COMP90024 Cluster and Cloud Computing Assignment1

HPC Twitter GeoProcessing

Xinjie Lan 910030, Liuyi Chai 971843

1.Introduction

The project is to design a simple parallelized application to identify Twitter usage around Melbourne which is categorized in sixteen individual grid cells. The program calculates several posts for each grid cell and the corresponding top five occurring hashtags for these grid cells. The application allows a given number of nodes and cores to be utilized. This report would briefly describe the approach we took to parallelize our code and analyze the performances of executing the solution on different resources.

The application is written in Python and implements MPI to run in a parallel way. The application runs on the University of Melbourne HPC facility SPARTAN with the dataset of bigTwitter.jsom which includes a great amount of data of the coordinates of tweets and melbGrid.json which specifies the ID and range of the gridded boxes. The program is tested under three scenarios:

- One node with one core
- One node with eight cores
- Two nodes with eight cores

2.Description of the program structure and code

The overall parallel structure of this program is Single-Program-Multiple-Data(SPMD) which is that all the processes execute the same piece of code but with different parts of data. All the processes parse both the melbGrid file and the bigTwitter file at the same time, but different process only deals with its own parsed dataset. After all the processes finish their jobs, the specified master node which is rank0 gathers all the results from other nodes/processes. The master node then combines all the results and calculates the rankings. The code can be summarized in three parts:

2.1 Load and parse the grid file

The first part aims to load the melbGrid file and extract the data. To implement this, we first initialize MPI instance and every process stores the grid data into a dictionary which contains each grid cell's ID and their range of coordinates. All the processes do the same work here.

2.2 Parallel parse the Twitter file and calculate the results

The main job of the second part is to parallel parse the bigTwitter file. When there is only one node, the program should read all the lines one by one, and this is accomplished by using a flag as shown in figure 1.

```
else:
    #Initilizing a swtich which makes sure that the program can run under two cases:
    # 1 multi-process
    # 2 single process

    if size > 1:
        dontSkip = False
    else:
        dontSkip = True
    # If there are more process, each process read their own line. If there are just one process, read
    if rank == (count+1)%size or dontSkip:
```

Figure 1: Flag & Rank Number

When multiple nodes/cores exist, each process only reads the line that belongs to it. In order to evenly allocate workload for every core that we are using, we decided to use modulo. If the rank number equals to the current line plus one total modulo number of cores, then this line belongs to the core that has the same rank number. When dealing with parsing, as shown in figure 2, we only stored the coordinates and the hashtags of each tweet into a dictionary and then appended the dictionary objects to a list.

For the hashtags, we assume that only a pattern matches “Space#abcSpace” can be viewed as a valid hashtag, so we used function split() with a range[1:-1].

```
#Store the coordinates into the dictionary
tweetDict["coord"] = singleCoord

#Getting the hashtag from text field
#By not using the regex, use split() to match the parttern of " #String "
rawText = row["doc"]["text"]
hastags = rawText.split(" ")[1:-1]
for hashtag in hastags:
    if hashtag.startswith("#"):
        hashtagList.append(hashtag.lower())
        tweetDict["hashtag"] = hashtagList

tweetData.append(tweetDict)
```

Figure 2: Storing tweet data

After parsing, each process classifies the coordinates into the cell it belongs to, and then count the number of the posts along with their hashtags in this cell. The results are stored in a dictionary type.

