**COMP90024 Assignment 1 Report**

COMP90024 assignment 1 requires us to process a large data file called "bigTwitter.json" and extract relevant information about "author\_id" and "full\_name". We will also perform three simple data processing tasks using sal.json. The project will require the use of parallel computing methods, MPI, the use of the HPC (Spartan) platform provided by the University of Melbourne and was tested on three different resource setups, 1 node and 1 core, 1 node and 8 cores, and 2 nodes with 8 cores.

Our report will be divided into several parts: reading the large data file, parallel computing model selection and its implementation, submitting tasks to Spartan, and analyzing the final results.

## 1.Reading the large data file

Initially, we made a foolish attempt to read the entire file, which resulted in an "out of memory" error. Afterwards, we decided to split the large file into several chunks, each of which could be successfully read. There were two difficulties with this approach. First, how to ensure that each chunk contained several complete tweets, thus ensuring the integrity of the data. Second, how to find the offset of beginning of each tweet with the least amount of time.

Table

Description automatically generated with low confidenceTo address the first issue, we traversed the entire "bigTwitter.json" file after reading it, found the beginning of each tweet ("\_id:" represents the start of each tweet), and stored their offsets in a list, as long as we choose to split the file based on these offsets, the data will not be missing(Figure 1),after that, we assign the number of chunks for the program by hand, in this project, we set the chunk size to 300. As for the second issue, we used the method **open("file\_name", "rb")** to directly read and compare the byte data, thus avoiding the decoding process. This enabled us to traverse the 18.7GB large data file at the fastest speed possible. The output showed that this method took only 40 seconds to find the offset of the beginning of each tweet.

Figure File reading

## 2. Parallel computing

### 2.1 Parallel computing model selection

Based on the knowledge from the lecture, we plan to choose between two parallel computing models. The first is the Master Worker/Slave Model, and the second is the Single-Program Multiple-Data In Model 1, one core is used as the master and the other cores act as slaves. The master is responsible for sending and receiving information from other cores, continually assigning tasks to different cores, and receiving and integrating results from other cores. Slave cores are responsible for receiving and computing tasks from the master core and returning the results to the master core. In Model 2, all tasks are equally distributed among all cores, and results are stored locally until the tasks are completed, and then they are gathered.

After consideration, we decided to use Model 2 for two reasons. First, for this project, the task workload of data processing is relatively evenly distributed. When all cores work together, we can obtain the optimal time. Second, since we divided the file into 300 parts, the master core will continually send tasks to slaves and receive results from them, which will increase unnecessary time. If we need to process several files of unknown sizes, with large files like bigTwitter.json and small files like tinyTwitter.json, Model 1 would be a better choice as it will help us reduce waiting time.

### 2.2 Method implementation

We use Python as the programming language for our project, and call mpi4py to implement MPI functionality. First, the program initiates the MPI world and obtains environment variables such as core sizes and core ranks using **MPI.COMM\_WORLD**, **comm.Get\_rank()**, and **comm.Get\_size()**.

After obtaining the number of cores, we use **comm.scatter()** to evenly distribute several chunks (in the actual project, there are 300) to eight cores. This way, all cores will only read their own parts from the bigTwitter.json file, each of which is composed of several chunks, eliminating out of memory errors. Additionally, since all cores run the program simultaneously, read the file, extract data, and accumulate and store results concurrently, we have implemented the idea of parallel computing.

Finally, we use **comm.gather()**, with root set to rank 0, to have it collect and store results from other cores, sort and print the final results.

## 3. Task submittion on Spartan

The example submit script is shown below, which is the specification of one node eight cores:

1. #!/bin/bash
2. #SBATCH --nodes=1
3. #SBATCH --ntasks-per-node=8
4. #SBATCH --time=0-00:10:00
5. module load mpi4py/3.0.2-timed-pingpong
6. module load python/3.7.4
7. srun -n 8 python3 As1.py
8. ##Job monitor command to list the resource usage
9. my-job-stats -a -n -s

Line1 known as "shebang" and indicates that this is a Bash script.

Line2,3,4 are SLURM directives, which are used to specify the configuration of the job. In this case, the job is configured to use one node (--nodes=1) with 8 tasks per node (--ntasks-per-node=8), and the maximum time allowed for the job is 10 minutes (--time=0-00:10:00).

