Sentiment Score Analysis in Melbourne using Twitter Feed

Soham Mandal[1190783], Trina Dey[1223966]
(COMP9002 - Cluster & Cloud Computing)
University of Melbourne
13 April 2021

Abstract

In the digital age, social media is one of the prominent sources of data that hosts an abundance of human sentiments in various forms. Twitter has a 280 character text field, which along with other data is publicly available for researchers to explore. In this report, we document the approach and methodology towards reading a large json Twitter file with a substantial amount of tweets from across Australia, locate and analyse the sentiments of these tweets specifically from Melbourne, display scores and note the performance. This process calls for heavy utilisation of resources and along with the scope of this subject, the main focus of this exercise aims to implement parallel computing on various server-node configurations on the SPARTAN HPC, note and compare the performance of the jobs and draw relation to Amdahl's Law.

Introduction

Our procedure reads a json file *bigtwitter.json* consisting of millions of tweets from Australia and maps the tweets into various grids-blocks based on the location data available for Melbourne Area in *melbGrid.json* and analyses the sentiment of the tweets from a keyword-score *AFINN.txt* file.
Due to the scale of the data, loading all of it into the memory at once on a shared computational subsystem is not feasible with regards to the efficiency in computational power as serially processing the data would utilise more resources and take more time. Thus we order the data in chunks for optimised processing. We parallelize the program and note the observations of the performance on various server-node configurations.

Literature review

Parallel computing is a type of computation where many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.[1] Parallelization of a process can be done at various levels such as hardware (*multiprocessor*), operating system (*multi processes - parallel or interleaved, threads*), Implicitly by Software (support provided by *languages* or *compilers*), Explicitly by Software(using message passing libraries) etc. Interprocess communication is the key to send or receive data across multiple processes and once complete, one of the multiple processes should gather the result and provide the output. In our case, we are going to use Explicit Software Parallelization.

Every program has some part which can not be parallelized and some part which can benefit from parallelization if the resources scale up. Thus, the overall process cannot speed up beyond a certain factor. Amdahl's Law gives the theoretical formula for speed up in latency of the execution of the task at a fixed workload that can be expected of a system whose resources are improved.[2] So if a program has a **fixed execution time σ** which cannot benefit from the parallelization and an **execution time π** which can benefit from parallelization , the simplest form of Amdahl's law can be expressed as S_{lat} -

$$S_{lat} = T(1) / T(n) = \frac{\sigma + \pi}{\sigma + \pi/n} \cong 1 / \alpha$$

Where

- S_{lat} is the speed up in latency
- T(1) is the total time of process without resources improvised
- T(n) is the total time of process where resources (processors in our case) are improvised by a factor of n.
- α - fraction of running time that sequential program spends on a non parallel parts of a computation and can be given by the formula

$$\frac{\pi}{\sigma} = \frac{1-\alpha}{\alpha}$$

There are several ways to achieve parallelization, two popular approaches are -

1. **Master- Slave approach** has a master process which orchestrates parallel execution by instructing each "slave" processor where (*i.e.*, at what program counter value) to begin executing and provides each slave processor with a set of live-in values needed to execute the assigned segment (or "task") of the program.[3]
2. **Divide and Conquer approach** is where the problem is solved by breaking them up into smaller subproblems, recursively solving the subproblems, then combining the results to generate a solution to the original problem.[4]

Methodology

We are going to describe the approach we took to solve our problem.

Data Preprocessing - The files *AFINN.txt* and *melbGrid.json* are processed only once and parsed into dictionaries *map_grid*, *sentiment_dict*, *sentiment_space_dict*. These are very small files, parallel processes can wait until the base process (with rank = 0) can broadcast the data to all other processes.

Partitioning - Once the broadcasted message from the base process is received, each process starts reading from a certain line number in file. The partitioning of data is done based on the number of lines or tweets present in the file. So if there are 1000 lines in a file and 4 processes reading it, we can define the limit of the each process to be number of lines divided by the number of processes and can define the start_index as rank * limit and end_index as start_index + limit. Thus the data is now read by each process starting from a line number and reads upto a limit.

