
    3. Computing time: Subtracting the total time by communication time and IO time to get the computation time.
  (c) The test data we use for two basic version is testcase06. And test data for advanced version is a new binary I generate with “ten million floating point number”. The reason we don’t compare basic and advance version under same data is because we cannot run the basic version with ten million numbers. Also, the run time for running testcase06 on advance version is too short to measure.

II. Time Profile

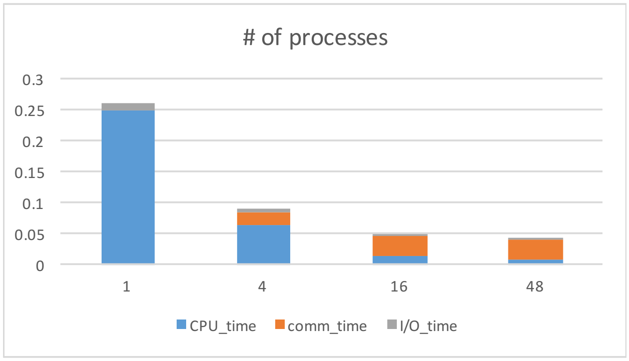
Basic Version(one buffer 65535 number)



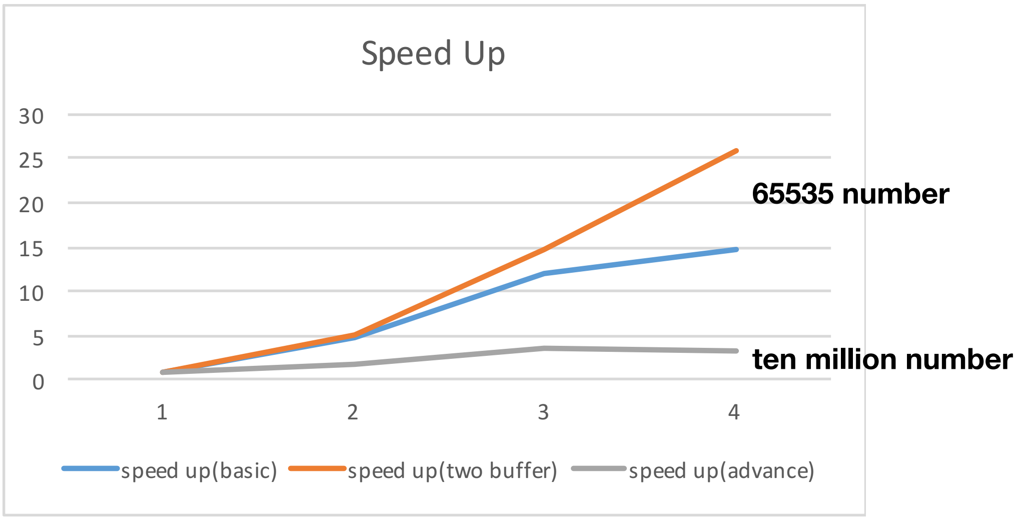
Basic Version(two buffer 65535 number)



Advance Version(one buffer ten million number)



III. Speedup



IV. Discussion

(a) After evaluating basic version of parallel odd-even sort, I found the scalability of basic version is bounded by the communication time. Therefore, I implemented another version of basic parallel odd-even swap sort to test this idea(slightly **improved** version), which let process communicate time reduced by 50% because we don't communicate for each phase. We can obtain almost twice performance on the slightly modified version.

(b) Secondly, I found the IO needed to be improved as well. Therefore, we try and error several possible cases and let each processes to obtain just needed numbers.

(c) Based on the experiments mentioned before, I implement advanced version, which focus on sending larger buffer to nearby processes and sort the local data! There is a huge speedup. But we did not compare them since the running time for same data on basic version is too huge!

(d) For both basic versions, communication time is the bottleneck! However, I think the slightly modified version(buffer size =2) scale pretty well! However, for advance version the scalability is not as expected. I think it is because the running time is bounded by the method of merging two buffer. I have implemented two version. One is simply merge two sorted array. However there is bug. Therefore, the implemented version becomes qsort two separate sorted array and choose the needed part, which creates lots of overhead.

