COMP90024 Assignment1 Report

Yifei Zhang: 1174267, Cheng Chang: 1270486

1. Introduction

In this assignment, we implemented a program that processes 1MB, 50MB and 100GB of data through implementing a parallelized application, then intends to find out the most active/happiest hour and day ever. During the implementation, we found out that brutally iterating through a 100GB file will be extremely time consuming. Hence, with the combination of "Amdahl's Law" and "Gustafson-Barsis' Law", various optimizations are being developed and shown in this report. As 1MB file is too small to make comparison between approaches, we are only going to demonstrate performance on 50MB and 100GB files.

2. How to Invoke Program

Three different slurm scripts that run a certain approach with node resources are attached with code. The script identifies the version of Python and mpi4py, resources will be used, and includes which approach will be running. To run the script on Spartan, first need to change the approach python file identified within the script, then using 'sbatch' command followed by script file name, for example, 'sbatch 1node1task.slurm'. To run scripts for different sizes of file (100GB, 50MB), you need to change the file path in the corresponding approach file. The approach file will be running on a 100GB file with the best approach we implemented (optimization3) by default, only need to type following commands to run each task: "sbatch 1node1task.slurm", "sbatch 1node8tasks.slurm", "sbatch 2nodes8tasks.slurm".

3. Approaches used in improving performance

In this assignment, we implemented 3 approaches and optimization to parallelize the problem: A1_optimization1.py (baseline), A1_optimization2.py (optimization2) and A1_optimization3.py (optimization3). Key differences of each optimization is explained below.

Data Reading: In baseline, we try to process the string from file line by line, and use regular expressions to match the string segment we need. Further in optimization2, we improved this reading method by changing it to byte chunk reading. Instead of reading each line in the file,

we calculate total bytes of the file then split it into several equally large chunks. However, this will cause programs to miss out information or duplicate reading. Hence we fix this problem in optimization3 by reading to the end of the line whenever the chunk reaches the end.

Data Storage: All approaches implemented in assignment are using the same storage method: using the default dictionary. Using time as key and number of tweets as value. This method will have O(1) time complexity when storing and reading data.

Data Processing: In baseline, optimization2 and optimization3 we use regular expressions to match string segments. Note that in baseline and optimization2, if there is no number followed by word “sentiment”, we simply don’t count that tweet, but in optimization3, we consider it as sentiment score 0.

Data Transmission: In baseline, we read each line of file or json object terms by terms, which means each rank will read a line then wait for the other rank to read. We improved this method in optimization2 and optimization3. We split the file into chunks of byte. Each rank no longer needs to wait while the other rank is reading, as each rank has their own chunk to work on. After we are done with data processing, we use the ‘gather’ method to collect results from each rank to the root rank. Then root rank will be responsible for final data merging and printing out results.

4. Compare Performance

To better visualise the performance of each approach, we created a table and bar chart for performance (number of time in second) of each approach when running a 50MB file 100GB file, under corresponding resources.

