


COMP90024 Cluster and Cloud Computing

ASSIGNMENT 1 REPORT

Josh Feng – 1266669 - chenhaof@student.unimelb.edu.au

Fan Yi – 1193689 - Yify@student.unimelb.edu.au

2025 SM1



1. Introduction

The University of Melbourne's Spartan is a high-performance computing platform that allows us to utilize parallel computing resources to process and analyze data. This project aims to parallelize sentiment analysis of a large Mastodon social network dataset and find the Top 5 results for the following four indicators:

- The 5 happiest hours in the data and the 5 saddest hours in the data
- The 5 happiest users in the data and the 5 saddest users in the data

We used the python language, adopted MPI to implement parallel processing, and finally successfully read and processed a 144g file in NDJSON format. And we test it on Spartan platform under three different resource configurations, 1 node 1 core, 1 node 8 cores and 2 nodes 8 cores.

2. Design

2.1 Data

The Mastodon dataset collects user post data captured on social networking platforms. This data is stored in NDJSON format, where each row is a separate post record, and the same specification are stored for each row. We will mainly analyse the sentiment scores according to two different categories:

- By hour:
 - o Sentiment scores (*doc.sentiment*) were accumulated hourly by extracting the posting time (*doc.createdAt*) of each post to find the top 5 happiest and saddest hours.
 - o We aggregated the data on hourly basis. For example, 2025-01-31T02:41:57.000Z will be converted to the hourly interval it belongs to, i.e. 2025-01-31 2am to 3am. We will then add the sentiment value of that post to the cumulative sentiment for that hour interval.
- By user:
 - o We used user ID (*doc.account.id*) and username (*doc.account.username*) as unique identifiers, the sentiment scores of all posts by the same user are accumulated to determine the top 5 happiest and saddest users.
 - o Note that, only if *doc.account.id* and *doc.account.username* are exactly the same, we recognise this is the same user.

We processed a 144g file, and for each data line, we only extracted *doc.createdAt*, *doc.sentiment*, *doc.account.username* and *doc.account.id*. if any line of data was missing any of this data, we simply drop it because it is incomplete.

2.2 Algorithm

We mainly used the python language for data processing, and we took advantage of the multi-core and multi-node resources of the Spartan HPC platform by using mpi4py library for parallel processing, mpi4py is python wrapper for MPI which allow us to call MPI function in python in order to run the code in parallel on multiple cores. Our algorithm can divide into two main steps.

```

1 total_lines = None
2 start_line = None
3 end_line = None
4
5 # core 0 (master core) calculate how many line will be in the file
6 if current core is 0 Then
7     total_lines = count_total_line(file)
8 end
9
10 # core 0 broadcast to other cores with total_lines
11 total_lines = BROADCAST(total_lines, root=0)
12
13 # each core calculate start_line and end_line it needs to process
14 # base on the rank (Own number) and size (total number of cores)
15 (start_line, end_line) = CALCULATE_LINE_RANGE(rank, total_lines, size)
16
17 output start_line, end_line

```

Figure 1: pseudocode for algorithm of parallel preprocess data

Figure 1 shows the parallel preprocess the data file. We first using master core count the total number of lines in the file using the system command `wc -l`, we call this command directly in the python file, and it quickly counts the number of lines. After that the master core will broadcast the total lines of file to the other cores, and each core will calculate its own interval of lines number that should be processed based on its own rank, total lines of file, and the total number of cores, that will ensuring the entire file is evenly divided and processed. The formula for calculating core's own start_line and end_line:

