**COMP90024 Cluster and Cloud Computing Assignment 1 Report**

**Lang (Ron) Chen 1181506 and Ying Zhu**

**Introduction**

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

1. Author ID of the Tweet authors of the top 10 Tweet authors who tweeted the most and their respective number of tweets (sorted from highest to lowest by number of tweets);

2. The number of tweets made at each Greater Capital City (rural tweets excluded) and the respective number of tweets;

3. The Author ID of Tweet authors of the top 10 Tweet authors who has made tweets from the most number of Greater Capital Cities (GCC), and detailed information including: how many distinct GCCs they posted from, the total number of Tweets they made in each GCC and the number of tweets they made at each individual GCC. (Sorted from highest to lowest by distinct GCCs posted from, with tiebreaker being total number of GCC tweets.)

The parallelisation functionality was implemented via the mpi4py (Message Passing Interface for Python) package, which allows nodes to send packages of information between themselves. The performance of the application was tested using an 18 gigabytes ‘BigTwitter.json’ file 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 each node).

Note that core and processor is used interchangeably in this report - core for hardware, processor for theory

**Instructions for invoking application**

A precondition for running this application is uploading the three .slurm files and the .py file within this package onto the Unimelb HPC, into the same directory. The twitter file to be processed and the sal.json file should also be in the same HPC file directory.

To invoke the application, one should call the command:

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 other information), as well as invoking the python application with the appropriate files.

**Application implementation details**

The python code was produced in a local setting using the smallTwitter.json data file and the sal.json data file downloaded from the assignment page on the LMS. For each tweet, we collect the tweeter’s id and the location (sal) by using regular expressions to look for such information on each line which our programme tallies.

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

For task 2 and 3, the problem of ambiguous sal names was faced in the development process, whereby one sal name may have appeared in several states. To resolve this, a separate mapping dictionary for the ambiguous sals were built, where the key is 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 output. In cases of ambiguity, the application uses the state part of the geolocation information to resolve. During this process, sals that do not belong in Greater Capital Cities (regardless of ambiguity status) are removed for ease of coding 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 by using the python mpi4py package. When the application is invoked, each processor will run the whole script from start to end, subject to their index within the cluster. Each processor first create a 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().

