COMP 90024: Cluster and Cloud Computing – Assignment 1

Social Media Analytics

Team 1
Name: Yixuan Chen, Jiasheng Yang
Student ID: 1174135, 1464801
Instructor: Dr. Richard Sinnott
Date: April 12, 2024

1. Project Description

The project designs and develops a parallel application on SPARTAN HPC to analyze the large Twitter dataset, concluding the happiest hour and day according to the sentiment scores, and the most active hour and day according to the tweets. The project's goal is to come up the optimal parallelization approach to maximum the performance of the application.

2. Introduction

The project is mainly based on **Message Passing Interface** technology and **parallelization** idea. We implemented four different parallel approaches to maximize the performance of computing resources and application; which were divided into two categories, i.e., whether the data was executed batch processing. The more specific categorization is based on whether implement the streaming processing during the loading data file. Since when the codes did not apply the stream processing method, all data in the JSON file will be loaded in the memory; if the data is too huge, resulting out of the memory. Therefore, we only utilized the Version 1.0 project, without streaming processing, to handle the 1mb and 50mb files, which set the solid foundation for us to further explore the ways of stream processing (Introduce them integrated with the Version 2.0 project).

The remain parts will focus on parallelization approaches, implement details, and analysis of the performance of these approaches.

Note: Considering the limitation of computing resources, Message Limits, and approaching deadline, we conducted the one version, **the Version 3.0 of Without Batch Processing, only Streaming Processing,** as the final submits.

3. Parallelization Approach

3.1. Without Batch Processing, only Streaming Processing

For Version 1.0 Step:

a. Start and Read file: The master process (rank=0) starts timing, opens a JSON file and load it to the memory.
b. Data partitioning: If there is only one process is running, called serial mode, only the master is used to execute to get the answer. Otherwise, the master counts the number of lines that each worker process (1<rank<size-1) handled and then reads each process portion of the data into memory.
c. Data processing: For multi-process mode, the master sends the assigned data portions to worker. When the worker receives the data from the master, it will extract the time, sentiment scores, and calculate the sentiment scores and tweets. After finishing data

analysis, the worker sends results back to master.

d. Result aggregation and Output: The master receives and aggregates statistical results once after finishing all data wrangling. Lastly, it concludes the highest sentiment scores and most tweets, and outputs the final results and execution times.

For Version 2.0 Step:

Except a little difference in logics for reading file and data partitioning, both master and worker execute the same remaining tasks as *Without Batch Processing, Version 1.0*.

a. Read file: The master opens a JSON file and reads contents of file line by line to its memory.

b. Data Partitioning: If there is only one process is running, called serial mode, only the master is used to execute to get the answer. Otherwise, the master allocates the each lines of data read to the other worker. In other words, each line is regards as a data chunk.

c. Data Processing: There is an extra step before wrangling the data on the worker, i.e., parsing the JSON Object. The other executions are same as Without Batch Processing, Version 1.0.

For Version 3.0 Step:

While working with the Version 2.0, we found that there are still out of memory occurring when child process send the result to master process, In version 3, the main difference is that only send final result to master process. By doing that, it not only save the memory to store every tweets result, but also save the communication time within the process.

3.2. With Batch Processing, integrating Streaming Processing

For Version 1.0 Step:

Except a little difference in logics for data partitioning, both master and worker execute the same remaining tasks as *Without Batch Processing, Version 1.0*.

a. Data Partition: The master counts the batch size and splits the whole list of JSON data into a certain batch according to it. Notably, the master makes sure that there is at least one tweet in each batch. The master allocates these batches to the worker.

For Version 2.0 Step (assume, considering the default):

Except a little difference in logics for reading file, data partitioning, and data receiving and sending, both master and worker execute the same remaining tasks as *With Batch Processing, Version 1.0*.

a. Read file: Before reading the data, the program declares the buffer size. According to the buffer size, the master opens and reads a JSON file into its memory chunk by chunk; the chunk size is the buffer size.

b. Data Partitioning: If there is only one process is running, called serial mode, only the master is used to execute to get the answer. Otherwise, the master sends chunk when receives the ready signal from the worker.

d. Data receiving and sending: For multi-process mode, once master received the ready signal, it sends the chunk to the corresponding worker. In that case, the worker not only sends the result of data processing back to the master, but also the ready signal. Data There is an extra step before wrangling the data, i.e., parsing the JSON Object. The other executions are same as Without Batch Processing, Version 1.0. Lastly, the master will send the ending signal back to the worker when completing the reading file.

4. Implementation Details

4.1. Stream Processing and Batching Processing

For Without Batch Processing, only Streaming Processing:

It implements a basic and simple streaming processing: read as the JSON Object line by line, regarding the each line as the chunk, eventually, parsing the JSON Object (each line already read and sent) on the worker.

```
if rank == 0:
    with open(filename, 'r') as f:
        for line in f:
```

```
else:
    line_count = 0
    with open(filename, 'r') as f:
        for line in f:
            dest_rank = line_count % (size - 1) + 1
            comm.send(line, dest=dest_rank, tag=1)
            line_count += 1
    for i in range(1, size):
        comm.send( obj: None, dest=i, tag=1)
```

From our test experiments, we find that the value of **'dest' and the method of sending and receiving data** is the key to implement the parallelization.