- Case 1: single core
 - o start_line = 0, end_line = total_lines-1
- Case 2: multi cores
 - o Core rank is 0:
 - start_line = 0, end_line = (total_lines // size)
 - o Core rank between 0 and last:
 - start_line = rank_id * (total_lines // size) + 1
 - end_line = (total_lines // size) * (rank_id + 1)
 - o Core rank is the last one:
 - start_line = (size-1) * (total_lines // size) + 1
 - end_line = total_lines-1

```

1 input rank, start_line, end_line, file_path
2 local_sentiments_hour = {}
3 local_sentiments_people = {}
4
5 open file_path As file
6 For line in file:
7     IF (line_number < start_line) Then
8         CONTINUE
9     IF (line_number >= end_line) Then
10         BREAK
11     record = PROCESSING_CURRENT_LINE(line)
12     IF record is True Then
13         local_sentiments_hour[record.hour] = local_sentiments_hour.get(record.hour, 0) + record.sentiment
14         local_sentiments_people[record.user] = local_sentiments_people.get(record.user, 0) + record.sentiment
15 end
16
17 all_sentiments_hour = comm.reduce(local_sentiments_hour, op=merge_together, root=0)
18 all_sentiments_people = comm.reduce(local_sentiments_people, op=merge_together, root=0)
19
20 If rank is 0 Then
21     happiest_hours = TOP_N(all_sentiments_hour, n=5, reverse=True)
22     saddest_hours = TOP_N(all_sentiments_hour, n=5, reverse=False)
23     happiest_people = TOP_N(all_sentiments_people, n=5, reverse=True)
24     saddest_people = TOP_N(all_sentiments_people, n=5, reverse=False)
25
26 output happiest_hours, saddest_hours, happiest_people, saddest_people

```

Figure 2: pseudocode for algorithm of parallel top-n calculation

The figure 2 is parallel top-n calculation. Each core opens the file independently and reads the data line by line in streaming fashion, but each core will only process the data that own to itself based on pre-calculated line number interval and will skip the rest of lines. Each core has two dictionaries, one for sentiment scores by hour, using the time interval as the key and the value as the cumulative sentiment value for that hour period. Another one for sentiment scores by user, using the username and id as the composite key and the value as the cumulative sentiment value for that that user. Finally, the master core will collect and merge the dictionaries from the other cores and get the top-n hours and users by sorting the sentiment value.

We used script to submit jobs to Spartan, the appendix shows three scripts that invoke different number of nodes and cores per job. For each script, there is “#SBATCH –nodes” which specifies the number of nodes requested, and “#SBATCH –ntasks-per-node” which specifies the number of tasks running on each node, each task is bound to one core. Also, all the task running under *cascade* partition, if there are tasks running under other partition, we will terminate this task and re-execute it.

3. Discussion

After executing the task, we found that the 144g ndjson file have a total of 42,000,000 lines of data, which 2 lines of data lack key values so they are invalid.

```
===== The 5 Happiest Hours =====
12-1am on 1st January 2025 with an overall sentiment score of +206.1521
11-12pm on 31st December 2024 with an overall sentiment score of +187.4655
5-6am on 1st January 2025 with an overall sentiment score of +135.9252
10-11pm on 24th December 2024 with an overall sentiment score of +117.2863
3-4pm on 25th December 2024 with an overall sentiment score of +114.6020

===== The 5 Saddest Hours =====
7-8am on 6th November 2024 with an overall sentiment score of -373.7672
1-2am on 11th September 2024 with an overall sentiment score of -305.5761
5-6pm on 30th January 2025 with an overall sentiment score of -256.5684
6-7pm on 30th January 2025 with an overall sentiment score of -226.9097
4-5pm on 3rd February 2025 with an overall sentiment score of -223.8335

===== The 5 Happiest People =====
gameoflife, account id 110237351908820391 with a total positive sentiment score of +9105.7419
TheFigen_, account id 112728392129924952 with a total positive sentiment score of +2541.4742
EmojiAquarium, account id 113441357479871732 with a total positive sentiment score of +2086.4567
choochoo, account id 113541715572887376 with a total positive sentiment score of +1978.7431
hnbot, account id 110006430181118061 with a total positive sentiment score of +1917.7936

===== The 5 Saddest People =====
realTuckFrumper, account id 109521050152429017 with a total negative sentiment score of -9094.0813
uavideos, account id 109471254002284799 with a total negative sentiment score of -5901.9603
TheHindu, account id 113443732887173058 with a total negative sentiment score of -5710.7226
uutisbot, account id 111873606251312850 with a total negative sentiment score of -4066.1382
MissingYou, account id 112071598249945636 with a total negative sentiment score of -3341.6033
```

Figure 3: result of output for mastodon-144g.ndjson

The figure 3 is our output showing the Top 5 happiest hours, saddest hours, happiest people and saddest people. And each of output data is sorted from top 1 to top 5.

