Cloud & Cluster Computing Assignment Report
Twitter Social Media Data Analysis

Zhuoyang Hao 1255309 & Zilin Su 1155122

1 Introduction

This project aims to develop an application designed to efficiently process a 100Gb Twitter dataset to determine the happiest and most active times. The report delves into the strategy of parallelization utilizing MPI through mpi4py, and evaluates its performance across varying numbers of nodes and cores. Moreover, the report explores the influence of Amdahl's Law and Gustafson-Barsis's Law on improving computational efficiency via parallelization. This analysis aims to provide insights into optimizing processing techniques for large-scale data analysis.

2 Approach to parallelization

2.1 Strategies on 50Mb file

We explored two distinct methods for processing a 50MB file: the Equal Size approach and the Equal Tweets approach. The Equal Size method segments the file into chunks of identical size, distributing them across various processes, while the Equal Tweets method divides the file into chunks containing an equal number of tweets. Both strategies performed efficiently with the 50MB file. However, when processing a 100Gb file, the Equal Tweets approach required a significantly higher amount of RAM, since it has to read the whole file into memory, which would cause an out-of-memory issue, leading us to abandon this method.

2.2 Implementation of Equal Size approach

We employed `mpi4py` to enable communication among different processors, thereby facilitating the parallelization of our code. Utilizing the `.Get_size()` and `.Get_rank()` functions allowed us to ascertain the number of processors (for instance, $n = 8$ for a configuration of 1 node with 8 cores) and the rank of the current process, respectively.

Given our objective was solely to read data without transmitting or modifying it, we refrained from employing broadcast or scatter communications for task allocation among processors. Instead, we segmented the 100Gb tweet file into n evenly sized files and leveraged the `.seek()` function for swiftly determining each processor's specific processing locations. The simultaneous processing of different parts of tweet file by n processors is central to achieving parallelization. For processors assigned reading positions mid-tweet, we opted to bypass the remainder of that tweet, proceeding directly to the next tweet's start. In the final stage, we utilized the `gather` operation to compile results from all processes, aggregating them at the root process by simply summing the counts of tweets and their sentiment scores.

2.3 Processing Steps

Below is the outline of our processing steps:

1. Specify the desired number of nodes and cores for executing the MPI program.
2. Determine the total size of the large file in bytes. Subsequently, calculate the processing chunk size for each node. This calculation should be based on both the total file size and the total number of nodes.
    - Allocate data chunks to each node for processing. Each node will process its assigned chunk of data line by line, according to its rank, and temporarily store the results in a dictionary.
    - Proceed to execute the computation tasks utilizing the pre-defined processing logic and strategies.
3. Aggregate the dictionaries from all sub-tasks into a single comprehensive dictionary at the root process.

3 Instructions on running our application

To run an `MPI` program locally, use the `mpiexec` command along with the specified number of processors, Python environment, main program script, and required parameters. For example, to run with just 1 core, use the following command:

```
mpiexec -np 1 python3 main.py
```

To submit an `MPI` job on Spartan, use the `sbatch` command to submit a job using a `slurm` script. Here is an example that submits an `MPI` Twitter analyzer program with 2 nodes and 8 cores (4 cores per node):

```
sbatch 2node8core.slurm
```

We changed the resource configuration by changing the value of `nodes` and `ntasks` in the script. It's worth noting that the multi-node partition has to be invoked in the script and here is an example:

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks=8
#SBATCH --time=0-12:00:00
##change the partition to sapphire if needed
#SBATCH --partition=cascade

