COMP90024 Cluster and Cloud Computing

Assignment 1 – Multicultural City

Report

Weimin Ouyang

340438

wouyang@student.unimelb.edu.au

Shiming Zheng

1149897

shimzheng@student.unimelb.edu.au

Parallel computing is a typical capability incorporated in high performance computing (HPC) systems (Lafayette, March 23, 2022). It can be achieved by splitting up the data or tasks between multiple processors, and thus tend to result in faster program execution time (Sinnott, March 16, 2022). For Example, Gustafson-Barsis' Law suggests that a parallelised program S using N processes can run $N - \alpha(N - 1)$ times faster than S running as a sequential program, where α is the fraction of running time on the sequential part in a parallel system (Sinnott, March 16, 2022).

The current project, therefore, aimed to investigate the impact of parallel computing on program execution time through exploring the multicultural nature of Sydney, by implementing a parallelised application leveraging the University of Melbourne HPC facility Spartan. Specifically, the application was designed to process a large Twitter dataset and identify the locations of, and the languages used in making, the tweets, with a provided grid mesh of Sydney, as well as language tags identifying human languages. The application was then executed using different resources available on Spartan, including 1 node and 1 core (1/1), 1 node and 8 cores (1/8) and 2 nodes and 8 cores – with 4 cores per node (2/8).

It was proposed that faster execution could be achieved with resources with 8 cores. However, the execution time with 8 cores would be less than 8 times faster as compared to that with only 1 core, given there would likely be extra time spent on resource allocation and collection. It was also proposed that the execution time with 2/8 would be slightly slower than that with 1/8, assuming communications between 2 nodes would be more time-consuming.

Method

Data Description

Data files used for the current project included:

- 'LangCode.json' file providing information on language names and their corresponding codes
- 'SydGrid.json' file providing latitude and longitude information of a grid mesh consisting of 16 cells for Sydney
- 'bigTwitter.json' file with more than 20GB of tweet data

Data Processing and Analyses

The flow of data processing for the current project can be described with the following steps:

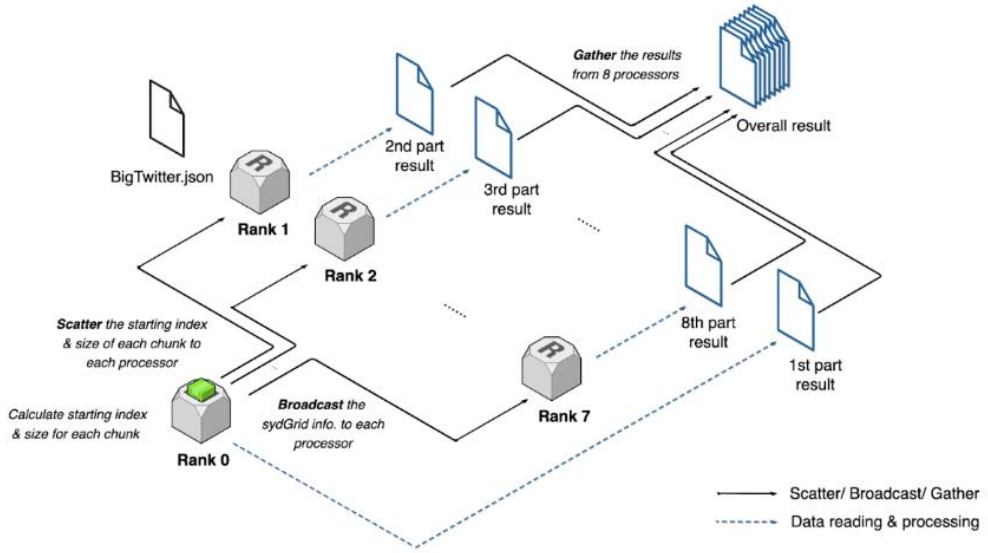
1. The 'LangCode.json' file was read into a single processor and saved as a 'langCode' dictionary in the form of {language code: language name}.
2. The 'SydGrid.json' file was read into a single processor and converted into two sorted lists of latitude and longitude values for the lines in the grid mesh. The latitude list was sorted from the largest to the smallest, representing grid lines from the top to the bottom; while the longitude list was sorted from the smallest to the largest, representing grid lines from the left to the right.
3. When a tweet line in the 'bigTwitter.json' was read and loaded by a processor, its geographical location in terms of latitude and longitude, and language code, were extracted. Tweets with either of these missing or with language code of 'und', were ignored, as suggested by the assignment description. In case when a line was not able to be loaded, it was caught as an exception and ignored.
4. Based on the extracted geographical information, each tweet was classified into one of the 16 cells in the Sydney grid. All tweets located at the outer boundaries were included, while tweets located at the inner grid lines were allocated according to a 'left, then lower' principle.
5. If the tweet has successfully been allocated to a cell, the cell number and the language it used were recorded with two separate dictionaries - 'cell_count' and 'cell_lang'. The 'cell_count' dictionary had the cells as the keys, and the recorded total number of tweets found in each cell as the values. The 'cell_lang' dictionary, on the other hand, was a nested dictionary with the cells as the keys, and dictionaries recording different languages appeared in each cell and their counts as the values.

