COMP90024 - CLUSTER AND CLOUD COMPUTING ASSIGNMENT 1 REPORT

Hongwei Chen (891632), Zhen Cai (1049487)

ABSTRACT

Parallel computing is a model for dividing computationally hard problems into a number of easier-to-solve ones, for multiple processors to execute at the same time. Message Passing Interface (MPI) standardizes the library routines for interprocess communication in a distributed-memory environment, and there are several open-source implementations available allowing programmers to parallel code that operate on different nodes. This report describes our method of using mpi4py to solve a 19.34GB twitter data analysis task. Our best result contributes a completion time of only 34 second. Investigated by Amdahl's law, we estimate that the bottleneck, i.e. completion time of non-parallelisable part, should be bounded up by 5.43 second maximum.

Keywords Twitter · HPC · mpi4py · slurm

1. INTRODUCTION

This project implements a parallelized message-passing application leveraging compute power of SPARTAN, an HPC facility at the University of Melbourne, which that program searches a large twitter dataset for Sydney, in order to identify possible geospatial insights on languages used in making Tweets. The application should run with varying hardware resources in physical partition, once for each of: 1 node and 1 core; 1 node and 8 cores; 2 nodes and 8 cores (with 4 cores per node). The wall-clock time and overall CPU and memory usage are measured.

Results of *2n8c.out* can be found in appendix. It counts the number of tweets and languages in each given cell, as well as lists the top-10 most popular languages and number of tweets produced in each. Tweets with no location information or made outside of the grid can be ignored. Mapping between language codes and their names is based on Twitter developer platform [1], and we decide those outside the scope will simply output their language code.

2. DATA

2.1. Grid

The *sydGrid.json* file specifies a (4×4) grid in Sydney, by including coordinates of each corner in the 16 cells. The JSON file has a brief structure as below:

```
1  { ..
2    "features": [
3      { ..
4        "geometry": {
5          "type": "Polygon",
6          "coordinates": [[
7            [long1, lat1], [..], [..], [..], [..]
8          ]]
9        }
10     }, {..}, ..
11   ]
12 }
```

A bounding box, as shown in line 7, is made up of 5 points, the first of which is the same as the fifth just for closure. In Figure 1, the elbow arrow and numbering reflect respectively the ordering of points and cells in *features* array.

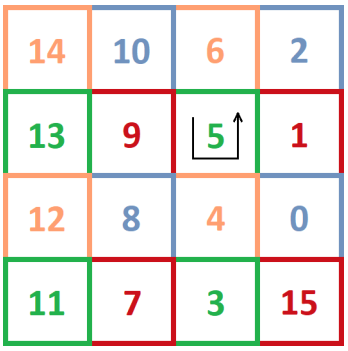

Fig. 1. The ordering of points and cells in *sydGrid.json*, and cell determination of on-border tweets. Zhen's drawing.

If a tweet occurs right on the upper or rightmost border or in the top-right corner of cell C, then it belongs to C. The remaining borders and corners are plainly considered "can-be-found" in other cells, for the 3×3 grid from A2 to C4 or perhaps from 10 to 0 here on our drawing (Figure 1). The bottom and leftmost borders of the entire mesh, including their endpoints, are the only ones not yet covered by our rule. Most of them give a natural solution, nonetheless it's worthnoting that we prefer to assign an on-border tweet at the corner to its left or bottom, e.g. the leftmost corner between 12 and 13 is determined in cell 13.

2.2. Tweets

Three Twitter datasets are provided, where they have the same structure but differ in size. We use the *bigTwitter.json* file for final analysis, while the other 2 files are solely used for software development and testing.

Except for line 1, each line is one tweet exclusively, so allowing for line-level parsing. We just extract what we need from a line since not all of the information of a tweet is relevant to our problem space. Thus we create a *ConciseTweet* data structure with only 2 properties: coordinates and lang(uage).

3. PARALLELISM

Our strategy shares a similar idea as MapReduce. It uses the Single Program Multiple Data (SPMD) model, which means that each process runs the same code but on separate sections of the data. At this point, data refers to *bigTwitter.json*, on which a non-trivial split is performed evenly. Each process stores a Python defaultdict(Counter) object *cell_stats* as a counting accumulator, i.e. in each cell, one counter on the number of tweets made in various languages. Then we simply design a new MPI operation to collect results from all processes to root, i.e. rank 0, to handle output. Data is read line by line to prevent memory overflow.

```
cell_stats <- defaultdict(Counter)
section <- split_sections(bigTwitter.json)      (*)
batch <- []
while (section has not ended) {                 (*)
  batch add read_one_line(section)
  if len(batch) >= 50 then    # size
    update cell_stats with batch
    batch <- []
  endif
}
op <- MPI.Op.create(addDictCounter)
cell_stats <- reduce_to_root(cell_stats, op)
```

Now let's take a closer look at the split below.

```
f <- open_in_bytes_format(bigTwitter.json)
sec_length <- len(f) // mpi_size
read_start, read_end <- sec_length*rank,
    sec_length*(rank+1)

# ensure last worker stop before EOF
if rank == size - 1 then
  read_end <- file_end - 1
endif

# ditch partial line
if rank != 0 then
  read_len <- read_one_line_from(read_start)
  read_start += read_len
endif
```

```
while (read_start <= read_end) {
  l <- read_one_line_from(read_start)
  read_start += len(l)
  batch add l
  if len(batch) >= 50 || read_start > read_end
      then
    update cell_stats with batch.decode(uft-8)
    batch <- []
  endif
}
```

