HPC Twitter GeoProcessing

San Kho Lin (829463) — sanl1@student.unimelb.edu.au

1 Introduction

In this project, I experiment parallel programming to implement a simple MPI[1] application leveraging The University of Melbourne HPC facility, SPARTAN[2]. Contrary to a typical parallel project, the objective is not to consider optimisation of the application to run faster i.e. *Speed Up* factor, but to learn about introductory to HPC and how basic benchmarking of application on a HPC facility can be achieved and, lessons learned in doing this experiment on a shared resource.

2 Problem Description

The problem statement to solve in this experiment is such that the application needs to search a large Geo-Coded Twitter dataset to identify tweet hotspots around Melbourne. The experimental dataset has a size of $\approx 10GB$ in JSON formatted file, namely *bigTwitter.json*. The Melbourne grid in JSON formatted file – namely *melbGrid.json* – is provided and, it includes the latitudes and longitudes of a range of gridded boxes as illustrated in the Figure 1. An individual tweet can be considered to occur in the box if its Geo-Location information – the tweet latitude and longitude given by the tweet coordinates[3] – is within the box identified by the set of coordinates in *melbGrid.json*.

3 Geo Search Implementation

A simple naïve way of implementation is to search a given point in all grid boxes. In this case, the worst-case search of the algorithm is linear for $G(N)$ number of grid boxes. A better approach is to use a simple **Divide and Conquer** strategy. It should also be noted that the file *bigTwitter.json* includes many tweets that are not in *melbGrid.json* grid, e.g. they are from other Australian cities or from other parts of Victoria. The application should filter/remove these tweets since only the tweets in the grid boxes identified *melbGrid.json* are of interest. Figure 1 shows the visual of how grid division and searching performs over the Map.

3.1 Algorithm Details

```
First search a point is within Super Cell using A1, D5 coordinates. If true then;
First cut by lattitude, any of B1...B4 ymin value, result two zones: z1, z2
z1 comprise of A1...A4, B1...B4
z2 comprise of C1...C5, D3...D5
If point fall into z1, then
Cut z1 by longitude, using A4 or B4 xmax value, result zVoid and zAB
If point is in zVoid then ignore and continue next tweet
If point fall into zAB, then
Cut zAB by lattitude, any of A1...A4 ymin value, result two zones: zA, zB
If point fall into zA, then
Cut zA by longitude, using A3 xmin value, result two zones: zA12, zA34
If point fall into zA12, then
Cut zA12 by longitude, using A2 xmin value, result now know the point either A1 or A2
If point fall into zA34, then
Cut zA34 by longitude, using A4 xmin value, result now know the point either A3 or A4
...
Algorithm perform similar approach on other half at every levels.
```

[1] Message Passing Interface
[2] https://dashboard.hpc.unimelb.edu.au/
[3] https://dev.twitter.com/overview/api/tweets

4 Parallel Implementation

For parallel implementation, the focus is more on handling of the big JSON file. Since the *bigTwitter.json* is $\approx 10GB$ and each tweet entry is line separated, it is possible to process line by line for memory efficient way – instead of reading the whole file into memory. There are generally two approaches: partition the data and process each partition in $P(N)$ processors or, a master process read the lines and distribute tasks to $P(N-1)$ processors for Geo-Searching.

For this project experiment, I choose **Supervisor-and-Workers** with dynamic task pulling approach. First, the supervisor process reads the tweet line and, waits for any worker process READY signal (i.e. MPI Tag). The supervisor process loops through line by line until end-of-file has reached. At worker process, it sends its READY signal for availability, and then receives the tweet line from the supervisor for Geo-Processing. This model works since processing of each tweet line is independent of each other. It is well-known as an Embarrassingly Parallel[4] problem.

Since the purpose of the application is to find the hotspots – i.e. finding the tweet count for each grid box, each worker maintains its own local data structure for the occurrence frequency. It is trivial to use a simple key-value data structure with fix keys for $G(N)$ grid boxes for counting. When the supervisor process reaches end-of-file, it sends termination EXIT signal to all workers. All workers are then sending their local counting data to the supervisor for summing and sorting statistics.

