The University of Melbourne
School of Computing and Information Systems
COMP90024 Cluster and Cloud Computing

Assignment 1, 2020

Marco Marasco 834482 Austen McClernon 834063

Introduction

In the last decade, social media has become more prevalent and pervasive in society. This boom has resulted in an exponential growth in the volume of data captured by each social media platform, but also the value of the data.

Social media provides a unique and (predominantly) uncensored insight into the behaviour and trends of consumers. Accordingly, this data has become a powerful tool for firms to use to help understand the market.

Before the raw data generated by the social media platforms can be used by firms, it must first be processed into a form with higher utility. Due to the sheer volume of data generated, this is not a trivial problem to solve.

The following report will demonstrate and discuss the advantages and disadvantages of using a parallelised solution for processing large social media datasets. Leveraging the University of Melbourne HPC facility Spartan, a large Twitter data-set for the city of Sydney was processed to find the top ten:

a. Most frequently occurring hashtags

b. Most commonly used languages for tweeting

Approach

For this implementation, a script was written in **Python 3.7.3** that utilises the Message Passing Interface (MPI), a standardised and portable message-passing system designed to function on a wide variety computers in parallel.

Each script execution was initialised with a specified number of nodes and cores per node. In executions of more than 1 core, the process with rank 0 was deemed to be the master process, and the others slaves. Once each process was created, it immediately began to process the data.

Given the size of the dataset, reading the entire file into memory had several drawbacks. Firstly, the size was too large for many of the nodes on the Spartan cluster, so using a simple Python IO operation to read and process the file as a singular JSON object was not viable. Secondly, distributing the entire file content to other processes introduced unnecessary strain on the network connecting them. Thirdly, reading in the file in its entirety resulted in an increased exposure to possible invalid JSON formatting and other formatting errata in the file. As such, the most effective method to read the data was for each process to individually open smaller "chunks" of the file.

For a given execution with n nodes and c cores, a file size of F bytes, the chunks were of size:

$$ChunkSize = \frac{F}{n \cdot c} \tag{1}$$

Where each chunk of starts at an increasing offset, distributed amongst the slaves such that $\sum chunks = size_{data.json}$.

To parallelise the execution, each processor at runtime calculated its own unique chunk of the file to read in. As no frequency related dependencies existed between chunks of the files, processing the entire file could be easily parallelised.

Once a process had identified its file chunk, it then began to further divide that chunk into smaller portions. Creating these smaller portions increased the risk of reading in some portion of the file that started or ended within a tweet inside the file, which could result in a decreased volume of processed tweets. Furthermore, the Python MPI library itself is

limited to reading in at most $2^{31} - 1$ bytes from a file at a time, however, reading in this many bytes caused memory errors in several of the servers in the Spartan cluster. Tests were conducted to find the optimal number of bytes to read in at any time, by assessing the runtime performance and the impact portion size had on the number of tweets processed (see appendix 1).

The result of these tests indicated as the portion size decreased, both the runtime and the number of tweets decreased. The decrease in runtime was relatively small, and as the main goal for this implementation was to process the tweets within this file, it was decided reading $\frac{2^{31}-1}{2}$ bytes at time was the optimal choice as it resulted in the lowest impact on the number of tweets.

One at a time, each smaller portion was then loaded into a byte array buffer for processing. Using Python's builtin Regex library, the hashtags and language of each tweet was extracted and the frequency of each hashtag and language were updated in their respective counters.

Once a process had finished reading its chunk of the file, it closed the file and returned the counters. If there were slave processes at runtime, once the master process finished processing its data it sent out a message to all slaves to send their processed data once ready. Upon receiving return signals from all slaves, The master process then collated all the counters, sent exit signals to the slaves and outputted the results.