6. After all tweets were processed, the cell numbers were converted into A1-4 through to D1-4 as illustrated in the assignment description, while the language codes were converted into language names against the previously created 'LangCode' dictionary.
7. 10 most frequently recorded languages were then identified for each cell, with ties broken according to the order they were first recorded for that cell.

Parallelisation

The large size of the bigTwitter dataset made it impossible to be completely read into memory for later processing. The parallelisation design for the current project, therefore, focused on parallelising file reading and subsequent data processing. The Single Program Multiple Data (SPMD) paradigm was adopted, with which each processor executed the same piece of code but on different parts of the data (Sinnott, March 16, 2022). In addition, Python language with the library 'mpi4py' was employed for implementation. The 'mpi4py' package provided a message passing interface (MPI) for Python to coordinate the cooperation of processors by passing messages to each through a common communications network (Lafayette, March 23, 2022). Figure 1 illustrates the steps taken to parallelise file reading and data processing for the current project.

Figure 1: Flow Chart of Designed Parallelisation



Parallelising file reading

For data reading, we adopted a chunking method that allowed all processors to read the 'bigTwitter.json' file in parallel. Firstly, in rank 0, using the file size and the total number of processors, the byte size of each chunk needed to be processed by each processor was calculated. According to the byte size for each chunk, we also calculated the starting index for each processor. Then, as shown in Figure 1, we scattered the starting indices and the sizes of the chunks to all processors by the MPI *scatter* function. As a result, rank 0 received its starting index 0 and the chunk size 2.42GB, meaning it was assigned to read and process the data file from 0GB to 2.4GB. Similarly, the starting index 2.42GB and the chunk size 2.42 GB were received by rank 1, indicating that it needed to read and process from 2.42GB to 4.84GB, and so on. This way, each processor was clear what particular part of the data they were going to read and process, thereby achieved parallelisation in file reading.

Parallelising data processing

Data were also processed in parallel following parallelised file reading. After steps 1 and 2 described in Data Processing and Analyses were completed in rank 0, the MPI *bcast* function was employed to broadcast the sorted lists of latitude and longitude values of the grid lines in the Sydney grid to each processor. With this information, after reading data in parallel, each processor was able to process their own data chunks following steps 3-5 as described in Data Processing and Analyses. Once parallelised data processing was

completed by all processors, the MPI *gather* function was used to gather the results from all processors to rank 0, which then finished data processing with steps 6-7 as described in Data Processing and Analyses to arrive at the final results (Figure 1).

Slurm Scripts

Commands were executed on Spartan HPC with Simple Linux Utility for Resource Management (slurm) scripts. Three separate scripts were produced, each requested a different set of resources. Figure 2 shows the scripts used for executing the application with 1/1, 1/8 and 2/8, respectively from the top to the bottom.

Figure 2: Slurm Scripts

```
#!/bin/bash


#SBATCH --partition=snowy
#SBATCH --job-name="1node1core"
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=1-0:00:00

# Load required modules
module --force purge
module load mpi4py/3.0.2-timed-pingpong
module load foss/2019b
module load python/3.7.4


#Launch multiple process python code
echo "1 node 1 core"
time srun -n 1 python3 master.py -data /data/projects/COMP90024/bigTwitter.json -grid /data/projects/COMP90024/sydGrid.json -code langCode.json
```

```
#!/bin/bash


#SBATCH --partition=snowy
#SBATCH --job-name="1node8core"
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=1-0:00:00

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

echo "1 node 8 cores"
time srun -n 8 python3 master.py -data /data/projects/COMP90024/bigTwitter.json -grid /data/projects/COMP90024/sydGrid.json -code langCode.json
```

```
#!/bin/bash


#SBATCH --partition=snowy
#SBATCH --job-name="2node8core"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=1-0:00:00

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

echo "2 nodes 8 cores (4 codes per node)"
time srun -n 8 python3 master.py -data /data/projects/COMP90024/smallTwitter.json -grid /data/projects/COMP90024/sydGrid.json -code langCode.json
```

Results

Data Processing Results

Figure 3 shows the data processing results. The most multicultural area in Sydney appeared to be cell C3 as it recorded the highest total number of tweets and the highest number of languages used, 4480 and 30, respectively. In contrast, cell D4 recorded the lowest total number of tweets and number of languages used: 3 tweets with only 1 language. Furthermore, English was the most widely used language in tweets for all cells in Sydney.