2.3 Gather data from all processes summarize the results

```
gatheredGridData = {}
if size >1:
    gridCount = comm.gather(gridCount,root=0)
else:
    gridCount = [gridCount]
```

Figure 3: MPI.gather

In part3, the master node gathers the data from all the processes by the MPI call comm.gather as shown in figure 3.

```
postRankingList = []
hashtagRankingList = []
for grid in gridData:
    if grid["gridId"] in gatheredGridData:
        #Rank the top hashtags for each cell by Counter.mostcommon
        #After sorted, compare each hastag from the begining, if there is a tie, count them both as the top5
        if "hashtags" in gatheredGridData[grid["gridId"]]:
            rankings = collections.Counter(gatheredGridData[grid["gridId"]]["hashtags"]).most_common()
            count = 0
            result = []
            rankingDict = {}
            if len(rankings) ==1:
                result.append(rankings[0])
            else:
                for i in range(0,len(rankings)-1):
                    if count == 5:
                        break
                    else:
                        if rankings[i][1] != rankings[i+1][1]:
                            result.append(rankings[i])
                            count +=1
                        else:
                            result.append(rankings[i])
            rankingDict[grid["gridId"]] = result
            hashtagRankingList.append(rankingDict)
            postRankingList.append([grid["gridId"],gatheredGridData[grid["gridId"]]["count"]])
```

Figure 4: Processing results

Since the gathered data is in a list, the master node firstly parsed the dictionary from the list and then categorized the data into a new list which is used for ranking. The master node compares each element in the sorted hashtag ranking list so that a tie will be displayed as well(Figure 4). From the ranking list, sort the list by a number of posts for each grid cell, and we counted the number of occurrences of the hashtags and stored the top5 into a post ranking list.

3. Utilizing Spartan

For the scenario that is using two nodes and eight cores, we used physical partitions and allocated four tasks to each node, and each task has one core. With the scenario that is using one node and eight cores, we assigned eight tasks which each one has one core. As for the one node one core situation, it is straightforward that only one task with one core is needed. Both one node situations used cloud partitions. The scripts for eight cores can be found in Figure 5, and the script for one node one core can be found in the zip file because of the length limits of this report.

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --output 2n8c8tasks.txt
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --time=0-00:15:00

# Load required modules
module load Python/3.6.1-intel-2017.u2
# Launch multiple process python code
time mpirun python3 Assignment1.py -i /data/projects/COMP90024/bigTweet.json
```

```
#!/bin/bash
#SBATCH -p cloud
#SBATCH --output 1n8c8tasks.txt
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --time=0-00:10:00

# Load required modules
module load Python/3.6.1-intel-2017.u2
# Launch multiple process python code
time mpiexec python3 Assignment1.py -i /data/projects/COMP90024/bigTweet.json
```

Figure 5: Slurm jobs for 2 nodes 8cores(left), 1node 8cores(right)

4. Results & Discussion

The output of the program is stored in zip file. The time performance can be found in figure 6. One-node-one-core allocation took the longest time to run the program. Although for the eight-core scenario it is almost three times faster than the one-core scenario, there are not many differences between one and two nodes.

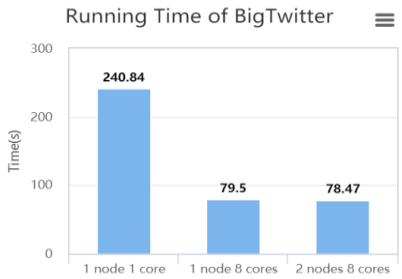

Figure 6: time performance for different resources

The reason that running in eight cores is faster might be that we designed a parallel system efficiently. Since we allocated cores for eight tasks, eight processes are doing the same calculation but with different parts of data at the same time. In this way, processing time is greatly reduced. However, even with eight cores, it is only three times faster. The reason for this might be that the master node does a time-consuming job itself when summarizing the results. It has to process a tedious job which needs to search in the list and extract necessary data then calculate the final results. This is also confirmed with Amdahl's law that with 75% of the program is parallelized, the speedup is up to 4 times faster. In the future, how to generate a more convenient result for the master node to work can be studied.

As the performance is similar for both one and two nodes with eight cores, we infer the reason might be that we used physical partitions for two nodes which provided better network environments and reduced the time that it takes to communicate between nodes. Also, since our program only gather the data when processes finish their calculating, communications between two nodes happened less frequently. Thus, even in a shared memory way, the performance is still good.