UNIVERSITY OF MELBOURNE

COMP90024 Cluster and Cloud Computing Assignment 1 Social Media Analytics

Chenyang Dong 1074314

Un Leng Kam 1178863

1 INTRODUCTION

This project implements a parallel program to count the number of different tweets made in the Greater Capital cities of Australia, identify the Twitter accounts (users) that have made the most tweets, and identify the users that have tweeted from the most different Greater Capital cities using a large Twitter dataset and a file containing the suburbs, locations and Greater Capital cities of Australia. The parallelisation approach is employed to increase the running time of the program, and to run efficiently given the provided memory resources on the HPC system SPARTAN.

1.1 Instructions to invoke the program

To invoke the program, copy or clone the contents of the zip file containing the source code into your home directory. Change into the `slurm` directory and it will consists of three files that allow the user to trigger the execution of the program in three different configurations. In order to invoke the program, run the command `sbatch scriptname.slurm` and replace scriptname with the filename of the script that you want to run (i.e. for 1 node - 1 core configuration, run `sbatch 1n1c.slurm`).

1.2 Assumptions on Geo-Locations

Given the geo-location data `sal.json` and the twitter files `tinyTwitter.json`, `smallTwitter.json`, and `bigTwitter.json` it is assumed that right-most name of the geo-location name string is always the state or territories of the location. Examples show in table 1.1.

Example	Description
Nelson Bay - Corlette, New South Wales	New South Wales is the right-most string
long point (campbelltown - nsw)	nsw is the right-most string

Table 1.1: Table of Geo-Location Examples

2 DATA PROCESSING

Locations of tweets in the `bigTwitter.json` may not produce an exact match in `sal.json`, location such as suburb name Richmond occurs several times in New South Wales, Victoria, South Australia, Tasmania etc and also appears as substrings of suburb names, e.g., North Richmond, Richmond Lowlands, Broad Water (Richmond Valley – NSW). Thus, such geo-location names are extracted and processed into new keys to allow matching with the geo-location of the tweets.

2.1 Pre-processing `sal.json`

This json file contains suburb names and which Greater Capital City the suburb belongs to, such information could be use for matching which Greater Capital City the tweet belongs to. However, suburb names contain in these files may not produce and exact match with the tweet location. For example, tweets may be tweeted in *"Central Coast, New South Wales"*, yet, the nearest match in `sal.json` is *"alison (central coast - nsw)"* or *"pretty beach (central coast - nsw)"*, as shown, it is unable to locate which Greater Capital City the tweet is tweeted which out further pre-processing. Hence, such suburb names is transform to new key-value pair for better matching with tweets.

Before	**After**
"alison (central coast - nsw)": "1gsyd"	("alison", "nsw"): "1gsyd", ("central coast", "nsw"): "1gsyd", ("alison", "central coast", "nsw"): "1gsyd"

Table 2.1: Table of Sal Keys Transformation Examples

As shown in table 2.1, the key-value pair is transform into multiple key-value pairs for matching, in the format (name, suburb in abbreviation/territory). Furthermore, suburbs located in rural areas are remove from `sal.json`, thus, forming a dictionary composed of suburb keys and Greater Capital Cities code value that can be use for tweet location matching.

2.2 Pre-processing `bigTwitter.json`

In this twitter json file, the same problem arises as mention above, hence, transformation is done on the suburb names. The names are first converted to lower cases, and then convert into a list of the key format (name, suburb/ territory).

Example	**Description**
"Nelson Bay - Corlette, New South Wales"	['nelson bay', ('nelson bay', 'nsw'), 'corlette', ('corlette', 'nsw')]

Table 2.2: Table of Tweet Transformation Examples

This allows the matching of tweet location to Greater Capital City easier, and without under-counting Greater Capital Cities tweet counts.

3 General Processing To Solve Tasks

For each tweet we convert its geo-location name string to the format shown in table 2.2, and match it towards the dictionary created using the technique outline in section 2.1. Where for each task we create a separate dictionary to store results, task 1 is in the format of {author id: tweet counts}, task 2 is in the format of {Greater Capital City: tweet counts}, and finally {author id: tweet counts in each Greater Capital Cities}.

