COMP90024 - Cloud and Cluster Computing, Assignment 1
HPC Twitter Processing

Steven Tang (832031)
stevent2@student.unimelb.edu.au

Qifan Deng (1077479)
qifand@student.unimelb.edu.au

April 2020

1 Introduction

Project 1 is about developing a parallel application to process a large Twitter dataset. A hybrid approach using MPI with OpenMP was taken.

The structure of this report is as follows. Section 2 outlines the architecture of the developed application, with rationale given for critical aspects of the design. Section 3 presents the performance of the application, with results discussed and analysed. Finally, potential improvements are suggested in section 4.

1.1 Compilation & invocation

1.1.1 Dependencies

- make, g++
- MPI
- OpenMP
- rapidjson (source included)

1.1.2 Compilation and invocation

```
$ make
$ mpirun [-np <n>] --bind-to none \
    ./tp <inputTwitter.json> <lang.csv>
```

2 Design and architecture

2.1 Assumptions made

Below, some assumptions and their implications are listed:

- The assignment's focus is not on the performance of the application. Only trivial performance optimisations were performed.
- It is not essential to optimise for the scenario where only one thread is available for a particular process. Currently, a sequential merging step is performed to combine results from multiple threads. This step can be skipped if there is only one thread.
- The data is assumed to be in the format that is defined in `bigTwitter.json`. That is, excluding the first and last lines, each line (delimited by `CRLF`) is a tweet in JSON[1], with a possible trailing comma.
  Additionally, such files may be truncated at the end of particular lines, as is the case with `(tiny|small)Twitter.json`.
- Input files can be concurrently read by multiple processes and threads.

2.2 Overview

The implemented application can be logically separated into three primary components.

At the top level, the input Twitter file is subdivided into n evenly-sized chunks, one for each MPI process. To achieve thread parallelism, these chunks are then further subdivided, with individual lines (or tweets in JSON) in those subdivisions parsed and processed. Results (stored in hashmaps) are then combined at both the thread and process levels to form the final result.

Because lines in the tweet file are not of fixed length and it would be expensive to determine and seek to the starting positions of particular lines, it was determined that it would be infeasible to perform divisions by using line numbers. Instead, divisions are performed at the byte level, with logic implemented to avoid duplicate processing of the same line.

2.2.1 Process

The division of the input file across multiple MPI processes is performed in `main.cpp`. The code for combining and printing results is located in `combine.cpp`.

To summarise, the following steps are taken:

[1] `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object`

1. Read the language file (which stores mappings between language codes and names).
2. Obtain the length of the input Twitter file.
3. Divide the input file into even chunks. During this step, the start and end byte positions for each process is calculated, based on the length of the file, the number of MPI processes and the rank of the current process.
4. Pass start and end positions to `threading.cpp` (see 2.2.2), and obtain two result hashmaps as the return value.
5. Processes with `rank` > 0 send their results to the master process (rank 0); the master process merges received results.
6. Format and print overall results.

2.2.2 Thread

The primary motivation behind using a threaded approach is to reduce the communication overhead that is incurred when there are many MPI processes (i.e. one for each core).

In `threading.cpp`, the section of the input file that is allocated to the current process (indicated by start and end byte positions) is further subdivided. To reduce load imbalance at the thread level, sections are subdivided into 200MB chunks, as opposed to having further even divisions between threads.

OpenMP is then used to schedule the parallel processing of the aforementioned chunks. During this process, lines in each chunk are extracted and passed to `line.cpp`, which extracts and merges results to the hashmap of the current thread. Once all the chunks have been processed, hashmaps from threads are merged in a critical section to form the overall result for the current process.

2.2.3 Line

Languages and hashtags are extracted and counted in `line.cpp`. Lines containing tweets are parsed into document objects (`DOM`) using `RapidJson`[2], a fast and self-contained JSON parser.

The language code is directly read from `DOM["doc"]["lang"]`, while for counting hashtags, the regular expression `"#[\d\w]+"` is used. With this expression, only strings which start with a `"#"` and consisting of alphanumeric characters and underscores are matched. Following this, the string is converted to lowercase. When examining the output of initial iterations, it was discovered that strings like `#1` and `#echobox` were matched by the expression. While such hashtags are valid, they were found in the description fields of users and URLs respectively.

To avoid these occurrences, two fields from the document were chosen. The first is `DOM["doc"]["entities"]["hashtags"]`[3], an array of hashtag objects, in which the string in `["text"]` is a hashtag which is then validated. The second is `DOM["doc"]["text"]`, which contains the text of the tweet. Regular expression matching is applied to this field to extract hashtags not found in the hashtags array. A set (the data structure) is then used to remove duplicates.

3 Results and analysis

3.1 Spartan

The developed application was executed on Spartan [1], the HPC system operated by the University of Melbourne.

The physical partition was chosen for its high speed networking and low latency [2]. Care was taken to increase reproducibility, by constraining the job to a particular group of nodes (using the `partition`, `constraint` and `exclude` options), and avoiding the concurrent execution of jobs by specifying `--dependency=singleton`. `contiguous` and `wait-all-nodes` were added to increase the accuracy of results and `--bind-to none` was specified to allow the application to make use of multiple cores through threading. This is presented in 3.2.