Invocation

Invocation of the Python application on the Spartan cluster was made via the SLURM workload manager. A SLURM script that set the parameters required for Spartan and the hob to execute was submitted to the workload manager to be executed later. For example, to submit a job to the workload manager:

```
sbatch script.slurm
```

Appendix 2 contains all SLURM scripts used to run jobs on the Spartan Cluster.Below is an example SLURM script for execution the application on the physical partition with 2 nodes and 4 cores per node:

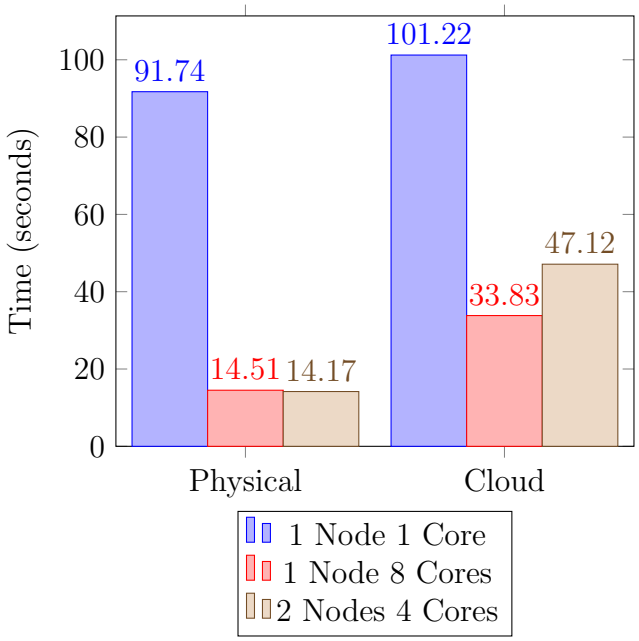
```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=0-01:00:00
#SBATCH --partition=physical

# Set CPU type.
#SBATCH --constraint=physg1

echo "2 Node 8 Cores"
module purge
module load Python/3.7.3-spartan_gcc-8.1.0
time srun -n 8 python3 app.py data/bigTwitter.json
```

Results

Table 1: Most Frequent Hashtag & Most Frequent Language

Hashtag	*Frequency*	*Language*	*Frequency*
auspol	19320	English(en)	3120644
coronavirus	10011	Undefined(und)	236874
firefightaustralia	6806	Thai(th)	132264
oldme	6418	Portuguese(pt)	122711
sydney	6312	Spanish(es)	73199
grammys	5076	Indonesian(id)	56797
scottyfrommarketing	5047	Japanese(ja)	49678
sportsrorts	4412	Tagalog(tl)	42240
iheartawards	4297	French(fr)	33607
assange	4179	Arabic(ar)	24550

Figure 1.1 Physical and Cloud Partition Run Time



Analysis

For the analysis of the results, the three different node configurations are defined as follows:

Configuration 1: 1 node, 1 core.

Configuration 2: 1 node, 8 cores.

Configuration 3: 2 nodes, 4 cores each.

Two different partitions on the Spartan network were tested for processing the Twitter file, the Physical and Cloud partitions. An initial inspection of figure 1.1 clearly demonstrates the benefits of code parallelisation in the domain of tweet processing. In the physical partition, the parallelised performance of configurations 2 and 3 were approximately 6 times

faster, and in the cloud partition approximately 3 and 2 times faster for configurations 2 and 3 respectively.

For configuration 1, the results would not expect to differ between the two partitions as all computation is performed within a single core. The results show a slight variation in runtime, but this is likely due to the superior processor designated for the physical partition executions. As the problem of processing the Twitter file was deemed to be parallelisable, utilising multiple processor cores was expected to significantly speed up the runtime. This however came with a runtime tradeoff for latency in processes reading the file and communicating with each other.

Comparing configurations 2 and 3, both configurations utilise the same number of cores, however, configuration 3 distributes its workload across two different nodes within the cluster. Distributing the work load across nodes introduces the issue of network latency in the cluster as processes on different nodes need to communicate and accordingly would result in a longer runtime for configuration 3 on average, given the network specifications outlined below:

- **Cloud:** 10gb Ethernet cables, with an added latency of the Neutron core service in the OpenStack.
- **Physical:** 25gb/56gb Ethernet with Mellanox SN2700 and SN2100 leaf switches with 100G connections to the partition core.

The inferior network infrastructure in the cloud partition would be expected to have a significantly higher impact on runtime for configuration 3, compared to the physical partition.

For configurations 2 and 3, the cloud partition's performance consolidates the theoretical behaviour of the results. It's performance in configuration 2 is 28.2% (13.29 seconds) faster than configuration 3.