4 Results For Each Task

Below presents the results for each task as required by the assignment specification, output can also be found in the output directory consisting of the same outputs by each configurations and the program run-time.

4.1 Task 1

Rank	Author Id	Number of Tweets Made
1	1498063511204761601	68477
2	1089023364973219840	28128
3	826332877457481728	27718
4	1250331934242123776	25350
5	1423662808311287813	21034
6	1183144981252280322	20765
7	1270672820792508417	20503
8	820431428835885059	20063
9	778785859030003712	19403
10	1104295492433764353	18781

4.2 Task 2

Greater Capital City	Number of Tweets Made
2gmel (Greater Melbourne)	2287625
1gsyd (Greater Sydney)	2220892
3gbri (Greater Brisbane)	860299
5gper (Greater Perth)	589771
4gade (Greater Adelaide)	466521
8acte (Australian Capital Territory)	202655
6ghob (Greater Hobart)	90891
7gdar (Greater Darwin)	46390
9oter (Other Territories)	182

4.3 Task 3

Rank	Author Id	Number of Unique City Locations and #Tweets
1	1429984556451389440	8 (#1921 tweets - 11gsyd, 1880gmel, 6gbri, 2gade, 7gper, 1ghob, 1gdar, 13acte)
2	702290904460169216	8 (#1226 tweets - 338gsyd, 263gmel, 234gbri, 128gade, 153gper, 43ghob, 21gdar, 46acte)
3	17285408	8 (#1209 tweets - 1061gsyd, 60gmel, 40gbri, 3gade, 7gper, 11ghob, 4gdar, 23acte)
4	87188071	8 (#402 tweets - 116gsyd, 86gmel, 65gbri, 29gade, 52gper, 15ghob, 5gdar, 34acte)
5	774694926135222272	8 (#272 tweets - 37gsyd, 38gmel, 37gbri, 28gade, 34gper, 36ghob, 28gdar, 34acte)
6	1361519083	8 (#266 tweets - 18gsyd, 36gmel, 1gbri, 9gade, 1gper, 2ghob, 193gdar, 6acte)
7	502381727	8 (#250 tweets - 2gsyd, 214gmel, 8gbri, 4gade, 3gper, 8ghob, 1gdar, 10acte)
8	921197448885886977	8 (#208 tweets - 49gsyd, 58gmel, 37gbri, 24gade, 28gper, 4ghob, 1gdar, 7acte)
9	601712763	8 (#146 tweets - 44gsyd, 39gmel, 11gbri, 19gade, 14gper, 8ghob, 1gdar, 10acte)
10	2647302752	8 (#80 tweets - 13gsyd, 16gmel, 32gbri, 3gade, 4gper, 5ghob, 3gdar, 4acte)

5 Approach to Paralleling

5.1 Initial Attempt

Our initial approach to implement a parallel program by utilising the python library `ijson` and `mpi4py`. Given the twitter json file would be too large to load into memory, such task is solve through the parallelisation approach. Using python's file operation syntax `with open('bigTwitter.json', 'rb')` is used, where 'rb' denotes to read in bytes which allows `ijson` to operate in. The method `ijson.items()` loads the json file one object at a time, by utilising such functionality of the method, memory resource limitation is mitigated. Given we are able to access the id of each node, as we iterate through each item a count is assigned to the item, `if count % size == rank` the json item is assigned to node with the rank id and processed by that node. However, there remain parts that could be parrallelised, since such method still requires every core to iterate every item. Hence, going from 1 node 1 core to 1 node 8 core was only a minute improvement.

5.2 Improved Implementation

Given the slight improvement in our initial attempt, we begin to think of parrallelisation solutions that resolves such issues. The main issue was that our initial approach requires to loop through every json object and assign the object to each core for processing, thus, it could be resolve by identifying sections of the file and assign such sections to each core for processing. Hence, reducing the run-time of the program as each core iterates lesser json objects, and we are able to process the whole file in parallel.

