COMP90024 Cluster and Cloud Computing
Assignment 1 - Social Media Analytics

Group Members

1244661 Natakorn Kam | 1329582 Huu Hieu Nguyen

1 Introduction

In this assignment, we will learn about the basic principles of HPC and how to implement a parallel application using the University of Melbourne's HPC facility SPARTAN. This application shall process a large Twitter dataset, which includes a file with following attributes: suburbs, locations, and greater capital cities in Australia. It will provide answers to these questions: top 10 tweeters based on the number of tweets, number of tweets made in each greater capital city, and identify those tweeters with most different cities. The main script is written in Python and then we use Slurm to send a request to the HPC. In this assignment, we will be testing 3 different scenarios: 1 node & 1 core, 1 node & 8 cores, and 2 nodes & 8 cores (with 4 cores each).

2 Approach Description

Twitter is a social network platform that allows people all over the world to interact with each other by posting statuses (tweeting), commenting and sharing each other's content. With its gigantic user base, Twitter is a great analytical data source, which could be useful in finding answers like sentiment analysis on particular products or services. However, working with this enormous size of data requires an HPC and parallelization programming technique to make it possible and executable under a manageable period of time. Therefore, the application is implemented in Python using Message Passing Interface for Python (mpi4py) to establish parallelization, so that the data can be processed and managed easier. In the following chapters, we will discuss in detail about the parallelization techniques, the way we cleaned and sorted data, as well as job submission.

2.1 Parallelization Techniques

Thanks to the university's infrastructure we have been granted access to SPARTAN, and it will be utilized effectively for this assignment. The main idea of the task is to efficiently distribute equal amounts of work to each core of the supercomputer. There are 3 major steps involved in this technique.

First, we need to count the total number of json objects in the file by opening it and then process the data in said directory. Based on the string ["id:"] we were able to count the number of objects in the json file. In the beginning, we stumbled upon a problem, which is the long execution time of the Python script using the library "ijson".

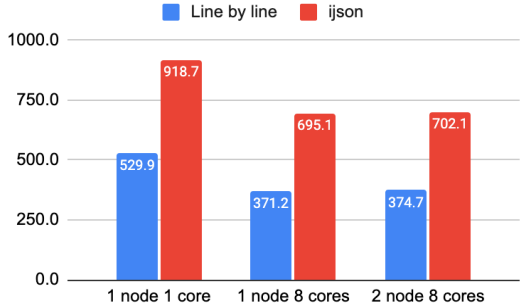

***Figure 1**: Comparison of runtime between read line by line and ijson*

It took a long time to run and give out results (Figure 1). After a while we managed to figure out the point at issue. Thus, we switched to reading the file line by line. It is better because instead of processing the data by json objects which consume a lot of memory resulting in longer processing time, reading the data from one line and then discarding it from the memory appears to be much more optimized (Figures 2 and 3).

```
with open(fileName, "r") as f:
    for record in ijson.items(f, "item"):
        # process file
```

***Figure 2**: open file using ijson*

```
with open(fileName) as f:
    for line in f:
        # process file
```

***Figure 3**: open file by reading line by line*

In the second step, we split json objects based on the number of available cores. Resulting in calculating the first and last json object that each core needs to process. After achieving the desired number, the data will be distributed among cores using the scatter() function from the package mpi4py.

Next, each core will process a given chunk, the file will read from the first line to the last of the responsible chunk. The matching process of place name and code is to check whether that place matches with the same attribute from "sal.json" file or not. After it passes the last chunk, we will exit the loop to reduce runtime. In other words, the core will process objects in their range only. Afterwards, the data will be merged.

To summarize, the constant runtime for every case will be counting json objects, which is what we have to do every time before distributing data equally. As a result of parallelization, the runtime would improve since the data is processed chunk by chunk.

2.2 Sorting And Cleaning Data

The cleaning process is focused on the name of the place since it will be paired with the suburb name in the "sal.json" file. First, we used arrangeSuburb() to sort data from the file "sal.json", and then we rearranged the geological locations to match its great capital city accordingly.

Secondly, in tweeted messages, there are locational attributes. We need to transform strings with capital letters to only small letters using lower(). However, some data needs to be removed and reorganized using cleanPlaces(). This function removes geographical descriptions with only "Australia" as its keyword since it is impossible to identify the suburb or city of these tweets. The cleanPlace() function also abbreviates the state name to a more convenient format and relocates those objects with capital city as their second attribute to their correct state.

