Cluster and Cloud Computing Assignment 1
HPC Twitter Processing

Xinzhe Li 825797
Zhaofeng Qiu 1101584

University of Melbourne — April 6, 2020

1 Introduction

The main goal of this project is to implement a parallelized application to identify the top 10 most used hashtags and languages among a considerable set of tweets on an HPC system. The performance of the application is also measured and discussed in the report. The running environment of the application is shown in Table 1.

HPC facility	SPARTAN [1]
Programming Language	Python 3
Package Used For MPI Programming	mpi4py
Dataset	bigTwitter.json

Table 1: Running Environment

2 Methodology

2.1 Dataset Handling

Pre-process tweets are available in JSON format (bigTwitter.json), which will run on the HPC platform. Some portions of it (smallTwitter.json and tinyTwitter.json) are extracted to be used as trials locally. Python is chosen to implement the dataset analysis logic.

2.2 Message Passing Interface (MPI)

MPI is a cross-language communication protocol used for parallel programming. Mpi4py is a powerful Python library that implements many interfaces in the MPI standard, including point-to-point communication, collective communication, blocking/non-blocking communication, inter-group communication, etc. We mainly use the gather function in it to collect parallelly processed data.

2.3 Slurm Script

The HPC platform we used is Spartan [1]. By writing slurm script to run our program, we measure our application's performance in different configurations. More detail is discussed in section 3.5.

3 Implement

3.1 Parallelization

Considering twitters in the dataset are independent, in our application, each process would only handle part of the data in the file parallelly. Since the size of a file is effortless to obtain, we can easily get the block size each process needs to handle by dividing the total size by the number of processes. In each process, it would traverse the data in its block and count the number of occurrences of each hashtag and language into two dictionaries respectively. After every process successfully manipulates the data, the rank-0 process would gather by MPI and do a sort to get the ranks. The parallelization process is shown in figure 1.

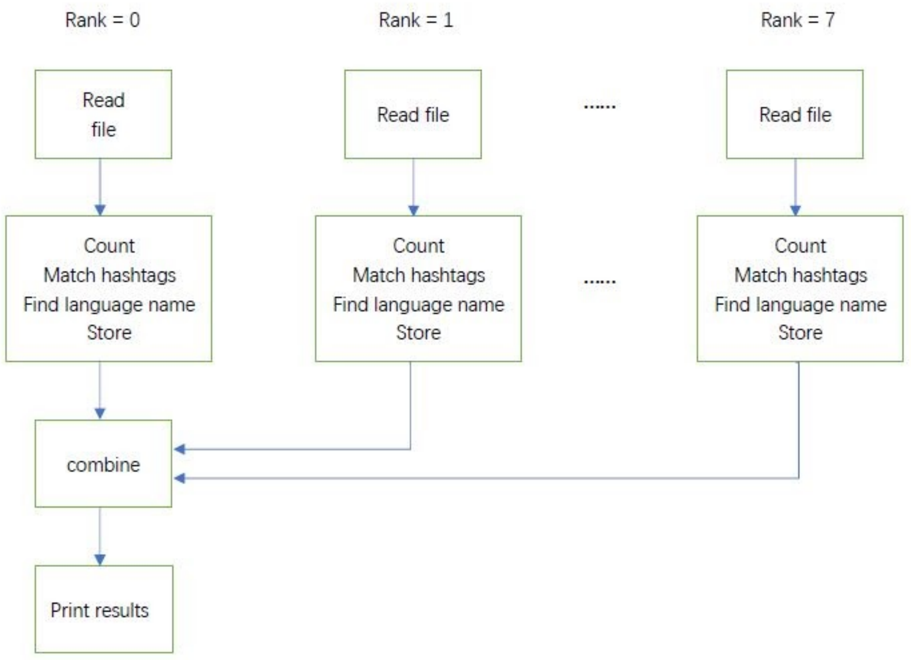

Figure 1: Parallelizaion

3.2 Parallel File Reading

In our application, each process would get its own file pointer and set its file position to a specific offset before reading. However, the JSON objects in the data file are separately by line, not by size. So, a simple trick is applied to address this issue: All processes, except the first one, would ignore the line it starts with (line with incomplete JSON data). They also read one extra line at the end of their assigned block. (Because the next process may ignore it) This trick ensures all JSON objects in the dataset to be completely recognized.

3.3 Use mmap

To improve the application performance, we use mmap [2] functions to read the dataset rather than normal file I/O functions. Using mmap can create a one-to-one mapping relationship between the file disk address and a virtual address in the process virtual address space. With the mapping relationship, the processes can use pointers to read the segment of memory, which can speed up the query and reading operations of the position of the file pointers.

3.4 Hashtags and Languages

Through manually analyze the JSON structure of the dataset, we find out that the attribute "hashtags" is what we want. It not only excludes incomplete hashtags like "#git..." which are incomplete due to being omitted but also contains all the hashtags in retweets. All hashtags are converted to lowercase for statistics. "Iso_language_code" and "lang" both record the languages and trials tell us they are the same. So, attribute "lang" is used in our final solution.To find the associated name (e.g. English) of a language code (e.g. en), we retrieve the supported languages and their codes from Twitter. For the languages not listed there, an ISO 639 list is used as a substitute.

3.5 Slurm

Slurm is a widely-used job scheduler for the HPC platform. A slurm script is used to invoke the job on Spartan. The slurm script for 2 nodes 8 cores is shown below. A series of sbatch commands allow us to specify the parameters for this particular execution. "–partition" allows us to choose whether to run this program on a cloud or a physical platform. "–time" is used for limiting the maximum runtime of our program. "–nodes" and "–ntasks-per-node" specifie the number of nodes and the number of cores per node, respectively.

