COMP90024 Cluster and Cloud Computing
Assignment 1 HPC Twitter GeoProcessing

Zijun Chen, Wenqing Xue

Introduction

The purpose of this assignment is to implement a simple parallelized program to parse a large Twitter dataset, and identify the total post counts and most frequently occurring hashtags based on the presumed geographical areas. The task is executed on University of Melbourne HPC facility SPARTAN with different distributions of resources, including 1 node and 1 core, 1 node and 8 cores, and 2 nodes and 8 cores (with 4 cores per node). Thus, we use message passing interface (MPI) in order to manipulate the communications between nodes.

Dataset

- **melbGrid.json**
  Used to divide the Melbourne area into 16 distinct gridded boxes based on its latitude and longitude of each of the corners of the boxes.
- **bigTwitter.json**
  Used to be analysed by ourself-implemented program. This JSON-based file contains 2.5 million tweets.

Folder Structure

```
/
├── src/           // source code root
│   ├── main.py    // main program
│   ├── helper.py  // helper class for main program
│   └── test.py    // unit test
├── slurm/         // slurm scripts
├── reslut/        // raw outputs
└── .travis.yml    // continuous integration configuration
```

Figure 1: Folder structure of the submission.

Slurm Script

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
module load Python/3.6.4-intel-2017.u2
time mpirun python src/main.py \
  melbGrid.json bigTwitter.json
```

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --time=00:03:00
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=1
module load Python/3.6.4-intel-2017.u2
time mpirun python src/main.py \
  melbGrid.json bigTwitter.json
```