(e) Although the CPU computing time is too short to be counted, we can still find the communication overhead did not scale with number of processes. I think the reason for running time stuck at 16 processes and 48 processes is because the qsort itself can perform better with the number of input in each 16 processes. Therefore the total time cannot be shortened by simply creating more processes.

V. Others

(a) Cross computer communication time:

# of nodes	# of process	total_time	comm_time
1	1	35.28	0
1	2	11.715	0.21
2	2	11.87	0.31
1	4	6.45	0.6075
2	4	6.74	0.765
4	4	6.84	0.767
1	8	3.89	0.84
2	8	4.02	0.91
4	8	4.2	1.12

In the above experiments we can observe that the time for communicating between nodes will significantly increase compared to in-node communication! The above communication time is coming from the average of every processes' communication time.

(b) I have implemented 6 version of code. And some discussion is covered in previous paragraph. The 6 version(all in 103061108/hw1 folder) optimization methods are:

(a) MPI_IO test file

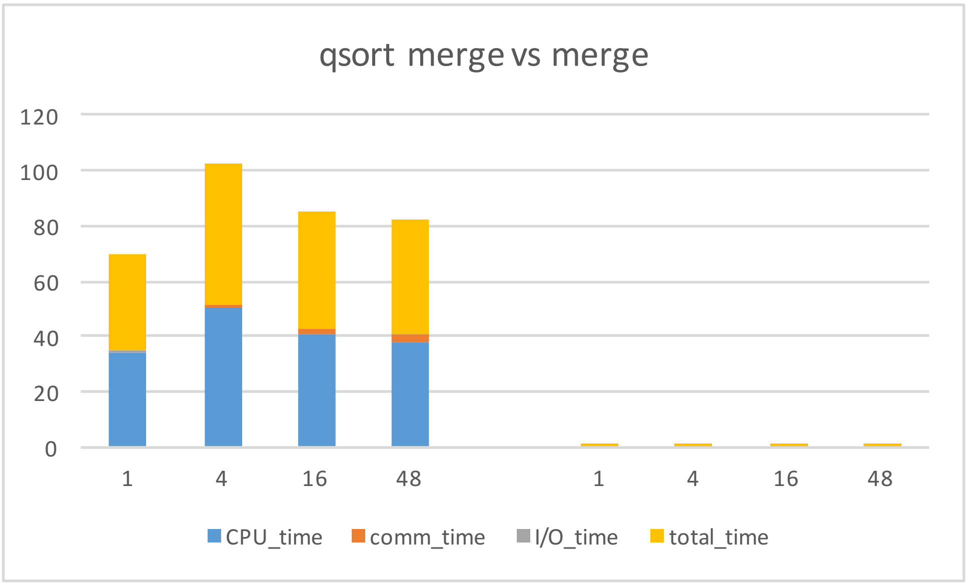
(b) MPI_IO with one float communication buffer(basic)

(c) MPI_IO with two floats communication buffer(basic)

(d) MPI_IO with quick sort and buffer with length of other processes' local buffer + merge two sorted arrays.

(e) MPI_IO with quick sort and and buffer with length of other processes' local buffer + quick sort two sorted arrays.

(c) We then want to compare the performance of two advance versions which the only implementation difference is the way to merge local data and received data!



We can observe that the advance version using sort to merge two sorted array is very time inefficient. The reason is because the sorting is always a very high complexity algorithm. The advance sorting algorithm is we sort the local data and merge sort them with near processes!

Experience/ Conclusion

1. The homework is very hard for me. But I learned a lot on how to evaluate the program we have produced!
2. I learned how to debug parallel program.
3. I encountered a lot of difficulty. But the biggest one is that after spending time(6 days), I cannot find the reason causing errors for big numbers in advanced versionQQ.
4. I think we don't have to focus on the error handling code since the focus of this course is on the parallel programming, which communication, computing and IO time is most important part.
5. So there is no necessity to design a spec which the number of processes is larger than the number of input float.