COMP90024 - Cloud and Cluster Computing
Assignment 1 - Report

By Juny Kesumadewi and Rizky Paskalis Totong

Introduction

This report describes the implementation and performance of a parallelised Python application that counts the number of languages used in a large Twitter dataset and identifies the grid location where Tweets were created. To compare the performance of this application using different machine configurations, we have run this application on The University of Melbourne's High Performance Computing (HPC) facility called SPARTAN.

Implementation

The application was designed so it can be run on multiple processors. It is implemented using `mpi4py` which is a Python binding of Message Passing Interface (MPI).

The application takes the following input files:

- A Twitter JSON file that contains information about Tweets to be allocated into a location grid. In this experiment, we use the `/data/projects/COMP90024/bigTwitter.json` file available on SPARTAN. This is a ~20GB text file that contains roughly four million rows of data.
- A JSON file contains a list of rectangular location grids represented in Polygon coordinates. The file on SPARTAN `/data/projects/COMP90024/sydGrid.json` was used in this experiment. It represents an area in Sydney.
- You can optionally provide a JSON file to map a Twitter language code to a language name. In this experiment, we use the file lin `$PROJECT_ROOT/data/language.json`.

The main tasks of this application are:

- To read JSON path `rows.doc.coordinates.coordinates` in `bigTwitter.json` file which represents a point coordinate. Then, we want to identify in which grid this point is located. To determine the grid, we use the JSON path `features.geometry.coordinates` in `sydGrid.json` file.
- To read JSON path `rows.doc.lang` in `bigTwitter.json` file that represents a language code and identifies which language it is using the data in `language.json` file.
- The application then collects the information from every Tweet in `bigTwitter.json` file and produces a summary report of the language count in each grid. The summary is output to stdout as well as a tab-separated text file.

Execution

This experiment compares the application performance on these three different configurations:

A. 1 node and 1 core
B. 1 node and 8 cores
C. 2 nodes and 8 cores (with 4 cores per node)

SPARTAN supports Slurm job scheduling tool. We have provided the following Slurm scripts to execute the application using the configurations above:

```
sbatch run1c.slurm  # configuration A
sbatch run8c1n.slurm  # configuration B
sbatch run8c2n.slurm  # configuration C
```

Each of the `run*.slurm` scripts contain the following bash instructions. You may need to change the `$dir` variable to point to the directory where your input files live.

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4

dir=/data/projects/COMP90024

module load foss/2019b
module load python/3.7.4
module load mpi4py/3.0.2-timed-pingpong

srun -n 8 python3 src/app.py -f $dir/bigTwitter.json --grid $dir/sydGrid.json

##Job monitor command to list the resource usage
my-job-stats -a -n -s
```

The SBATCH instructions part of the scripts vary as following:

- Configuration A: Run only one process

  ```
  #SBATCH --nodes=1
  #SBATCH --ntasks=1
  ```

  ```
  srun -n 1 python3 src/app.py -f $dir/bigTwitter.json --grid $dir/sydGrid.json
  ```

- Configuration B: Run eight processes, and all processes must be run on the same node

  ```
  #SBATCH --nodes=1
  #SBATCH --ntasks=8
  ```

- Configuration C: Run a total of eight processes. Split them equally across two nodes.

