COMP90024- Clustering and Computing
Assignment1

Sejin Kim 1025590
Rajeong Moon 972583

1. Introduction

Understanding user interest and defining target user groups is getting more important than ever in social network service platforms. However, finding meaningful information from the huge amount of data gained from a number of users can be costly in time or money-wise.

Our initial problem lies in making a program that processes huge amounts of data input in a parallel manner, using MPI. To be more specific, the problem situation we wish to solve is to build a program that works in parallel to process data with large size(big data) with the error. Information of interest is 10 most frequently used hashtags and languages in a large given JSON file.

2. Environment Setting

⑴Used modules

The program includes packages such as mpi4py, JSON, emoji. A key module to make this program run in parallel is mpi4py, which is a python library for MPI that enables to make program works in parallel using multiple cores. Another important module is the JSON module which can enable us to parse JSON files and get information easily by searching keys.

⑵Spartan server setting

In this project, we evaluate our program in Spartan, which is a server in a cloud computing environment. The test was conducted under the condition of 1 node 1 core, 1 node 8 cores, 2 nodes 8 cores. Slurm script for this setting and job scheduling will be presented in section 3.

Python version 3.7.3-spartan_gcc-8.1.0 is used with needed libraries imported. This version was chosen in an attempt to match versions with the developing environment on the local machine.

⑶Dataset

To evaluate the performance of the developed program, a JSON file(bigTwitter.json) which has 20 GB data was used. When processing big data in the cloud environment, approaches to prevent error due to memory exhaustion and speed up the running time are considered.

3. Program Implementation

⑴Program invocation - *Slurm job*

As mentioned in section 2, Spartan is a shared cloud computing environment, the program needed to be invoked with job scheduling called slurm. Slurm job is scheduled by submitting a slurm script using the command 'sbatch <slurm_file_name>'. Below will demonstrate a brief view of the slurm script used to invoke the program for the project.

Single node slurm(1 node 1 core)	Multi node slurm(eg. 2 node 8 core)
#!/bin/bash #SBATCH --partition=physical #SBATCH --nodes=1 #SBATCH --ntasks=1 #SBATCH --cpus-per-task=1 #SBATCH --time=0-5:0:00	#!/bin/bash #SBATCH --partition=physical #SBATCH --nodes=2 #SBATCH --ntasks=4 #SBATCH --cpus-per-task=1 #SBATCH --time=0-5:0:00

# The modules to load: module load Python/3.7.3-spartan_gcc-8.1.0 # The job command(s): mpirun python CCC-Assignment1.py bigTwitter.json 1n1c	# The modules to load: module load Python/3.7.3-spartan_gcc-8.1.0 # The job command(s): mpirun python CCC-Assignment1.py bigTwitter.json 2n8c

(2)Parallelize approach

-mpi4py and communicator

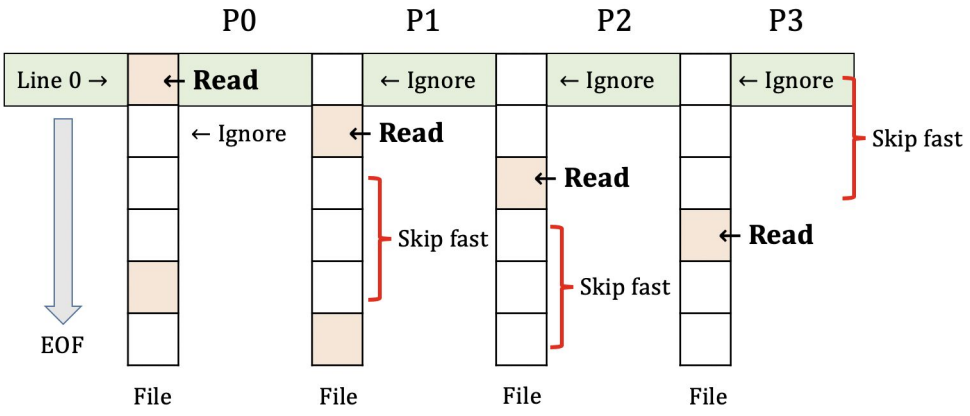
The parallelization of code starts with importing MPI packages. 'comm' is an MPI communicator instance and rank is the process number. Size indicates how many processors are used in a program. In MPI parallel programming, 'rank' and 'size' are used often enough to handle processors. It can be implemented as shown in the snippet below and these mpi4py operations are used in the project.

from mpi4py import MPI comm = MIP.COMM_WORLD rank = comm.Get_rank() size = comm.Get_size()	recvbuf = comm.alltoall(sendbuf) #scatter and gather at the same time recvbuf = comm.gather(sendbuf, root=0) #gather data from send buf and receive to recvbuf

-How each node read big size file to avoid memory exhaustion (resource distribution)

Reading data from a file of large size on a single processor consumes a lot of time and puts a burden on memory as well. To deal with this issue, the program does not load the JSON object entirely, but every processor in the program partially reads the object line by line inside the file.

