COMP90024 CLUSTER & CLOUD COMPUTING ASSIGNMENT 1 REPORT

A PREPRINT

Xulin Yang
904904
xuliny@student.unimelb.edu.au

Lu Wang
1054195
lu.wang4@student.unimelb.edu.au

April 14, 2020

ABSTRACT

Twitter is a popular social media for people to share their opinions and comments for events, and in this project, we will implement a search application to analyse the top 10 commonly used hashtags and languages in the Sydney Twitter data set. This application is designed to be executed on the Spartan to practice and learn knowledge about high performance computing and Slurm script.

***K*eywords** twitter data · high performance computing · MPI · mpi4py · Spartan

1 Introduction

This parallelized application leverages the Spartan which is a high performance computing (HPC) facility of the University of Melbourne, it is implemented to identify the highest frequent occurrence (top 10) of hashtags(#) and languages using in Twitter database for Sydney, and also gives the count of their appearing times. This search application would be executed on three different running resources,including 1 node and 1 core, 1 node and 8 cores as well as 2 nodes and 8 cores.

2 Design Description

Twitter, which is one of the most popular social media around the world, allows people to share their opinions. These comments are valuable data resources for many research fields[3], and analysing such massive data requires high performance computing techniques. In this application, we will search Twitter data set and find the most commonly used hash-tags and languages with counting their appearing times, *while the difficulties are how to achieve the communication between multiple processors on multiple compute nodes and enable cross-node parallelization*[2]. Therefore, to tackle these issues, this application is implemented by Python language, and using the library 'mpi4py' to achieve these goals. We will describe the application design from two aspects, including scripts and design techniques.

2.1 Scripts

Scripts are used to submit jobs to Spartan, and each script acquires different resources. In this application, using Slurm is to submit script jobs to Spartan. The format of scripts is "#SBATCH –type=value", for example, the required resources are set with four lines, including nodes, ntasks-per-node, cpus-per-task and time. The 'cloud' is referred to Spartan. We assume that the number of cores equals to the number of processors, and one processor can have one task at a time. See Appendix for the original slurm scripts.

2.2 Parallel technique

We will discuss techniques we used to parallel our search application to benefits the parallel optimization in the following sections.

2.2.1 Parallel file processing

Algorithm 1: parallel twitter file reading & processing

Result: hash_tag_count, language_code_count
total_row ← $read\ from\ file$
offset ← $read\ from\ file$
number of lines to be processed ← total_row - offset

lines_per_core ← number of lines to be processed / number_of_processors
line_to_start ← 1 + lines_per_core * processor's_rank
if *current processor is last processor* **then**
  line_to_end ← 1 + line_to_start + lines_per_core
else
  line_to_end ← 1 + number of lines to be processed
end

for *line_index, line* ← $read\ from\ file$ **do**
  if *line_to_start* ≤ *line_index* ≤ *line_to_end* **then**
    do the summarize of hash_tag and language_code task;
  end
end

We use Single-Program Multiple-Data parallel design strategy for this project. In the implementation of the above algorithm, **firstly**, we read the first line from the 'bigTwitter.json' as shown below to calculate *number of lines to be processed*.

```
{"total_rows":215443567,"offset":211386044,"rows":[
```

Secondly, we calculate the range of lines for each processor to process as shown above. It should be noted that the plus 1 in this step is to ignore the first line in the json file as mentioned in the first step. It also should be noticed that last processor not only process the given range, but also the remaining lines as in the indivisible $number\ of\ lines\ to\ be\ processed\ /number\_of\_processors$ case.
Thirdly, each processor reads from the start of file until the end of the range of line to be processed.
Last but not the least, in our codes, we use bcast function to broadcast the result of number of lines to each processor in order to save the file reading time as well as the calculation time.