Communication - The communication from the base process to the other processes is done by the *mpi4py.*[5] library with the function MPI ***comm.bcast(data, root =0)*** which broadcasts the melbGrid and Affin scores in a dictionary to all other processes.

Processing - Once we start reading the lines there are three major computations we perform.

- Parse the line into the json to take out coordinates and text of a tweet.
- Check if the tweet is in Melbourne by comparing the lat lng bounds of a tweet with the bounds mentioned in the *melbGrid.json* & which cell id it belongs to.
- Calculate the sentiment score of the text.

There are two interesting observations we want to share.

1. **There are two ways of reading the json.** Either we can use the *json* library which converts the text into a json object. Or we can use *ijson* library which converts the text into a nested dictionary in python. We noted that the execution time of *ijson* is double the *json* library.
2. **AFINN.txt contains hyphenated text and phrases.** Splitting the sentence on space will skip some of the sentiments. For ex - "*cool stuff in St. Kilda*" when split on space generates *cool* and *stuff* separately and any matching to "*cool stuff*" as a phrase in AFINN.txt will be missed.
   a. Searching each phrase from *AFINN.txt* in the sentence meant searching 2477 phrases in each sentence with a loop count 2477 per sentence.
   b. So instead we approached splitting the *AFINN.txt* in two parts - phrases vs words. There were only 15 phrases in the file. Now we search phrases in sentences but tweet words as a lookup in the parsed dictionary which decreased our loop count to 15 + number of words in a tweet of 280 characters.

Performance Boost (on smallTwitter.json with 1 node 8 cores**)**

- For approach (a) ~360 seconds.
- For approach (b) ~2 seconds.

Aggregation - Once each process has parsed their assigned number of lines, the data comprising lat, lng, number of tweets, and sentiment score of the cell is stored in a dictionary with key as **cell id**. However, we need the data as one combined output. So we gather the data back using the **comm.gather*(data, root=0)*** function from the mpi4py library & merge it and print it as an output in the base process.

Program Execution - To run the programs on SPARTAN, we schedule our job using ***sbatch*** and ***slurm*** file. SLURM is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters.[6]

SBATCH commands in a slurm file

--partition would set the partition on the HPC.

--nodes defines the number of nodes the program will run on.

--ntasks-per-node specifies the number of tasks to be run on each node and not the number of cores.

srun or *mpirun* enables parallelisation on different processors. The default is one task per cpu, but note that the ***--cpus-per-task*** option will change this default.

We have purged all previously loaded modules under our home directory and loaded the required modules.

In the final program, we have passed two command line arguments - the total number of lines in the tweet file and path of the file to be read.

```
#!/bin/bash
#SBATCH --partition=snowy
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
```

```
#SBATCH --time=0-1:00:00

module purge
module load foss/2019b
module load python/3.7.4

count=$(wc -l bigTwitter.json| awk '{print $1}')
time srun -n 8 python  HappyCityAnalysis.py
bigTwitter.json $count
```

To run the code, copy all the data files AFINN.txt, melbGrid.json, bigTwitter.json, HappyCityAnalysis.py, HappyCityAnalysis.slurm to your home directory. On bash schedule the job using below command.

```
$ sbatch HappyCityAnalysis.slurm
```

Output

Run Time Performances for different node-core configuration is shown below. We see that as we increase the number of processors, the run time decreases and performance increases.

