Cluster and Cloud Computing Assignment 1
HPC Twitter GeoProcessing Report

Hanxun Huang - Student Id: 975781
Xu Wang - Student Id: 979895

Professor: Prof Richard Sinnott

School of Computing and Information Systems
University of Melbourne
Australia
April 2019

1 Project Description

This project aims to implement a parallelized application leveraging the University of Melbourne HPC facility SPARTAN. Identify Twitter hashtags and the number of Tweets around Melbourne by searching a large geocoded Twitter data set.

2 Project Implementation

Our project is implemented in Python with the mpi4py library that works with OpenMPI. We followed the Single-Program Multiple-Data architecture that split data into smaller batches that each sub-process only process a small amount of the data and gather the result. It is a very common architecture in real-world practice. For example, PyTorch uses a similar method that divides input data into smaller batches to allow neural networks to train on multiple GPUs.

In our project, we mainly implemented 2 files, util.py and main.py.

Util.py:

- search_result class: Stores the result corresponding to a grid. Stores the number of posts, grid id, hashtags and provide functions that process the result to produce Top 5 most common hashtags.
- grid_data class: Stores single entry of grid JSON data, provide a function that parses single JSON entry and a function check if given geo-coordinates belong to this instance.
- twitter_data class: Stores single entry of twitter JSON data, provide a function that parses single JSON entry into an instance.
- util class: Search function that takes a list of grid_data and twitter_data returns a list of search_result class. As well as loading JSON file helper function.

Main.py: The main uses "argparse" library to parse the arguments. We added the path to the Twitter data file and grid data file as arguments so that the program is more robust we can test it on a different data set without changing the source code. The batch size is also an argument that specifies the number of lines data to be processed by a sub-process in a batch.

The main function will check if there are multiple processes available, if it is not, it will just simply load the data and do the search. If there are multiple processes available, the root process acts as a handler that load numbers of lines of data and send it to sub-processes. Each sub-process receive a batch of data needs to be processed then invoke the search function and stores the search result in a list. When all the data are read and send to sub-processes, the root will send a signal let sub-processes know to break out the loop and prepare to

gather the result data. The root will gather the result from different sub-process and merge them into a single list to produce the output.

Spartan Scripts:

```
#!/bin/bash

#SBATCH --account=COMP90024
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --time=1-0:0:00

# check that the script is launched with sbatch
if [ "x$SLURM_JOB_ID" == "x" ]; then
   echo "You need to submit your job to the queuing system with sbatch"
   exit 1
fi

# Run the job from this directory:
cd /home/$USER/COMP90024-2019SM1-Project1/

# The modules to load:
module load MPICH/3.2.1-GCC-6.2.0
module load Python/3.5.2-GCC-6.2.0

# The job command(s):
mpirun python3 main.py --grid_file_path "../data/melbGrid.json" \
--twitter_data_file_path="../data/bigTwitter.json"  --batch_size=15
```

3 Result and analysis

During the implementations, we identified the most resource consuming part of this project is the parse JSON object process. In fact, iterating over the "smallTwitter.json" only took about 10ms. However, iterating and parsing it into Python JSON object took about 70ms. According to Amdahl's law (Hill and Marty 2008), if we want to maximize the performance of the parallel program, we need to reduce the amount of time needed for the non-parallelizable part. Therefore, we let the root process only iterating over the data without parsing them into Python JSON object to maximizing the performance. The sub-process receive a chunk of data to parse and do the search. The root process will load the next batch of data and send it to the next sub-process and so on. It is possible to further improve the performance by fine-tuning the batch size to minimize the effect of the sub-process waiting for new data or root waiting to send new data. Theoretically, we can also add a queue structure to stores the data that need to be processed which also can improve the performance.

