# COMP90024 Cluster and Cloud Computing - Project 1 Report

Xin Shen 1266811

## Introduction

This report documents the parallelized data analysis pipeline developed for COMP90024 Project 1. This pipeline leverages the Python library *mpi4py* to achieve parallelization of tasks on multiple processors on the University of Melbourne’s High Performance Computer - **Spartan**.

### Aim

The aim of this project is to read, process and analyse 144Gb of social media data from Mastodon to identify the following patterns:

1. The 5 happiest/saddest hours
2. The 5 happiest/saddest users

The data is in the Newline Delimited JSON (NDJSON) format, and columns that are useful for our analysis are:

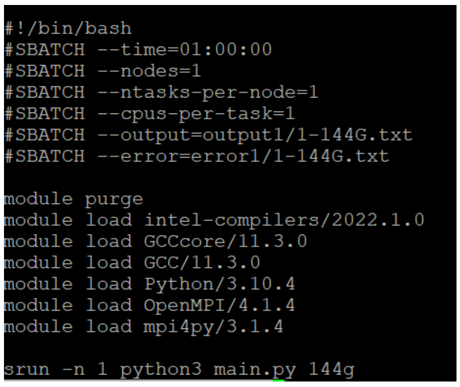
1. doc.createdAt: time of creation of a post
2. doc.account.id: account id of the author of the post
3. doc.account.username: account username of the author of the post
4. doc.sentiment: precalculated sentiment score of the post

By integrating these columns, we will be able to identify solutions for the targeted problems.

## Application Pipeline

### Slurm Scripts

1 node 1 core

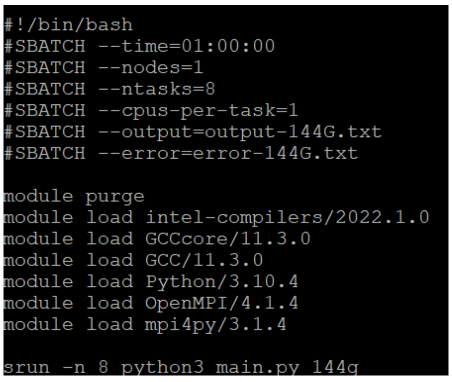


```
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --output=output1/1-144G.txt
#SBATCH --error=error1/1-144G.txt

module purge
module load intel-compilers/2022.1.0
module load GCCcore/11.3.0
module load GCC/11.3.0
module load Python/3.10.4
module load OpenMPI/4.1.4
module load mpi4py/3.1.4

srun -n 1 python3 main.py 144g
```

1 node 8 cores

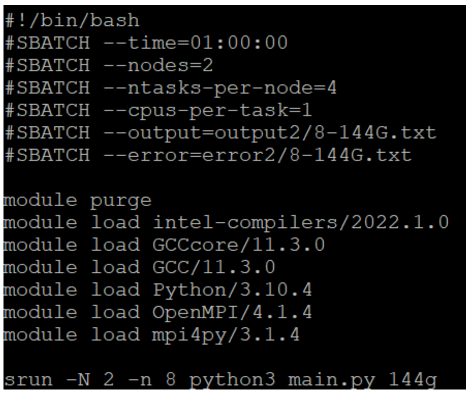


```
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --output=output-144G.txt
#SBATCH --error=error-144G.txt

module purge
module load intel-compilers/2022.1.0
module load GCCcore/11.3.0
module load GCC/11.3.0
module load Python/3.10.4
module load OpenMPI/4.1.4
module load mpi4py/3.1.4

srun -n 8 python3 main.py 144g
```

2 node 8 cores



```
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --output=output2/8-144G.txt
#SBATCH --error=error2/8-144G.txt

module purge
module load intel-compilers/2022.1.0
module load GCCcore/11.3.0
module load GCC/11.3.0
module load Python/3.10.4
module load OpenMPI/4.1.4
module load mpi4py/3.1.4

srun -N 2 -n 8 python3 main.py 144g
```

The above Slurm scripts will schedule our task on Spartan, assign the number of nodes and cores, set directory of output and error files, load modules required to run the program, and finally use the srun command to execute the program.

## Initiate Parallelization: Scatter Data

For this project, we run the decomposition of data sequentially on the root processors, then distribute the decomposed result to each child processor. By referring to the size variable from the communicator, we can decompose our data using the following method:

1. $Chunk\ size\ =\ Data\ Size\ /\ Number\ of\ Processors$
2. To prevent breaking a data row in the mid-way, we read from start byte + chunk size until we reach the first b"\n" character denoting the end of a complete NDJSON line. The starting byte of the next chunk will be the next byte after the previously located b"\n" character.
3. At the end of the decomposition, we will yield a list of dictionaries in the form of *[{startByte: x, endByte: y}]*.

The Scatter method of the MPI communicator will distribute each element of the list to its corresponding processor. By using Scatter(dictionary), we send one dictionary recording the start and end points of a data chunk to processors.

## Perform Parallelization: Reading Data

Once each processor receives their assigned breaking points, we initiate the reading through of the data file. By using Python's built-in file handling methods such as *open(), seek()* and *tell()*, we are able to read from the assigned starting point, and stop once reaching the endpoint. In order to prevent overloading the storage, instead of reading and saving the entire file, we use *yield* keyword to process each row of data on the fly, extract and store only meaningful information and discard before moving on to the next row.
While iterating through lines in the data file, for each specific row, we perform the following actions:

1. Eliminate unrelated columns and only keep the aforementioned useful columns
2. Since we integrate timestamps by hours, we use regex to transform *yyyy-mm-ddThh:mm:ss* to *yyyy-mm-ddThh*
3. Integrate by time: create a dictionary which timestamp is the key and corresponding sentiment score is the value, for each iteration, if we identify an existing timestamp, we append its sentiment score, otherwise, we initialise a new item.
4. Integrate by user: similar to timestamp, but using account id as key, and a tuple of (sentiment score, account name) as the value.