2.2.2 Parallel top-n calculation

Algorithm 2: parallel top-n calculation

Input: hash_tag_count, language_code_count;
Result: reduced_hash_tag_count, reduced_language_code_count
if $number\_of\_processors > 1$ **then**
  reduced_language_code_count, reduced_hash_tag_count ← $merged$ from each processor with hash_tag_count, language_code_count
  if $current\_processor\ is\ root\_processor$ **then**
    split_language_code, split_hash_tag ← $split$ equally w.r.t. number_of_processors from reduced_language_code_count and reduced_hash_tag_count
  else
    do nothing
  end
  local_language_code, local_hash_tag ← $scatter$ split_language_code, split_hash_tag to each processor
  do top-n calculation on local_language_code, local_hash_tag;
  reduced_language_code_count, reduced_hash_tag_count ← $merged$ from each processor with local_language_code, local_hash_tag
else
  hash_tag_count, language_code_count ← $do\ topn\ calculation$ on local_language_code, local_hash_tag
end

The **Algorithm 2** is strictly after **Algorithm 1** in the implementation.
In the parallel part, **firstly**, we use reduce function to perform the merge operation. It should be noticed that as the

reduced object are Collections.Counter object, they can simply use the operator.add function to add counts for the same keys compared to the addition of duct's add operation is just appending operation (i.e. replace the value in the same key case). **Secondly**, we use numpy.array_split to have a faster approximate equally divided operation. It should be noted that the *split* only happens on the *root_processor*. **Thirdly**, we use scatter function to perform the sending operation to each processor. **Fourthly**, *do top-n calculation* is just using Counter.most_common(*n*) from the Python standard library. **Fifthly**, in the last merge operation, we use the customized merge function to achieve a better performance. As the two local objects to be merged are sorted, we can just do a similar merge operation as in the merge sort and cut-off when the merged object has *n* elements. **To sum it up**, this algorithm just splits the merged counts to each processor and do calculation separately to gain the advantage of parallel computing. The **advantage** is that we have a smaller amount of count objects to do the calculation. The **drawback** is that we have the additional split data operation time as well as the scatter and reduce operations time. Compared to the drawback, we benefit more on the advantage in the large data processing case.

3 Discussion

The execution task is to read the data file and output execution time as well as top 10 hashtags and languages. The amount of the data in the bigTwitter.json is 4057525 lines. For the 1 node 1 core, only one processor does the whole task. For the other two resources with 8 cores, there are 8 processors doing the tasks separately. Obviously, the 8 processors finish tasks faster than only 1 processor.

3.1 Execution results

Top 10 most commonly used hashtags		
1	#auspol	19,891
2	#coronavirus	10,125
3	#firefightaustralia	6,814
4	#oldme	6,419
5	#sydney	6,376
6	#scottyfrommarketing	5,185
7	#grammys	5,085
8	#assange	4,689
9	#sportsrorts	4,516
10	#iheartawards	4,297

Figure 1: Top 10 hashtags

Top 10 most commonly tweeted languages		
1	English (en)	3107116
2	undefined (und)	252117
3	Thai (th)	134571
4	Portuguese (pt)	125858
5	Spanish (es)	74028
6	Japanese (ja)	49929
7	Tagalog (tl)	44560
8	Indonesian (in)	42296
9	French (fr)	38098
10	Arabic (ar)	24501

Figure 2: Top 10 Languages

The top 10 commonly used hashtags and languages are showing as above. The number 1 hashtages used word is 'auspol' appearing for 19,891 times, and English is the most commonly used language. We use data from ["doc"]["metadata"]["iso_language_code"] to summarize the languages. We use ["doc"]["text"] to summarize the hashtags by python's builtin regex:

```
re.findall("#[a-zA-Z0-9_]+")
```

3.2 Variation description

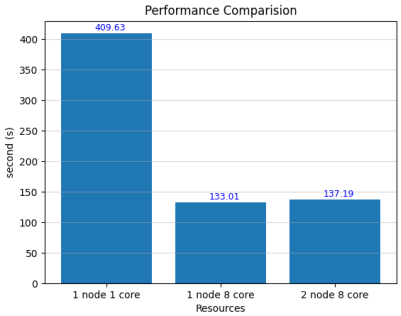

Figure 3: Performance Comparison

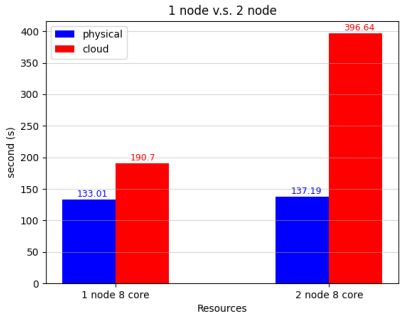

Figure 4: Node Comparison

From the results of executing three times on each Spartan resources, the differences are obvious, as listed in the above figures. From Figure 3, in the 1 node and 1 core, the total execution time is 307.96% and 298.59% larger than 1 node

