COMP90024 Assignment 1 Report

Team Members:
Rongxiao Liu 927694
Xudong Ma 822009

1. Introduction

This project aims to identify the areas that have the most Twitter posts and the most frequent hashtags in these areas around Melbourne, according to given files ‘bigTwitter.json’ and ‘melbGrid.json’.

2. Slurm Scripts

(1) 1 node 1 core:

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
module load Python/3.6.4-intel-2017.u2
export PYTHONIOENCODING=UTF_8
echo '1 node 1 core'
time mpirun python3 Assignment_1.py
```

(2) 1 node 8 cores:

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
module load Python/3.6.4-intel-2017.u2
export PYTHONIOENCODING=UTF_8
echo '1 node 8 cores'
time mpirun python3 Assignment_1.py
```

(3) 2 nodes 8 cores:

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
module load Python/3.6.4-intel-2017.u2
export PYTHONIOENCODING=UTF_8
echo '2 nodes 8 cores'
time mpirun python3 Assignment_1.py
```

3. Approaches

Design logic:

In this project, we use python3 to program. The unimelb HPC platform SPARTAN provides a multi-processors and multi-nodes environment, which is a Multiple Instruction Multiple Data streams (MIMD) architecture, to scale the computational capability horizontally. In MIMD architecture, processors work independently and can execute different instructions on different data. ‘mpi4py’ library is used to implement process communication. Moreover, our program follows a Single-Program Multiple-Data (SPMD) model, which can run the same code on all processes but handle different data when processors are more than one. Another key idea is Divide and Conquer that divides the tasks into parts so that each process only needs to handle their own parts in the file simultaneously, and finally merge their results. As a result, the running time can be shortened.

According to ‘Amdahl's Law’ - $S = 1/(\alpha+(1-\alpha)/N)$, the speed-up (S) can be increased by adding more processors (N) or increasing proportion of parallel code ($1-\alpha$)/decreasing proportion of non-parallel code (α).

(1) add processors (N): We will test the performance of three sets of condition including 1 node 1 core, 1 node 8 cores and 2 node 8 cores.

(2) decreasing non-parallel program (α): In this assignment, some overheads such as gathered data calculation and results printing cannot be parallelized. However, two main tasks ‘file reading’ and ‘lines handling’ can be parallelized easily. To make as more program as possible can be parallel, these two parts should be allocated to all processes equally. The ‘file reading’ part is parallelized by allocate respective offset and size of file so that each process only read 1/8 size of the file (if 8 processes). The ‘lines handing’ part is also parallelized because each process only handle the lines they read from respective part in file (1/8 as well if 8 processes).

Code implementation:

