Assignment 1

Group Members: Harish Kannan, Sarathi Thirumalai Soundararajan
Professor: Dr. Richard O. Sinnott
GitHub: CCP_MPI_Python

1 Problem Description

The task for this programming assignment is to implement a parallelized application leveraging the University of Melbourne High-Performance Computing (HPC) facility, SPARTAN. The application is to process a large dataset extracted from Twitter. The main objectives of the application are:

- To identify the happiest hour ever recorded in the dataset.
- To determine the happiest day ever recorded in the dataset.
- To find the most active hour ever, i.e., the hour with the highest number of posts in the dataset.
- To ascertain the most active day ever, i.e., the day with the highest number of posts in the dataset.

The program should be designed with efficiency in mind, making use of SPARTAN's parallel computing capabilities to handle the large volume of data involved.

2 Implementation

The script uses the following packages for the two approaches used for the implementation.

- `pandas`: For data analysis and manipulation.
- `json`: For parsing JSON data.
- `ijson`: For incremental JSON parsing.
- `time`: For operation timing.
- `datetime`: For date and time manipulation.
- `timezone`: For timezone handling.
- `ZoneInfo`: For timezone data.
- `matplotlib.pyplot`: For plotting data.
- `seaborn`: For statistical data visualization.
- `mpi4py`: For parallel computing with MPI.

3 Approach 1 - Batch Processing

3.1 Algorithm

The algorithm analyzes a dataset of tweets to identify the happiest hour and day based on sentiment scores, and the most active hour and day based on tweet volume. It reads tweets in chunks, processes them to calculate sentiment scores and counts, and then determines the time periods with the highest sentiment scores and the highest number of tweets. Finally, it outputs the happiest and most active times, displaying specific dates, times, and sentiment scores or tweet counts, alongside the execution time of the script.

Further, we create a bar plot to visualize the processing times of a batch processing task under three different computational resource configurations: "1 node & 1 core", "1 node & 8 cores", and "2 nodes & 8 cores". It constructs a DataFrame to organize these configurations alongside their respective processing times in seconds, then plots this data. Annotations on each bar display the precise processing times, and the plot is titled "Batch Processing Approach Visualization".

Algorithm 1: Twitter Data Analysis

```
Data: A file containing Twitter data in JSON format.
Result: The happiest and most active hour and day based on sentiment and tweet counts.
Function cvt_data(a):
    Data: A string representation of a JSON object.
    Result: The JSON object or an empty dictionary on failure.
    // Attempt to parse JSON, return empty dict if fails.
    $result \leftarrow \{\}$;
    if a is valid JSON then
        $result \leftarrow$ parse a into JSON object;
    return result;
Function happiest_hour(final_data):
    // Update global dictionary with sentiment values by hour.
    for data in final_data do
        if data is valid and has sentiment then
            Update global sentiment dictionary with data;
Function happiest_day(final_data):
    // Update global dictionary with sentiment values by day.
    // Similar to happiest_hour function.
Function most_active_hour(final_data):
    // Update global dictionary with tweet counts by hour.
    // Similar to happiest_hour function.
Function most_active_day(final_data):
    // Update global dictionary with tweet counts by day.
    // Similar to happiest_hour function.
Program Main():
    // Initialize global dictionaries for results.
    // Read JSON file in chunks and process.
    Initialize global dictionaries;
    while not end of file do
        Read a chunk of data from file;
        Convert each line of chunk using cvt_data;
        happiest_hour(converted data);
        happiest_day(converted data);
        most_active_hour(converted data);
        most_active_day(converted data);
    // Determine and print results for happiest and most active times.
    Print "happiest hour", "happiest day", "most active hour", "most active day";
    return;
```

3.2 Result

High CPU utilization across all configurations suggests that the task is computationally intensive. Notably, memory usage remains consistently low, demonstrating the analysis's memory efficiency. The reduction in execution time when moving from a single core to multiple cores, particularly with the configuration utilizing two nodes with four cores each, underscores the benefits of parallel processing.

This optimal configuration resulted in the lowest wall-clock time of 00:58:02. The analysis consistently identified the happiest time on Twitter as between 1pm-2pm on 31 December 2021 and the most active time as between 12pm-1pm on 21 May 2022. These findings remained invariant across the different computational setups, highlighting the robustness of the analysis method.

