Cluster and Cloud Computing Assignment 1
HPC Twitter GeoProcessing

Kazi Abir Adnan
Student ID: 940406
kadnan@student.unimelb.edu.au

Daniel Gil
Student ID: 905923
glid@student.unimelb.edu.au

1. Introduction

Large-scale data processing is common within the context of social network analysis. The problem is defined to search a large Twitter dataset to identify twitter activity around Melbourne and calculating the most 5 frequent hashtags in a fixed locality. A dataset of localities in Melbourne is given with a range of gridded boxes with the latitude and longitude of the corners of boxes. The task is to show the tweet counts in each box in descending order of grids along with the top 5 hashtags in each of them. These statistics are a representation of Twitter activities around each grid cell. The Twitter dataset is of size around 10 GB and contains 2.5 million tweets. In this report, a solution is given using the parallel programming paradigm Message Passing Model (MPI) in a python application to search the dataset using different configurations of resources in the HPC (High Performance Computer) service *Spartan*.

2. Methodologies

A standard sequential python application was built at first to explore the dataset and test with small size twitter files: *tinyTwitter.json* and *smallTwitter.json*. Given the dimension of *bigTwitter.json*, two methods were implemented to parallelize the parallelizable portion of the code in a distributed environment.

2.2 Method 1: Point to Point Communication - Sending and Receiving with MPI

The first method implemented used basic send and receive calls using MPI. First, a $rank\ 0$ was assigned to a $master\ process$ and an index $i\ \{i \in 0,1,\ldots,k\}$ was given to $slave\ processors$ where k is the total number of cores configured in slurm script and captured by MPI as the *rank* of the processor. Each *ranked* processor will process the tweets to capture hashtags according to the grid configuration and deliver post and hashtag counts. For every $rank\ not\ equal\ to\ 0$, corresponding to $slave\ process$,a signal from the $master$will trigger for them to send the data. In a similar procedure, the *master* will process the tweets and deliver a count, but it will also ask for the other processes to send the data to marshal all counts.

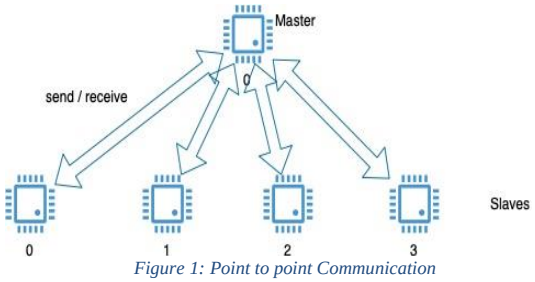

Figure 1: Point to point Communication

2.3. Method 2: Collective Communication - Scatter and Gather with MPI

The communications above involved only the master and a slave process. In this approach we wanted to coordinate all processes in the communication using $rank = 0$ as coordinator and split the work across all possible processes. First, the coordinator $rank\ 0$ reads the file and count all the lines available to distribute between all the processes. An interaction with the operating system is necessary to count the lines in the file optimally. The bash command $wc\ -l$ to get the counts. An algorithm was implemented to split the lines into chunks for every processor available according to the $size$ and lines counted in the input file. The *coordinator (master)* use MPI $comm.scatter$ to distribute tuples of $(< init\ line >, < finish\ line >)$ across nodes and cores.

```
[gild@spartan-login1 data]$ time wc -l bigTwitter.json
2500002 bigTwitter.json
```
Figure 2: Line count using bash command