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --time=00:03:00
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
module load Python/3.6.4-intel-2017.u2
time mpirun python src/main.py \
  melbGrid.json bigTwitter.json
```

Figure 2: Slurm scripts for 1-node 1-core / 1-node 8-core / 2-node 8-core.

Approach

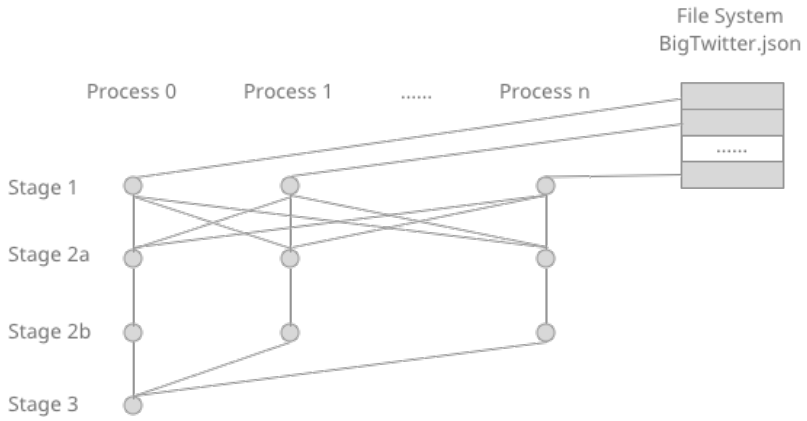

Figure 3: Execution sequence model.

- **Sequence**

  Stage 1

  Load the Melbourne grid data. Each process will be assigned with a trunk of the whole file based on its rank. The trunk is actually a start and end offset of the file in bytes, so it can be in middle of a line. The trick here is to align the offset to a near newline character without processes having the same line or skip a line. Process the trunk of file line by line individually.

  Stage 2a

  Each process will be assigned with some girds and then collect the data of assigned grids from all the other processes.

  Stage 2b

  Each process will merge the data collected in stage 2a individually.

  Stage 3

  The process with rank 0 will collects results of each grid from other process and make the final output.

- **Data Structure**

  In stage 1, we mainly use Dict in python which is actually hash table and it can provide amortized $O(1)$ time complexity for adding one to the counter of a given hashtag.

  In stage 2a, we convert the counter result to a much more compact data structure (Tuple[int, List[Tuple[str, int]]]: a int for tweet number and a list for each hashtag with its count) in order to reduce the communication time.

  In stage 2b, we convert the compact data back to Dict to merge the data.
  As for finding the top-k hashtags in all n hashtags, we first use hash set to remove all duplicate occurrence number in $O(n)$ time. Then using binary heap to find the top-k value in $O(n \log k)$ time. Finally, we filter the hashtags list and sort the result in $O(n + k \log k)$ time. We are aware there are an algorithm called BFPRT which can find the top-k value in $O(n)$ time. However, since the k is pretty small ($k = 5$) in this case, so the current approach will still give us $O(n)$ time complexity.

Result Analysis

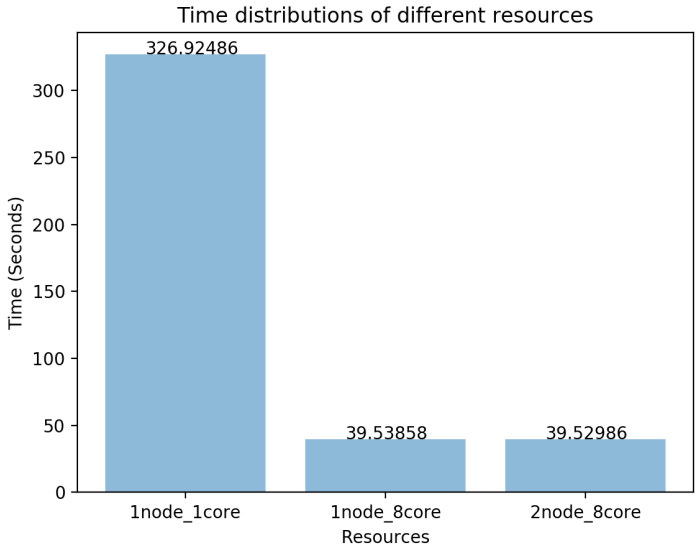

Figure 4: Execution time plot figure.

In general, similar to what we expected that the 1-node 1-core scenario is the slowest one. Both 8-core cases have almost the same performances and are about 8 times faster than the 1-core case. According to the Amdahl's Law, the results suggest that almost all the parts of our program can be parallelized. That is correct since stage 1 and 2b are the two stages that can be parallelized (each core works individually and there is no communication between cores happened in those two stages), and the stage 1 is the most time consuming task which takes more than 99% of the total execution time. Additionally, it is clear to see that 8-core cases is much faster than 1-core case in both stage 1 and stage 2b.

For stage 2a, even though there is no communication cost in the 1-core 1-core case, it is the slowest one since there is only one core to handle the work of compacting and marshalling the result from the counter. Therefore, this core holds 8 times of data comparing to each core in 8-core scenario.

As for the performance similarities between 2-node 8-core and 1-node 8-core scenario, the reason is that in both cases there are 8 cores and the only differences are: 1. different cores in one node might share some part of the resources such as memory, L3 cache. 2. cores in different nodes might have high network latency when they communicate with each other. However those are very tiny factors since the processed results are pretty small so it did not make a noticeable influence.

One thing we can improve is that instead of using this complicated distributed merging strategy, a naive single core gathering and merging might work better. The reason is that the result suggests it spend more time on communication than the actual merging. Actually the whole stage 2 took only 0.1s, so optimizing this stage can not make much difference anyway.

For stage 3, its obvious that the 1-node 1-core case is much faster since the the master core is the only core which holds all the result so there is no cost on gathering data.

It should be noted that all the analysis above is theoretical and it is based on results from one

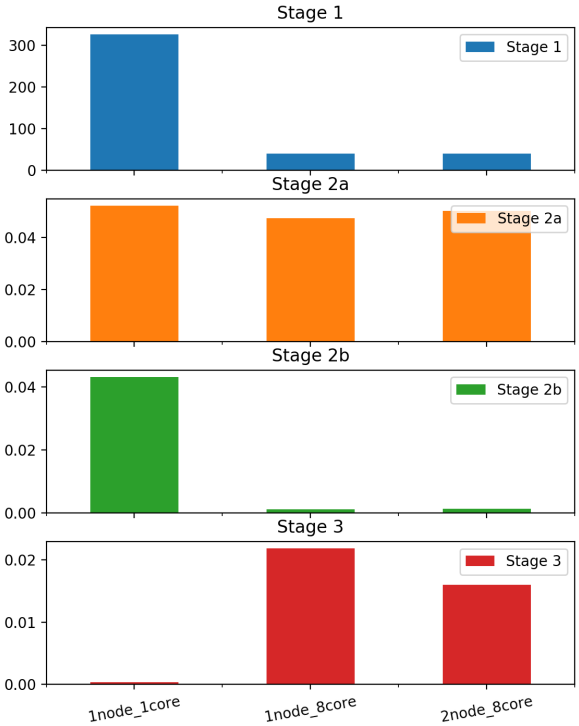

Figure 5: Execution time plot for each stage figure.

execution of each case which can not be fully trusted. There are lots of factors that can slightly influence the execution time such as the IO load on the file system. We tried to minimize the influence of this problem by running three processes at the same time, which can at least keep the load of server in the similar level. Also, the time used in both stage 2 and 3 are extremely short, therefore a small influence can totally change the result. Since we are using Python, the GC might also be a problem if it decides to clean all the objects creates in stage 1 in stage 2.