**COMP90024 Assignment 1 Report**

**Student: Yixuan Liang, Hongzhi Fu**

**Student ID: 1094908, 1058170**

**Large File Loading**

To accomplish parallelism across all processes, each process should be capable to access any tweet entries in a file. A naïve method is to use f.readlines() function built-in Python library to load the entire tweets JSON file to the main memory and access a specific tweet via list index. However, because of limited memory space, it is not advisable to load 14 gigabytes of JSON file into the memory, so what we need to do is read a batch lines of tweets for each iteration, load them into memory, do some calculations, and finally release the memory space. Although the benefit of limited file reading is to solve the problem of memory overflow, each process must iterate the big file to access a random line of data, rather than read them in constant time complexity. Hence, we come up with an idea that pre-computes each line of offset with respect to the head of the file.

When opening a big file, we have a file pointer f, which points to the header of the file. To read a specific line of text, we should first set the current file pointer to a new position which is relative to the file header, e.g. f.seek(offset) means set the file pointer f to offset position with respect to the file header. Next, we can read a line of data by invoking f.readline() method. The file pointer should know the offset value corresponding to each line before accessing a random line of data, thus we can store these offset values into an external file line2offset.txt in advance. So we can have random access to a line of tweet data by accessing offset values in line2offset.txt to help the file pointer seek the right position in a file stream. The intuition is shown in Figure 1.

文本

描述已自动生成

**Figure 1**. The Intuition of Process from Line to Offset

The script to compute each line’s offset is line2offset.py , where we iterate the whole file, count the number of characters in each line, and compute the accumulated number of characters as the offset values. For example, as the first line has 48 characters, the offset value of the first character in the second line should be 0+48=48, which will be then stored in the second element in the text file line2offset.txt. Before processing a random tweet, we should first load the offset values from disk into main memory as an array, and access the corresponding offset value via array index. Lastly, set the file pointer to the offset value and we can read a specific line of data.

The running time of computing offset values on bigTwitter.json is about 2 minutes and has the space of 43Mb. It means we have fast access to any tweet data by pre-computing the offset values.

**Parallel Computing Approach**

The MPI program is written in Python language using the mpi4py package. The program starts by initiating the MPI world and getting environment variables such as node sizes, node rank, and node status. All nodes are assigned with different functions, one node being the master node and the rest being the worker nodes. The master node is responsible for allocating tasks to different worker nodes whenever they are ready, in other words, idle for more computational tasks, while worker nodes are mostly responsible for implementing parallel computing. The process includes a master node giving instructions to each worker node on which part of the calculation should be done repeatedly until all tasks are completed, and each worker node performs the assigned task and sends back the result to the master node. In the end, the master will aggregate all results and return a final output.

In our final solution, the architecture is shown in Figure 2 with an example of the program working on three nodes. The communication between the master node and each worker node is independent with each other. Therefore, there will not be any dependencies between worker nodes and thus avoid unnecessary waiting between worker nodes. The approach for this solution is to implement tag during the MPI point-to-point communication between master node and worker node. The process consisted of four steps:

* Step 1: An idle worker node sends a READY tag to the master node.
* Step 2: The master node receives the message, assigns the indexes of the tweets needed to be processed by this worker node with a START tag.
* Step 3: The worker node receives the indexes. It loads the corresponding tweets from the huge file without opening the entire file and processes the tweet. Once it is finished, the worker node sends back the result to the Master node and with a DONE tag. When the master node receives the result, it aggregates them with the previous batches.
* Step 4: When all tasks are assigned, the master node will send an EXIT tag to the worker node when it gets another READY tag. The worker will terminate itself by not doing anything. When all the worker nodes terminate themselves, the master node will return the final results.

The design of the architecture is iterated through several versions. In our first attempt, we used an allocation method that would make the worker nodes dependent on each other. Even though they are calculating at the same time parallelly, new tasks would not be assigned to the worker node until all worker nodes are finished with their last batches of tasks. This would result in unnecessary waiting time if some of the worker nodes are much slower than the other nodes.

手机屏幕截图

描述已自动生成

**Figure 2.** An Example of Architecture of Master-Worker Communication

In our second attempt, we changed our architecture to the point-to-point communication with tags, which is shown in our final solution. However, the result did not improve much when we tested them out using the bigTwitter.json on Spartan. The results for the two architectures are 647.8 and 122.9 seconds respectively when using 1 node 8 cores. This means the resources assigned to the program are fairly the same in terms of computational power. In these two attempts, we used the master node to read through each tweet and assign a batch of tweets to each worker node. When we change this approach to our final solution, where the master is only assigning indexes of the tweets and worker nodes read the file themselves, the process time is reduced in one-fifth when using the same amount of resource.

**Submit Script Explanation**

The example submit script is shown below, which is the specification of one node eight cores.

#!/bin/bash

#SBATCH --nodes=1

#SBATCH --ntasks-per-node=8

#SBATCH --time=0-1:00:00

module load foss/2019b

module load python/3.7.4

srun -n 8 python3 tweet\_mpi.py

The command #SBATCH --nodes=1 means we will use only one node for computation. For the resources we want to use within a node, we can modify ntasks-per-node parameter. In the above example, we will use 8 cores for each node. If we want to change the maximum computational resources to 1 hour, we can use command

#SBATCH --time=0-1:00:00. After specifying node and core parameters, we need to configure OpenMPI and Python environment by running module load foss/2019b

and module load python/3.7.4. The final step is to run Python script for processing twitter data.

**The Final Result**

The result for our solution processing the bigTwitter.json file on Spartan using 1 node 1 core, 1 node 8 cores, and 2 nodes 8 cores are shown in the bar chart below.

**Figure 3.** Comparison of Running Time between 3 Different Specifications

Timeline

Description automatically generated

**Figure 4.** Final Output for bigTwitter.json

The bar chart demonstrates parallel computing specifications (1n8c and 2n8c) have a significant improvement on performance compared with sequential computing (1n1c). Numerically, the running time of parallel computing is only one-seventh of that of sequential computing. The main reason why 2n8c is slower than 1n8c is the communication time between nodes is longer than communication between processes, because of network overhead.

The final output of our solution is shown in Figure 4, with C2 being the most happiest city in Melbourne area with the most tweets collected.