  ```
  #SBATCH --nodes=2
  #SBATCH --ntasks=8
  #SBATCH --ntasks-per-node=4
  ```

Results

The execution of the three configurations produces the same output as shown in Figure 1. It shows that for every grid, English is the most popular language used. The grid C3 has the largest number of Tweets and the most variety in languages used.

Cell	#Total Tweets	#Number of Languages Used	#Top 10 Languages & #Tweets
A1	21	2	English-20, French-1
A2	22	2	English-21, Japanese-1
A3	11	3	English-9, Indonesian-1, Thai-1
A4	85	4	English-80, French-2, Turkish-2, Spanish-1
B1	184	5	English-179, Tagalog-2, Catalan-1, Danish-1, Haitian-1
B2	504	12	English-482, Chinese-5, Italian-4, Japanese-3, German-2, Tagalog-2, Estonian-1, Finnish-1, Hindi-1, Indonesian-1
B3	641	12	English-610, Chinese-5, Indonesian-5, Japanese-5, Romanian-4, Portuguese-3, Spanish-3, Catalan-2, Danish-1, German-1
B4	469	10	English-450, Japanese-8, Portuguese-3, Italian-2, Dutch-1, French-1, German-1, Indonesian-1, Spanish-1, Turkish-1
C1	46	4	English-42, French-2, German-1, Haitian-1
C2	95	6	English-84, Indonesian-6, Spanish-2, Chinese-1, French-1, Romanian-1
C3	4480	30	English-4186, Japanese-56, Spanish-42, Indonesian-32, Catalan-24, Portuguese-15, Chinese-13, Tagalog-13, German-12, Romanian-11
C4	962	21	English-888, Haitian-17, Indonesian-6, Japanese-6, Romanian-6, Estonian-5, Italian-5, Spanish-5, Tagalog-4, Catalan-3
D1	97	6	English-90, Portuguese-2, Welsh-2, Catalan-1, Lithuanian-1, Spanish-1
D2	33	2	English-31, Spanish-2
D3	160	7	English-154, Catalan-1, Estonian-1, Portuguese-1, Spanish-1, Tagalog-1, Thai-1
D4	3	1	English-3

Figure 1: Application Output

Parallelisation

To obtain an exact count and not the estimate of Tweet languages and locations, gathering data from every Tweet in the input file is necessary. This means every line in the file that represents a Tweet needs to be parsed. On a very large file, this is where the majority of the processing time occurs. Hence, this is the part of the application that we want to parallelise.

The approach we took is to do data parallelisation by splitting the number of rows to be processed equally among all workers. Each process reads every line of the file from the beginning to the end. However, they only parse the lines that were allocated to them. Each worker is allocated a start line and they will skip the next N-1 lines before parsing another line. N is the number of processes running. See Figure 2 for an illustration of this approach. Important points behind our approach are:

	Worker 1	worker 2	Worker 3
Line 2	Parse	Skip	Skip
Line 3	Skip	Parse	Skip
Line 4	Skip	Skip	Parse
Line 5	Parse	Skip	Skip
Line 6	Skip	Parse	Skip
...	...	...	...

Figure 2: Line allocation in a 3-worker example

- We know that the Twitter input file can be very big. To **avoid using a large memory space**, the master process does **not** preprocess the input Twitter file and sends chunks of files to the workers. Instead, the master process only sends instructions or configurations to all the workers. Each worker reads the file line by line by opening a stream. Worker reads the instructions sent by the master process to apply the logic to choose which lines to parse and skip the lines it does not have to parse.
- In parallel application, the running time is the time it takes by the worker that finishes last. When work is not distributed equally among workers, the application will take longer to finish. We deliberately chose to allocate alternate lines to each worker instead of allocating consecutive lines.

This is to avoid burdening a single worker with most of the work when good data are not spread equally in the file. Note that when a row of data does not have location or language information, they are considered bad data and are to be skipped.

- For this experiment, to ensure identical results among different runs and configurations, it is important that the sort algorithm used to produce the ranking summary is a stable sort algorithm.

Performance

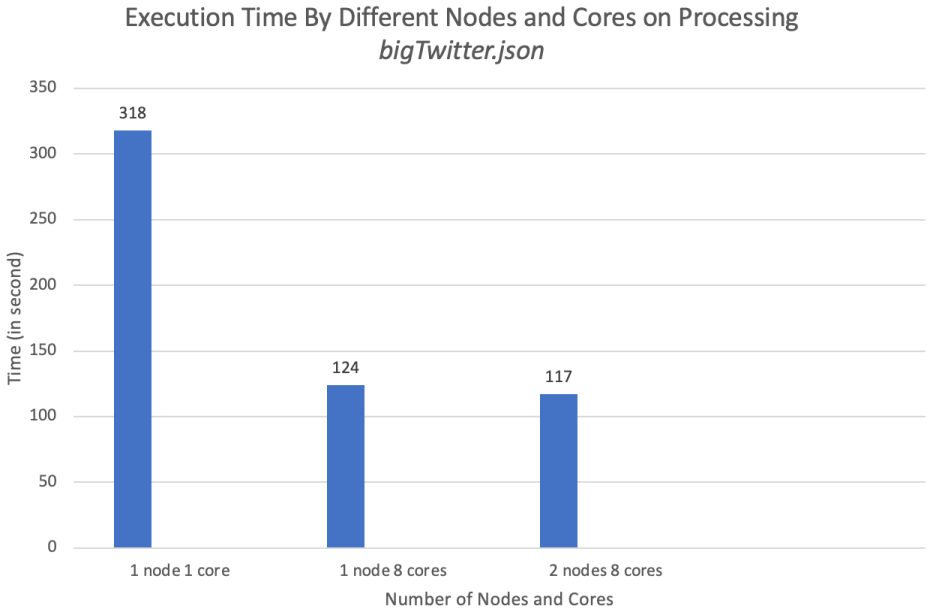

Figure 3: Performance Measurement of Parallel Processing

The performance evaluation is based on processing the `bigTwitter.json` file. We evaluate scenarios of different numbers of nodes and processors to investigate thoroughly how parallelism affects the overall time performance measured in seconds. Figure 3 clearly shows that multiple processors (1 node 8 cores and 2 nodes 8 cores) are 2.5 times faster than a single processor (1 node 1 core). Therefore, the performance of multiple processors is superior. This is expected because of the task splitting among processors in which different processors concurrently parse different parts of the file. Aside from measuring only by the number of processors, we also assessed the comparison between a single node (1 node 8 cores) and multiple nodes (2 nodes 8 cores). Multiple nodes show slightly better performance than a single node by only a 7 seconds gap, hence there is no strong evidence to show that multiple nodes could offer better performance than a single node with a similar number of processors. In our application, there is very minimal communication occurring among processes, hence network latency does not have a significant impact on the performance of multiple node configuration.

Summary

In summary, data parallelisation designed effectively can reduce the running time of an application when run on multiple processors. For our application, running the same number of processors on a single node or two nodes only has very trivial performance differences. This is mainly due to the nature of our application which does not require a significant amount of communication among processes.