3.2 Slurm file

```
#!/bin/bash
#SBATCH --job-name="comp90024_p1"
#SBATCH --partition=physical
#SBATCH --constraint=physg4
#SBATCH --exclude=spartan-bm[027-030]
                  ,spartan-bm[018-023]
#SBATCH --contiguous
#SBATCH --wait-all-nodes=1
#SBATCH --time=0-0:10:00
#SBATCH --dependency=singleton

module purge
module load OpenMPI/3.1.0-GCC-8.2.0
make

/usr/bin/time -v mpirun --bind-to none ./tp \
/data/projects/COMP90024/bigTwitter.json \
lang.csv
```

[2] https://rapidjson.org/

[3] https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/entities-object#hashtags

The slurm file was executed with command line arguments `--nodes`, `--ntasks` and `--cpus-per-task` specified with the values (1, 1, 1), (1, 1, 8) and (2, 2, 4) for 1 node 1 core (1N1C), 1 node 8 cores (1N8C) and 2 node 8 cores (2N8C, i.e. 4 cores per node) respectively. The argument `--wait-all-nodes` is set to `1` to indicate that execution should begin when every node is ready. In this case, the time costs will not be affected by different starting times of nodes.

3.3 Results and Analysis

Results were obtained from the `physg4` group on the physical partition of Spartan.

Nodes in `physg4` have Intel(R) Xeon(R) Gold 6154 CPU processors and are connected by 25Gb networking with 1.15 μsec latency [2].

Rank	Language	Occurrence
1	English (en)	3,107,116
2	undefined (und)	252,117
3	Thai (th)	134,571
4	Portuguese (pt)	125,858
5	Spanish (es)	74,028
6	Japanese (ja)	49,929
7	Tagalog (tl)	44,560
8	Indonesian (in)	42,296
9	French (fr)	38,098
10	Arabic (ar)	24,501

Figure 1: Top 10 languages from `bigTwitter.json`

Rank	Hashtag	Occurrence
1	#auspol	19,858
2	#coronavirus	10,103
3	#firefightaustralia	6,812
4	#oldme	6,418
5	#sydney	6,237
6	#scottyfrommarketing	5,158
7	#grammys	5,085
8	#assange	4,689
9	#sportsrorts	4,483
10	#iheartawards	4,293

Figure 2: Top 10 hashtags from `bigTwitter.json`

Note that in Figure 1, `und` is mapped to `undefined` as it does not appear in the documentation[4].

[4] `https://developer.twitter.com/en/docs/twitter-for-websites/twitter-for-websites-supported-languages/overview`

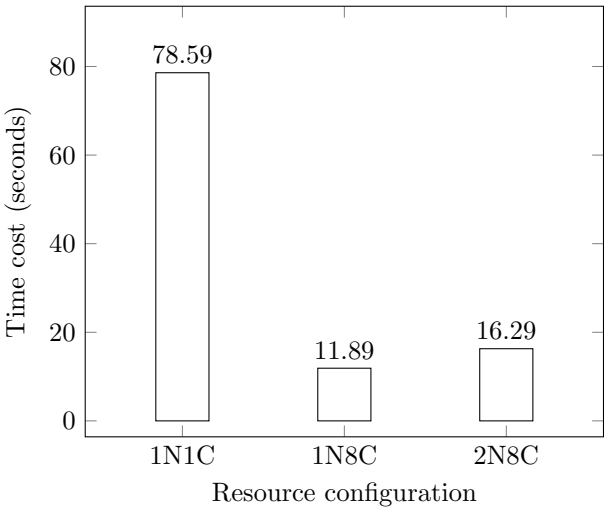

Figure 3: Execution time (s) of application on `bigTwitter.json`; `1N1C` stands for 1 node 1 core, `1N8C` stands for 1 node 8 cores and `2N8C` stands for 2 nodes and 8 cores (with 4 cores per node)

Figure 3 presents the time costs. It can be seen that 1N1C ran the longest at 78.59 seconds. The costs for 1N8C and 2N8C are similar, with the time spent by 1N8C being slightly shorter at 11.89 seconds when compared to 16.29 seconds for 2N8C. Both are about 5 times faster than 1N1C.

The cost of 1N1C can be considered to the baseline. With 8 cores in parallel, say 1N8C, the time cost is reduced. However, the theoretical speedup of 8 is not achievable as there are parts of the program which cannot be easily parallelised (e.g. the merging of results). Additionally, there are various overheads associated with the OpenMP and MPI runtime libraries and various I/O operations (e.g. network communication for message passing and reading `bigTwitter.json`).

Finally, it is unclear why 2N8C is somewhat slower than 1N8C, though it can be seen from the `/usr/bin/time` output that the CPU utilisation for `2N8C` was lower. Though further experimentation and measurements will be required, it is possible that the program was often blocked by I/O operations.

4 Further improvements

- Currently, the issue of load imbalance between nodes is not addressed. This could be a problem when nodes are heterogeneous, as faster nodes will be idle when waiting for slower nodes to finish. Possible solutions include using a distributed task queue (in place of MPI) or having the master node manage

and distribute smaller work units. However, it is important to note that the tradeoff between computation time and communication time (due to network overhead) must be examined when such alternatives are considered.

- Optimisations can be applied to the merging of results at the process level to further reduce network communication.

- Further investigation can be made on file reading. It is unknown whether it would be better to have multiple threads reading the file concurrently or a single thread which is responsible for reading the file and distributing the data to other threads.

  The current 200MB chunk size is also rather arbitrary and requires further investigation.

5 Conclusion

In this report, a parallelized application for processing a large Twitter dataset was introduced. The performance characteristics of the application under different resource configurations was described and analysed.

From the results, it can be seen that because the application has been parallelized, it is able to take advantage of the multi-node and multi-core distributed-memory environment that is provided by Spartan. However, care must be taken when configuring resources, which includes network and disk I/O, as well as node and core distributions. It is worthwhile to read the documentation of a HPC cluster to choose the best group of nodes or partition to run the application under.

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] "Spartan Documentation," Spartan Documentation. [Online]. Available: https://dashboard.hpc.unimelb.edu.au/. [Accessed: 14-Apr-2020].