By employing `f.seek()`, `f.tell()`, and `os.path.getsize()`, such approach can be accomplished. Through `os.path.getsize()` we are able to obtain the file size of the twitter file and an approximate file chunk bytes range given the number of cores through integer division and multiplication with rank id. With the received chunk size, we are able to use `f.seek()` to locate the beginning byte of the core's corresponding chunk, and we iterate through each line of the chunk using `f.readline()` and applying string concatenation. For each json object closing bracket is met, we try to load the previous string concatenation as a json object and process it as a tweet to solve the three tasks, and finish when `f.tell()` returns we have met the end of our respective chunk. However, as we are seeking for bytes of the file, some cores may begin in the middle of a json object, hence, `try-except` is utilised when we try to `json.loads()` a string concatenation and the fear of ending in the middle of a json object is mitigated as an if condition is set in-place to ensure ending with a correct closing string of a json object.

6 Runtime Results

6.1 Discussion

The program's runtime results under different configurations are presented in figure 6.1. The '1 node 1 core' took the longest and completed the execution in 786.932 (13m7.932s). The '1 node 8 cores' took 105.295 seconds (1m45.295s) and '2 nodes 8 cores' took 102.549 seconds (1m42.549s) which are approximately the same. Both 8 cores configurations are around 7.5 times faster than 1 core, which is slightly less than the ratio of cores as ideally 8 cores would be 8 times faster than 1 core, according to Amdahl's law. This can be explained by the time consumed by the MPI module communicating the data between the root core and other working cores when we gather the results from other cores. However, the 1 node 1 core configuration is taking such a long time is also because of the usage of the code `f.tell()` which checks everytime an ending symbol of a json object is seen, which takes up a lot of time, as seen by the comparison of the program with '1 node 1 core' using the ijson approach which runs around 2.2 times faster when there is no `f.tell()` method used.

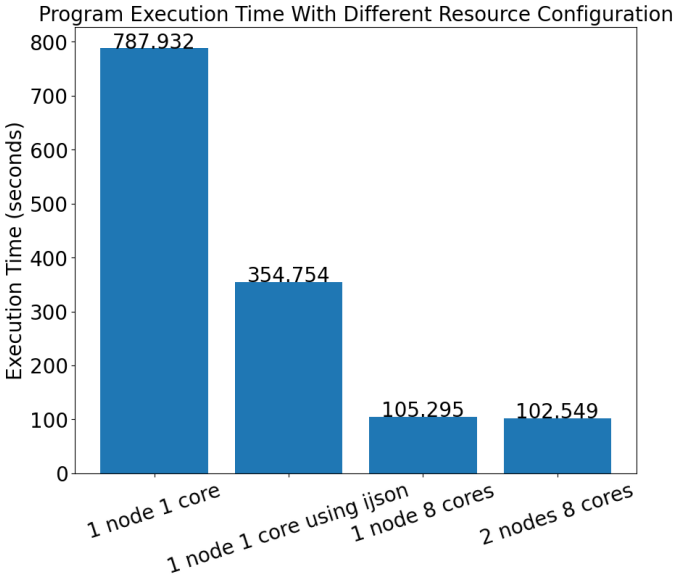

Figure 6.1: Execution time with different resource configurations (using 'seek' approach unless specified)

Moreover, the execution took slightly longer with 1 node and 8 cores than with 2 nodes and 8 cores. The communication overhead between the nodes in a distributed system may be higher than the communication overhead between cores in a single node. This could account for the slight disparity in execution time between two configurations. Based on the results, it is clear that increasing the number of cores can reduce execution time for such issues, as long as the solution is efficient and fully utilizes parallelization, as demonstrated by the significant difference in improvement outlined in subsections 5.1 and 5.2, regardless of how the cores are distributed across nodes. However, given the necessary communications between cores and nodes, and processes that couldn't be parallelised such as combining results or sorting the results from each core, the extend parallelisation can improve the program's execution time depends on the developer's implementation and the nature of the problem.