2.3 Script Submission

In order to submit the code to SPARTAN we used 3 separate Slurm scripts, requesting a different number of cores and/or nodes each time. As shown in Figure 4, we use the following script to request for 1 node and 8 cores, which also automatically notifies via mail when the task is finished. The time limit for the job is 10 minutes.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --job-name="1nodes_8cores"
#SBATCH --mail-user=natakornk@student.unimelb.edu.au
#SBATCH --mail-type=END
#SBATCH --time=0-0:10:00
```

***Figure 4**: Slurm Script*

3 Result Analysis

After submitting the job using slurm, we receive the result in format .out which contains the print command in Python and performance report of CPU and RAM usage. The output also includes counting objects time, splitting data time, data processing time, and merging time. These output will be discussed in this part accordingly.

3.1 Results

The submission results are shown below. To answer the first question, we can look at Figure 5 illustrated below. It can be seen that Melbourne and Sydney are ranked 1st and 2nd respectively with by far the most number of tweets posted. This can be easily explained by the population of these 2 cities, its population is about twice the population of Brisbane – the city ranked number 3.

Figure 6 displays the answer to the second question, we notice that the user with the most tweets has about 68,477 in total, which is, coincidentally, more than twice in comparison with the 2nd ranked user. With further investigation, we found out that this user's first tweet was in February 2022, which converts to roughly 163 tweets per day.

Lastly, the explanation to question 3 is shown in Figure 7. All of the 10 ranked users have tweeted from 8 greater capital cities except other territories. The person with the most tweets has about 1,920 posts coming from those places.

Author Id	Number of Tweets
1498063511204761601	68,477
1089023364973219840	28,128
826332877457481728	27,718
1250331934242123776	25,350
1423662808311287813	21,034
1183144981252280322	20,765
1270672820792508417	20,503
820431428835885059	20,063
778785859030003712	19,403
1104295492433764353	18,781

***Figure 5**: Question 1 Result*

Greater Capital City	Number of Tweets
2gmel	2,286,484
1gsyd	2,117,747
3gbri	858,073
5gper	586,232
4gade	460,604
8acte	202,493
6ghob	90,639
7gdar	46,387
9oter	3

***Figure 6**: Question 2 Result*

Author Id	Number of Unique City Locations	Number of Total Tweets Made	Count Tweets in Each City
702290904460169216	8	1920	#1880gmel, #2gade, #13acte, #7gper, #10gsyd, #6gbri, #1ghob, #1gdar
1429984556451389440	8	1209	#1061gsyd, #40gbri, #23acte, #60gmel, #11ghob, #3gade, #4gdar, #7gper
17285408	8	1116	#252gmel, #111gade, #46acte, #218gbri, #40ghob, #20gdar, #288gsyd, #141gper
87188071	8	393	#28gade, #65gbri, #50gper, #110gsyd, #86gmel, #34acte, #15ghob, #5gdar
774694926135222272	8	272	#37gbri, #28gade, #28gdar, #34acte, #37gsyd, #34gper, #38gmel, #36ghob
921197448885886977	8	260	#36gmel, #193gdar, #12gsyd, #9gade, #1gper, #6acte, #2ghob, #1gbri
1361519083	8	250	#214gmel, #8gbri, #8ghob, #3gper, #4gade, #2gsyd, #1gdar, #10acte
502381727	8	205	#37gbri, #24gade, #28gper, #46gsyd, #58gmel, #7acte, #4ghob, #1gdar
601712763	8	146	#44gsyd, #10acte, #39gmel, #1gdar, #14gper, #8ghob, #19gade, #11gbri
2647302752	8	80	#32gbri, #3gdar, #3gade, #4gper, #4acte, #13gsyd, #16gmel, #5ghob

***Figure 7**: Question 3 Result*

3.2 Performance Analysis

We can make some conclusions and compare the results using Figure 8, which shows the script execution time for the file bigTwitter.json. We can easily identify that the format that took the longest to process is 1 core & 1 node. In terms of speed, 1 core & 8 nodes and 2 cores & 8 nodes both achieved identical results. The usage of each CPU is also close. However, we can still notice that

running with 1 core puts a little more strain on the CPU, as well as consuming more memory (5MB to 4MB). The reason for this might be the data being processed by a singular node.

The reason that 1 node & 8 cores have the best performance could be because each core receives a chunk of work, and then exits fast when it finishes. After all the calculations, the result is merged into the same node. The 2 node & 8 cores setting comes second closely, the few lagging seconds could be because of communication time between 2 nodes. The execution time per core improved almost 10-fold by running 8 cores in comparison.

The entire process is shown in Figure 9 with results from 1 node & 1 core setting. The yellow highlighted part is the parallel process, which means that the constant time would be about 122+38 = 160 seconds. Those parallel processes are then distributed among each core so that time usage is decreased. However, it is not significantly reduced because the data needs to be read from the start. This means the last chunk of the data is the bottleneck of the process. We tried using seek() to start reading files from the appropriate start line, but the result worsened.

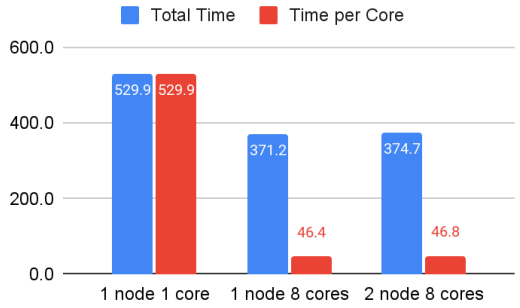

***Figure 8**: Performance of each scenario*

Process	Time (seconds)
check number of object	122.4
split time	0.0
generate data	366.2
count data	1.1
merge time	38.0

***Figure 9**: Process Steps*

4 Conclusion

After finishing this assignment we were familiarized with the fundamentals and first steps to work with HPC by analyzing a large set of real data from a social network platform Twitter. We learnt that HPC is an extremely powerful tool that can be shared among multiple users, and to utilize it we need to apply the concept of parallel programming, and in our case, we used an installable MPI package for Python – mpi4py. To achieve desirable results, we used a script in Slurm format to execute the Python file through the HPC. We also experimented with a few different settings to understand more about the mechanism of SPARTAN by altering the requested number of nodes and/or cores.