Figure 3: Data Processing Results

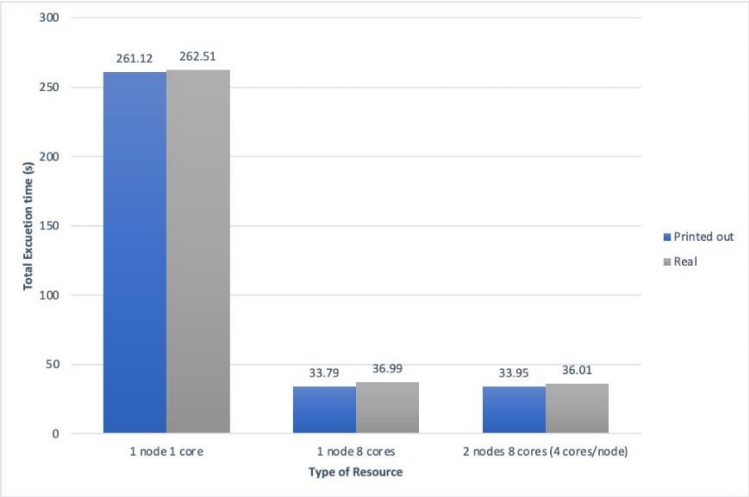
Cell	#Tweets	#Number of Languages Used	#Top 10 Languages & #Tweets
A1	21	2	[(English, 20), (French, 1)]
A2	22	2	[(English, 21), (Japanese, 1)]
A3	11	3	[(English, 9), (Indonesian, 1), (Thai, 1)]
A4	85	4	[(English, 80), (French, 2), (Turkish, 2), (Spanish, 1)]
B1	184	5	[(English, 179), (Tagalog, 2), (Haitian,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

Execution time results

Total execution time

Total execution time results demonstrated significantly reduced program execution achieved with 8 cores as compared to 1 core. Specifically, the program execution time recorded by the system (real) and the print statement (printed out) with 1/8 was 7.10 and 7.73 times faster than that with 1/1, while the program execution with 2/8 was 7.29 and 7.69 times faster than the time for 1/1, almost identical to that with 1/8 (Figure 4). In addition, while the printed out execution time appeared to be about 0.2 seconds faster for 1/8 compared to 2/8, the system recorded execution time for 1/8 was slightly slower than that with 2/8. It is worth noting that the bar chart reveals small gaps between the printed out total processing time and the system recorded total processing time, an indication of time spent on resource allocation and collection.

Figure 4: Execution Time Results on BigTwitter data



Execution time breakdown

To better understand the nature of parallelisation in the current project, we investigated the breakdown of total execution times with 1/8 (Figure 5). The time breakdown shows that a theoretical speedup of about 7.78 times could be achieved according to Gustafson-Barsis' laws, as $\alpha = \frac{1.107}{34.763} = 0.032$; $S = 8 - 0.032(8-1) = 7.776$.

Figure 5: Execution Time Breakdown

Unit: second	Initial Processing	I/O Reading and Processing (Time Subject to Parallelisation)	Results Gathering	Results Preparation	Time Independent of Parallelisation	Total Printed Execution Time
1 Node and 8 Cores	0.05	33.5	1	0.057	1.107	34.763

Discussion

The results of the current project were partly in line with our proposals.

On one hand, we observed significantly reduced execution time with 8 cores, as compared to 1 core. Though our actual results did not reach the theoretical speedup proposed by Gustafson-Barsis's law, we were very close, especially with the printed out total execution time. Given that the application was run on an HPC and there appeared to be extra time spent on resource allocation and collection etc, the results, in our opinion, still well reflected the theoretical result.

We were slightly surprised by the system recorded 2/8 result, given it was faster than that of 1/8. However, given their identical printed out total execution time, it was reasonable to assume that resource allocation and collection for our particular 1/8 run took longer than our particular 2/8, and therefore the unexpected results.

References

Lafayette, L. (2002, March 23). *The Spartan HPC system at the University of Melbourne* [Lecture recording]. Canvas@Unimelb. https://lms.unimelb.edu.au/canvas?in_c=sinfo-homepage-quick-links|source=students|medium=button|content=lms

Sinnott, R.O. (2022, March 16). *Overview of distributed and parallel computing systems* [Lecture recording]. Canvas@Unimelb. https://lms.unimelb.edu.au/canvas?in_c=sinfo-homepage-quick-links|source=students|medium=button|content=lms