Below is the xxd [2] output for the row of the *bigTwitter.json* file. ".." has hex dump of "0d0a", that is, CR-LF.

```
0000000: 7b22 746f 7461 6c5f 726f 7773 223a 3130
    {"total_rows":10
0000010: 3030 2c22 726f 7773 223a 5b0d 0a7b 2269
    00,"rows":[..{"i
```

We open the file in bytes and then decode using UTF-8 afterwards, rather than straightforward open in UTF-8 as usual. This is because Python's 'r'-flag file reading optimizes for CR-LF, i.e. replacing \r\n with \n. Then one section for each process can be determined, but because one section can start or stop halfway down a line, we proceed the current line and position our starting point at the head of the following, then let the process, whose rank is 1 less, finish the partial line up. Iteratively, read lines from data until it reaches the end.

4. SUBMISSION ON SPARTAN

Bash scripts are used to submit SLURM jobs on SPARTAN.

```
bash run.sh /data/projects/COMP90024
```

The file *run.sh* essentially does 4 steps:

1. make a symbolic link to files,
2. load modules,
3. install Python dependencies,
4. *sbatch* jobs on 3 different resource configurations.

The modules used are *foss/2021a*, *mpi4py/3.0.2-timed-pingpong*, and *python/3.9.5*. In any case, we don't use *sbatch* for module loading; instead, we run module commands in our bash script early, then the setup will be propagated to all nodes in the allocation.

SLURM directives have a syntax of *#SBATCH [option]*. For example, the following is the *2n8c.slurm* file to request 2 nodes with 4 cores each, and a 1-minute time limit. Line 3 refers to the output path. Line 12 is used to view statistics when job has completed. There isn't much of a difference between this and the other 2 SLURM jobs.

```
#!/bin/bash

#SBATCH -o output/2n8c.out
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH -t 1:00
```

```
cd ${SLURM_SUBMIT_DIR}

srun -n 8 python3 count.py bigTwitter.json sydGrid.json

my-job-stats -a -n -s
```

5. PERFORMANCE ANALYSIS

From the results of executing on each of the 3 Spartan resources, the difference is obvious, as shown in Figure 2. The completion time of 1n8c and 2n8c are temporally near, as both are in physical partition. Such partition is well suited to multi-node tasks since each node is connected by high-speed 50Gb networking with only 1.5 µsec latency. [3] Despite the transmission delay between nodes, 2n8c time is surprisingly shorter than 1n1c by 1 second. In our opinion, intermittent MPI speed downgrades in 1n8c and machines' variable speed and utilization state are very likely to be the main reasons for this. Overall, compared to 1-node-1-core, we achieve a 680% and 700% speedup respectively.

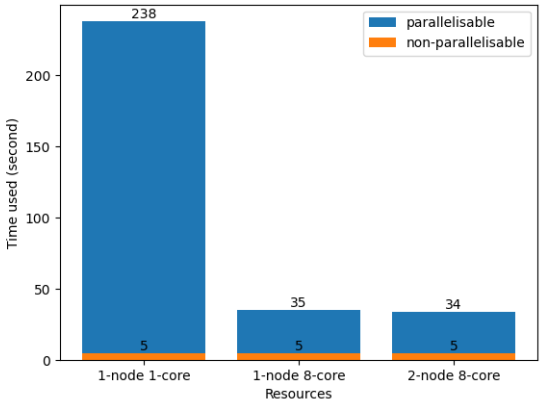

Fig. 2. Performance comparison.

According to Amdahl's Law, for a fixed problem size, a few sequential instructions will have a limiting factor on program speedup, hence increasing number of processors may not have positive feedback. Let the actual non-parallelisable part (e.g. printing output) costs time σ. Regardless of whether we can further optimize to make the code run faster, we all have

$$\frac{238-\sigma}{8}+\sigma \leq (35+34)/2 \implies \sigma \leq 5.43.$$

Because network delay and parallelism overhead are there, which lengthen the execution, a $\leq$ indication is used above. This model must be inaccurate, but one that nonetheless is considered sufficient for an intuition, that a significant portion of a job can be distributed among a range of worker processes.

6. CONCLUSION

In this project, we parallelized the solution of the twitter data analysis task to a great extent. In the process, we also gained experience in calling the Message Passing Interface via Python library mpi4py and using the SLURM job queuing system. For any future improvement, we suggest refrain from hardcoding the comprehensive language code list, instead crawl the list or consider the Twitter API, so leads to an increased amount of reliability and readability.

7. REFERENCES

[1] Use Cases, Tutorials, & Documentation — Twitter Developer Platform. (n.d.). *Supported languages and browsers.* https://developer.twitter.com/en/docs/twitter-for-websites/supported-languages

[2] Linux Documentation. (n.d.). *Xxd(1): Make hexdump/do reverse - Linux man page.* https://linux.die.net/man/1/xxd

[3] (2022, April 2). *Spartan Documentation.* https://dashboard.hpc.unimelb.edu.au/status_specs/