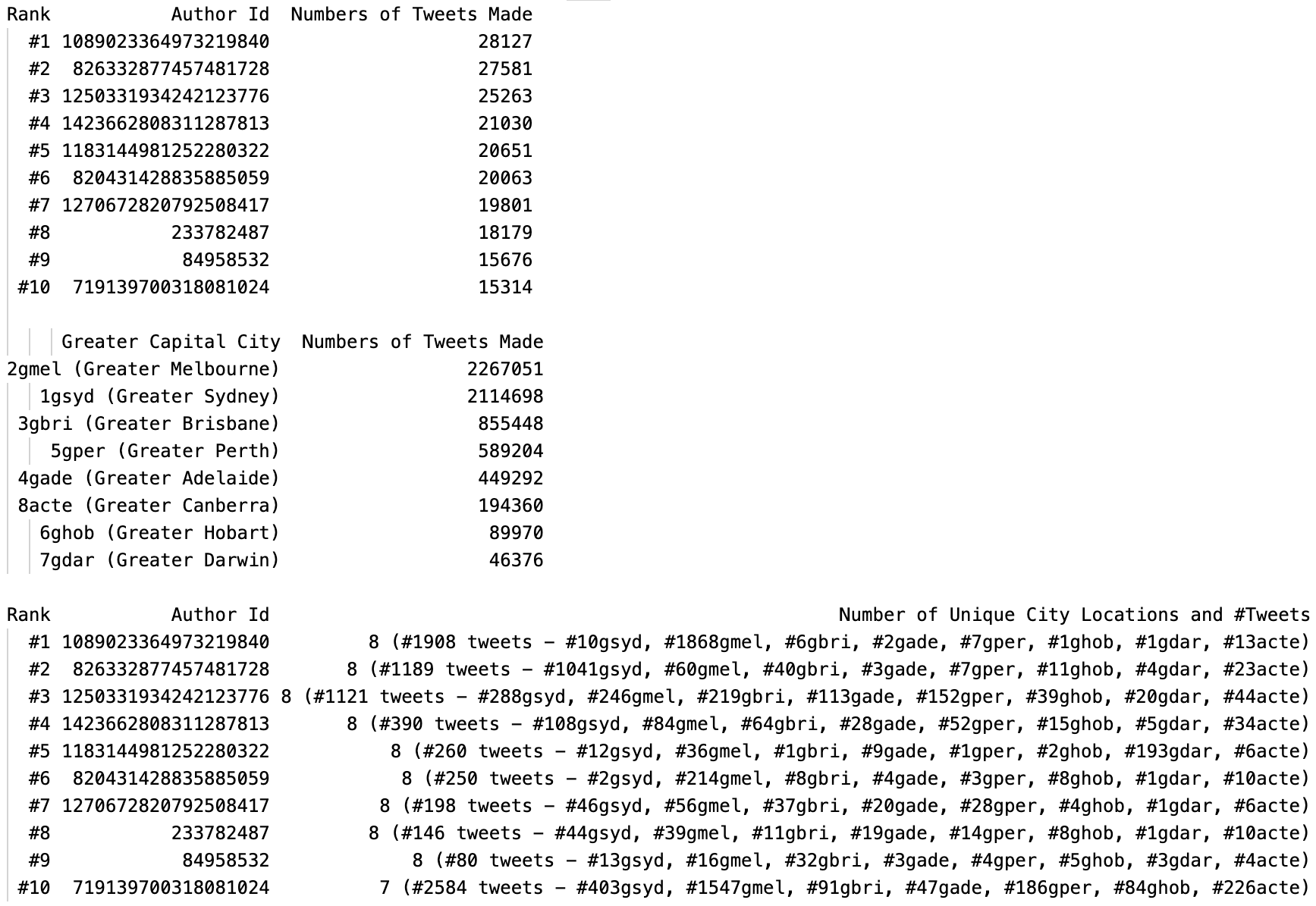
Then, the total size of the file (in terms of number of bytes) to be processed is recorded, and implicitly divided into *size* near-equal sections through using integer division, giving the byte indexes which separates each adjacent block of the file. These are known as approximate boundaries (each block gets an approximate start and approximate end byte index). The intention is to use python 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 doesn’t 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 byte index of the ‘\n’ character in every line read, is greater or equal to the index of the upper boundary 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 is a whole object which is not divided between processors. Failing to address such problems may lead to undercount, as the programme does not accept a tweet in the tally unless it has a user\_id and appropriate geolocation information.

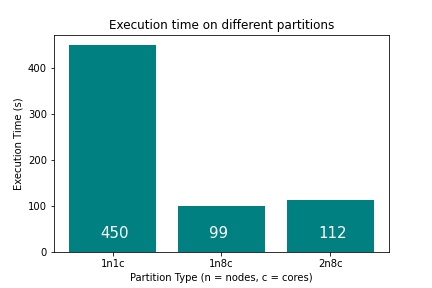
To resolve this issue, a helper function is built to calculate the approximate boundaries, using an important property of .json files in that the lines containing the end of each primary object is either *‘ },\n’ or ‘ }\n’*. For each block, we begin to read lines from the approximate starting byte index, 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 programme 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 the file back to the root processor using gather().

Once each processor completes iterating through the file and hence collecting the statistics into two dictionaries, the comm.gather() function makes each processor send its gathered information back to the 0th (our set root) processor, which the 0th processor will collect up 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**



**Performance Results**

*Figure 1. 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 |
| --- | --- | --- |
| 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, and expectedly so as it is effectively a serial application which should run slower than its parallelised counterparts, as the primary task it performs is a loop over tweet information which is highly parallelizable. 1N-8C holds the shortest execution time, 13 seconds less than 2N-8C, which is also explainable and expected as 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’s overhead costs for parallelisation was higher from having to send and receive information across different 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 at 56.25s; but instead at 99s 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 (i.e. sorting the tallies to present the top 10) and the overhead costs of parallelising (i.e. gathering the data) establishes an upper bound to how much speed reduction an application can achieve by increasing the number of processors and leveraging parallelisation. This phenomenon is best represented by Amhdal’s Law.

Using Amhdal’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.