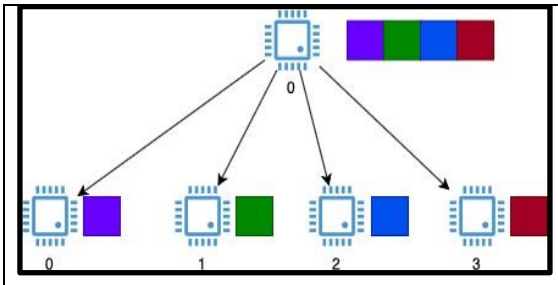

Figure 3: Scatter

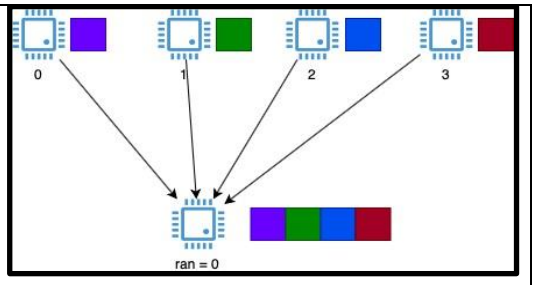

Figure 4: Gather

Each process reads the file and count the tweets for the grid only considering the number of lines distributed from the *coordinator (master)*. The $mpi.gather$ takes all the elements from many processes and gathers into one single process in the *coordinator (master)* $rank = 0$.

3. Execution

3.1 File reading & Data Structure

- First from `melbGrid.json` file we parsed the grid's boundary and created a list of `class Grid`.
- A line by line approach is used instead of loading whole Twitter json file.
- While reading each line, we took the field `['doc']['coordinates']['coordinates']` to get the tweet location and `['doc']['text']` field to get the raw tweet.
- If the location falls into any of the grid, then parsed hashtag from tweet using regex `(?<=\s)#\S+(?=\s)` which explicitly looks for hashtag type '***space#ANYTHINGspac*e'** hashtags.
- If a tweet falls into boundary then we sorted the grids alphabetically which ensures the requirement.
- We tested several data structure to store the information. We started with *dataframe* of python which worked good for small Twitter files. Big Twitter file is big and failed to store information at once, we read file line by line. Finally, we ended up using *dictionary* to store counts for each grid storing hashtag and post counts.

3.2 Job submission

Job Scheduling System Slurm was used to submit jobs into Spartan HPC cluster. A script was configured for each configuration to allow python code execution according to the parameters given using two different approaches. A Slurm script is shown in the next figure to describe the structure followed by every configuration.

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --time=0-00:03:00
#SBATCH --partition=physical
#SBATCH --account=comp90024

# Load required modules
module load Python/3.5.2-goolf-2015a

# Launch multiple process python code
time mpiexec python3 main_p.py bigTwitter.json
echo "2 nodes 8 cores - Point-to-Point Communication"
echo "-----------------------------------------------"
time mpiexec python3 main_sg.py bigTwitter.json
echo "2 nodes 8 cores - Collective Communication"
~
~
"two_nodes_eight_cores.slurm" 16L, 486C
```

Figure 5: Slurm file for two nodes and eight cores

The job was submitted following the procedure to submit a batch script to Slurm:

```
[gild@spartan-login2 src]$ sbatch two_nodes_eight_cores.slurm
Submitted batch job 8041232
```

Figure 6: Job Submission

A separate slurm script was built for each of the configurations: (1) One Node - One Core, (2) One node - Eight Cores and (3) Two Nodes Eight Cores. The scripts execute the same code, but parameters were passed to slurm according to the configuration needed. For example, the parameters for configuration (2):

```
1. #!/bin/bash
2. #SBATCH --nodes=1
3. #SBATCH --ntasks=8
4. #SBATCH --time=0-00:06:00
5. #SBATCH --partition=physical
6. #SBATCH --account=comp90024
```

Finally, to automate the task we built a *runme.sh* which submitted all the slurm scripts to the scheduler:

```
~
"runme.sh" 4L, 112C
#!/bin/bash
sbatch one_node_one_core.slurm
sbatch one_node_eight_cores.slurm
sbatch two_nodes_eight_cores.slurm
```

Figure 7: Shell script to run all configurations

3.3 Python Project structure

```
.
├── README.md
├── data
|   ├── melbGrid.json
|   ├── smallTwitter.json
|   └── tinyTwitter.json
├── output
|   ├── slurm-8090461.out
|   ├── slurm-8090463.out
|   └── slurm-8090465.out
├── slurm
|   ├── one_node_eight_cores.slurm
|   ├── one_node_one_core.slurm
|   ├── runme.sh
|   └── two_nodes_eight_cores.slurm
└── src
    ├── main_p.py
    ├── main_sequential.py
    └── main_sg.py