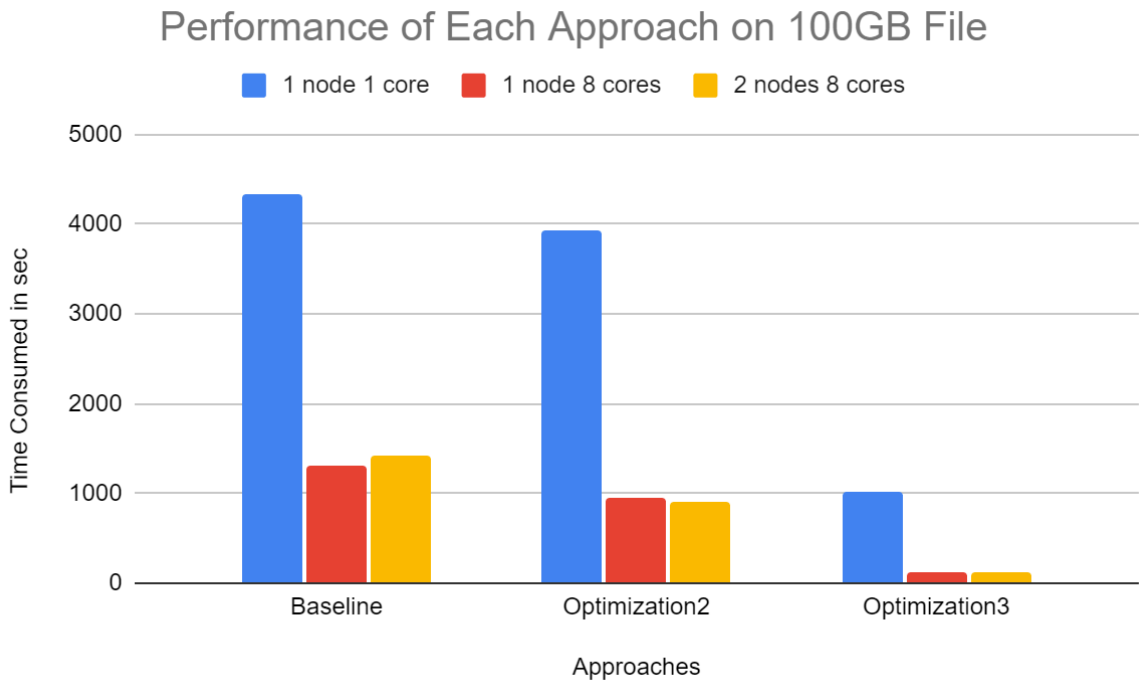
Performance of 50MB file			
Approach / Resources	**1 node 1 core**	**1 node 8 cores**	**2 nodes 8 cores**
Baseline	1.871 sec	1.093 sec	1.523 sec

Optimization2	1.714 sec	0.478 sec	0.452 sec
Optimization3	0.821 sec	0.226 sec	0.233 sec

Table 1: performance of each approach running 50MB file

Performance of 100GB file			
Approach / Resources	**1 node 1 core**	**1 node 8 cores**	**2 nodes 8 cores**
Baseline	4336.799 sec	1308.079 sec	1424.035 sec
Optimization2	3928.893 sec	942.280 sec	895.101 sec
Optimization3	1007.259 sec	129.270 sec	127.344 sec

Table 2: performance of each approach running 100GB file



Graph 1: Performance of each approach when running on 100GB file

5. Analysis

The results obtained from running the three different approaches both on a 50MB and 100GB file presents valuable insight into the effectiveness of parallelization in handling large scales

of data. Optimization 3 shows to be the best algorithm in processing both 50MB data and 100GB data across all configurations as evidenced in table 1 and table 2. The efficiency is significantly demonstrated in multiple cores and nodes due to parallelism.

Parallelism particularly improves performance by distributing the workload across multiple processing units, such as dividing the data into smaller chunks and assigning them to different processing units and all ranks are able to read and process the data at the same time, hence minimising the required processing time.

According to "Amdahl's Law", the theoretical speedup with parallelism would be:

$$Speedup = 1/((1 - P) + P/N)$$

If we are able to achieve 100% parallelization, theoretical speedup should be 8 times with 1 node 8 cores configuration. In the case processing 100GB data with optimization 3, 1 node 8 cores configuration is 7.79 times faster than 1 node 1 core configuration, and 2 node 8 cores is 7.9 times faster than 1 node 8 cores. The multiple core processing is not able to achieve the theoretical 8 times speedup is due to the serial parts of the programing which can not be parallelized, such as only the root node is calculating and printing the final result. In these files, 100% parallelization is not possible, however, according to "Gustafson-Barsis' Law", when parallelizable tasks increase, the speedup ratio will also increase.

The performance difference between 1 node 8 cores and 2 node 8 cores can be related to the data reading stage. Although data is split by bytes to each rank, the reading process is still processing line by line. This leads to some lines including emoji which have larger size to be taking more time to process hence ranks ready smaller bytes need to wait for this progress. Furthermore, this performance difference can be also due to transmission overhead. While parallelism enables concurrent processing, communication between nodes also introduces latency then causes performance difference.

In conclusion, the report presents how parallelization programming enhances performance with large datasets. Through optimization like bytes chunk reading and parallel processing, outstanding performance improvements are able to be achieved.