Figure 1 below demonstrates how the file reading process happens on each node. Each node has its file object but rather than reading the whole file separately on each node, each processor handles the file object by using the 'lineNumber' variable starting from 0 to EOF.



[Figure 1. File reading process on each node]

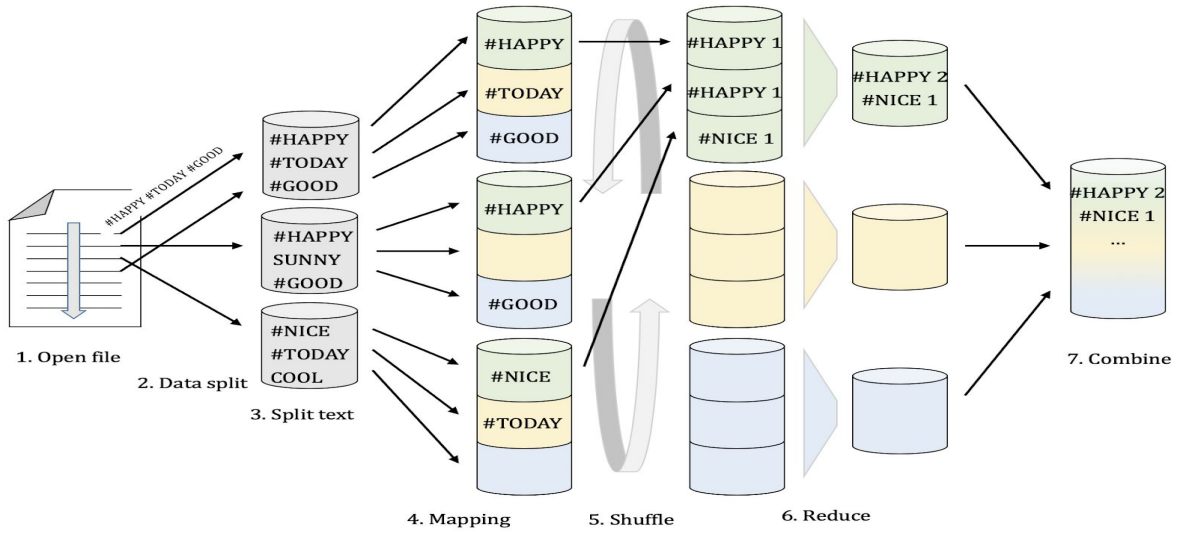
To read and access inner JSON objects fast, each processor ignores most lines of the file and only considers specific lines by checking the remainder of the line number of the file divided by its process id. If the remainder is the same as the processor ID, the processor loads the inner JSON string into the object, and if not, ignores the data of the line. Thus, processors can skip the set of data fast. For example, if action 'read' takes 10 seconds and 'ignore' in the above Figure 1 takes 1 second, the program can save the 36 seconds in this way. Another advantage of this approach is that the program does not need to distribute data from the root(master)node to others. Each processor can get the non-overlapping partial data by reading data at an interval of the number of processors. This approach of reading a single line and parsing it can be done as each line of a given JSON file consists of '<complete JSON object>,\n' .

-How to shuffle sets of data between processors

To count the number of the same hashtags, in the mapping process(below Figure 2), each processor produces a unique integer value of the hashtag string to use it as a sort of hash value to be used when searching for the same

hashtags using matching integers. As a result, every same hashtag belonging to a different processor can be stored on the same index, which is produced as a hash-like integer value in the earlier step, of the list which stores entire hashtags on a processor. Then to count the number of a hashtag in the entire JSON file, the program should gather data which has the same index value of the list in all processors into one of the processors. In this process, the program uses 'comm.alltoall()' the operation to execute scatter and gather at the same time. By using this operation, the program does not have to send MPI messages, one by one.

(3) How it works (text preprocessing - hashtag extracting - shuffling - reducing - merging)



[Figure 2. Data processing flow]

To extract data from large size data in a fast and efficient way, the concept of Map Reduce is used for the project. Figure 1 depicts the 7 major processes of data processing. Firstly, the program opens the JSON file, not loading the entire data. Then each processor of the program starts to read line by line. When the processor reads the line, it parses the string data which is allocated to the processor based on the processor id into a JSON object. Therefore, as presented in Figure 1, every processor read the file at the same time, but only the line corresponding to its processor id is processed. In this way, duplicate data processing does not occur. After the processor obtains the JSON object, the processor extracts specific fields of data and splits to filter whether a split text is a hashtag or not. In the 4 stages, each processor calculates an index like hashing function to mapping data into one of the partitions in the processor. Due to the partition key value, data sets only have the same partition key are gathered into the same processor and do the reduction process. In this phase, the processor has the only big dataset which consists of unique hashtags. Lastly, every processor sends the data into a processor to combine whole data.

