Report for Project 1

Author: Lu Chen Student Number: 883241

Part 1. How the application is invoked

File: test_big_1.slurm. (Other slurm files can be found under 'slurm_files' directory)

```
#!/bin/bash
#SBATCH -p physical
#SBATCH --time=01:00:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4

module load Python/3.4.3-goolf-2015a
time mpiexec python test_big.py
```

Here is the bash script that I used to submit my application. After running this script, my application will be handed on to the Server Workload Manager Slurm and it will help allocate the reasonable server resource for my job and give me feedback.

The first line invokes a shell environment, which serves as a command interpreter to read from the users' input and give instructions to the Linux kernel. The next four lines starting with #SBATCH are used to request server resources. For example, in my case, I need to specify running on the physical partition since I use Python to write application. You can also specify the maximum walltime by "--time". Therefore, your job will be terminated if its run time reaches this limit and it can help prevent waste of resources. Then, in my request, I tell the job scheduler that my job needs 2 nodes and 4 cores for each node to run parallel tasks. Slurm can help reserve these nodes and cores for my job before it's running.

The remaining part is just the commands that illustrate what operations need to be done when my job starts. In my case, a python module is first loaded into my current environment, which contains mpi4py and openMPI. Then my application is invoked by "mpiexec python" command. Here, I use "time" Linux command to record the time elapsed between invocation and termination of my application. In the output file, you can see three process time statistics – real, user and sys. Among them, 'real' time records actual run time.

sbatch test_big_1.slurm

This command is used to submit my job and my job will be in the queue of physical partition.

Part 2. Approaches I parallelized my code

Step 1: Master node reads melbGrid file and broadcasts ranges of interest

At first, the master node opens melbGrid json file and extracts the range for each box as well as range for whole area. They will be used by all processors to determine the location of coordinate pairs. Master node broadcasts these data.

Step 2: Each processor processes a specific section of Instagram json file.

Each processor reads in the json file line by line. But it only processes the specific lines. Rank 0 handles line 0, rank 1 handles line 1, rank 2 handles line 2……. Only the current line will be loaded into memory. The json object of this line is first parsed to a python dictionary. Then this json object will be checked whether it has the key "coordinates". If so,

it will continue to be checked whether it is inside the whole range of interest using the grid range broadcasted by master node. After that, it will be determined which box it belongs to. At last, a new key “box_id” will be added to record this json object. The dictionary data of this json object will be appended to a ‘data_of_interest’ list.

Step 3: Each processor calculates the sum of posts for each box, row and column
Each processor sums up posts in ‘data_of_interest’ according to key ‘box_id’. For example, if the processor is going to sum up posts in line A, it will count the posts in box_id which starts with ‘A’.

Step 4: Gather post numbers and print out results
Post numbers calculated by all processors will be gathered into master node. And master node will sum them up by each box, row, column, sort the results and print them out.

Part 3. Final results for application’s running on bigInstagram.json file

Number of posts in each box

Box	**C2**	**B2**	**C3**	**B3**	**C4**	**B1**	**C5**	**D3**
Number of Posts	174836	21477	18070	6299	4209	3311	2638	2447
Box	**D4**	**C1**	**B4**	**D5**	**A2**	**A3**	**A1**	**A4**
Number of Posts	1857	1595	1001	783	477	335	262	115

Number of posts in each row

Row	**C**	**B**	**D**	**A**
Number of Posts	201375	32088	5087	1189

Number of posts in each column

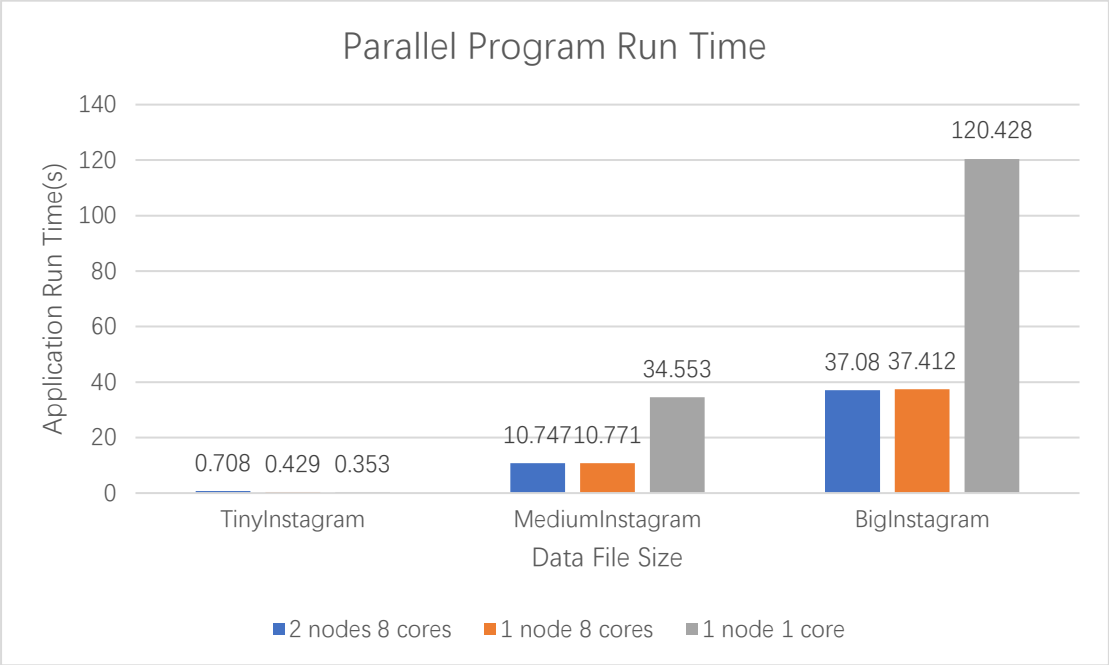
Column	**2**	**3**	**4**	**1**	**5**
Number of Posts	196817	27151	7182	5168	3421

Part 4. Variations in performance

Here is the table of run time for different sizes of json file on different cores and nodes.

Run time(s)	TinyInstagram	MediumInstagram	BigInstagram
2 nodes 8 cores	0.708	10.747	37.08
1 node 8 cores	0.429	10.771	37.412
1 node 1 core	0.353	34.553	120.428

Here is the corresponding bar chart.



We can clearly see:

(1) Run time on 2 nodes 8 cores and 1 node 8 cores is becoming more and more similar when file size grows bigger and bigger. The reason is that essentially, they both run on 8 cores. The slight difference lies in the communication time between cores, which depends on the actual resource location and network conditions.

(2) For medium size and big size files, we can calculate the speed up S(8) for them. We can see they have almost the same speedup ratio on 8 cores.
Medium size file: S(8) = T(1)/T(8) = 34.553 / 10.771 = 3.208
Big size file: S(8) = T(1)/T(8) = 120.428 / 37.412 = 3.219
This result can be explained by Gustafson-Barsis Law: S(N) = α + N(1-α), N stands for n processors, α stands for the fraction of serial run time in each processor.
The speed up S is decided by the serial run time and parallel run time in the program. Therefore, it can be concluded that the proportions of parallelized part running on medium and big sized files are approximately the same.

(3) However, for tiny Instagram file, running on 1 core is even faster than running on 8 cores. It means the proportion of parallel run time in each processor is not the only factor that affects speed up S. We also need to consider the communication delay between cores when they gather, scatter or broadcast information. And for tiny Instagram file, this delay becomes dominant in the whole running process since the whole run time is too tiny. But for files of larger size, this delay becomes less important due to long run time.