For With Batch Processing, integrating Streaming Processing:

It implements a common and advanced streaming processing: split and read JSON Object according to the batch size (This approach is only can be applied for the small size data since the computing resource, Message Limits, is allocated from the SPARTAN HPC too small.)

```
    else:
        batch_size = max(len(tweets) // (10 * (size - 1)), 1)  # Ensure at least 1 tweet per batch
        tweet_batches = [tweets[i:i + batch_size] for i in range(0, len(tweets), batch_size)]

        for i, batch in enumerate(tweet_batches):
            dest = (i % (size - 1)) + 1
            comm.send(batch, dest=dest)

        # Signal end of data
        for i in range(1, size):
            comm.send( obj: None, dest=i)

# Worker processes
else:
    while True:
        tweet_batch = comm.recv(source=0)
        if tweet_batch is None:
            break
        results = process(tweet_batch)
        comm.send(results, dest=0)
```

4.2. run.slurm

As shown in the following code, it set all the nodes and cores information in the one variable, utilizing the loop to generate the temporary files for them. Lastly, store all the result in a new file. And grep the execution time in a file so that generates the figure of the performance to analysis.

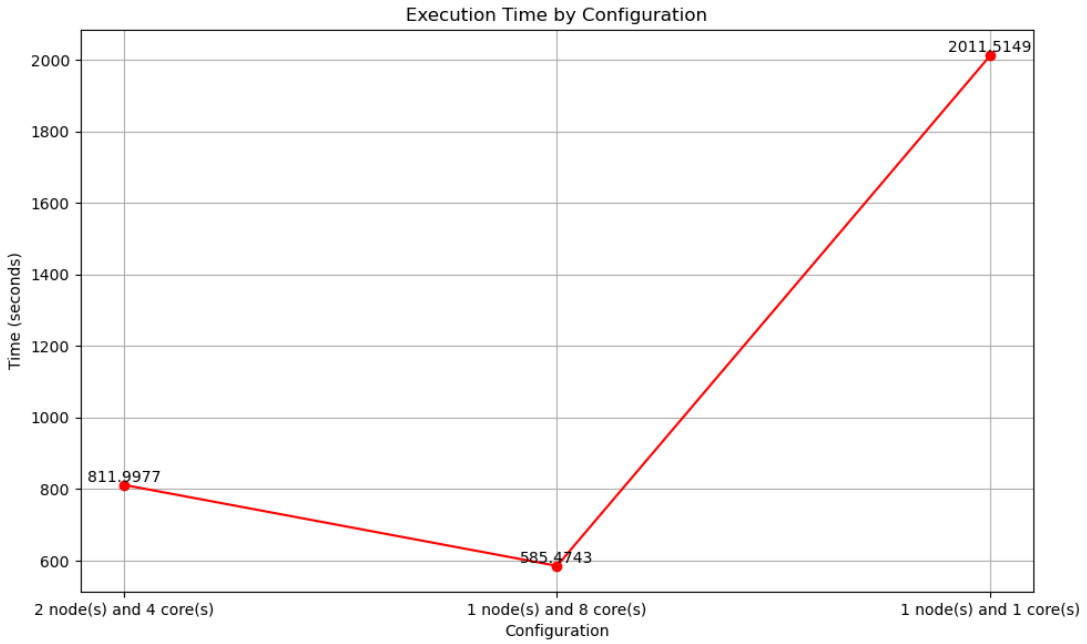
```
#!/bin/bash

declare -a configs=("1 1" "1 8" "2 4")

result_file="result.txt"
if [ ! -f "${result_file}" ]; then
    touch "${result_file}"
fi

execute_time_file="execution_times.txt"
if [ ! -f "${execute_time_file}" ]; then
    touch "${execute_time_file}"
fi

slurm_log_folder="slurm_log"
if [ ! -d "${slurm_log_folder}" ]; then
    mkdir "${slurm_log_folder}"
fi
```

```
for config in "${configs[@]}"; do
    IFS=' ' read -r -a array <<< "${config}"
    nodes=${array[0]}
    ntasks_per_node=${array[1]}

    temp_script="temp_script_${nodes}_${ntasks_per_node}.slurm"

    # create the
    cat > ${temp_script} <<EOL
#!/bin/bash
#SBATCH --job-name=run_${nodes}_${ntasks_per_node}
#SBATCH --output=${slurm_log_folder}/%j.out
#SBATCH --error=${slurm_log_folder}/%j.err
#SBATCH --time=01:00:00
#SBATCH --nodes=${nodes}
#SBATCH --ntasks-per-node=${ntasks_per_node}

module --ignore_cache load mpi4py

output=\$(srun python main.py)

echo "${nodes} node(s) with ${ntasks_per_node} core(s) result:" >> "${result_file}"
echo "\$output" >> "${result_file}"
echo "" >> "${result_file}"

exec_time=\$(echo "\$output" | grep "Execution Time:" | awk '{print \$3}')
echo "${nodes} node(s) and ${ntasks_per_node} core(s): \$exec_time seconds" >> "${execute_time_file}"
echo "" >> "${result_file}"

# delete the temporary slurm script
rm -- "\$0"
EOL

    # submit the temporary job
    sbatch ${temp_script}
done
```

5. Analysis

According to the line chart, we can find that the execution time of the application decreases while the number of processes with the same nodes is increasing, which indicates that our method is parallelized to some extent and maximizes the use of computational resources.

Additionally, in the case of the same process, the execution time is increasing as the number of nodes is increasing. This might be due to the increased communication overhead between nodes. Additionally, for multi-node mode, the data will be partitioned among nodes. When the number of nodes increases, the time required for the data allocation my increases as well, especially the dataset is very huge.



6. Conclusion

All in all, we gained basic knowledge about MPI, cluster computing, and parallel programming. We will continue to improve the application via the exploration in the class to come up more optimal parallel solutions to maximize the computing resource during our spare time. This project will be our “calling card” for our future study and career.