Table 1: Resource Usage Summary

Configuration	CPU Usage (%)	Memory (RAM)	Execution Time	Wall-clock Time
1 node, 1 core	99.7	0.1% [5MB of 4195MB]	5776.58 seconds	01:36:22
1 node, 8 cores	99.6-99.9	0.0% [5MB of 33555MB]	3690.11 seconds	01:01:35
2 nodes, 8 cores (4/node)	93.6-99.6	0.0% [5MB of 33555MB]	3476.48 seconds	00:58:02

Table 2: Analysis Summary and Resource Usage

Configuration	Happiest Time	Most Active Time
1 node, 1 core	31 Dec 2021, 1pm-2pm Score: 3658.95	21 May 2022, 12pm-1pm Tweets: 39061
1 node, 8 cores	31 Dec 2021, 1pm-2pm Score: 3658.95	21 May 2022, 12pm-1pm Tweets: 39061
2 nodes, 8 cores (4/node)	31 Dec 2021, 1pm-2pm Score: 3658.95	21 May 2022, 12pm-1pm Tweets: 39061

4 Approach 2 - Streaming Approach

4.1 Algorithm

Algorithm 2: Analysis of Twitter Data for Happiness and Activity

```
Data: A JSON file with Twitter data
Result: Determine the happiest and most active hour and day based on sentiment scores and tweet counts.
Initialize dictionaries for storing hourly and daily sentiment scores and tweet counts: ;
happy_hour_dict, happy_date_dict, mtweets_hour_dict, mtweets_date_dict ← {};
Open the JSON file and read data in a streaming manner using ijson;
while data exists do
    foreach item in data do
        happiest_hour(item);
        happiest_day(item);
        most_active_hour(item);
        most_active_day(item);
Determine the happiest and most active times: ;
happy_hour_dict ← sort and find the maximum value by sentiment;
happy_day_dict ← sort and find the maximum value by sentiment;
mtweets_hour_dict ← sort and find the maximum value by count;
mtweets_day_dict ← sort and find the maximum value by count;
Output the results;
Print happiest hour, happiest day, most active hour, most active day;
Print execution time;
Function happiest_hour(data):
    if data is valid and contains sentiment then
        Update happy_hour_dict with sentiment value for the hour extracted from timestamp;
Function happiest_day(data):
    if data is valid and contains sentiment then
        Update happy_date_dict with sentiment value for the day extracted from timestamp;
Function most_active_hour(data):
    if data is valid then
        Update mtweets_hour_dict by incrementing the count for the hour extracted from timestamp;
Function most_active_day(data):
    if data is valid then
        Update mtweets_date_dict by incrementing the count for the day extracted from timestamp;
```

ijson is an iterative parsing tool for Python with the standard Python iterator interfaces, capable of parsing and extracting data from JSON files repeatedly. It works by processing data in smaller pieces via generator iterators and yield expressions, allowing interaction with huge JSON files without encountering memory limits, making it suitable for processing enormous data sets.

Some of the Key Features of ijson include:

- Efficient Memory Consumption
- Standard Python Interfaces
- Iterative Processing

```
with open("/home/harishk/twitter-100gb.json") as f:
    data = ijson.items(f, "rows.item")
    for i in data:
```

Here, we initially used `ijson.parse`, which is a lower-level function to generate a three-element tuple, namely: a prefix, an event name, and a value. This function is used to identify the appropriate feature required. Then, we can use the `ijson.items` function, which will return each item under the `data` key using `data.item` path. The `for` loop will then iterate through each item in the `data` key, giving us a row-wise list.

4.2 Result

Table 3: Resource Usage Summary

Configuration	CPU Usage (%)	Memory (RAM)	Execution Time (s)	Wall-clock Time
1 node, 1 core	99.6	0.1% [5MB of 4195MB]	4459.51	01:14:24
1 node, 8 cores	99.2	0.0% [5MB of 33555MB]	2359.36	00:39:24
2 nodes, 8 cores (4/node)	93.3-99.3	0.0% [5MB of 33555MB]	2441.46	00:40:46

Table 4: Happiest and Most Active Times

Configuration	Happiest Time	Most Active Time
1 node, 1 core	31 Dec 2021, 1pm-2pm	21 May 2022, 12pm-1pm
	Score: 3658.95	Tweets: 39061
1 node, 8 cores	31 Dec 2021, 1pm-2pm	21 May 2022, 12pm-1pm
	Score: 3658.95	Tweets: 39061
2 nodes, 8 cores (4/node)	31 Dec 2021, 1pm-2pm	21 May 2022, 12pm-1pm
	Score: 3658.95	Tweets: 39061