Conversely, the physical partition's results show a decrease in runtime for configuration 3. In this experiment, very few messages are passed between the nodes and said messages are very small compared to the physical partition network capacity. This results in network latency to have less of an impact on the total runtime for configuration 3. As a result, it is possible the 2.4% (0.34 seconds) decrease in total runtime between configurations 2 and 3 could have been caused by inferior performance in the cores during configuration 2's execution.

Whilst the results in the physical partition contradict the expected behaviour, the 28.2% increase between configurations 2 and 3 in the cloud partition clearly supports the expected runtime behaviour between configurations. By distributing the work across multiple nodes, the runtime was more heavily impacted by the network latency.

Conclusion

The results of this report strongly indicate that if a computational task can be identified as parallelisable, implementing a parallelised solution will significantly decrease the total time required to complete the task. In this experiment, utilising 8 different cores resulted in a peak 6x faster runtime compared to only using 1 core. However, the results also showed careful consideration must be made into how this parallelisation is configured. For a given HPC cluster, increasing the number of nodes (whilst maintaining the total number of cores) for a task exposes the runtime to more network latency. Knowing the network infrastructure of the available HPC system is key to optimally configuring parallelisation tasks.

Appendix 1

Table 2: Portion Size v.s. Physical Node Runtime

$\frac{2^{31}-1}{N}$	1 *Node* 1 *Core*	1 *Node* 8 *Cores*	2 *Nodes* 4 *Cores*
2	91.743	14.513	14.173
4	87.006	14.482	15.751
6	85.162	13.976	13.93
8	85.064	14.051	14.402
10	84.75	13.886	13.957

Table 3: Portion Size v.s. Cloud Node Runtime

$\frac{2^{31}-1}{N}$	1 *Node* 1 *Core*	1 *Node* 8 *Cores*	2 *Nodes* 4 *Cores*
2	101.223	33.825	47.117
4	126.776	23.096	22.549
6	178.492	55.192	45.765
8	178.079	138.878	22.75
10	186.572	55.469	43.925

Table 4: Node Configuration Hashtag Frequency vs. Portion Size

Tag_{nodes_cores}	$\frac{2^{31}-1}{2}$	$\frac{2^{31}-1}{4}$	$\frac{2^{31}-1}{6}$	$\frac{2^{31}-1}{8}$	$\frac{2^{31}-1}{10}$
auspol_{11}	19,320	19,320	19,320	19,320	19,320
auspol_{18}	19,320	19,320	19,320	19,320	19,320
auspol_{24}	19,320	19,320	19,320	19,320	19,320
coronavirus_{11}	10,011	10,011	10,011	10,011	10,011
coronavirus_{18}	10,011	10,011	10,011	10,011	10,011
coronavirus_{24}	10,011	10,011	10,011	10,011	10,011
$\text{firefightaustralia}_{11}$	6,806	6,806	6,806	6,806	6,806
$\text{firefightaustralia}_{18}$	6,806	6,806	6,806	6,806	6,806
$\text{firefightaustralia}_{24}$	6,806	6,806	6,806	6,806	6,806
oldme_{11}	6,418	6,418	6,418	6,418	6,418
oldme_{18}	6,418	6,418	6,418	6,418	6,418
oldme_{24}	6,418	6,418	6,418	6,418	6,418
sydney_{11}	6,312	6,312	6,312	6,312	6,311
sydney_{18}	6,312	6,312	6,312	6,312	6,312
sydney_{24}	6,312	6,312	6,312	6,312	6,312
grammys_{11}	5,076	5,076	5,076	5,076	5,076
grammys_{18}	5,076	5,076	5,076	5,076	5,076
grammys_{24}	5,076	5,076	5,076	5,076	5,076
$\text{scottyfrommarketing}_{11}$	5,047	5,047	5,047	5,047	5,047
$\text{scottyfrommarketing}_{18}$	5,047	5,047	5,047	5,047	5,047
$\text{scottyfrommarketing}_{24}$	5,047	5,047	5,047	5,047	5,047
sportsrorts_{11}	4,412	4,412	4,412	4,412	4,412
sportsrorts_{18}	4,412	4,412	4,412	4,412	4,412
sportsrorts_{24}	4,412	4,412	4,412	4,412	4,412
iheartawards_{11}	4,297	4,297	4,297	4,297	4,297
iheartawards_{18}	4,297	4,297	4,297	4,297	4,297
iheartawards_{24}	4,297	4,297	4,297	4,297	4,297
assange_{11}	4,179	4,179	4,179	4,179	4,179
assange_{18}	4,179	4,179	4,179	4,179	4,179
assange_{24}	4,179	4,179	4,179	4,179	4,179