Line 5,6 load the necessary modules for MPI and Python. Line5 loads the mpi4py module with a specific version that supports timed pingpong tests, while line6 loads Python version 3.7.4.

Line7 uses the "srun" command to launch our Python program "As1.py" with 8 MPI tasks (-n 8). This means that the program will be run in parallel with 8 processes.

Line8 just comment

Line9 monitor the resource usage of the job.

To submit the job on spartan, we use command: **sbatch one\_node\_eight\_cores.slurm**, after task submitted, we use command: **squeue -u user\_name** to show our submitted task’s status or cancel the task use **scancel task\_id**.

## 4.Final results analysis

### 4.1 Task result

Task 1 counts the number of authors and their associated cities in the data using regular expressions and a pre-defined dictionary of city names. Tasks 2 and 3 extract the author name, place, and state name from the tweets. The program then checks the extracted place and state name against a JSON file (sal.json) that maps locations. If a match is found, the corresponding geographic coordinate is used to determine which “gcc”region the tweet is from. Finally, the program counts the number of tweets from each “gcc” region and outputs the results. The program ignores tweets from outside of Australia, unmatched places in sal.json, and rural places.

Our application has analyzed the bigTwitter.json file and determined that the author with ID 1498063511204761601 has posted the most tweets, with a total count of 68,477 (including rural locations). This author has also posted at 8 unique locations. Additionally, our application has identified that the location with the highest number of tweets is Greater Melbourne, with a total count of 2,267,051 tweets made.

### 4.2 Performance on different nodes and cores

The data in Figure2 shows that using either one or two computer nodes, each with eight cores, resulted in significantly faster processing times compared to using only one node with one core. Specifically, using one node with eight cores was approximately 3.7 times faster than using one node with one core, and using two nodes with eight cores was approximately 3.2 times faster than using one node with one core.

Based on the choice of parallel model and implementation, we can provide a reasonable explanation for the results. Compared to a single core, an eight-core configuration is like dividing the 18.7GB bigTwitter.json file into eight equal parts, with each core only needing to process one-eighth of the original file. In contrast, a single core would have to process the entire file alone. Even though the eight-core mode incurs additional time for the scatter and gather of results, due to the file's large size, the eight-core mode still results in a significantly shorter runtime than the single-core mode.

Additionaly, the two node and eight core configuration was slightly slower than the one node and eight core configuration, with a processing time improvement of approximately 1.1 times. This is likely due to the additional communication and coordination required between the 2 nodes, which results in a longer runtime.

Overall, the results demonstrate that parallelizing the code using MPI significantly improves the runtime performance of the application.

Figure

### 4.3 Exploration and Optimization

During the process of testing the project's runtime, we discovered several interesting points. Firstly, in some cases, the "one node, one core" configuration completed the task faster than the "one node, eight cores" configuration. Upon analyzing the results, we found that the CPU usage of the "one node, eight cores" configuration was significantly lower than that of the "one node, one core" configuration in that particular experiment. The CPU usage of all cores was less than 50%, and even 25% of the CPU usage was less than 25%. This tells us that, due to bandwidth or other hardware issues, multi-core processing may sometimes be slower than single-core processing.

Secondly, we found that the size of the chunk did not have a significant impact on the execution time. This is because regardless of how the chunk size varies, each core needs to complete the same amount of work, which is always to read one-eighth of the file. We just need to ensure that the chunk size is not too large to cause an out-of-memory error.

There is still potential for further optimization in the program. When gathering the results, we might be able to use algorithms and write our own gather method, using "comm.reduce(own\_function)" to collect our results.

In conclusion, splitting the file into chunks and processing them in parallel is an effective approach to handling large datasets when memory is a limiting factor. This approach allows us to work with large amounts of data without running into memory limitations. We observed a speedup of approximately 3.7 times on 1 node and 8 cores and 3.2 times on 2 nodes and 8 cores. Additionally, we observed a slight overhead in completing the job on two different nodes with four cores each compared to using all cores on a single node. Overall, our experiment demonstrates the importance and effectiveness of parallelizing code using MPI for large-scale data processing tasks. By distributing the workload across multiple nodes and cores, the runtime of the application can be greatly reduced, making it possible to process large amounts of data in a reasonable amount of time.