5 Slurm Script

The Slurm script used for the streaming approach was configured with 1 node and 1 core, with a maximum execution time limit of 4 hours. Necessary packages, including mpi4py, were loaded using the module command. A virtual environment, ‘venv‘, was created for the ijson package installation, which was not included in the loaded modules. The environment setup and deactivation were handled as follows:

```
python3 -m venv venv
source venv/bin/activate
pip install ijson
deactivate
```

The ‘srun‘ command executed the Python script, and ‘my-job-stats‘ was used to monitor resource usage. The script execution and job status monitoring were performed with ‘sbatch‘ and ‘squeue‘ commands, respectively.

Adjustments were made for configurations involving 1 core with 8 nodes, and 2 cores with 8 nodes, with appropriate adjustments in the Slurm script for each scenario.

The batch processing approach utilized the same Slurm script, excluding the requirement for the ‘venv‘ virtual environment, as the ijson dependency was omitted. The script modification involved changing the run command to:

```
srun python3 new_script.py
```

6 MPI4PY

MPI (Message Passing Interface) is a standardized and portable message-passing system designed to facilitate process communication. We utilized mpi4py, a Python package that adheres to MPI standards, to enable Python applications to leverage multiple processors across clusters, supercomputers, and workstations. mpi4py supports a variety of use cases, including point-to-point and collective communication of any Python object that can be pickled, thus enabling efficient parallelism through message exchange and execution synchronization.

Our application leverages MPI with the following commands:

```
comm = MPI.COMM_WORLD
size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
```

Here, `comm` is the default communicator that includes all available processes. `size` represents the total number of processors allocated to execute the application, while `rank` returns the rank of this process within the communicator. We use the condition `ind % size == rank` to allocate data to specific processes based on rank, facilitating computation and storage of results. Predominantly, we employ `comm.gather` to collect all results at the process with rank 0, acting as the master node, to aggregate the final outcomes:

```
happy_hour_dict = comm.gather(happy_hour_dict, root=0)
happy_date_dict = comm.gather(happy_date_dict, root=0)
mtweets_hour_dict = comm.gather(mtweets_hour_dict, root=0)
mtweets_date_dict = comm.gather(mtweets_date_dict, root=0)
```

This methodology enabled parallel processing of a vast dataset, significantly accelerating the computation as the number of cores increased, demonstrating the efficacy of mpi4py in distributed computing environments.

7 Comparative Study

Stream processing outperforms batch processing in speed, making it ideal for real-time analysis due to its efficient handling of data on-the-fly. This efficiency is particularly beneficial when quick insights are required from streaming data. Conversely, batch processing, while slower, excels in scenarios where data is pre-collected and immediate analysis isn't critical. It leverages parallel processing effectively, especially with increased computing cores, ensuring consistent and reliable outcomes. The choice between stream and batch processing hinges on the analysis needs: real-time insight versus comprehensive review of large datasets.

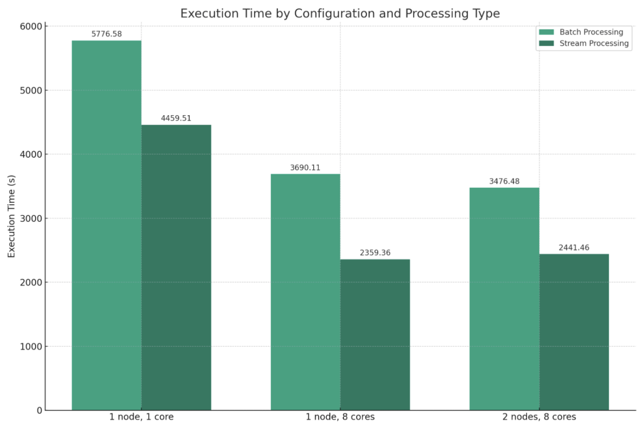

Figure 1: Execution Time by Configuration and Processing Type

Scaling up from a single core to eight cores on one node significantly reduces execution time: by about 36.12% for batch processing and 47.09% for stream processing. Extending to two nodes with four cores each further reduces batch processing time by approximately 39.82% and stream processing time by around 45.25% from the baseline single-core setup. These results underscore the substantial performance gains achievable through parallel processing, demonstrating a direct correlation between the number of utilized cores and improved processing efficiency.