Cluster and Cloud Computing Assignment-1 Report

Jiacheng Ye 904973

Yujing Guan 1011792

1 Overview

In this project, we are going to explore the basics of HPC and how resources impact application performance. We will need to implement a simple, parallelized application leveraging the University of Melbourne HPC facility SPARTAN. The application will use a large Twitter dataset and a small sentiment dictionary to calculate overall tweets' sentiment scores.

In this report, we are going to address how the system is implemented, what the results are, and how to interpret them.

1.1 Dataset

There are three JSON files. TinyTwitter.json is the smallest dataset. It is mainly used for development. SmallTwitter.json contains more data, about 30MB big, and it is used for validating. BigTwitter.json is about 20+ gigabytes large file. It is used for final evaluating. All files' formats are the same, which contains tweet and location information collected by Twitter API.

There is another .txt file, a simple dictionary of terms related to sentiment scores, called AFINN. It is written in plain text, line by line. Each line contains a word and its corresponding sentiment score, separated by space.

2 Application/System Overview

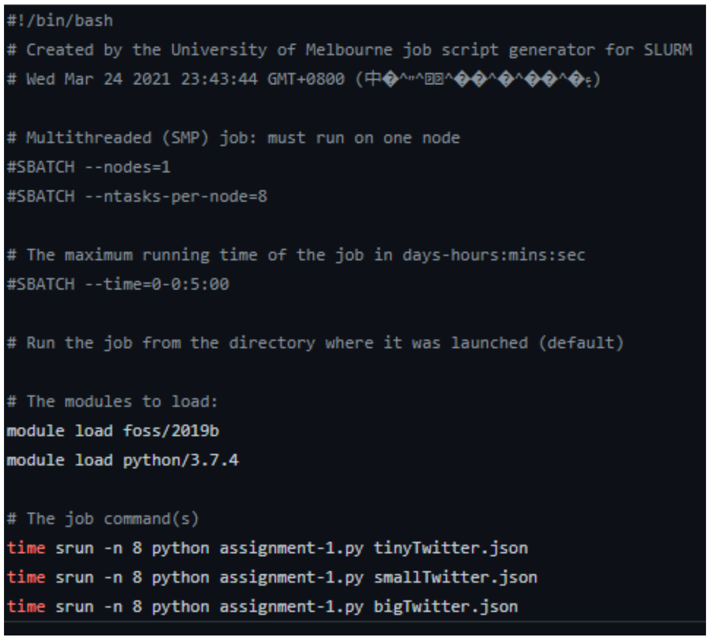
All codes are run in Spartan, a High-Performance Computing (HPC) system at the University of Melbourne. Spartan is a batch system; all jobs are submitted by the scheduler Slurm. Except for the batch command file, all codes are implemented in Python 3. Parallel computing is done via MPI, particularly with the mpi4py library.

2.1 Script

To run the application, a slurm command is needed:

```
sbatch slurm-script-name.slurm
```

All scripts are written in Slurm standard. Here is an example:



```
#!/bin/bash
# Created by the University of Melbourne job script generator for SLURM
# Wed Mar 24 2021 23:43:44 GMT+0800 (中�^"^��^��^�^��^�;)

# Multithreaded (SMP) job: must run on one node
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8

# The maximum running time of the job in days-hours:mins:sec
#SBATCH --time=0-0:5:00

# Run the job from the directory where it was launched (default)

# The modules to load:
module load foss/2019b
module load python/3.7.4

# The job command(s)
time srun -n 8 python assignment-1.py tinyTwitter.json
time srun -n 8 python assignment-1.py smallTwitter.json
time srun -n 8 python assignment-1.py bigTwitter.json
```

"--nodes=1" specifies how many resources to use. "--ntasks-per-node" specifies how many cores to use for each node. "--time" determines the maximum running time of the job. "foss/2019b" is a special module in Spartan, which contains gcc and mpi4py.

2.2 The application

2.2.1 Parallel computing, MPI

We use mpi4py to make the application run parallel. After initialization, we choose rank 0 as the master and rest ranks as the workers (slaves). The master is responsible for calculating the sentiment scores and communicating, I.e., sending out requests asking for the results. The workers then will send the results back to the master after they finish their work. After the master collects all the results, it will communicate with the workers and ask them to close the communication channel. Each rank will only preprocess a subset of tweets, whereas the row number mod number of cores equals the rank (I.e., only process the tweets which belong to the same module group). For example, with 1 node 8 cores, master rank will process tweets with row number 0, 8, 16... (as 0 % 8 = 0, 8%8 = 0, 16%8 = 0). By doing so, we ensure that there will be no tweets to be processed more than once while not introducing extra message passing.

2.2.2 Data processing

We first read AFINN into a dictionary, whereas keys are the leading characters ("a", "b"...), and values are lists of lists, the outer list contains words that start with the leading characters, and the inner list contains the word and its corresponding score. The lists are sorted by the words in ascending orders as well. By doing so, we can accelerate the checking speed when we calculate the sentiment scores, as we will only check a subset of all the words.

Raw strings are loaded by json.loads(). We choose this method over regex because it is more reliable. Also, as long as we don't load redundant strings, it runs as fast as regex.

To only load the substring we want, after a tweet is read, we extract 2 fields from it first, "coordinates" and "text". This can be done in linear time. Then re-construct them into json format, and loads the string with json.loads().

Each tweet’s location is calculated by comparing its x, y coordinates with each cell’s coordinates. For example,

if (cell_1_x_min < x <= cell_1_x_max && cell_1_y_min < y <= cell_1_y_max)

Then this tweet belongs to cell 1.

