Twitter Data Analytics with SPARTAN

1.Introduction

1.1. Background and Objectives

In this report, we present our implementation of parallel code logic, tasks, and analysis methods to perform social media analytics on a large Twitter dataset. Our objectives are to:

1. Identify the Twitter accounts (users) that have made the most tweets.
2. Count the number of different tweets made in the Greater Capital cities of Australia.
3. Identify the users that have tweeted from the most different Greater Capital cities.

In this case, we use a dataset consisting of three different JSON files: *bigTwitter.json*, *smallTwitter.json*, and *tinyTwitter.json* for testing. Additionally, we use the *sal.json* file containing suburbs, locations, and Greater Capital cities of Australia.

1.2. Tools

For our application, we implemented the solution in C++ due to its efficiency and performance benefits. C++ offers low-level control of resources and memory management, resulting in faster execution times compared to Python. Furthermore, C++ allows us to utilize MPI libraries for parallelization and communication between nodes and cores effectively.

2.Methodology

2.1. Data Preprocessing and Application Parallelizing

Before processing the tweets dataset, we first need to establish a map *place_map* that records the names of all places and its corresponding Great Capital city (GCC). By choosing place name as the key and GCC as the value in the *place_map*, the code can efficiently look up the corresponding GCC for each place name when processing the tweet dataset.

We divide the large tweets JSON file into *n* equal parts by bytes size and assign each part to one of the *n* processes for reading and analysis. Each process will only read one portion of the original document and process the data in the corresponding file portion. This parallelization strategy takes advantage of the multiple processes, reducing the overall processing time.

We adopt a line-by-line reading approach and only index specific fields. This method helps reduce memory consumption and improve program performance. By using an ifstream object to read the file line by line and the *getline*() function to obtain the content of each line, we only search for and process fields relevant to the task objectives, such as username and place name. This approach offers higher efficiency when dealing with large datasets but may require dealing with higher code complexity.

For task 2, we simply establish a map and record the tweets number. Then we use MPI_Gather to send the tweet number of all greater capital cities to the rank 0 process. The rank 0 process then add the tweet numbers from all processes and obtain the final result.

For task 1 and task 3, we use a different approach. To enhance the performance, instead of sending the names and values of all users, each process only sends the names and values of some users. Once the data is assigned, each process performs the following steps:

- Each process processes its assigned portion of the data and identifies its top 10 users with the most tweets or most different Greater Captital cities. Utilizing MPI_Allgather, every process shares its top 10 list with all other processes.
- Upon receiving the top 10 lists from other processes, each process generates a map that records the user names and their corresponding values for all users in the combined top 10 lists.
- Each process then sends the tweet counts of the users in its map to process 0. Process 0 aggregates the tweet counts for all author IDs and sorts the users based on their total tweet counts or Greater Capital cities numbers.
- Finally, process 0 selects the overall top 10 users with the highest tweet counts or Greater Capital cities numbers.

This approach offers a significant speed improvement compared to a method that requires sending the entire user dataset among the processes. By only sharing the top 10 users for each process, we greatly reduce the communication overhead and the amount of data exchanged between processes. However, this method has a trade-off in terms of accuracy. Since we are only considering the local top 10 users from each process, there is a possibility that some users with a high tweet count or Greater Capital cities numbers may be overlooked if they are not in the top 10 of any process. Despite this limitation, our method provides a substantial performance boost, making it a practical choice for large-scale social media analytics.

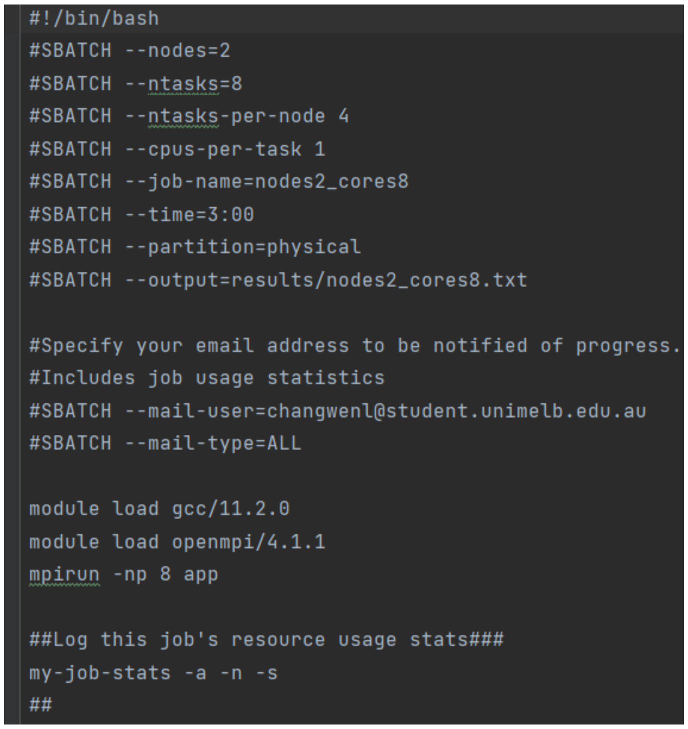
2.2. Project Deployment and Execution on Spartan

We write a cpp file and a header file for each task, and another file to invoke the parallelization and execution of all tasks.