However, benchmark performance is not the goal of this project, we did not do any further investigation into this issue.

$$T(1) = \sigma + \pi \tag{1}$$

$$T(N) = \sigma + \frac{\pi}{N} \tag{2}$$

$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \frac{\pi}{N}} = \frac{1 + \frac{\pi}{\sigma}}{1 + \frac{\pi}{\sigma} \times (\frac{1}{N})} \tag{3}$$

$$\pi/\sigma = \frac{1 - \sigma}{\sigma} \tag{4}$$

$$S = \frac{1 + \frac{1-\alpha}{\alpha}}{1 + \frac{1-\alpha}{N\alpha}} = \frac{1}{\alpha + \frac{1-\alpha}{N}} \approx \frac{1}{\alpha} \tag{5}$$

(α : Fraction of running time that sequential program spends on non-parallel parts of a computation)

According to Amdahl's law (Hill and Marty 2008), the theoretically maximum speed up is 1 divide by the number of cores. It is limited by the non-parallelizable part of the program. In our implementations, we utilized root as handler and the rest process will perform the parallelizable computations. The root process only handles the load JSON operation, if the batch-size is perfectly fine-tuned according to the processor's speed and the communication overhead is near 0, we could achieve the perfect performance in our approach. In our implementation, except the communication overhead, there is part of a program that is not parallelizable. Such as, read the JSON file, split the data into batches and merge the final results. Our method is also limited by there is a core(root) that did not used for parallelizable computing instead act as a handler. It can be improved by each sub-process load it's own data and merges the result in the end.

Configuration	System Timer	Program Timer
1Node-1Core	343.24s	335.84s
1Node-8Core	57.81s	57.42s
2Node-8Core	61.31s	59.96s

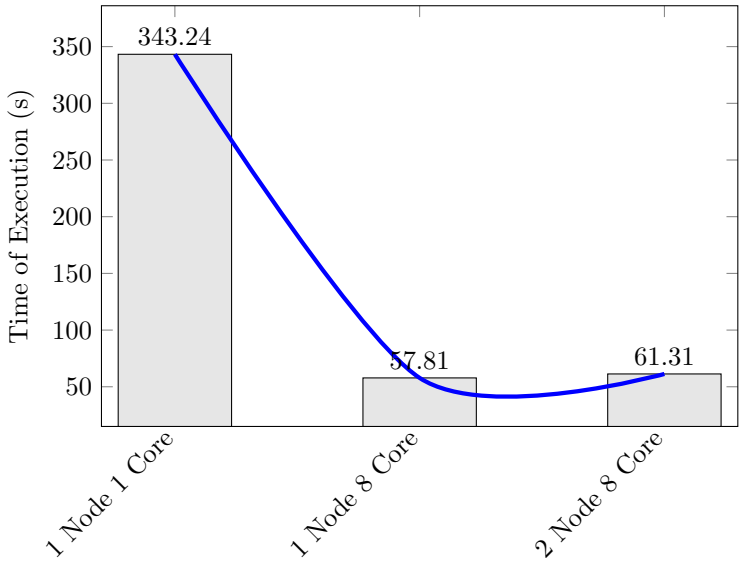
Table 1: Result Table for the bigTwitter.json dataset
(Note: System Timer use the 'time' command, program Timer use Python built-in time method)

We experimented our program on different Spartan Configurations, the T(1) or the 1-core configuration is around 340s. Therefore, the theoretical maximum performance for 8-core configuration is $\frac{340(s)}{8}$=42.5(s). According to our experiments result (Table 1), the performance is around 60s. This result confirms

the Amdahl's law (Hill and Marty 2008), that the speed up depends on the proportion of the program are parallelizable.

The time consuming for 1 node is slightly lower than that of 2 nodes. The main reason of this is because when allocating resources, communication between nodes is required.The real execution time for different configurations is shown below (Chart 1).

Chart 1: Time of execution on each configurations



The final result indicating in the Melbourne grid area, zone C2 has the most tweet posts and zone A4 has the least. C2 corresponding to the CBD Area and Southbank, which are the most populated area around the Greater Melbourne area. Within 16 grids in total, the hashtag "#melbourne" ranks first in 7 grids.

4 Conclusion

To conclude, the 1 node 8 core shows better performance among the three configurations mentioned in the requirements. It uses the lowest time to complete the task in our approach. Our experiment result has confirmed the Amdahl's law (ibid.) when applying for a certain program in the parallel computing environment.

References

Hill, Mark D and Michael R Marty (2008). "Amdahl's law in the multicore era". In: *Computer* 41.7, pp. 33–38.