4 directories, 14 files
```

Figure 8: Project structure

3.4 Run the project

The easiest way to run the whole project is to go to the slurm folder and run the '*$ ./runme.sh*' command in the command line.

4. Evaluation

The next figure summarizes a comparison between the two approaches in real time execution:

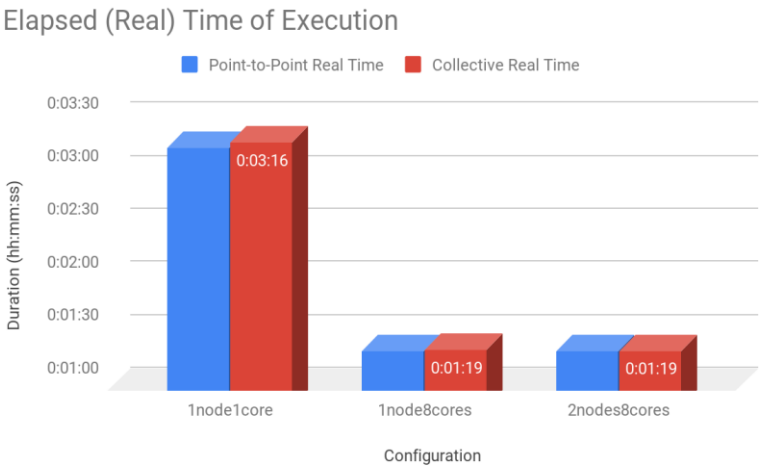

Figure 9: Elapsed time comparing two methods

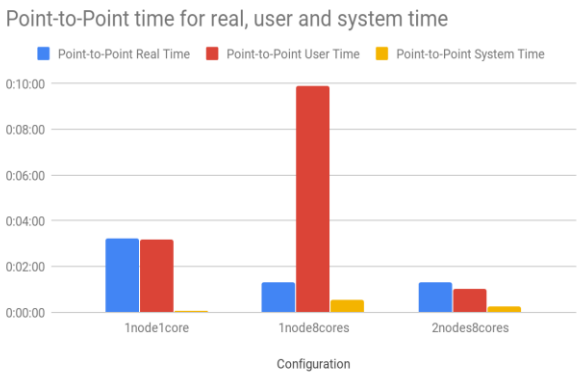

Figure 10: Execution times for Method 1: Point to point communication

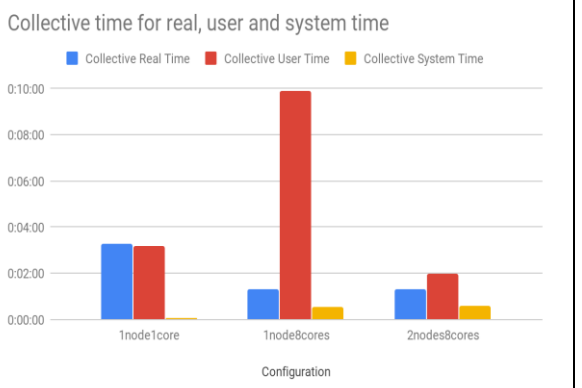

Figure 11: Execution times for Method 2: Collective communication

5. Results

```
C2 : 145096 posts
B2 : 79633 posts
C3 : 53299 posts
B3 : 26550 posts
C4 : 19699 posts
B1 : 16507 posts
D4 : 14497 posts
D3 : 13388 posts
C1 : 8427 posts
B4 : 5450 posts
A3 : 5036 posts
A2 : 4165 posts
C5 : 4106 posts
D5 : 3248 posts
A1 : 2533 posts
A4 : 310 posts
```