Output for each server node configuration

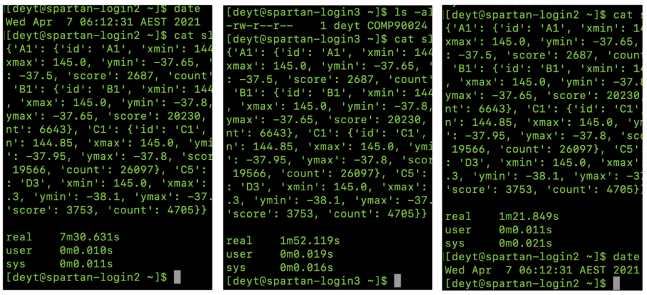

figure 1

- 1 Node 1 Core ~ 450.631 seconds.
- 1 Node 8 Core ~ 112.119 seconds
- 2 Node 4 Core each ~ 81.119 seconds

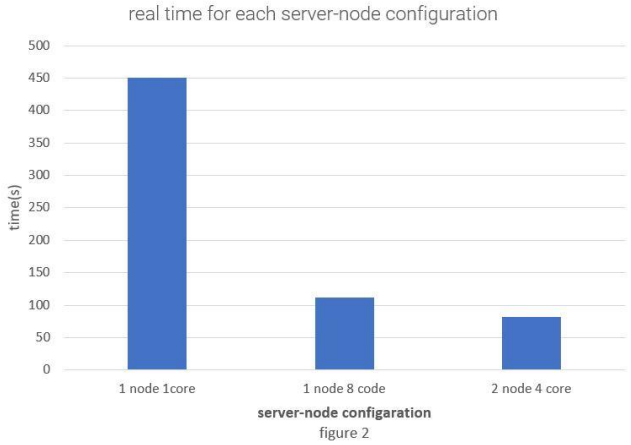

figure 2

Comparison of user vs sys time for different node core configurations.

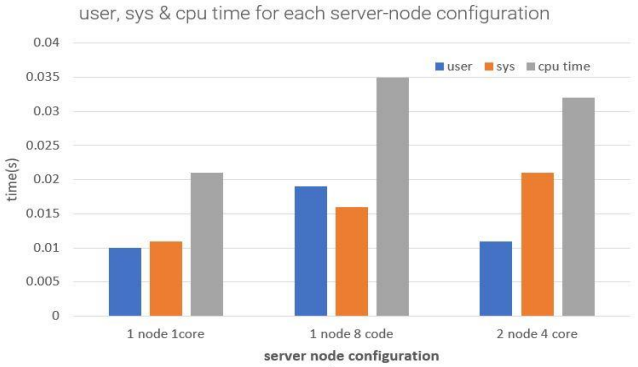

figure 3

- *usr* is the amount of time the CPU was busy executing code in user space.[7]
- *sys* is the amount of time the CPU was busy executing code in kernel space.[7]
- *cpu* is the amount of time the CPU was busy executing overall code once the process started.[7]

Comparison for Real Time noted for different node core configurations.

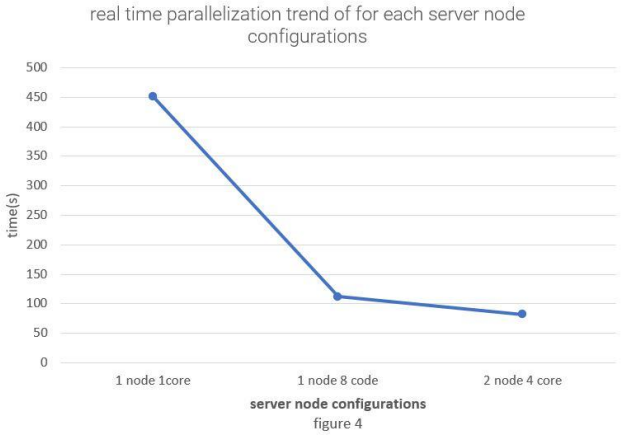

figure 4

Real time is the actual time taken from the start of a program to the end. In other words, it is the difference between the time at which a task finishes and the time at which the task started.[8] This would also involve any time the processor is waiting for parallelization or I/O.