## Finalise Parallelization: Gather and Summarise Data

At the end of iterations, all processors produce two dictionaries: *sentiment_byHour* and *sentiment_byUser*. We use the *Gather* method of MPI communicator to retrieve and collect all generated dictionaries from child processors to the root and perform the final summarisation of the overall dataset. We merge all dictionaries under the same criteria, ensure we append the sentiment score instead of update, and eventually produce two complete summarisation dictionaries. By sorting the two dictionaries with respect to the sentiment score, we can identify the top 5 and last 5 keys.

## Results and Performance

The system returns the following results:
**Happiest Hours:**

1. 2025/01/01 12AM to 1AM has a sentiment score of 206.152
2. 2024/12/31 11PM to 12AM has a sentiment score of 187.465
3. 2025/01/01 5AM to 6AM has a sentiment score of 135.925
4. 2024/12/24 10PM to 11PM has a sentiment score of 117.286
5. 2024/12/24 3PM to 4PM has a sentiment score of 144.602

**Saddest hours:**

1. 2024/11/06 7AM to 8AM has a sentiment score of -373.767
2. 2024/09/11 1AM to 2AM has a sentiment score of -305.576
3. 2025/01/30 5PM to 6PM has a sentiment score of -256.568
4. 2025/01/30 6PM to 7PM has a sentiment score of -226.910
5. 2025/02/03 4PM to 5PM has a sentiment score of -223.833

**Happiest Users:**

1. User *gameoflife* with ID 110237351908820391 has a sentiment score of: 9105.742
2. User *TheFigen_* with ID 112728392129924952 has a sentiment score of: 2541.474
3. User *EmojiAquarium* with ID 113441357479871732 has a sentiment score of: 2086.457
4. User *choochoo* with ID 113541715572887376 has a sentiment score of: 1978.743
5. User *hnbot* with ID 110006430181118061 has a sentiment score of: 1917.794

**Saddest Users:**

1. User *realTuckFrumper* with ID 109521050152429017 has a sentiment score of: -9094.081
2. User *uavideos* with ID 109471254002284799 has a sentiment score of: -5901.960
3. User *TheHindu* with ID 113443732887173058 has a sentiment score of: -5710.723
4. User *uutisbot* with ID 111873606251312850 has a sentiment score of: -4066.138
5. User *MissingYou* with ID 112071598249945636 has a sentiment score of: -3341.603

The time taken to run through this application is as follows:

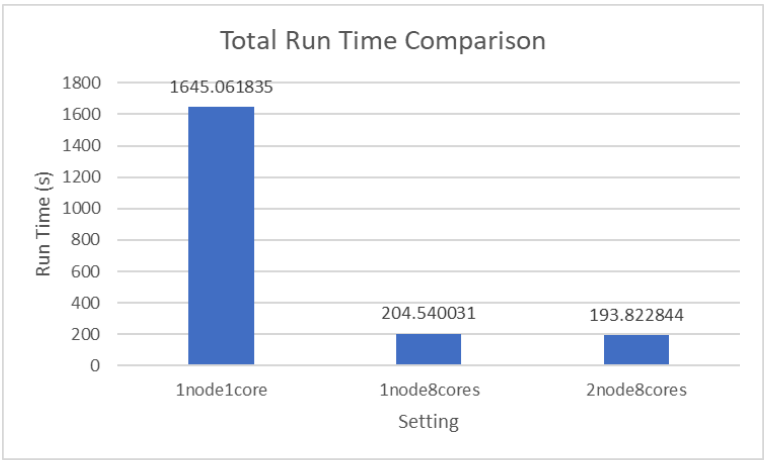


Figure 1: total run time under three settings

| Settings | Scatter | Process | Summarise |
|---|---|---|---|
| 1n1c | 0.02 | 1644.68 | 0.36 |
| 1n8c | 0.73 | 202.56 | 1.25 |
| 2n8c | 0.77 | 192.15 | 0.87 |

Table 1: Process time of individual components

We ran the same application under 3 settings: 1 node 1 core, 1 node 8 cores, and 2 nodes 8 cores, so that we can examine the change in efficiency with parallelization. As shown in Figure 1, we see that by distributing data processing from 1 core (sequential) to 8 cores, we managed to reduce the total run time by more than 8 times, significantly boosting the efficiency.
However, as suggested by Amdahl's Law, there exists an inevitable bottleneck in parallelization. Regardless of the proportion of parallelizable tasks in an application, it always requires a sequential process to scatter and gather results, and the run time of our application will always be bound by the time taken to perform these actions. In fact, on some occasions, introducing parallelization even increases the runtime of sequential tasks. Referring to Table 1, we can observe that when running sequentially, it requires only 0.02 seconds to distribute data, as no actual decomposition is needed; whereas under parallelization, it requires at least 0.73 seconds to assign each processor their subset of data. Similarly, running on multiple processors also required longer time to summarise results. Therefore, despite the vast reduction in run time when using multiple processors, the total improvement in efficiency is always bounded by the time for sequential tasks.

## Conclusion

This project demonstrates a complete process of performing parallelized data analysis using multiple processors on a HPC, using an example of identifying the happiest/saddest hours/users on the social media platform of Mastodon. By comparing the application's performance on: 1 node 1 core, 1 node 8 cores and 2 nodes 8 cores, the derived result evidence that parallelization indeed can largely improve the efficiency. Nevertheless, it also indicates that such improvement in performance is limited by the non-parallelizable tasks in this application, such as scattering data to sub-processors and gathering results from them (Amdahl's Law).