HPC Twitter GeoProcessing Report

Zichun Zhu, Xiaoyue Ma

1 Introduction

This report aims to describe the implementation of a parallelized project applying the University of Melbourne HPC facility SPARTAN. A brief description of how to identify Twitter usage and the top-5 hashtags from the given JSON files by using this application will be given.

There are four JSON files. The melbGrid.json file illustrates the geography information, and two small-size raw data files to test the application. After testing, it is necessary to search the total number of Twitter and the frequency of top-5 hashtags in each area from the *bigTwitter.json* file.

What is more, 1 node and 1 core, 1 node and 8 cores, and 2 nodes and 8 cores with 4 cores per node are utilized.

2 Methodology

2.1 HPC Parallelism

In general there are two approaches to parallelize a program: divide work by data, and divide work by steps. The chosen is highly depend on the specification of work. In geo processing, the number of steps per data entity is relatively small, while the size of whole data could be very large. Thus dividing work by data, and assigning working on a piece of data to different processes is more tractable.

2.2 MPI

There are different ways to classifiy parallel computers. The widely used classification, Flynn's taxonomy distinguishes computer architecture along instruction stream and data stream. According its definition, most current High Performance Computer(HPC) is in Multiple Instruction, Multiple Data(MIMD) type. Every process can work concurrently on different data stream, but those data and processors could be physically located in different place. To share the data between the processors, we applied MPI in the implementation.

Message Passing Interface(MPI) is an useful open source tool in multi-core programming which contains many functions to share message in processes. The main difference between OpenMP and MPI is that the processes using MPI does not shared memory between each other, while OpenMP is a shared memory multiprocessing programming library. MPI therefore able to be used in wide application regardless the physical architecture of platform.

3 Brief Description of Project Design

Two python files, __main__.py and reader.py, are designed for both sequence and parallel computing.

The main implements are as follows (Figure 1).

What is more, considering that the size of data could be up to 15 GB, even though divided into chunks it is still a heavy load message transmission. So rather than reading the whole data and then sending out pieces of raw data, the master process scans and records the line indexes, and then scatters the indexes instead. A simple calculation can imply the efficiency and sufficiency. In *smallTwitter.json*, there are about 7000 lines, each is one tweet, and the size of the file is about 25 MB. If we assume tweets density is even, a 15 GB file may contains:

$$tweet\_num = 25MB/15GB * 7000 \approx 4.2million$$

If those tweets are represented in indexes, and each takes size of *int*, then 15 GB tweets file can be abstracted to:

$$indexes\_size = 4.2million * 24bytes \approx 100MB$$

The load of transmission could be largely reduced by representing tweets in form of indexes.

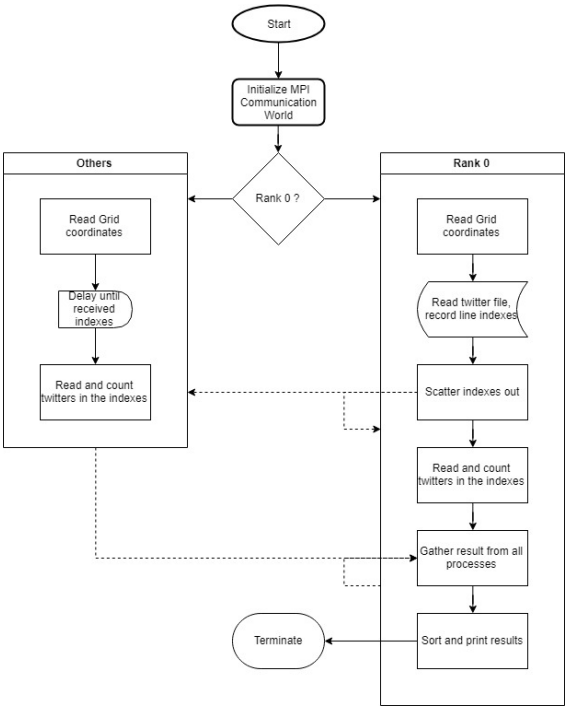

Figure 1: Main Implements

4 Result

As we can see from the Figure 2 and the Table 1, after using parallelized application (1 node 8 tasks and 2 nodes 8 tasks), the running time of searching BigTwitter.json file seems to be about 2.7 times faster than the serial one (1 node 1 task). And the two parallel programs are very similar, with only a slight time difference.

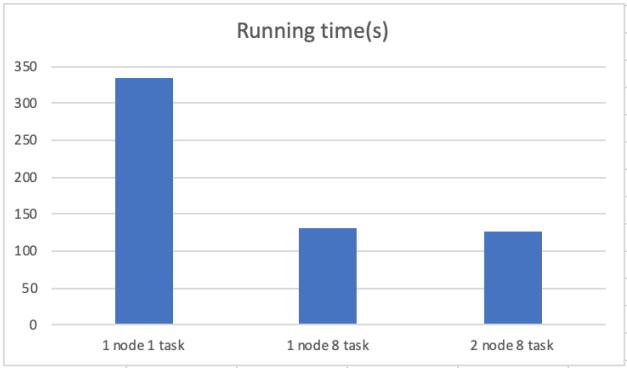

Figure 2: Runtime

In the theory, according to Amdahl's law and the Gustafson-Basis's law, with the increase of the processes, the speedup could be higher. Thus, it is obvious that 1 node 1 task, the non-parallel program costs the longest time to run the data. Furthermore, the runtime of 2 nodes 8 task and 1 node 8 tasks, the parallel programs may similar, but due to the inter-

Speedup(s)	Real	User	Sys
1 Node 1 Task	342.685	328.387	12.025
1 Node 8 Task	122.154	961.023	13.059
2 Node 8 Task	140.352	376.636	181.978

Table 1: Runtime

communication between the nodes, the 2 nodes 8 tasks will spend more time.

However, in practical, 2 nodes 8 tasks spends slightly faster running time than 1 node 8 tasks. There are two reasons could explain this phenomenon.

Firstly, since each node's memory is shared and since this job is likely sharing the node with other jobs, which could impact the performance and memory access of your job on the node.

And also, with the usage of line indexes, the communication time between processes could be shorten, thus, the time difference between within single node and cross two nodes could be neglected.

Appendices

A Slurm Script

A.1 1 Node 1 Task

```
#!/bin/bash
#SBATCH --partition=physical
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --output="1node1task.out"

module load Python/3.5.2-goolf-2015a

time mpiexec python3 __main__.py
```

A.2 1 Node 8 Task

```
#!/bin/bash
#SBATCH --partition=physical
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --output="1node8task.out"

module load Python/3.5.2-goolf-2015a

time mpiexec python3 __main__.py
```

A.3 2 Node 8 Task

```
#!/bin/bash
#SBATCH --partition=physical
#SBATCH --nodes=2
```

```
#SBATCH --ntasks-per-node=4
#SBATCH --output="2node8task.out"

module load Python/3.5.2-goolf-2015a

time mpiexec python3 __main__.py
```