HPC using MPI on Instagram Files

Siddharth Malhotra (934336)

Department of Computing and Information System
The University of Melbourne

COMP90024 – Cluster and Cloud Computing – Assignment 1

A report submitted for the degree of

Master's of Science

March 2018

Problem Statement:

The project deals with an ordered enumeration of posts found in an Instagram file based on a set of coordinates of Melbourne. The Instagram file here, namely, 'bigInstagram.json' is sized at over 3.5 GB; while the set of coordinates are maintained in a 'melbGrid.json' file is sized at 8KB. These are 16 grids which are divided based on the coordinates such as 'xmin', 'xmax', 'ymin', 'ymax'. We perform high performance computing provided by the Spartan facility, which has, compilers and packages pre-installed for several platforms. The idea is to implement a Message Passing Interface (MPI) over a set of nodes and clusters. We first begin our testing on a 1 node 1 core platform, then 1 node 8 core platform and finally on a 2 node 8 core platform. Through this report, we wish to explain the functionality of our code, an overview of the message passing implementation and finally we present the results of our study.

We implement the project on Python 3.6 as Python 3.6 has a solid library support for text analysis and string processing. Further, the MPI functionality is intuitive and well built. For local testing we use the Anaconda environment and install MPI packages. On Spartan this is pre-installed, and the scripts are run through the slurm script which are batch files allowing access to number of cores and nodes we need.

Technique:

2.1 Pre-processing

Perhaps, one of the most fundamental tasks when dealing with large amounts of data is to perform through text analysis. This has two major advantages, fast task processing (due to less amount of data code as to traverse) and efficient debugging. Since the data provided is structured. We use regular expressions for extracting data from the 'bigInstagram.json'. In order to accelerate Regex process, we perform it as part of the MPI execution, covered in section 2.2.

As part of the pre-processing we maintain a count of the number of lines our program will have to traverse. This helps us for the following purposes:

- Partitioning the data and allocation of resources.
- Checking for exceptions in the tweet file (which aren't tweet) and to ignore them without traversing them.

2.2 Computing and MPI

We perform the computation of the count of the lines in the Instagram file and calculation of the number of partitions; we check for the begin and end of each partition/chunk of data. The reason we do this is because we want each core to maintain its reference of where it should begin and end reading. Thus, this allows us to efficiently manage the performance of our system, wherein, the only information each core is supposed to know is what is its range of computation.

We perform a readline() on each of the 'bigInstagram.json' file in this range and use Regex expressions for building a list of coordinates in question. Further, a dictionary of the blocks is maintained with reference to the 'melbGrid.json'. Finally, a loop is run through the system checking the occurrence of coordinates into the dictionary wherein, we increment each time we find an occurrence. It is important to note that this computation is performed on individual core levels.

Post the count and accumulation of these numbers from individual cores, we perform a gather of the final_result at tbe master node accessible via root=0. The the root node's main purpose is the collection of the results.

Finally, we display the results based on the blocks, rows and columns. We initiate the block sum with 0 value. Further we set the default values of the rows and columns to 0. We access these values by picking out values through the positions of their characters. Since this is a dictionary, we order them based on the sorted() function from the Python library. We reverse these values to order them in descending order.

Throughout the entire project we run a clock().This is run prior to the invocation of the main function and stops post the printing of the results.

Invocation:

3.1 SLURM scripts

A) 1 Node 1 Core:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=1:00:00
#SBATCH --partition=physical

module load Python/3.5.2-goolf-2015a

mpirun python3 Assignment1.py
```

B) 1 Node 8 Core:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=0-1:00:00
#SBATCH --partition=physical
module load Python/3.5.2-goolf-2015a
mpiexec python3 Assignment1.py
```

C)1 Node 8 Core:

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --time=0-1:00:00
#SBATCH --partition=physical
module load Python/3.5.2-goolf-2015a
mpiexec python3 Assignment1.py
```

3.2 Results:

In block	C2: 175838 posts B2: 22062 posts C3: 17184 posts B3: 5815 posts C4: 4145 posts B1: 3311 posts C5: 2613 posts D3: 2269 posts D4: 1889 posts C1: 1522 posts B4: 900 posts D5: 717 posts A2: 479 posts A3: 363 posts A1: 262 posts A4: 85 posts
In row	C: 201302 posts B: 32088 posts D: 4875 posts A: 1189 posts
In column	2: 198379 posts 3: 25631 posts 4: 7019 posts 1: 5095 posts 5: 3330 posts

3.3 Execution time graph: