

COMP90024 Cluster and Cloud Computing Project 1 Report

Yi Yang, Claire Zhang

Abstract

This report provides an insight into various approaches performed to implement a parallelized application that searches a large geocoded Twitter dataset in Sydney to identify tweets hotspots by leveraging the University of Melbourne HPC facility SPARTAN. The report has been divided into three parts, including a brief introduction to the work structure, parallel methodologies and the result analysis.

1 Project Structure

1.1 Data

The big Twitter dataset is a 20Gb JSON file containing 4057524 tweets, including user information, locations, text content etc.

The 'sydGrid.json' file consists of 16 polygon form geo-coordinates over the Sydney regions.

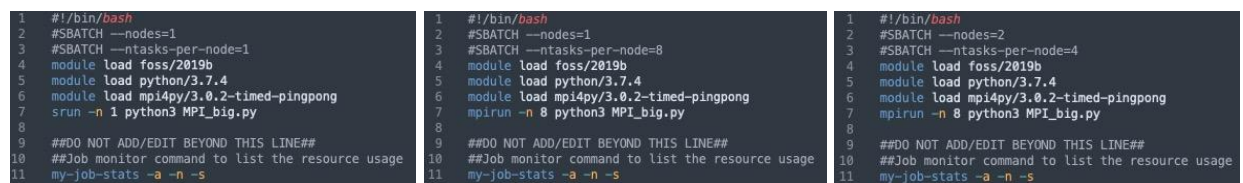
1.2 Code Structure

The MPI_big.py contains the main logic of how to process the data and generate the final result. It can be used in both one or multiple cores that are provided for the job. If there are multiple cores provided, it is designed to split to the equivalent load of work for each core.

1.3 Script Structure

In order to submit the job to SPARTAN, three *.*slurm* scripts in Figure 1 were written based on the different nodes applied. All of the slurm scripts contain the partition is used and the number of nodes, the number of tasks per node that is provided in the job and which module needs to be loaded. There are the nodes and cores applied:

- 1 node 1 core
- 1 node 8 core
- 2 nodes 8 core



```
1 #!/bin/bash
2 #SBATCH --nodes=1
3 #SBATCH --ntasks-per-node=1
4 module load foss/2019b
5 module load python/3.7.4
6 module load mpi4py/3.0.2-timed-pingpong
7 srun -n 1 python3 MPI_big.py
8
9 ##DO NOT ADD/EDIT BEYOND THIS LINE##
10 ##Job monitor command to list the resource usage
11 my-job-stats -a -n -s
```

```
1 #!/bin/bash
2 #SBATCH --nodes=1
3 #SBATCH --ntasks-per-node=8
4 module load foss/2019b
5 module load python/3.7.4
6 module load mpi4py/3.0.2-timed-pingpong
7 mpirun -n 8 python3 MPI_big.py
8
9 ##DO NOT ADD/EDIT BEYOND THIS LINE##
10 ##Job monitor command to list the resource usage
11 my-job-stats -a -n -s
```

```
1 #!/bin/bash
2 #SBATCH --nodes=2
3 #SBATCH --ntasks-per-node=4
4 module load foss/2019b
5 module load python/3.7.4
6 module load mpi4py/3.0.2-timed-pingpong
7 mpirun -n 8 python3 MPI_big.py
8
9 ##DO NOT ADD/EDIT BEYOND THIS LINE##
10 ##Job monitor command to list the resource usage
11 my-job-stats -a -n -s
```

Figure 1: Slurm Script

2 Parallel programming with MPI

Since we have different numbers of cores provided, we try to partition the job to let them process in their own line. To handle this approach we try to partition the job according to the number of nodes that process the job in their own line. That is why we try to read the large data file line by line and only extract the tweet that has a coordinate that is useful for us to check whether it is within the cell. The design of this is to avoid loading all of the tweet information in the master which is more efficient for data processing.

We have used the gather and beast method in the MPI4py library. Since we calculate the start-line and end-line of each rank manually, we do not use the scatter method. The design is to divide the input data into different parts and send them to each node to work on. After reaching the barrier, each node will check the tweet location and extract the language of the tweet. Then we used the gather method to collect each processes' result and gather them to one at core 0 as a final result. We converted our result to a large dictionary which is easy to print out with respect to different cells.

3 Result And Analysis

The final output has been generated to a ***.out** file and it has also been shown in the appendix[1]. and it has also shown in the appendix. It shows the #total tweets, #language used and #top 10 languages the corresponding frequency in each area, which has been divided into grid cells.

The following table and graph compare the runtime of different nodes and cores on the application. From the results, 1c8c and 2n8c have less running time than 1 node 1 core, which is expected since having several cores can deal with different data threads simultaneously, leading to a quicker data process. 1n8c and 2n8c have quite similar running times. Compare 1n8c to single-core(1n1c), it has saved 56% running time. For 1n8c and 2n2c, they have similar running times but 2 nodes have slightly better performance. That might imply that multi-node doesn't necessarily improve performance.

Numbers of Node & Core	Run Time (s)
1n1c	339
1n8c	147
2n8c	143

Table1: Multi-node/core Performance

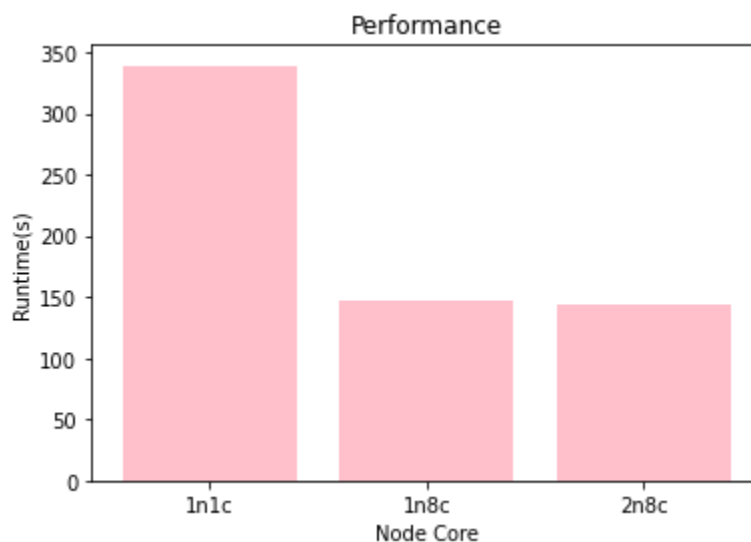


Figure 2: Performance bar

In the 1n8c process, we can see that although each of the first 7 nodes is processing the same amount of tweets, the processing time is actually incremental. Thus the multi-core is not completely parallelised.

In addition, it is worth noting that there is a significant pending time for the program to access the cloud partition, especially when some partitions are busy. Therefore, seeking faster pending time, such as choosing more free cloud partitions to allocate resources can be something to improve efficiency.

4 Appendix

[1] Figure 3: Process Time

```
rank 0 has processed 507190 lines takes 41.525233432650566
rank 1 has processed 507190 lines takes 53.571213375777006

rank 2 has processed 507190 lines takes 67.69564106315374

rank 3 has processed 507190 lines takes 76.69164279103279

rank 4 has processed 507190 lines takes 85.3331178240478

rank 5 has processed 507190 lines takes 100.95810535177588

rank 6 has processed 507190 lines takes 106.68208773434162

rank 7 has processed 507193 lines takes 116.51337845623493
```