This comparing algorithm ensures that if a tweet locates at a boarder, it will belong to the left cell or the top cell as required in the specification. The only issue with this is that the tweets, which lies on the left most boarder or on the bottom most boarder, are treated as outside the grid.

The tweet then will be splitted by using re.split(). The regex pattern we use is

r'(?:[!.?,\'\"\‘\’\“\”\s]|http://\S+|https://\S+)'

"?:" throws the eliminators after splitting the string. Punctuations listed in the brackets are specified in the specification. "http" and "https:" are used to remove all short links, to further increase the performance of the application. We don't split the string by space only because strings like "good!nice" should match both "good" and "nice". There is a problem with our method. In AFINN, there is a phrase "can't stand". And our method will never match it. However, since there is no "can't stand" in any tweet, we decided to ignore this issue.

As mentioned in the specification, if AFINN contains strings like "cool stuff", "cool", and "cool stuff" appears in tweets, then the application should match "cool stuff" but not "cool". If there is "cool stuff cool" in tweets, and AFINN contains both "cool stuff" and "stuff cool", then it should match both of them. To deal situations like this, we decided to use a slicing window. The window size varies between 1 to the maximum phrase length. We now use word list [this, is, cool, stuff] and "cool stuff" as the longest matching word to demonstrate our method.

When we load AFINN, we calculated the maximum length which appears in it (in this case, 2). Then we use 2 pointers, start and end, which indicates the start and end of the slicing window. Initially, start is 0 and end is 3. Initially, start = 0, end = 2.

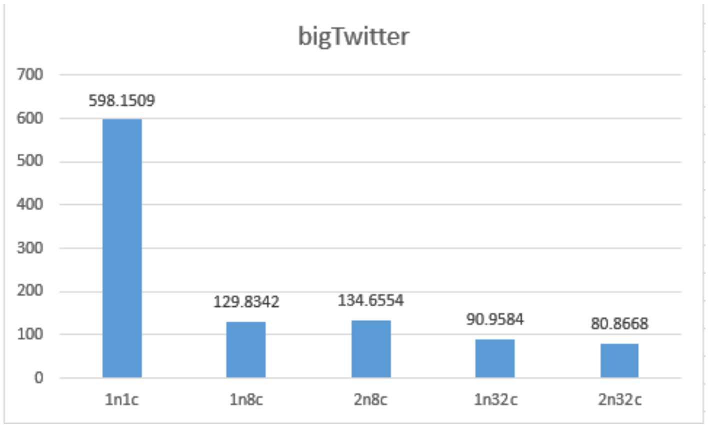
a) The word we get is "this is cool". Then we check if "t" actually exists in AFINN dictionary. If so, continue with 3, otherwise, start + 1, end = start + 2, repeat 2.
b) For each word in AFINN["t"], we check if word equals "this is cool".
    1. If not, check if "this is cool" is larger than the word. If so, end – 1, then we get word "this is", and repeat 2 with this new word. As we sort AFINN in ascending order, and if current AFINN word is larger than "this is cool", then all words after it will definitely not match.
    2. If after we loop through all the words, and there is still no match, then end –1, repeat 2. As it indicates there is not matching with length end-start.
    3. If end is equals to start, which means there is no matching start at "this". Then start + 1, end = start+2, go back to 2.
    4. If there is a match, add it to the results, and start + 1, end = start +2, go back to 2.

So, the words will be checked are: [this is, this, is cool, is, cool stuff, stuff] (assuming "t", "i", "c" and "s" are all in the AFINN dictionary). Note we will not check "cool", as 3.d says, it there is a match, move to next word directly.

However, there is an issue with our method. Strings like "cool, stuff" should only match "cool" and "stuff". But our method will split it into [cool, stuff], and reconstruct it into "cool stuff" when we are slicing the window through the word list. We didn't have time to deal with it, however. As a result, our final sentiment scores will be slightly higher than others.

3 Results

As there are noisy when we are timing the running time, we decide to run each command 10 times, and take the average as our result. We will not test the running time on TinyTwitter nor SmallTwitter. As the files are too small, and most of the time are spent on MPI communication. The result on BigTwitter shows as follow:



Obviously, the performance increases as soon as we introduce parallel computing. It runs about 4.6 faster at 1n8c compared to 1n1c. Although it should run 8 times faster ideally, there are some parts cannot be accelerated according to Amdahl's law. Our results support this theory as well. As 1n8c, 2n8c, 1n32c and 2n32c don't have that dramatic performance increase. The I/Os, preprocessing of dataset, the MPI communication, these non-reducible running times finally determine the maximum speed up while using infinite.

Our results also demonstrate the Gustafson-Barsis's Law, which suggests that with enough processors and remaining tasks, speed up will always meet the requirement. As we increase to 1n32c and 2n32c, the running time is further reduced by a small amount.

Another interesting but not surprising result is that 1n8c runs faster than 2n8c. As there will be lag when we are passing data between 2 physical nodes.

The performance will be more accurate to measure if we use C++ or C, as we can read file directly from kernel by using mmap. So that the noisy introduced by I/Os can be ignored.

4 Conclusion

In this project we calculate the sentiment scores using parallelizing computation by taking advantage of SPARTAN. Our outcomes accord with the Amdahl's low. That is, the speedup cannot be infinitely increase by adding more parallelizing processors. The maximum speedup finally is determined by the serial portion. So, in order to further increase performance, reducing running time of serial portion is a right way. Also, the outcomes shows that the communication time between nodes cannot be ignored and we may reduce time by using computation nodes which have low network delay.