To compile this project, run ‘mpic++ -o app parallel.cpp task1.cpp task2.cpp task3.cpp’. Then an executable file “app” will be generated.

To execute this project, run “mpirun -np 8 app”. If you want to invoke different number of cores, simply change it to another number.

To execute the project on Spartan, you may refer to the slurm batch file in the zip file. The figure below demonstrates how to use slurm file to execute project on 2 nodes with 8 cores.



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node 4
#SBATCH --cpus-per-task 1
#SBATCH --job-name=nodes2_cores8
#SBATCH --time=3:00
#SBATCH --partition=physical
#SBATCH --output=results/nodes2_cores8.txt

#Specify your email address to be notified of progress.
#Includes job usage statistics
#SBATCH --mail-user=changwenl@student.unimelb.edu.au
#SBATCH --mail-type=ALL

module load gcc/11.2.0
module load openmpi/4.1.1
mpirun -np 8 app

##Log this job's resource usage stats###
my-job-stats -a -n -s
##
```

Figure 1: Spartan Script for 2 Nodes 8 Cores

3. Results and Analysis

The actual result tables for all tasks are attached in the zip file. One may find it under the result folder. The wall-clock time with different settings are demonstrated in figure 2. The wall-clock time is the total time that Spartan spends on the job, which includes job initialization time, execution time, IO time, etc.

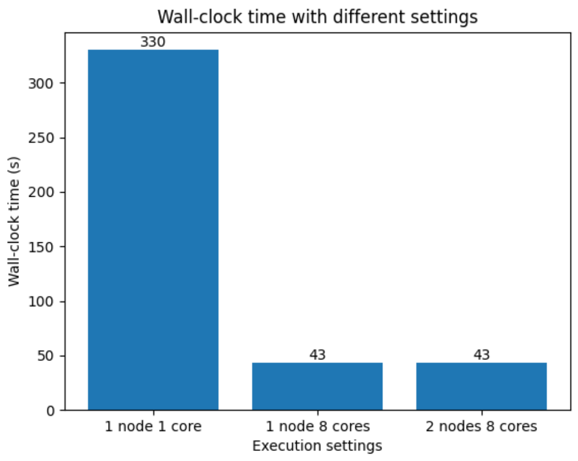


Figure 2: Running Time with Different Execution Settings

The wall-clock time between 1 node 8 cores and 2 nodes 8 cores is insignificant. The first possible reason is that there is only very few data transfers between different cores/nodes, therefore the communication time is insignificant and do not have a noticeable impact on overall execution time. The second possible reason is that the structure of Spartan is highly optimized, which decreases the communication overhead between different nodes.

The speed-up and efficiency is calculated below:
For 1 node 8 core, speedup = 330 / 43 ≈ 7.67; efficiency = 7.67 / 8 ≈ 0.96
For 2 node 8 core, speedup = 330 / 43 ≈ 7.67; efficiency = 7.67 / (2 * 4) ≈ 0.96
Now applying Amdahl's Law:
$Speedup = \frac{N}{N(1-P)+P}$,
P: proportion of parallel portion
N: number of processors

Then, let's analyze the case of 1 node 8 core then we can solve for $P \approx 0.885$. This means that the parallel portion accounts for about 88.5% of the program's runtime, and the serial portion accounts for the remaining 11.5%. Now, let's analyze the case of 2 nodes 8 cores (4 cores per node). Since the speedup is the same as 1 node 8 core, we know that the proportion of the parallel portion P is still about 88.5%.

This suggests that the program performs well in both configurations, with a large proportion of parallel portions and a relatively small serial portion. Therefore, we can conclude that the parallel portion accounts for a significant proportion of the program's runtime, and the serial portion (some sequential codes in this project, such as the data transferring (e.g. MPI_Gather) codes) is relatively small.

4. Conclusion and Future Work

Our project achieved high speed-up, fast execution and low memory usage. The wall-clock time is 330 seconds for 1 core, 43 seconds for 8 cores. The speed-up is 7.67, which is fairly high. The memory consumption is less than 1 GB. We achieve high speed-up and fast execution by using better parallelization approach, and we decrease memory usage by reading files line by line.

In future work, we can consider further optimizing the code performance by utilizing non-blocking communication operations: Instead of using blocking communication operations like MPI_Gather, we can implement non-blocking operations like MPI_Igather. Non-blocking operations allow the program to continue executing other tasks while communication is taking place, potentially improving performance.