COMP90024 Cluster and Cloud Computing Assignment 1 Report

Lang (Ron) Chen 1181506 and Ying Zhu 1174301

Introduction

This application is designed to gather geographic location information and create three tallies from a large set of data, with the ability to be run as a paralleled program. The three tallies to be presented are:

1. The author ID of the top 10 tweet authors who tweeted the most and their respective number of tweets (sorted in descending order by number of tweets);

2. The respective number of tweets made at each Greater Capital City (tweets made at rural area are excluded) in descending order;

3. The author ID of the top 10 tweet authors who have made tweets from the most number of Greater Capital Cities (GCC), along with detailed information, including how many distinct GCCs they posted from, the total number of tweets they made at GCC areas and the number of tweets they made at each individual GCC. (Sorted in descending order by distinct GCCs posted from, with a tiebreaker being the total number of GCC tweets.)

The parallelisation functionality was implemented via the mpi4py (Message Passing Interface for Python) package, which allows processors to send packets of information in between themselves. The performance of the application was tested using an 18 gigabytes 'BigTwitter.json' dataset on the University of Melbourne HPC system SPARTAN. Its speed performance was evaluated on '1 node 1 core', '1 node 8 core' and '2 nodes 8 cores' (4 cores per node).

Note that the terms "core" and "processor" are used interchangeably in this report, with "core" referring to hardware and "processor" referring to theory.

Instructions for invoking Application

A precondition for running this application is to upload the three slurm files and the solution python file within this package to the same directory on the Unimelb HPC. Additionally, the Twitter dataset to be processed and the sal.json file, which contains the suburbs, locations and Greater Capital Cities in Australia, should also be in the same HPC file directory.

To invoke the application, one should use the following commands:

```
1 node 1 cores: sbatch 1n1c.slurm

1 node 8 cores: sbatch 1n8c.slurm

2 nodes 8 cores: sbatch 2n8c.slurm
```

The slurm scripts contain requests for required computational resources (i.e. nodes = number of nodes; ntasks = number of cores, etc.), as well as invoking the python application with the appropriate files.

Application implementation details

The python code was developed in a local setting using the smallTwitter.json dataset and the sal.json data file downloaded from the assignment page on the LMS. For each tweet, we collect the tweeter's author id and the location (sal) by using regular expressions to search for this information line by line, thereby avoiding memory exceptions

For all tasks, if a tweeter's id or location information (or both) is missing, it does not contribute to the tally of any of the three tasks.

During the development process of task 2 and 3, the problem of ambiguous sal names arose, where one sal name may have appeared in several states. To resolve this, a separate mapping dictionary for ambiguous sals was created. The keys are the ambiguous sals, and the values are dictionaries containing the state names as keys and the original sal name in the sal.json file as values. In cases of ambiguity, the application uses the state part of the geolocation information to resolve it. During this process, sals that do not belong in Greater Capital Cities (regardless of ambiguity status) are removed for ease of developing the solution and also in keeping with the spirit of removing as much unnecessary data as possible to reduce processing time on the big data.

Method of Achieving Parallelisation

As aforementioned, parallelisation was achieved using the python mpi4py package. When the application is invoked, each processor runs the whole script from start to end, subject to their index within the cluster. Each processor first creates an MPI.COMM_WORLD object *comm*, which tells them what processor index (*rank*) they are within the cluster by comm.Get_rank(), and also the total number of processors in this cluster (*size*) by comm.Get_size().

Then, the total size of the file (in terms of number of bytes) to be processed is recorded, and implicitly divided into *size* near-equal sections through integer division, giving the byte indexes that separate each adjacent block of the file. These are known as approximate boundaries (each block gets an approximate start and approximate end byte index). The intention is to use python's inbuilt functions seek() to jump to the appropriate start of a block of data a given processor is supposed to process, based on their *rank*, so that it does not have to iterate over all previous file lines which are supposed to be processed by lower-indexed processors; and also terminate the processing function when tell(), which returns the current byte index of the '\n' character in every line read, is greater or equal to the index of the end byte index. However, as each primary json object (denoting one tweet) in the file has a different byte length, integer division cannot guarantee the first and last primary json objects read by the processor are whole objects not divided between processors. Failing to address such problems may lead to undercount, as the program does not accept a tweet in the tally unless it has both author id and appropriate geolocation information.

To resolve this issue, a helper function is built to calculate the approximate boundaries, using an important property of .json files in that the lines containing the end of each primary object is either *' },\n' or ' }\n'*. For each block, we begin to read lines from the approximate starting byte index, and record the byte position as the accurate start using tell() when we reach the first line denoting an end of primary object; we similarly do this for the approximate end byte index to get the accurate end. This way, each processor is guaranteed to have been reading full json objects.

When the program is actually processing each line of the file using readline(), after starting at accurate start byte index using seek(), at every line which is an end of the primary json object, the program will use the tell() function to check whether the byte index of the '\n' in that line has equaled or exceeded the accurate end byte index. If the aforementioned condition is satisfied, then this processor stops processing the file and sends its tallies back to the 0th processor using gather() .

The 0th processor will collect what was sent by all other processors using gather() as three lists of dictionaries where the output of each core is placed in the list at its indexed place. A function is then used on only the 0th processor to compile the comm.gathered dictionaries into one overall dictionary so that the final results can be outputted.