2nodes8cores.slurm

```
#!/bin/bash
#SBATCH --partition physical
#SBATCH --time=00:20:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --job-name=2node_8core
#SBATCH --output=2n8c.out

module load Python/3.6.4-intel-2017.u2-GCC-6.2.0-CUDA9
time mpiexec python src/twitterAnalysis.py
```

4 Results and Discussions

Real-time is the focus here because the primary purpose of HPC is to pour in more computing resources to save time when dealing with massive amounts of data and very complex questions. The result of the three different configurations is shown in figure 2. "real" refers to actual elapsed time. "user" and "sys" represents the amount of CPU time spent in user mode and kernel mode, respectively.

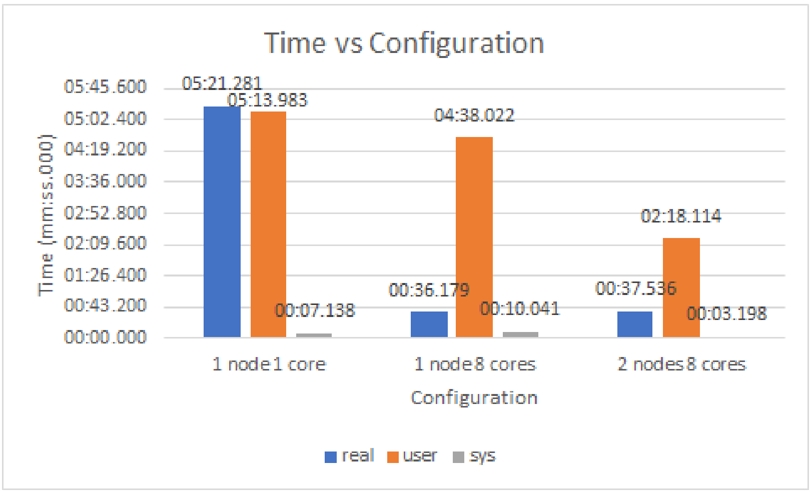

Figure 2: Time vs Configuration

4.1 Time

By comparing the real running time, user CPU time and kernel CPU time, we can find that:

$$real \times N \approx (user + sys) \times M \tag{1}$$

in which N is the number of cores and M is the number of nodes. Considering our application need to do Heavy I/O activity, "real" would be a better choice for evaluate the performance. ("real" includes the time waiting for I/O to complete)

4.2 1 core vs 8 cores

Compared to 1 core, 8 cores outperform by a significant margin. Parallelization contributes to this success as now 8 processes are performing the task, and each of them should theoretically only do $\frac{1}{8}$ of the total amount of work. However, we usually would not expect 8 cores to take less than $\frac{1}{8}$ the time of 1 core. As Amdahls Law points out, the speed can not scale accordingly with the number of cores because of the parts that can not be parallelized. Besides, splitting a task always creates overhead as now each process needs to initialize its variables and data structures. Due to the dynamic and uncontrollable (at least from our side) environment of Spartan, this could be just an accident. (In our local test, the time to use four cores is always greater than the time to use one core.)

4.3 1 node vs 2 nodes

Theoretically, 1 node is preferred than 2 nodes, when the number of cores is the same. If all the cores are located in the same node, it is expected that they can exchange messages in the same memory. If cores are located in multiple nodes, their communication may need to go through a physical cable or network, and that would take more time. When we set the "partition" on the slurm script from "physical" to "cloud", the difference is more obvious. The difference is shown in Figure 3.

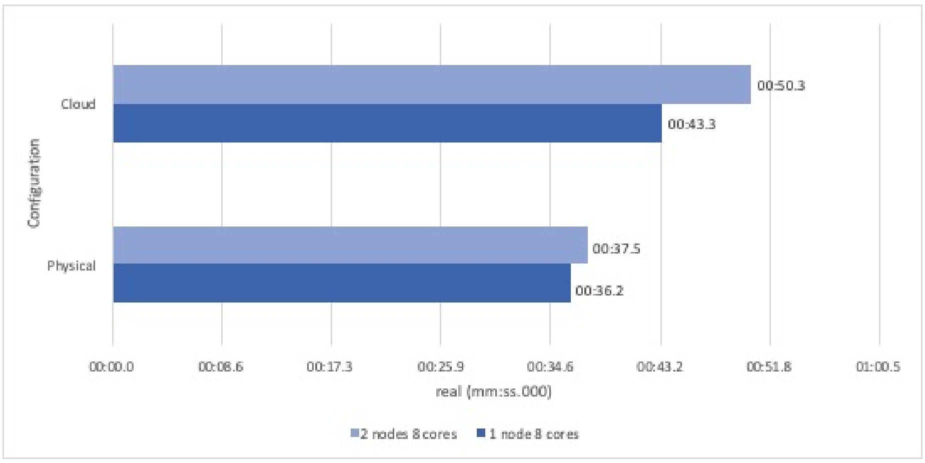

Figure 3: Real time vs Nodes

5 Conclusion

Our implementation successfully utilizes the HPC platform and produces an efficient solution to identify the top 10 most used hashtags and languages in a massive set of tweet data. The performance is measured, and the results, to a large extent, match the expectation. During this process, we review and reinforce HPC concepts, familiarize ourselves with MPI programming and extend our knowledge in practice.

References

[1] L. Lafayette, G. Sauter, L. Vu, and B. Meade, “Spartan performance and flexibility: An hpc-cloud chimera,” 2016. [Online]. Available: doi.org/10.4225/49/58ead90dceaaa

[2] “Memory-mapped file support.” [Online]. Available: https://docs.python.org/3/library/mmap.html