COMP90024 Cluster and Cloud Computing

Assignment 1 ***HPC Instagram GeoProcessing***

Tiange Wang 903588

1. Introduction

The report primarily demonstrates that how to count the number of Instagram posts in Melbourne coordinate grids through MPI and Python, in order to provide specific foundation information for the further data analysis in the next step.

The datasets used in this project includes:

- bigInstagram.json
  This is the 6G JSON file contains thousands of posts and coordinates.
- melbGrid.json
  This is the JSON file includes Melbourne coordinate grids.

The development platform is Python with package mpi4py, and the operating platform is Spartan – HPC facility of Unimelb.

2. Approaches description

- create_mel_grid (file)

The function reads IDs and X-Y coordinates of each grid from melbGrid.json, creates a dictionary type of set 'grids' to save the relevant data in the array 'mel_grid' and ignores other useless and unrelated data. In addition, it adds the key 'row' and 'column' to indicate which row and column this grid belongs to, to prepare for the following output (group by row & group by column). Meanwhile, it adds the key 'related_post_number' to count the number of posts which is in the area of this grid.

```
with open(file) as f1:
    features = json.load(f1)
    for each_row in features["features"]:
        # grids = {}
        # Created a dictionary structure set called 'grids'
        grids = dict()
        grids["id"] = each_row["properties"]["id"]
        grids["xmin"] = each_row["properties"]["xmin"]
        grids["xmax"] = each_row["properties"]["xmax"]
        grids["ymin"] = each_row["properties"]["ymin"]
        grids["ymax"] = each_row["properties"]["ymax"]
        # Take 'A' 'B' 'C' 'D'
        grids["row"] = each_row["properties"]["id"][:1]
        # Take '1' '2' '3' '4' '5'
        grids["column"] = each_row["properties"]["id"][1:2]
        grids["related_post_number"] = 0
        mel_grid.append(grids)
```

Figure 1. The create function.

- get_coordinate (file)

The function reads data line by line from bigInstagram.json to save the coordinate information of the posts which contain coordinates in the array 'coordinates'. It ignores the unrelated data and symbols, especially the first line and commas and line breaks at the end of each line. Since the function json.load () shows error when reading the 6G large file, it uses json.loads () instead to read the file line by line.

However, json.loads also has limitations. For instance, the parameter has to be the integrated JSON string. Thus, I use slicing to remove commas and line breaks to obtain the part of JSON string. After browsing the tiny file, I found that there is no comma in the penultimate line. Hence, I use different slicing for operating this particular case.

```
coord = json.loads(each_row[:-2])
coord = json.loads(each_row[:-1])
```

Figure 2. Display of slicing.

Meanwhile, in the data processing aspect, I use the structure 'try … except continue' to skip these posts which do not have coordinates data. Particularly, for parallel tasks, as the function comm.scatter() which would be utilised in the main function only support the equal division, I have to use the function numpy.array_split () from numpy package to do the unequal split task based on the number of nodes beforehand.

- related_post_number (grids, longitude, latitude)

The function reads longitude and latitude data of posts from the array 'coordinates' and decides which grid the post belongs to. Then it adds the number of the key 'related_post_number' in the array 'mel_grid'. I use IF statements to ignore the coordinates which are out of the bound of Melbourne grids.

```
def related_post_number(grids, longitude, latitude):
    for each_grid in grids:
        try:
            if (longitude >= each_grid["xmin"] and longitude <= each_grid["xmax"]) and \
                    (latitude >= each_grid["ymin"] and latitude <= each_grid["ymax"]):
                each_grid["related_post_number"] = each_grid["related_post_number"] + 1
        except:
            continue
```

Figure 3. The display of the function.

- main function

After call create_mel_grid (file) and get_coordinate (file), the array 'mel_grid' and 'coordinates' have obtained the related data. In the step of calling the 'related_post_number' function, it judges each coordinate and saves the result to the array 'valid_number'. Then it prints the grid result, row result, column result and the total time.

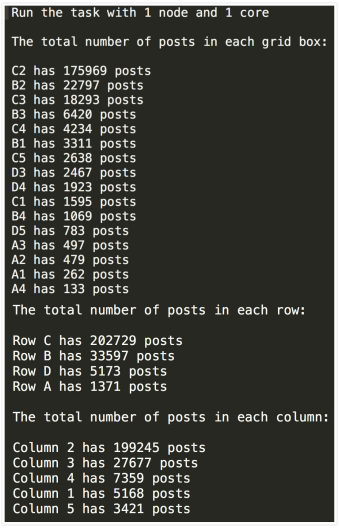
Particularly, for the circumstance of multi-nodes, it uses the MPI scatter function comm.scatter () to distribute the tasks to different nodes and the gather function comm.gather () to collect and integrate the processing result from these nodes. Nonetheless, the output function of single-node-mode does not support the split data structure. Hence, the alternative output function is required to process the data structure 'array[array[…], array[…]…]' and process two times loop.

```
coordinate = comm.scatter(coordinates, root=0)
for each_data in coordinate:
    related_post_number(mel_grid, each_data["longitude"], each_data["latitude"])
valid_number = comm.gather(mel_grid, root=0)
```

Figure 4. The process for multi-nodes.

3. Analysis and conclusion

After running the SLURM code on Spartan, I gain the result (shown below) and compare the total time of three resources: 1 node 1 core/ 1 node 8 cores/ 2 nodes 8 cores.



```
Run the task with 1 node and 1 core

The total number of posts in each grid box:

C2 has 175969 posts
B2 has 22797 posts
C3 has 18293 posts
B3 has 6420 posts
C4 has 4234 posts
B1 has 3311 posts
C5 has 2638 posts
D3 has 2467 posts
D4 has 1923 posts
C1 has 1595 posts
B4 has 1069 posts
D5 has 783 posts
A3 has 497 posts
A2 has 479 posts
A1 has 262 posts
A4 has 133 posts

The total number of posts in each row:

Row C has 202729 posts
Row B has 33597 posts
Row D has 5173 posts
Row A has 1371 posts

The total number of posts in each column:

Column 2 has 199245 posts
Column 3 has 27677 posts
Column 4 has 7359 posts
Column 1 has 5168 posts
Column 5 has 3421 posts
```

Figure 6. The output result

	1 node 1 core	1 node 8 cores	2 nodes 8 cores
Total time (s)	155.23015	153.58968	153.46628

Figure 5. The compared result of different resources.

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=00:10:00

echo 'Run the task with 2 node and 8 core'

module load Python/3.5.2-goolf-2015a
mpiexec python Instagram_rank.py
```

Figure 7. The code of SLRUM.

Shown in figure 5 is the comparison between different resources. It illustrates that the operation speed improves tinily after increasing nodes and cores. However, in general, the total time of 2 nodes 8 cores is the smallest. It demonstrates that the parallel programming enhances the executing speed, and all the nodes and cores could gain a task and process calculation at the same time. Thus, the resources are fully utilised to maximise the performance.

Nevertheless, for this program, the decrease of the time between 1 node and 2 nodes is not apparent, which only saves almost 2 seconds. In my opinion, the primary reason is that I only split the tasks in the process of judging whether a coordinate belongs to the Melbourne grids and which grid it is. Meanwhile, it also takes some time when distributing tasks to nodes. In the process of testing, I also found the program reads the file line by line through json.loads (string) is slower than use json.load (file) directly. If the program distributes the task of reading files to each node, the advantage of parallel programming would be more prominent.