module load mpi4py/3.1.4
module load Python/3.10.4
time mpiexec -n 8 python3 main.py
my-job-stats -a -n -s
```

4 Result

Table 1 illustrates the happiest hour and day and the most active hour and day as analyzed from our application data.

Happiest Hour	2021-12-31 13:00-14:00	3658.95 Sentiment Score
Happiest Day	2021-12-31	30402.26 Sentiment Score
Most Active Hour	2022-05-21 12:00-13:00	39061 Tweets
Most Active Day	2022-05-21	424372 Tweets

Table 1: Result

Job	Node	Core	Real Time	CPU Efficiency	Memory Efficiency	Partition
57840303	1	1	00:31:47	99.8%	3.4%	Sapphire
57858286	1	8	00:04:07	97.17%	3.4%	Sapphire
57861310	2	8	00:05:24	89.00%	1.8%	Sapphire
57854832	1	1	00:55:21	99.25%	3.53%	Cascade
57864512	1	8	00:07:08	96.90%	3.41%	Cascade
57854837	2	8	00:07:30	94.64%	1.82%	Cascade

Table 2: HPC Run Time Summary

5 Performance Analysis

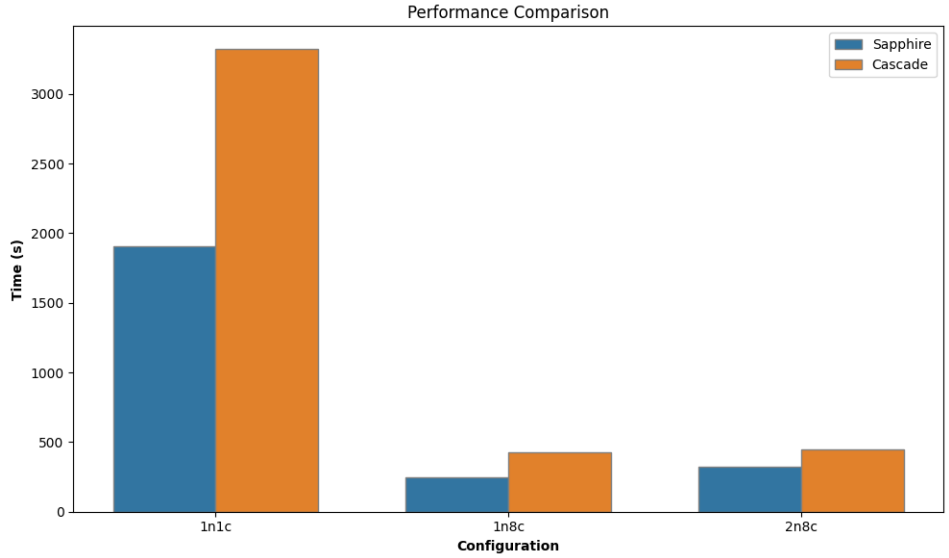

Figure 1: Performance on Different Partitions, Nodes and Cores

During our experimentation phase, we encountered significant variations in execution times when running identical scripts through the same `slurm` scheduler at different intervals. We attributed these discrepancies to the scripts being executed on different partitions within the Spartan HPC system. To address this, we explicitly specified the target partition in our SLURM script to ensure consistent execution environments.

Figure 1 illustrates the superior performance achieved on the Sapphire partition, utilizing a single node with one core, in comparison to its Cascade partition counterpart. This discrepancy is primarily due to the Sapphire partition's higher peak performance per node, as detailed in the Spartan HPC system specifications. Moreover, the data presented in Figure 1 demonstrates notable reductions in execution time when employing 1 node with 8 cores and 2 nodes with 8 cores, respectively. These results underscore our application's scalability, showcasing its ability to efficiently leverage increased

Node	Core	Time for 100Gb(s)	speed up	Time for 50Mb(s)	speed up	Partition
1	1	1907		3.4		Sapphire
1	8	247	7.72	2.2	1.55	Sapphire
1	1	3321		3.3		Cascade
1	8	428	7.76	2.1	1.57	Cascade

Table 3: speed up for 100Gb and 50Mb files with different cores

computational resources for enhanced performance.

In both the Sapphire and Cascade partitions, we observed that the execution time for configurations using 2 nodes with 8 cores was consistently longer than those utilizing 1 node with 8 cores. This outcome underscores the principle that inter-node communication tends to be less efficient than intra-node communication due to the inherent latency in data transmission between nodes. The diminished CPU efficiency observed in the 2-node, 8-core configuration serves as further evidence of this communication overhead.

Table 3 illustrates the performance improvement achieved by scaling up from a single-core, single-node configuration to 8 cores on a single node. The speedup factors observed are 7.72 and 7.76 for the sapphire and cascade partitions, respectively. These results closely approach the theoretical maximum speedup of 8, indicating our application is highly parallelizable with minimal sequential constraints.

However, it is crucial to interpret these results within the framework of Amdahl's Law, which posits that the potential for performance improvement in a system is inherently limited by the portion of the system that is not parallelizable. This principle is particularly relevant in our context, where specific segments of our program remain inherently sequential—most notably, the initial line read operation (`tweet_file.readline()`) to ensure the completeness of tweets at the start of each data chunk, and the final result aggregation executed at rank 0. These segments are not parallelizable and thus impose a ceiling on the achievable system-wide speedup.

In addition, the speed up of processing 50Mb file is only about two on both partitions, which is much smaller than that for the largest 100Gb file. This result is consistent with Gustafson-Barsis's Law: if we were to scale the problem size with the number of cores (i.e., process larger files with more cores), we might continue to see linear or near-linear speedups.

6 Conclusion

This project demonstrates the practical implications of Amdahl's Law through parallel processing of a large JSON file with MPI. Our results confirm that while parallelism can drastically reduce processing times, the speedup is inherently limited by the sequential fraction of the task. As evidenced in our experiments, detailed in Table 2, the gains from parallelism are less pronounced when the computation involves smaller datasets or requires significant inter-node communication, resulting in increased overhead.

Consistent with Gustafson-Barsis's Law, we find that parallelism is most beneficial when applied to large-scale problems. In scenarios where the problem size does not justify parallel overhead, such as with small datasets, a single core may suffice. Thus, the optimal use of parallel computing requires careful consideration of the dataset size and the computational overhead to realize meaningful improvements in performance.