Most happy areas in Melbourne

Cell	Score	Count	Score-Count %
B4	5740	6643	86.41
A2	4120	4904	84.01
D5	3753	4705	79.77

Least happy areas in Melbourne

Cell	Score	Count	Score-Count %
A4	57	381	14.96
A1	770	2752	27.98
B2	32085	107386	29.88

Conclusion

As a part of the exercise we recorded the real, user and sys time for different configurations in Spartan. We observe that 1 node 1 core takes 450.631 seconds, further scaling the cores to 1 node 8 cores as estimated - we observe that the real time significantly reduces to 112.119 seconds which is ~4 times less and tweaking the configuration to 2 nodes and 4 cores we realise that the time required for the program reduces to 81.33 seconds which is ~5 times less than 1 node 1 core and has a ~27% reduction in time than the 1 node 8 core setup.
Extending our findings we realise that after a point the execution time π of the program stabilizes or stops benefiting from parallelization . Hence establishing relativity to Amdahl's law.

Recommendations

Please note that the solution designed to the problem is not generalised completely and we think can be improvised in two ways -

1. We can change the program into a true Master-Slave approach where the master keeps on reading the file chunks in bytes by setting the file pointer using *file.seek()* and keeps sending the slaves the chunks.
2. In our code we have defined the limit on the number of lines to be read by each process. The *bigTwitter.json* was 14.5GB in size. But if the file size increases, the program can still go out of memory. So we need to add another for loop which iterates over ***read_files()*** function when each process has a total number of lines greater than a threshold, say 50,000, then the loop should break it into chunks of 50,000 and iterate over the number of lines. This is similar to what we have done in our program for size == 1.

References

1. "Parallel computing - Wikipedia." https://en.wikipedia.org/wiki/Parallel_computing. Accessed 13 Apr. 2021.
2. "Amdahl's law - Wikipedia." https://en.wikipedia.org/wiki/Amdahl%27s_law. Accessed 13 Apr. 2021.
3. "MASTER/SLAVE SPECULATIVE PARALLELIZATION AND ...." ftp://ftp.cs.wisc.edu/sohi/theses/zilles.pdf. Accessed 13 Apr. 2021.
4. "Automatic Parallelization of Divide and Conquer Algorithms *." https://people.csail.mit.edu/rinard/paper/ppopp99.pdf. Accessed 13 Apr. 2021.
5. "Tutorial — MPI for Python 3.0.3 documentation." https://mpi4py.readthedocs.io/en/stable/tutorial.html. Accessed 13 Apr. 2021.
6. "Slurm Workload Manager - Overview." https://slurm.schedmd.com/overview.html. Accessed 13 Apr. 2021.
7. "CPU time - Wikipedia." https://en.wikipedia.org/wiki/CPU_time. Accessed 13 Apr. 2021.
8. "Elapsed real time - Wikipedia." https://en.wikipedia.org/wiki/Elapsed_real_time. Accessed 13 Apr. 2021.

Bibliography

- Sched MD. n.d. Slurm Workload Manager - Overview. Accessed February 7, 2020. https://slurm.schedmd.com/overview.html.
- Wikipedia. 2005. "Amdahl's law." Amdahl's law - Wikipedia. https://en.wikipedia.org/wiki/Amdahl%27s_law.
- Wikipedia. 2006. "CPU time." CPU Time - Wikipedia. https://en.wikipedia.org/wiki/CPU_time.
- Wikipedia. 2009. "Elapsed real time." Elapsed real time - Wikipedia. https://en.wikipedia.org/wiki/Elapsed_real_time.
- Wikipedia. 2010. "Parallel computing." Parallel Computing - Wikipedia. https://en.wikipedia.org/wiki/Parallel_computing.
- Zilles, Craig B. 2002. *MASTER/SLAVE SPECULATIVE PARALLELIZATION AND APPROXIMATE CODE*. WISCONSIN: UNIVERSITY OF WISCONSIN - MADISON. https://ftp.cs.wisc.edu/sohi/theses/zilles.pdf.
- Rugina, Radu, and Martin Rinard. n.d. "Automatic Parallelization of Divide and Conquer Algorithms." https://people.csail.mit.edu/rinard/paper/ppopp99.pdf.