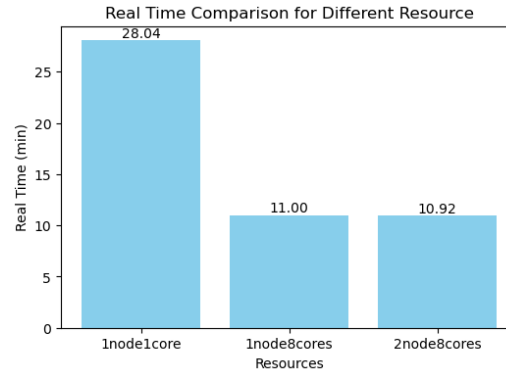


Figure 4: Comparison for different resource in real time

Figure 4 shows the results of executing times on each Spartan resources. We can see that the time taken by the job in the configuration of 1node1cores is about 28.04 minutes, and the execution time of the jobs in 1node8cores and 2node8cores are very close to each, they are both around roughly 11 minutes. The results illustrate that the execution time of single core takes more time than the multi-core execution, which proves that parallelisation significantly improves the processing speed. The different between 1node8cores and 2node8cores is not significant, this may because the additional overhead associated with multi-node communication and the limited size of the data. And base on Amdahl's law, as the number of parallelisms increases, the serial portion or communication share becomes performance bottleneck which can lead to multiple nodes and cores will not significantly shorten the total runtime.

We can use the time of 1node1core and 1node8cores to calculate the how many parts of our program can parallelisable. We can first get the value of speedup: $S = \frac{T(1node1core)}{T(1node8cores)} = \frac{29.04}{11} = 2.64$, Then we get that: $S = 2.64 = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-P)+\frac{P}{8}}$, we can get $P \approx 0.70995$, means that around 70.995% of the program can utilise multiple cores for simultaneous execution, which explains the significantly lower time spent on 8 cores compared to single core runtime. Finally, we can get: $S = \frac{1}{(1-0.70995)+\frac{0.70995}{N}}$, plotting the formula in graphing software (figure 5), we can see that as the number of cores N tends to infinity, S tends to 3.4 which means that even if more cores are added, the theoretical maximum speed is only about 3.4 time faster than a single core.

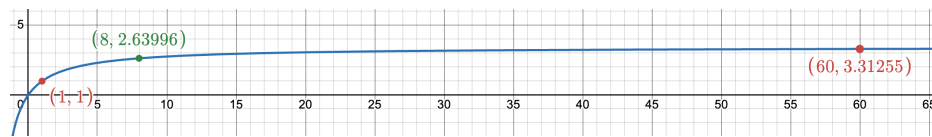


Figure 5: plot equation into graph software

4. Conclusion

From this project, we have learnt about HPC, Slurm and implementing the program on Spartan, also we learned to utilise python for parallel computing development. Our program has achieved the desired results in HPC environments, where runtimes can be dramatically reduced with multiple cores. In the future, we can further optimise the data segmentation strategy by improving the algorithms and use more advanced parallel techniques to further increase the efficiency of large-scale data processing.

Appendix

1 node 1 core

```
#!/bin/bash
#SBATCH --job-name=1node1core_job
#SBATCH --output=/home/chenhaof/result/1n1c_result_job_%j.out
#SBATCH --error=/home/chenhaof/result/1n1c_result_job_%j.err
#SBATCH --time=02:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

module load SciPy-bundle/2022.05

# The path of the file must be provided
time mpirun python3 main.py /home/chenhaof/mastodon-144g.ndjson
```

1 node 8 cores

```
#!/bin/bash
#SBATCH --job-name=1node8core_job
#SBATCH --output=/home/chenhaof/result/1n8c_result_job_%j.out
#SBATCH --error=/home/chenhaof/result/1n8c_result_job_%j.err
#SBATCH --time=02:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1

module load SciPy-bundle/2022.05

# The path of the file must be provided
time mpirun python3 main.py /home/chenhaof/mastodon-144g.ndjson
```

2 nodes 8 cores

```
#!/bin/bash
#SBATCH --job-name=2node8core_job
#SBATCH --output=/home/chenhaof/result/2n8c_result_job_%j.out
#SBATCH --error=/home/chenhaof/result/2n8c_result_job_%j.err
#SBATCH --time=02:00:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1

module load SciPy-bundle/2022.05

# The path of the file must be provided
time mpirun python3 main.py /home/chenhaof/mastodon-144g.ndjson
```