Technical Report for COMP90024 Cluster and Cloud Computing Project 1

Xinyao Niu 90072 xinyaon@student.unimelb.edu.au
March 31, 2020

Abstract

This is a technical report for analysing the performance for the same job but running in sequential or parallel. There are mainly three parts of this report, including a brief introduction to the code and script structure, the approach that we used to parallel the job, and finally the performance analysis.

1 Project Structure

1.1 Code Structure

There are mainly two parts that related to the code, they are ***testCluster.py*** and ***utils.py***.

The ***testCluster.py*** consist of the main running logic for the project. It split job equally if there are multiple core available, or running sequentially if there are only one core provided.

The ***utils.py*** contains other useful helper functions. Including the regular expression to extract the hashtag from twitter text (did not counting what's potentially in retweet and quote), the function that used to processed each line into a standard json structure, a lazy file reader which wouldn't dump everything in a file into the main memory and a function for pretty illustration of the final data.

1.2 Script Structure

There are many different scripts that I used on spartan. They can be mainly classified into two group based on usage, the job submission and house-keeping.

For the submission part, it includes ****.slurm*** and a base script ***submit.sh*** that used to submit specific jobs. All the ****.slurm*** constructed based on the partition it used and how many cores and nodes it will be provided in the job.

For the house keep part there are only one bash script called ***run.sh*** that used to clean all the code that I submit in order for me to renew the code using ***scp***.

1.3 Workflow

First, clean all the ****.out*** files that contains the output from previous run if they are no long needed. And then submit the job using the script in ***submit***. Finally, look at the output and either run the house keep script to renew the code locally or using vim to edit small part and re-submit the job using ***submit***.

2 Strategy to Parallel

There are two main sections in ***testCluster.py***. If only one core are provided, the task will be running sequentially, otherwise code will be paralleled.

I actually implemented two models for this project that trying to parallel the code, the first one is a master-slave model which master hold the IO object and lazily reading from the document and act like a job queue which will cache at most 100 lines of twitters. Each slaves (core other thank rank 0) ask master for job to do and at the end the result will goes toward master. The other approach is that tasks are partitioned based on total number of nodes (including node 0 which there is no core dedicating on IO only) and each of them process their own line. Finally, just like approach one, the result are collected and summarized at core 0. In the next section, the second approach will be discussed more as that is my final submission. Since approach one somehow only works on my personal laptop on bigTwitter and never terminate on spartan, it will only be discussed in an less-quantitative way.

3 Analysis of Performance

There is the overall plot for the different in time for different amount of resources available[1]. Note that the algorithm used to run the job is the second approach that mentioned above which the job will be partitioned and each core do their own. It is not surprisingly to find that with 7 more core added, 71% of time has been saved. However, when the core number has been doubled again, the improvement is not significant any more (only 14% less when comparing with 8 cores and 75% when comparing with 1 core). Another figure2 is about the time that each core needs to finish it own jobs and estimated optimal cost in that situation. Based on these information, we can conclude that

- The time cost significantly dropped for the first 7 cores added as the part that can be paralleled has been successfully distributed. However, with more cores been added, the part that cannot been paralleled starts to dominate the total time cost, for example, the time that we spend on file accessing.
- The drawback can be easily illustrate from the second figure. As I could not find a way to access a file from arbitrary position, the node that take in charge of the final partition of the job becomes the bottle-neck of the total task, or more specifically the reading speed. For 8 cores, it takes roughly 63% of time on reading the file from the beginning till the point that need to be processed, and this number increase to 78% for the 16 cores. As there are more cores added, the proportion will keep increasing unless a proper way to randomly access the file just like memory.

It is worth notice that even though the first approach that I mentioned only runs on my laptop, it actually performed slight worth than the approach that I submit if implemented in an non-blocking way. Even though the whole job will only read the file once, the time using on sending lines to other cores and one-less core doing job are still critical in this scenario.

