AMS 598 Project 1 Report

Tinghui Wu 114719023

Sep 30, 2022

1 Introduction

The aim of this project is to implement map reduce algorithm on MPI using Python. We'd need to read 20 data files contained 1.5 millions integers ranging from 0-100 and find the top 5 integers with highest frequencies.

2 Methods

2.1 Mapper

Since we have several text files, we can separate files to different nodes. In our cases using 4 nodes, each node would process 5 files. Then, each node would read the integers from the file and collect the counts into a dictionary. After finishing, the dictionary would be saved as a pickle file in a temporary folder.

```python
def mapper(file_list, tmp_dir, rank):
    '''This mapper function collect data into a map (dictionary)
    '''

    counts = {}
    for file_name in file_list:
        f = open(file_name, 'r')
        for number in f:
            number = int(number)
            if not number in counts:
                counts[number] = 1
            else:
                counts[number] += 1
        f.close()

    with open(os.path.join(tmp_dir, 'tmp_{}.pkl'.format(str(rank))), 'wb') as tmp_file:
        pickle.dump(counts, tmp_file, pickle.HIGHEST_PROTOCOL)
```

2.2 Reducer

In the reducer process, each node would gather the total amount of the integers from the 4 previous pickle file. For example, the first node (node 0) would sum up the numbers of integer 0-25, the second node would process 26-50, etc. Since we only need to report the top 5 most frequent integer, we can sort and send out the top 5 integer separately so that we only need to sort 25 numbers in each node.

```python
def reducer(tmp_dir, n_per_node, k, rank):
    '''This reducer function merge all counts together
    '''
    total = np.zeros(n_per_node)
    for curr_file in os.listdir(tmp_dir):
        with open(os.path.join(tmp_dir, curr_file), 'rb') as read_file:
            curr_map = pickle.load(read_file)
        for i in range(n_per_node):
            target = i + rank * n_per_node
            total[i] += curr_map[target]

    index = np.argsort(total)[::-1][:k]
    count = total[index]
    index += rank * n_per_node
    return index, count
```

2.3 Gather

After the reducer was done, we would use gather function to send the top 5 number to the root node. In our case, the root node would receive 20 numbers in total. Then, the root node would sort the last 20 frequencies again to find the top 5 integers.

3 Results

3.1 Main Output

After running the code, we got the following results in Table 1.

Integer	Frequency
23	966572
17	965845
48	965580
88	965565
71	964051

Table 1: Results of integer counts

3.2 Timing Analysis

Furthermore, we timed the program in each step:

- Step 1: Gathering the data file names and all the preparing
- Step 2: Mapping
- Step 3: Reducing
- Step 4: Gathering and report the result (mainly the root node)

and print the timing as Figure 1.

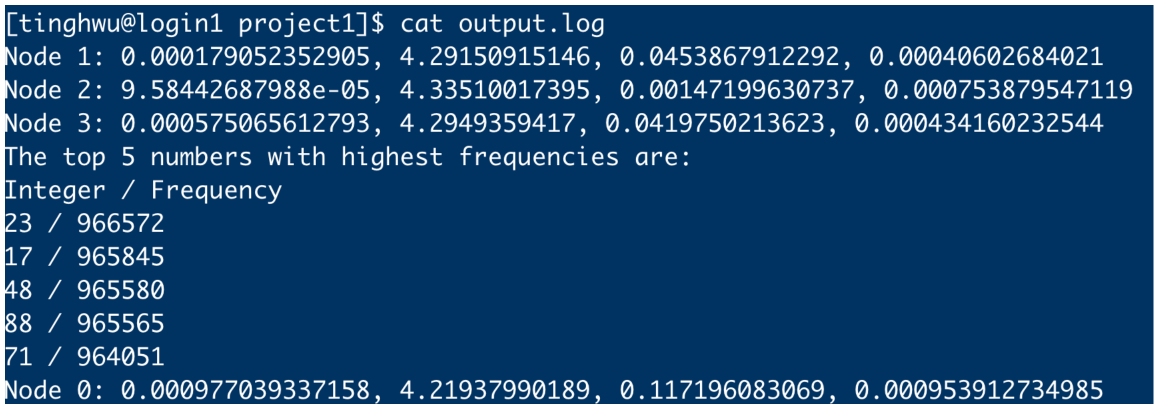
We can see that the step 2: Mapping took most of the time to run (4 seconds on 5 files). And the rest of the process took around 0.1 second. Therefore, we can roughly estimate the total time taken would be

$$\text{time} = \frac{4}{5}\frac{20}{\text{node}} + 0.1 = \frac{16}{\text{node}} + 0.1$$

We also run the code using different nodes and reported the time as in Figure 2, which was similar to our estimation.

Number of nodes	Time (seconds)
1	15
2	8
4	5

Table 2: Time taken running in different number of nodes



```
[tinghwu@login1 project1]$ cat output.log
Node 1: 0.000179052352905, 4.29150915146, 0.0453867912292, 0.00040602684021
Node 2: 9.58442687988e-05, 4.33510017395, 0.00147199630737, 0.000753879547119
Node 3: 0.000575065612793, 4.2949359417, 0.0419750213623, 0.000434160232544
The top 5 numbers with highest frequencies are:
Integer / Frequency
23 / 966572
17 / 965845
48 / 965580
88 / 965565
71 / 964051
Node 0: 0.000977039337158, 4.21937990189, 0.117196083069, 0.000953912734985
```

Figure 1: Total output from the code

4 Future Work

Here are possible improvement that some parameters could be more flexible in the future:

- Number of file to be read by each node: for now the total files should be perfectly divided by the number of nodes. We should modify to let the files separated to each node no matter what.
- Type of the counted items: for now the counted items were fixed to integers started from 0. It could be improved to accept more item other than integers such as float or text.
- Number of the counted items: similar to the number of files, for now we assume that the number of counted items could be perfectly divided by the number of nodes, which could also be improved by making the distribution more flexible.