The key class ‘Grid_cell’ used in this program is to define each area’s range and count the posts in these areas. This class includes some methods: ‘get_id’, ‘get_count’, ‘get_hashtags_freq’, ‘is_include’, ‘add_count’, ‘add_hashtag’.

At the beginning, for each area in ‘melbGrid.json’, we create an instance of ‘Grid_cell’ class for each area based on its coordinates range and append them in a list (grid_cells).

After that, we get the location range of the file that current processes need to handle based on file size and processes amount (divided equally), and start to handle this part of file line by line: if this tweet is in this area (using ‘doc’-‘coordinates’-‘coordinates’ value and ‘is_include(postgeo)’ function to judge), call function ‘add_count()’ of this instance and ‘add_hashtag(hashtag)’ for each hashtag in this tweet.

Finally, when each process finishes their program, they will gather required data to process 0 through ‘gather()’ method in ‘mpi4py’ library. Then process 0 will merge these results and print the final result.

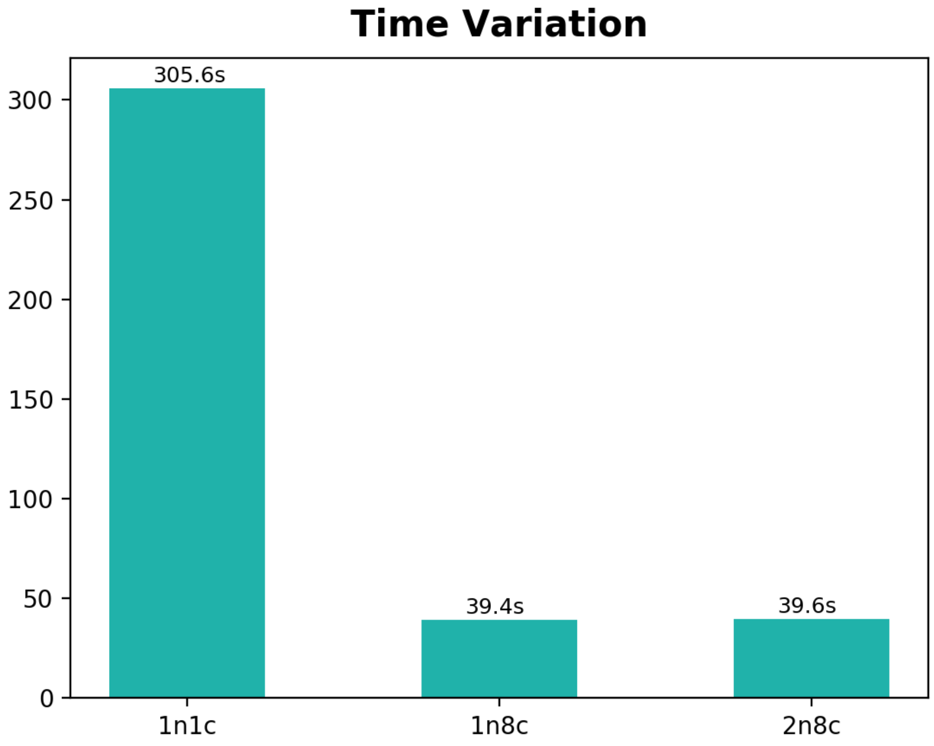
4. Performance variations

When it comes to running time of these three conditions, the time spent on 1n1c is very long (T(1) = 305.629s), which is a baseline that will be used to compare.

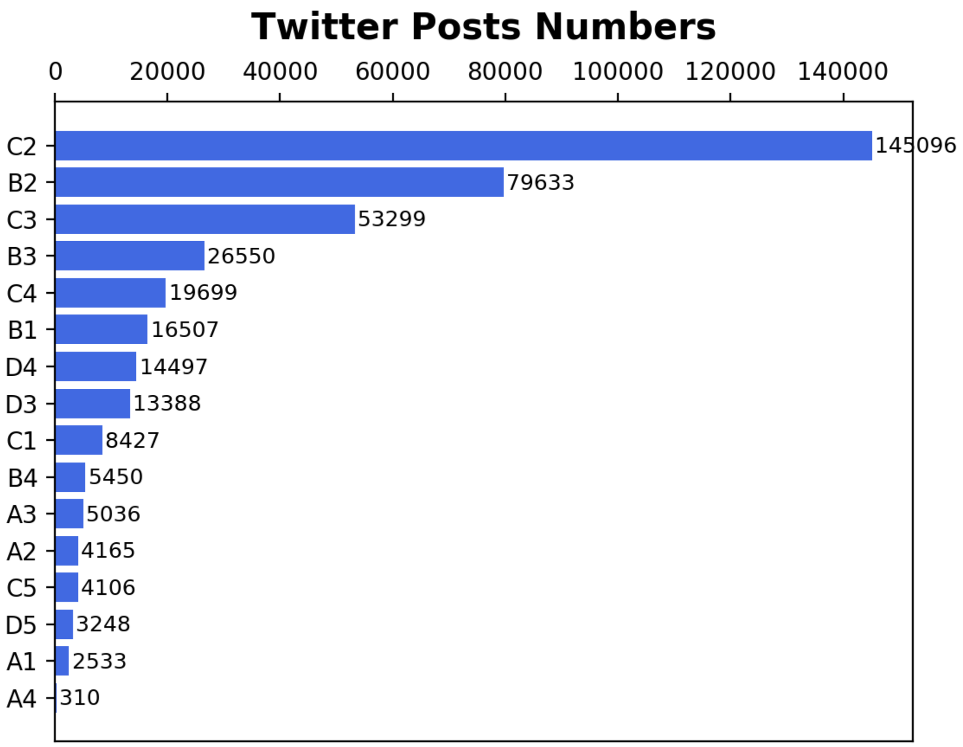
When we execute the code on 1n8c, the time is significantly shortened 8 times to 39.439s (T(8)). It is because the task is divided into 8 parts and there are 7 more cores sharing the load at the same time. Finally, we get S(8) = T(1)/T(8) = 305.629s/39.439s = 7.749. Based on 'Amdahl's Law': $S = 1/(\alpha+(1-\alpha)/8) = 7.749$, we can get α = 0.5% (99.5% program is parallelized). The reason why the proportion of parallelized program is so large is that the file we handle is very big (more than 10GB).

However, when it comes to a similar condition that has 8 cores as well but at 2 separate nodes (2n8c), the time increases a little bit (39.628s) but is still much shorter than the 1 cores baseline (S = 305.629s/39.628s = 7.712). The reason that 2n8c runs slower than 1n8c may be that the communication time between 2 separate nodes is much longer than communication time between cores at the same node. Considering that in our design, processes need communication only at 'gathering data' stage, the spent time are not much difference.

Another important factor that causes running time variation is that these jobs are dynamically allocated to different nodes. The performances of these nodes may vary to some extent, which depends on differences on resource usage.



The three outputs from 1 node 1 core (1n1c), 1 node 8 cores (1n8c) and 2 node 8 cores (2n8c) are the same: (ranked areas on posts number as below; hashtag results are too long to show, details in “1n1c.out”, “1n8c.out”, “2n8c.out” files)


Twitter Posts Numbers
C2
B2
C3
B3
C4
B1
D4
D3
C1
B4
A3
A2
C5
D5
A1
A4