4 Appendix

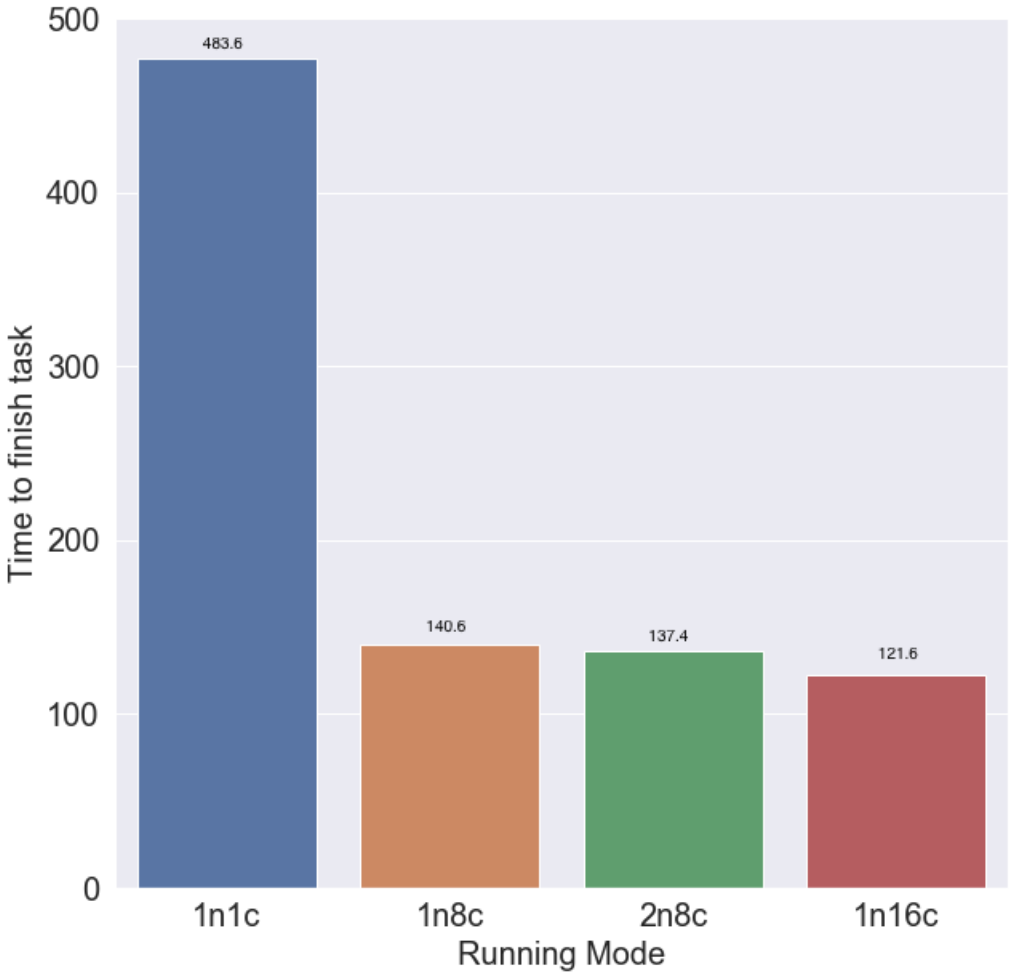

Figure 1: Job running time plot

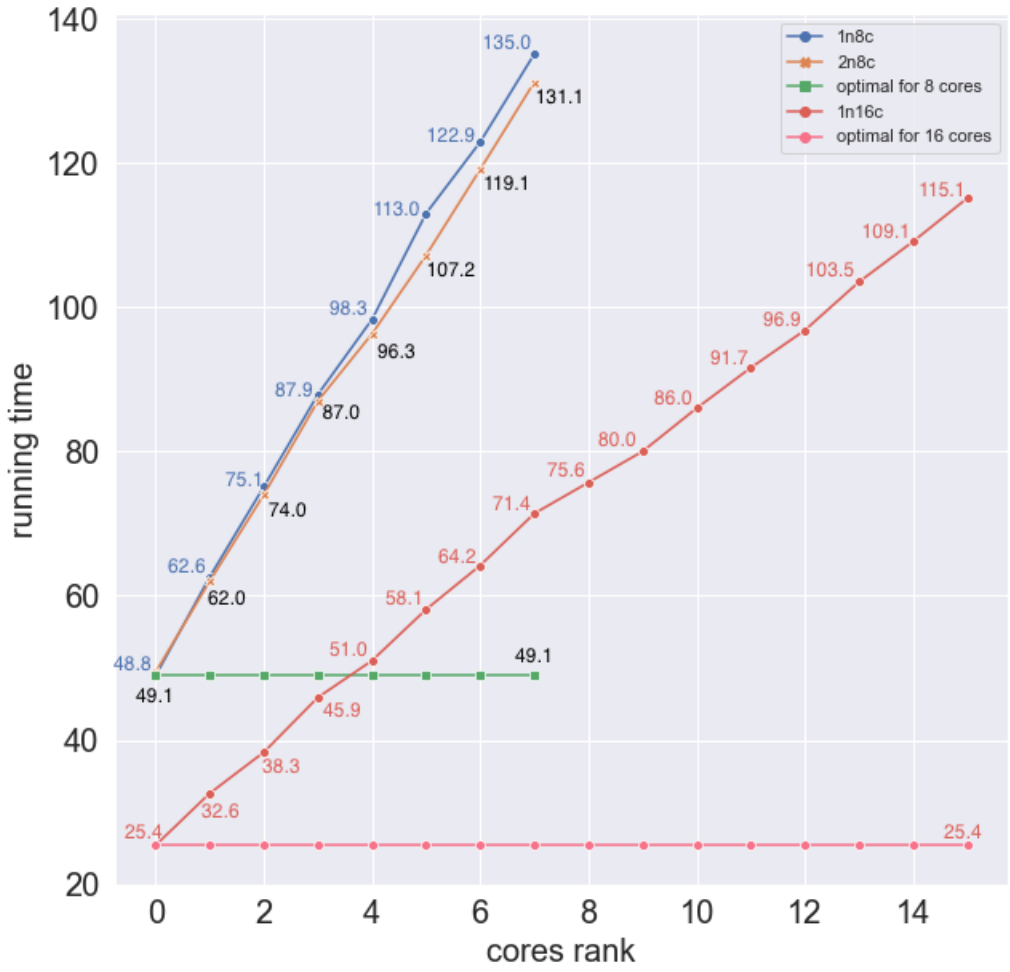

Figure 2: Core processing time comparison in each mode