Application output on BigTwitter.json

Rank	Author Id	Numbers of Tweets Made
#1	1089023364973219840	28127
#2	826322877457481728	27581
#3	1250319342421237776	25263
#4	1423662808311287813	21030
#5	1183144981252280322	20651
#6	820431428835885059	20063
#7	1270672820792508417	19801
#8	233782487	18179
#9	84958532	15676
#10	719139700318081024	15314

Figure 1: *Output for Task 1: count the number of tweets made by the same individual based on the "bigTwitter.json" file and return the top 10 tweeters in terms of the number of tweets made*

Greater Capital City	Numbers of Tweets Made
2gmel (Greater Melbourne)	2267051
1gsyd (Greater Sydney)	2114698
3gbri (Greater Brisbane)	855448
5gper (Greater Perth)	589204
4gade (Greater Adelaide)	449292
8acte (Greater Canberra)	194360
6ghob (Greater Hobart)	89970
7gdar (Greater Darwin)	46376

irrespective of where they tweeted

Figure 2: *Output for Task 2: count the number of tweets made in the various capital cities by all users based on the "bigTwitter.json" file, listed in descending order.*

Rank	Author Id	Number of Unique City Locations and #Tweets
#1	1089023364973219840	8 (#1908 tweets - #10gsyd, #1868gmel, #6gbri, #2gade, #7gper, #1ghob, #1gdar, #13acte)
#2	826322877457481728	8 (#1189 tweets - #1041gsyd, #60gmel, #40gbri, #3gade, #7gper, #11ghob, #4gdar, #23acte)
#3	1250319342421237776	8 (#1121 tweets - #288gsyd, #246gmel, #219gbri, #113gade, #152gper, #39ghob, #20gdar, #44acte)
#4	1423662808311287813	8 (#390 tweets - #108gsyd, #84gmel, #64gbri, #28gade, #52gper, #15ghob, #5gdar, #34acte)
#5	1183144981252280322	8 (#260 tweets - #12gsyd, #36gmel, #1gbri, #9gade, #1gper, #2ghob, #193gdar, #6acte)
#6	820431428835885059	8 (#250 tweets - #2gsyd, #214gmel, #8gbri, #4gade, #3gper, #8ghob, #1gdar, #10acte)
#7	1270672820792508417	8 (#198 tweets - #46gsyd, #56gmel, #37gbri, #20gade, #28gper, #4ghob, #1gdar, #6acte)
#8	233782487	8 (#146 tweets - #44gsyd, #39gmel, #11gbri, #19gade, #14gper, #8ghob, #1gdar, #10acte)
#9	84958532	8 (#80 tweets - #13gsyd, #16gmel, #32gbri, #3gade, #4gper, #5ghob, #3gdar, #4acte)
#10	719139700318081024	7 (#2584 tweets - #403gsyd, #1547gmel, #91gbri, #47gade, #186gper, #84ghob, #226acte)

Figure 3: *Output for Task 3: identify the top 10 tweeters that have tweeted in the most Greater Capital cities and the number of times they have tweeted from those locations based on the "bigTwitter.json" file, listed descending order.*

Performance Results

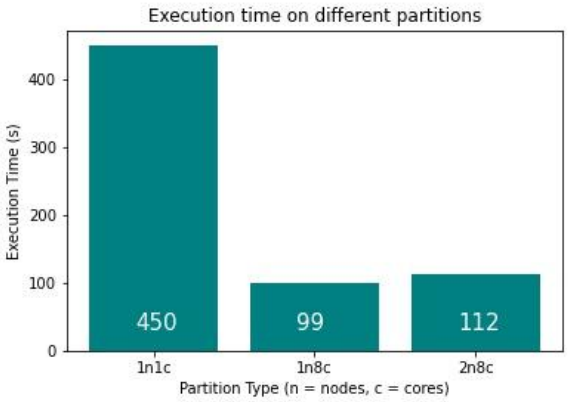

Figure 4: *Bar chart of Execution time on different partitions*

The execution time of the three different partitions are presented in the following table:

Partition Type	Partition Type Abbreviation	Execution Time (in seconds)
1 Node 1 Core	1N-1C	450
1 Node 8 Cores	1N-8C	99
2 Nodes 8 Cores (4 cores each node)	2N-8C	112

1N-1C's execution time was the longest out of the three, which was expected since it is effectively a serial application that should run slower than its parallelised counterparts. This is because the primary task it performs is a loop over the primary json objects containing tweet information, which is highly parallelizable.

1N-8C holds the shortest execution time, which is 13 seconds less than 2N-8C. This can be explained by the fact that communications between processors on the same node should be faster than communicating between processors across different nodes. In other words, 2N8C's execution time was longer than 1N8C's even though they both utilised 8 processors, because 2N8C had higher overhead costs for parallelisation from having to send and receive information across different physical nodes.

However, the speedup of 1N-8C did not reach 8 times, even though it had 8 times the computational power compared to 1N-1C. If it did, it should have had an execution time of 56.25s. Instead, at 99s, it is only 4.54 times faster than the serialised application. (Using the result of 1N-1C, which we chose to as the better representative of the 8 processor execution time as it is not affected by the cross-nodes communication costs, T(8) = 99, T(1) = 450 and S(8) = 4.54 for this particular program in the context of the Amdahl's Law).

This phenomenon can be explained by the fact that the non-parallelizable parts of the application, such as sorting the tallies to present the top 10, and the overhead costs of parallelising, such as gathering the data, establish an upper bound on how much speed reduction an application can achieve by increasing the number of processors and leveraging parallelisation. This phenomenon is best represented by Amdahl's law.

Using Amdahl's law, we estimate that 1/4.54 = 0.22 of the execution time in this program (using 8 cores on 1 node on the Spartan HPC) is not parallelisable.