```
C2 : [('#melbourne', 10399), ('#australia', 1537), ('#job', 1390), ('#jobs', 964), ('#melbourne…', 505)]
B2 : [('#melbourne', 840), ('#australia', 156), ('#auspol', 149), ('#nashsnewvideo', 123), ('#brunswick', 94)]
C3 : [('#melbourne', 489), ('#australia', 145), ('#coffee', 102), ('#oneorigin_specialtycoffee', 71), ('#food', 70)]
B3 : [('#melbourne', 168), ('#australia', 60), ('#fitness', 55), ('#pocus', 50), ('#health', 42)]
C4 : [('#alishas1million', 96), ('#vote5sos', 85), ('#melbourne', 82), ('#auspol', 75), ('#savedallas', 46)]
B1 : [('#melbourne', 176), ('#kca', 72), ('#australia', 60), ('#vote5sos', 55), ('#bds', 49)]
D4 : [('#nowplaying', 1730), ('#melbourne', 30), ('#dandenong.', 27), ('#job', 26), ('#jobs', 22)]
D3 : [('#melbourne', 80), ('#beach', 60), ('#summer', 29), ('#joesuggtofollowconniekirk', 28), ('#australia', 23)]
C1 : [('#jackfollowme', 104), ('#jackdm', 90), ('#camto4mill', 72), ('#christmasfollowparty', 62), ('#jamesyamm', 58)]
B4 : [('#melbourne', 33), ('#smallzywelcomeshome5sos', 26), ('#fire', 19), ('#vicfires', 19), ('#socialmedia', 18)]
A3 : [('#vicfires', 19), ('#saveconstantine', 12), ('#girls', 11), ('#epping.', 11), ('#love', 8)]
A2 : [('#krazykristmas', 165), ('#egsholidaygiveaway', 57), ('#rc1lmilliongiveaway', 53), ('#vdaywithbeth', 37), ('#craigieburn.', 31)]
C5 : [('#melbourne', 53), ('#australia', 17), ('#dandenongs', 14), ('#travel', 13), ('#1000steps', 12)]
D5 : [('#cat,', 25), ('#somebodytocon', 23), ('#melbourne', 22), ('#lost', 20), ('#voteukvampettes', 18)]
A1 : [('#melbourne', 14), ('#tattoo', 11), ('#worldjudo2014', 9), ('#vobis', 8), ('#drawing', 8)]
A4 : [('#lnp', 4), ('#vicfires', 3), ('#birthday', 3), ('#qldvotes', 3), ('#joy', 2)]
```

6. Key Findings and Discussion

Point-to-Point communication

- One node one core configuration shows the elapsed time as a sum of the time spent in user tasks and system/kernel calls like accessing the file which reflects no parallel tasks at all.
- The final time was improved significantly and cut to more than half. As more cores were working, we have more time spend in user tasks like counting the tweets and go over the lines assigned to the core. In addition, as every core opened the file our system times increased in proportion to the number of cores.
- Splitting the work into two nodes didn't improve the overall time as the total number of cores remain the same. However, the overall time used in user related tasks dropped significantly, also the system calls time went down to half of the time in the previous configuration with one node.
- The total elapsed time remained constant when using eight cores, however, the time for user-related tasks dropped significantly using two nodes.

Collective communication

- The patterns are similar for one node using one or eight cores configuration.
- Time increased in user and systems calls time have doubled for two nodes configuration showing the work being done by two different nodes

7. References

1. L. Dalcin, P. Kler, R. Paz, and A. Cosimo, *Parallel Distributed Computing using Python*, Advances in Water Resources, 34(9):1124-1139, 2011. http://dx.doi.org/10.1016/j.advwatres.2011.04.013
2. L. Dalcin, R. Paz, M. Storti, and J. D'Elia, *MPI for Python: performance improvements and MPI-2 extensions*, Journal of Parallel and Distributed Computing, 68(5):655-662, 2008.http://dx.doi.org/10.1016/j.jpdc.2007.09.005
3. L. Dalcin, R. Paz, and M. Storti, *MPI for Python*, Journal of Parallel and Distributed Computing, 65(9):1108-1115, 2005. http://dx.doi.org/10.1016/j.jpdc.2005.03.010
4. Lev Lafayette, Greg Sauter, Linh Vu, Bernard Meade, "Spartan Performance and Flexibility: An HPC-Cloud Chimera", OpenStack Summit, Barcelona, October 27, 2016. doi.org/10.4225/49/58ead90dceaaa