Assignment 1

COMP90024

Bing Xu (833684)

Jiaqi Wang (908406)

Semester 1 2021

1 Introduction

The emerging popularity of social media platforms has hard large implications on the volumes of data that need to be routinely processed. The widespread adoption of social media has brought big data in the order of gigabytes, terabytes or even petabytes that needs to be efficiently analysed. This data can be used to gain novel insights into current sentiment, and its value to advertisers, researchers and developers cannot be overstated.

Nonetheless, the size of this data gives rise to a number of issues. Clearly, these volumes of data cannot simply be loaded into memory in their entirety and processed. Likewise, these datasets may be too large to be processed using the computing power of just a single computer. One solution to deal with this challenge is cluster and cloud computing. This report demonstrates how such an analysis of social media data can be parallelised, using a large 14 gigabyte twitter dataset.

Using the HPC facility at the University of Melbourne, SPARTAN, we calculated the sentiment score for different areas in Melbourne, with the goal of finding which area has the happiest or most miserable people. Tweets were mapped by their geographical data to regions of Melbourne demarcated in `melbGrid.json` and sentiment for each tweet was calculated using the `AFINN.txt` file as a dictionary mapping words to sentiment scores. The following report discusses the details of this implementation, the performance improvements associated with parallelisation, and which configuration of nodes and cores may yield the best performance.

2 Approach

The program is written in `Python3`, using the `MPI4Py` library, a Python implementation of the popular Message Passing Interface protocol to facilitate communication between a number of computers in parallel. The main entry point into the program is the `parser.py` script.

2.1 Data processing

One of the central issues when handling such large datasets is how the dataset should be accessed by the program. For large datasets, certain techniques, such as reading the entire file into memory using the typical python IO paradigm (where the file is read line by line), are not feasible. To address this, we instead assigned each processor a chunk of the tweets to read and be responsible for processing.

Given N nodes, C cores and a file size of F, the file was divided into even-sized chunks according to the following equation:

$$ChunkSize = \frac{F}{N \times C} \tag{1}$$

The chunk that each worker is responsible for is determined by the rank the worker's rank. For example, the first worker, i.e. the worker with rank = 0, is responsible for reading in the first chunk, which corresponds to the file beginning from $0 - ChunkSize$ bytes.

More generally, the chunk offset, aka where the worker begins reading the file from, is calculated according to the following equation:

$$Offset = Rank \times ChunkSize \tag{2}$$

From this point, the chunk was divided up again to be read into a buffer from which the tweets could then be processed. Important to note is that the division of the file into smaller portions using the aforementioned method may lead to a loss of some of the data. This is due to the boundary between two chunks potentially occurring in the middle of a tweet. As a result, these tweets which are corrupted will not be able to be processed as we have lost the relevant information needed to calculate the tweet's sentiment and determine its location.

2.2 Result aggregation

We experimented with two different approaches for communication and data aggregation between the workers. In Approach 1, we used peer-to-peer communication, where upon completion of the processing of its allocated portion of the file, an assigned master process would manually call each worker to send it their respective results. In Approach 2, the `mpi.gather` method was employed, and results were gathered into a single python dictionary. The run times of these respective implementations are presented in Figures 1 and 2.

2.2.1 Tweet processing

Furthermore, the processing of tweets was done using the python `regex` library. A regular expression was used to extract each tweet, and another was used to find from each tweet its contents (the tweet itself) and attached location data.

2.2.2 Cell assignment

We attempted to assign each tweet to a grid from `melbGrid.json` based on its geodata. A tweet was assigned to a given cell in the grid if the tweet's corresponding longitude and latitude were within the cell's boundaries, with ties for tweets lying on boundaries between grids broken as per the assignment spec. All tweets originating outside of this grid were discarded.

2.2.3 Sentiment extraction

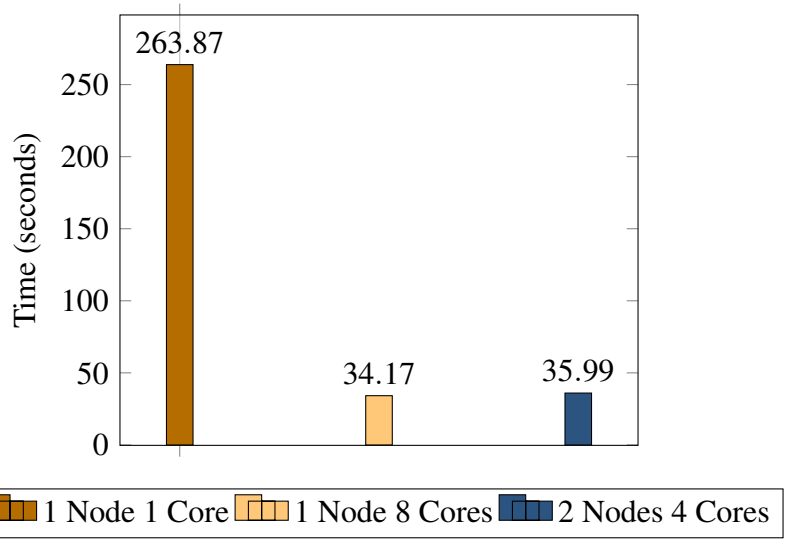
A modified version of the Aho-Corasick pattern matching algorithm was used to scan the contents of each tweet for the presence of words contained in the sentiment file `AFINN.txt`. Aho-Corasick utilises an automaton constructed from a dictionary (`AFINN.txt`), and this automaton only needs to be constructed once and can then be used to pattern match any string against words in the dictionary. The general Aho-Corasick algorithm permits non-exact matches, and so a modification was necessary. A check was added to see whether the matched pattern was an exact match before counting it as a match. Usage of this algorithm permitted the calculation of a tweet's sentiment in $O(n+z)$ time; where n is the length of the tweet and z is the total number of occurrences of words from `AFINN.txt` in the tweet. This enabled our implementation to be extremely fast.

3 Results

Table 1: Sentiment score for each region of Melbourne

Cell	#Total Tweets	Overall Sentiment Score	Average Sentiment Score per Tweet
A1	2752	+635	0.2307
A2	4904	+3829	0.7808
A3	5824	+2469	0.4239
A4	381	+47	0.1234
B1	21232	+10082	0.4748
B2	107385	+23501	0.2188
B3	34493	+17080	0.4952
B4	6642	+4864	0.7323
C1	10530	+6319	0.6001
C2	246825	+153656	0.6225
C3	69900	+34614	0.4952
C4	26096	+16843	0.6454
C5	5581	+2089	0.3743
D3	16220	+6544	0.4035
D4	16536	+8121	0.4911
D5	4705	+3068	0.6521

Figure 1: Run Time - p2p communication



3.1 Invocation

The SLURM scripts used to submit the jobs can be found in the appendix. The python script is executed as follows: `time srun -n {NUM_PROCESSORS} python3 run.py /path/to/json/`

3.2 Discussion

Figure 1 above displays the results of running the script with the three different configurations. Using 1 node and 8 cores, we recorded a run time of 34.17 seconds, approximately 7 times faster than the baseline configuration of 1 node and 1 core. This is to be expected - given this particular task falls under the class of 'embarrassingly parallelisable' problems, this improvement in run time is typical.

Perhaps more interestingly, the third configuration, consisting of 2 nodes with 4 cores each, recorded a slightly slower run time than the 1 node 8 core configuration. Despite these two configurations having the same number of cores, 2 nodes and 4 cores took marginally longer to finish execution. This can potentially be explained by the increased overhead introduced by the need for communication across nodes, as with two cores network latency now plays a role in inter-process communication. Nonetheless, in this case inter-process communication is relatively infrequent, and thus the run times of the two configurations only differ slightly.

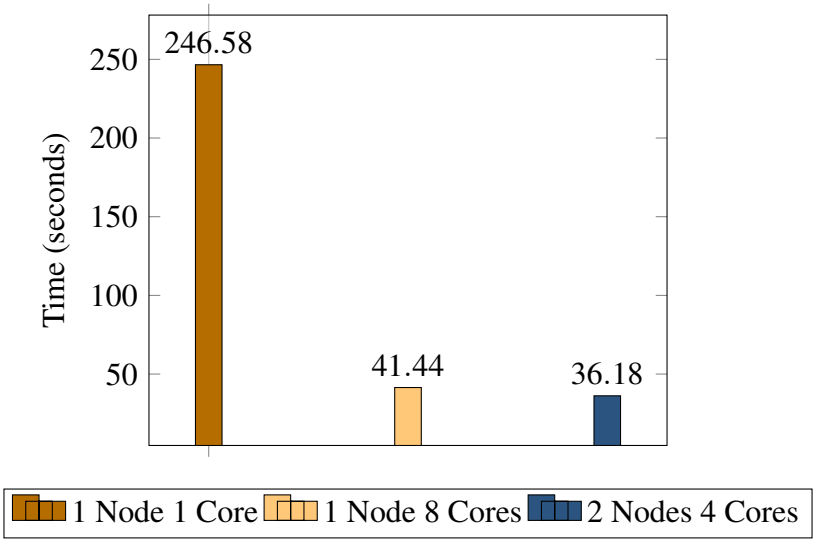
3.3 Conclusion

Finally, from Table 1, we note that the residents of the region A2 are the happiest in Melbourne, as on average, each tweet originating from A2 has the highest sentiment score. We also observe that, from Table 2 in the Appendix, the happiest regions in Melbourne appear to be those outside of metropolitan Melbourne. However, this result could also simply be due to people from these regions being under-represented on twitter.

4 Appendix

Figure 2: Run Time for the implementation using mpi.gather() to collect results

Figure 2: Run Time - mpi.gather



4.1 SLURM scripts

Figure 3: 1node1core.slurm

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:10:00
#SBATCH --ntasks-per-node=1
#SBATCH --partition=physical

echo "1 Node 1 Core"
module purge
module load foss/2019b
module load python/3.7.4
time srun -n 1 python3 run.py /home/bingx1/comp90024ass1/data/bigTwitter.json
```

Figure 4: 1node8core.slurm

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=0:10:00
#SBATCH --ntasks-per-node=8
#SBATCH --partition=physical

echo "1 Node 8 Cores"
module purge
module load foss/2019b
module load python/3.7.4
time srun -n 8 python3 run.py /home/bingx1/comp90024ass1/data/bigTwitter.json
```

Figure 5: 2node8core.slurm

```
#!/bin/bash
#SBATCH --partition=physical
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=0:10:00

echo "2 node 8 cores"
module purge
module load foss/2019b
module load python/3.7.4
time srun -n 8 python3 run.py /home/bingx1/comp90024ass1/data/bigTwitter.json
```

Table 2: Regions in Melbourne ranked by average sentiment score

Cell	#Total Tweets	Overall Sentiment Score	Average Sentiment Score per Tweet
A2	4904	+3829	0.7808
B4	6642	+4864	0.7323
D5	4705	+3068	0.6521
C4	26096	+16843	0.6454
C2	246825	+153656	0.6225
C1	10530	+6319	0.6001
C3	69900	+34614	0.4952
B3	34493	+17080	0.4952
D4	16536	+8121	0.4911
B1	21232	+10082	0.4748
A3	5824	+2469	0.4239
D3	16220	+6544	0.4035
C5	5581	+2089	0.3743
A1	2752	+635	0.2307
B2	107385	+23501	0.2188
A4	381	+47	0.1234