Table 5: Node Configuration Language Frequency vs. Portion Size

$Language_{nodes\_cores}$	$\frac{2^{31}-1}{2}$	$\frac{2^{31}-1}{4}$	$\frac{2^{31}-1}{6}$	$\frac{2^{31}-1}{8}$	$\frac{2^{31}-1}{10}$
English_{11}	3,120,644	3,120,629	3,120,613	3,120,604	3,120,581
English_{18}	3,120,641	3,120,626	3,120,606	3,120,593	3,120,577
English_{24}	3,120,641	3,120,626	3,120,606	3,120,593	3,120,577
Undefined_{11}	236,873	236,872	236,871	236,868	236,871
Undefined_{18}	236,874	236,873	236,871	236,871	236,870
Undefined_{24}	236,874	236,873	236,871	236,871	236,870
Thai_{11}	132,264	132,262	132,262	132,261	132,262
Thai_{18}	132,261	132,263	132,261	132,263	132,261
Thai_{24}	132,261	132,263	132,261	132,263	132,261
Portuguese_{11}	122,710	122,711	122,711	122,709	122,711
Portuguese_{18}	122,711	122,710	122,711	122,709	122,709
Portuguese_{24}	122,711	122,710	122,711	122,709	122,709
Spanish_{11}	73,199	73,199	73,198	73,198	73,197
Spanish_{18}	73,198	73,199	73,199	73,199	73,198
Spanish_{24}	73,198	73,199	73,199	73,199	73,198
Indonesian_{11}	56,797	56,797	56,796	56,797	56,796
Indonesian_{18}	56,797	56,797	56,797	56,797	56,797
Indonesian_{24}	56,797	56,797	56,797	56,797	56,797
Japanese_{11}	49,678	49,678	49,677	49,678	49,675
Japanese_{18}	49,678	49,678	49,677	49,678	49,678
Japanese_{24}	49,678	49,678	49,677	49,678	49,678
Tagalog_{11}	42,240	42,240	42,240	42,239	42,240
Tagalog_{18}	42,240	42,240	42,239	42,240	42,239
Tagalog_{24}	42,240	42,240	42,239	42,240	42,239
French_{11}	33,607	33,607	33,607	33,606	33,607
French_{18}	33,607	33,607	33,606	33,607	33,606
French_{24}	33,607	33,607	33,606	33,607	33,606
Arabic_{11}	24,660	24,660	24,660	24,660	24,660
Arabic_{18}	24,660	24,659	24,660	24,659	24,659
Arabic_{24}	24,660	24,659	24,660	24,659	24,659

Appendix 2

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=0-01:00:00
#SBATCH --partition=physical

# Set CPU type.
#SBATCH --constraint=physg1

echo "1 Node 1 Core"
module purge
module load Python/3.7.3-spartan_gcc-8.1.0
time srun -n 1 python3 app.py data/bigTwitter.json
```

Figure 2: SLURM Script to execute on 1 node, 1 core.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=0-01:00:00
#SBATCH --partition=physical

# Set CPU type.
#SBATCH --constraint=physg1

echo "1 Node 8 Cores"
module purge
module load Python/3.7.3-spartan_gcc-8.1.0
time srun -n 8 python3 app.py data/bigTwitter.json
```

Figure 3: SLURM Script to execute on 1 node, 8 cores.

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=0-01:00:00
#SBATCH --partition=physical

# Set CPU type.
#SBATCH --constraint=physg1

echo "2 Nodes 8 Cores"
module purge
module load Python/3.7.3-spartan_gcc-8.1.0
time srun -n 8 python3 app.py data/bigTwitter.json
```

Figure 4: SLURM Script to execute on 2 nodes, 4 cores each.