4. Result and program performance

-Result

The program gathered information of interest from text tags of the JSON file which is what the user wrote. It also gathered the same information from retweeted and quoted_status tags. This approach enables to include hashtags or language code not only from what the user wrote but the weight of importance(influence) of each hashtag and language by how many times certain tweets are retweeted or quoted.

To get results below, the program used regex expressions to find hashtags rather than using tokenizers in nltk library. This approach is for getting only desired result on hashtags that completes hashtag rule of Twitter, rather than getting any tokens to start with "#" but are not considered as actual hashtags.

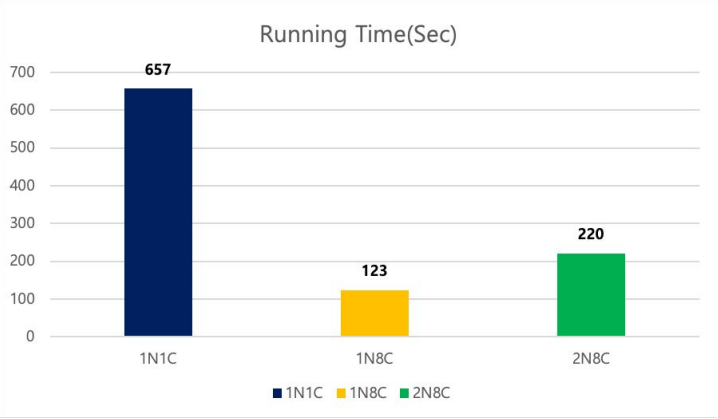
Top 10 hashtags and languages used in 'bigTwitter.json' are presented below.

<Top 10 hashtag used>		*<Top 10 languages used>*	
1. #AUSPOL	35,014	1. English (en)	4,860,171
2. #CORONAVIRUS	17,628	2. Undefined (und)	358,515
3. #มาพ่องเพิ่งอะไร	15,060	3. Thai (th)	259,178

4.	#FIREFIGHTAUSTRALIA	12,964	4.	Portuguese (pt)	167,718
5.	#OLDME	12,776	5.	Spanish (es)	117,940
6.	#GRAMMYS	10,103	6.	Japanese (ja)	73,689
7.	#ASSANGE	9,232	7.	French (fr)	64,890
8.	#SYDNEY	8,539	8.	Tagalog (tl)	64,450
9.	#SCOTTYFROMMARKETING	8,507	9.	Indonesian (in)	57,879
10.	#SPORTSPORTS	8,117	10.	Korean (ko)	33,128

In the top 10 frequently used languages, language code[1] 'und' is collected and dealt as a kind of language although it actually means 'undefined'. We adopted this point of view as it is stated on the problem that to consider language code to count frequently used language and in the given JSON file 'und' is paired with 'iso_language_code' key.

-Program performance



[Figure 3. The result of running time(sec) with 3 different conditions]

As a result of the project, Figure 3 shows the running time of different computing environments. The above 1N-1C's result is the benchmark to compare the performance of a parallel approach. 1N-8C and 2N-8C use a total of 8 processors when the program is running on the Spartan. From the result, 1N8C shows the best performance with 123 seconds then 2N8C follows by 220 seconds. Compared to the single-core approach, both parallel programs produced time savings of about 60 per cent. Compared with the performance of 1N1C, it can be seen that 1N8C has about 4 times better and 2N8C has about 3 times better performance. However, when comparing the results of two parallel programs using the same number of cores, the 1N8C and 2N8C showed approximately twice the performance difference. 1N8C does not need to pass from one node to another in exchanging messages between processes like 2N8C because 8 processors are running in the same node. Considering the above results, it can be seen that the execution time of a task is reduced if the tasks are processed simultaneously in multiple nodes through parallel programming properly.

5. Conclusion and recommendation

The result in section 4 implies performance improvement regarding data processing time. With increased processors, the time getting result is much less than doing the same job in one node.

During the development phase, the first approach we took was the master node has access to all data and sends it to other nodes line by line. It was a very costly procedure that took hours to finish its job. Changing the data access method as mentioned in section 3 (2), the performance result improved significantly from hours to less than 3 minutes. This conveys the importance of load sharing on the data access phase as well instead of only on the data processing phase. An improvement can be suggested regarding parallel data accessing methods. Although a parallelized data approach was implemented as mentioned in section 3 (2), the same issue can be handled in a more elegant way by implementing MPI I/O. With MPI I/O technique, it is expected to be able to access one common shared file and read part of it, which can save resources even more.

[1] IOS language code as they are referred in https://developer.twitter.com/en/docs/twitter-for-websites/twitter-for-websites-supported-languages/overview