and 8 cores and 2 nodes and 8 cores respectively. 8 cores on both 1 node and 2 nodes have similar executing time in this case as they are in *physical partition*. From Figure 4, we also compared 8 cores on 1 node and 2 nodes running on **physical** and **cloud** machine on Spartan. In other words, the time they need on physical partition for the searching jobs is similar when running on the local machines. However, 1 node 8 cores of Spartan *cloud partition* would need 190.7 seconds to finish the jobs, while 2 nodes 8 cores would need nearly twice amount of running time than 1 node required. This is because there is communication time cost between nodes in the *cloud partition*. *Cloud: This partition is best suited for general-purpose single-node jobs. Multiple node jobs will work, but communication between nodes will be comparatively slow. Physical: Each node is connected by high-speed 25Gb networking with 1.15 µsec latency, making this partition suited to multi-node jobs (e.g. those using OpenMPI).* [1]
The drawback for this algorithm is that the last process reads the file from the start to the end which requires more time on IO operation compared to other processes. It could be improved to read start from the desired part of the file, but we didn't find the way for this improvement.
From the results showing above, we can see that more cores can have higher performance than less cores until we reach the parallel optimization limited as described by the Amdahl's law, and communication between nodes can lead to more time cost. So, single-node jobs on Spartan perform comparatively better than multiple node jobs, while the time they need on physical machine is almost the same. However, when the amount of available cores in one node no longer satisfy our demands, we can increase the number of nodes to satisfy it.

4 Conclusion

Through this practice of parallel computing, we have learnt knowledge about HPC, Slurm scripts,and implementing an application on Spartan. Additionally, we also learnt how to use Python to develop parallel computing application.
The bottleneck (i.e. the non-parallelisable part) is still the file reading time. For future improvements, we can probably use more advanced data parallelism techniques instead reading from the start of the file each times. In other words, we can use data splitting tools like Hadoop, etc.

References

[1] Lev Lafayette, Greg Sauter, Linh Vu. Bernard Meade, "Spartan Performance and Flexibility: An HPC-Cloud Chimera", OpenStack Summit, Barcelona. October 27, 2016. doi.org/10.4225/49/58ead90dceaaa.

[2] Richard Sinnott. Cluster and Cloud Computing Assignment 1 – HPC Twitter Processing. 2020.

[3] SHUCHENG CUI. Elastic distributed computing System for tweet sentiment analysis Using machine learning algorithms. June 2015.

5 Appendix:

5.1 Original Slurm Script

5.1.1 1 node 1 core

The following is the script for submitting analysing Twitter data to 1 node and 1 core executing resource on Spartan.

```
#!/bin/bash

#SBATCH --account=COMP90024
#SBATCH --partition=physical
#SBATCH --output=1node1core-physical.txt
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mail-user=xuliny@student.unimelb.edu.au
#SBATCH --mail-type=End
#SBATCH --time=1-0:0:00

# Run the job from this directory:
cd /home/$USER/COMP90024/HPC-Geo-Data-Processing/slurm
# The modules to load:
module load Python/3.6.1-intel-2017.u2

# The job command(s):
time mpirun python3 "../src/main.py" -country "../src/language.json" -data "../data/bigTwitter.json"
```

5.1.2 1 node 8 cores

```
#!/bin/bash

#SBATCH --account=COMP90024
#SBATCH --partition=physical
#SBATCH --output=1node8core-physical.txt
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --mail-user=xuliny@student.unimelb.edu.au
#SBATCH --mail-type=End
#SBATCH --time=1-0:0:00

# Run the job from this directory:
cd /home/$USER/COMP90024/HPC-Geo-Data-Processing/slurm
# The modules to load:
module load Python/3.6.1-intel-2017.u2

# The job command(s):
time mpirun python3 "../src/main.py" -country "../src/language.json" -data "../data/bigTwitter.json"
```

5.1.3 2 node 8 cores

```
#!/bin/bash

#SBATCH --account=COMP90024
#SBATCH --partition=physical
#SBATCH --output=2node8core-physical.txt
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --mail-user=xuliny@student.unimelb.edu.au
#SBATCH --mail-type=End
#SBATCH --time=1-0:0:00

# Run the job from this directory:
cd /home/$USER/COMP90024/HPC-Geo-Data-Processing/slurm
# The modules to load:
module load Python/3.6.1-intel-2017.u2

# The job command(s):
time mpirun python3 "../src/main.py" -country "../src/language.json" -data "../data/bigTwitter.json"
```