5 Conclusions

Appendix describe how the program can invoke on local development as well as an example job script for running on SPARTAN HPC cluster. The program output result observes that tweets hotspots concentrated from Melbourne CBD[5] region and having sparse towards outer suburbs.

Figure 2 shows the breakdown of different runs for the program. It observes that it does not achieve a major speed up and, the communication start up time is more expensive than sequential Geo-Searching time in this case. Perhaps the data partition approach might yield a better speed up. Perhaps the problem computation is much simpler to solve it in sequentially.

Nevertheless, by experimenting this project, I have learnt how to do parallel programming with Python/mpi4py for MPI, as well as hands-on experience on running it on shared HPC resources and basic benchmarking of the HPC application.

6 Appendix

For local development setup, the program can be invoked by:

```
mpiexec -n 8 python app.py
```

A SLURM job script for 2 nodes 8 cores (4 cores per node) run as follow:

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --time=00:30:59
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=4
module load Python/3.4.3-goolf-2015a
mpirun python app.py
```

[4] https://en.wikipedia.org/wiki/Embarrassingly_parallel

[5] Central Business District

Figure 1: Simple Divide and Conquer over Melbourne Grid

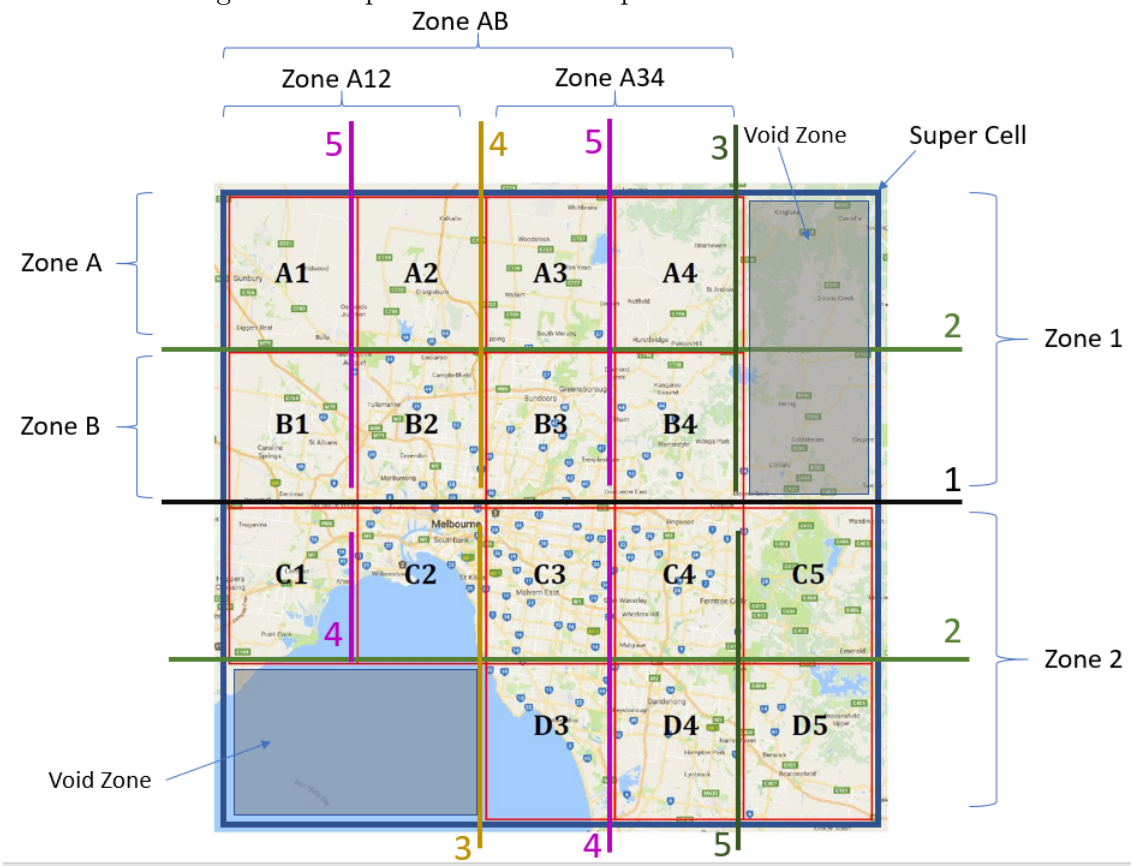
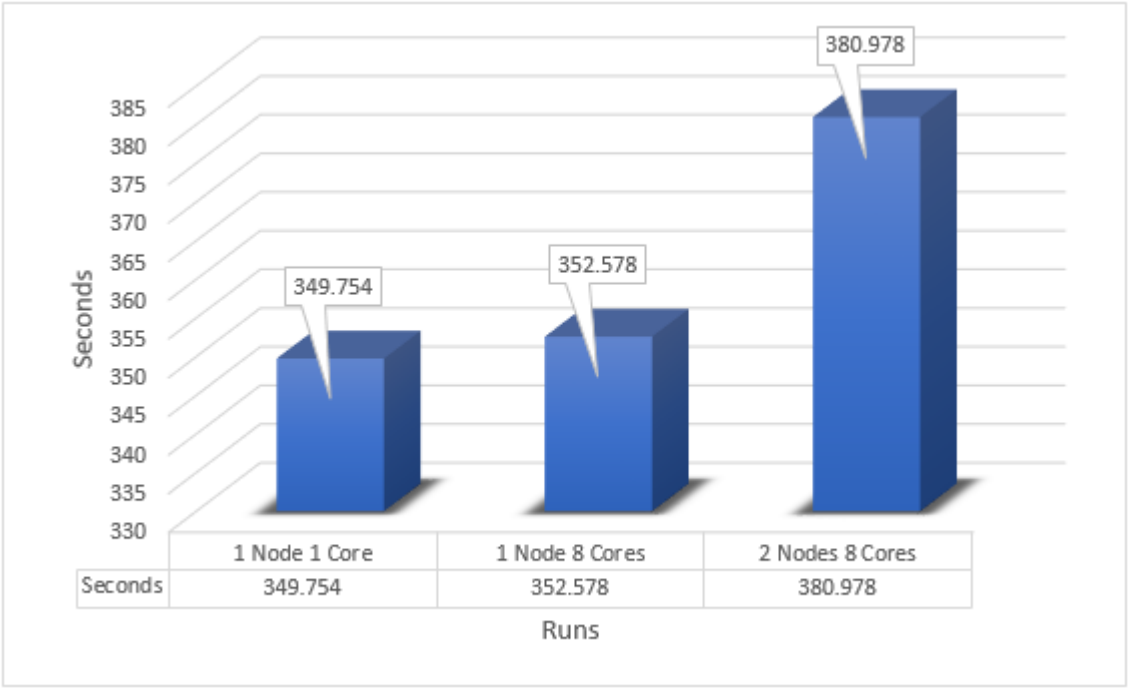


Figure 2: Elapse Time For Executions



Program Output Result

```
Order (rank) the Grid boxes based on the total number of tweets made in each box
-------------------------------------------------------------------------------
C2: 139332 tweets
B2: 78634 tweets
C3: 52769 tweets
B3: 26216 tweets
C4: 19452 tweets
B1: 16379 tweets
D4: 14545 tweets
D3: 13308 tweets
C1: 8374 tweets
B4: 5454 tweets
A3: 5038 tweets
A2: 4142 tweets
C5: 4029 tweets
D5: 3163 tweets
A1: 2546 tweets
A4: 308 tweets
Order (rank) the rows based on the total number of tweets in each row
-------------------------------------------------------------------------------
C-Row: 223956 tweets
B-Row: 126683 tweets
D-Row: 31016 tweets
A-Row: 12034 tweets
Order (rank) the columns based on the total number of tweets in each column
-------------------------------------------------------------------------------
Column 2: 222108 tweets
Column 3: 97331 tweets
Column 4: 39759 tweets
Column 1: 27299 tweets
Column 5: 7192 tweets
```