

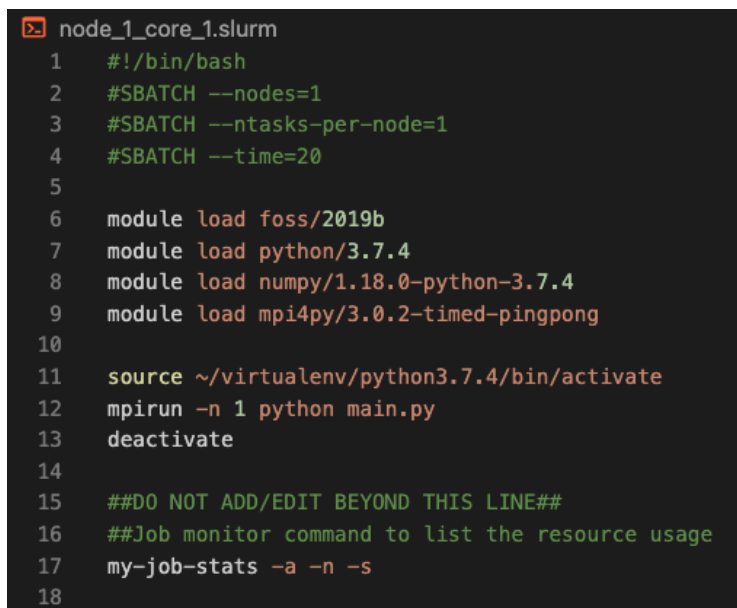
Report Structure

There are three sections in this report – Job Execution, Result Discussion and Appendix.

- Job Execution includes subsections about slurm scripts, tweet processing and application flow.
- Appendix is where the results are. In order to clearly show the pictures, they took 2 pages. Therefore, the report (text) length is under 4 pages.

Job Execution

About slurm scripts



```
node_1_core_1.slurm
1  #!/bin/bash
2  #SBATCH --nodes=1
3  #SBATCH --ntasks-per-node=1
4  #SBATCH --time=20
5
6  module load foss/2019b
7  module load python/3.7.4
8  module load numpy/1.18.0-python-3.7.4
9  module load mpi4py/3.0.2-timed-pingpong
10
11 source ~/virtualenv/python3.7.4/bin/activate
12 mpirun -n 1 python main.py
13 deactivate
14
15 ##DO NOT ADD/EDIT BEYOND THIS LINE##
16 ##Job monitor command to list the resource usage
17 my-job-stats -a -n -s
18
```

The above picture captures the slurm script for submitting the 1 node 1 core job to Spartan with `sbatch` command as instructed. The other two scripts are attached in the zip file. The only difference between the three scripts is line 2 and line 3 on which the node and core numbers are different from job to job.

`ijson` and `pandas` are installed in the virtual environment. `ijson` is not used and the reason is discussed in the later section.

About Processing Tweets

Assumptions and facts about tweets and sal json

The necessary directory structure for the code to run is described below.

1. Tweets json and sal json are stored under a directory named `data`, and `data` should be in the same directory as `main.py`.

2. ``functions.py`` and ``config.py`` should be in the same directory as ``main.py``.

After exploring the format of tweet json (twitter-data-small.json) and sal json, the below assumptions are made, upon which the data filtering are implemented.

Tweet json:

1. The location information is in an array which the key is ``places``. The array contains exactly one object.
2. The location information about suburb is stored using a key called ``full_name``.
 - a. There is at most one comma in the value.
 - b. Before the comma, there is at most one hyphen. In this case, it is treated as two suburbs.
 - c. Possible patterns: `alison`, ``alison, new south wales``, or ``alison, central coast``

Sal json (facts and assumptions):

1. The most complicated pattern of key string is, for example `"alison (central coast - nsw)"`. The `" - "` (space-hyphen-space) and brackets appear at most once in a key to format the corresponding location information.
 - a. Possible patterns (not true, just examples): `alison`, `alison (nsw)`, or `alison (central coast - nsw)`.
2. A key string without specifying additional information with brackets is unique across `sal.json`.
3. Suburbs which are used in multiple states should have information in brackets.

Approach of filtering tweets

The first part of filtering tweets is the conversion of `sal.json` into Python data structure. Each process converts essential information in `sal.json` into a pair of dictionaries. Keys of the dictionaries are tuples.

- The first dictionary stores information of key string and gcc code from `sal.json`.
- The second dictionary does similar things, except that it also appends state information in tuples for those having tuple[1] as ``None``.
 - a. For example, the key tuple of melbourne in one dictionary would be `("melbourne", None)`, another one would be `("melbourne", "victoria")`.
 - b. For those without ``None`` in the tuple, their keys are identical in both dictionaries.

The second part is to convert the location information in tweets into a tuple, which is then used to map values in the sal dictionaries. If the tweet tuple finds a matched key in the first sal dictionary, a corresponding count would be computed. Otherwise, try the second sal dictionary. If still no match, then no capital city count would be computed for the tweet.

The above matching process using two dictionaries is implemented because of the uncertain consistency in tweet location. For example, I am not sure whether a location of a tweet in Melbourne would be shown as `"melbourne"` or `"melbourne, victoria"`, which would result in different key tuples in my code. Therefore, the two-dict matching process is implemented to capture the uncertainty.

Some remarks on the approach

- When generating dictionaries from sal.json, although all processes perform the same operations, it is considered appropriate because sal.json is a small file.
- The two-dict matching would not count ambiguous location from tweets in capital counts. For example, an ambiguous location would have a tuple ("richmond", None). It would not find a match in the first dictionary because all Richmonds should not have None in key tuples based on the third assumption of sal.json. Trivially, no match would be found in the second dictionary because no tuple in the second dictionary has 'None'.
- On the string level, strings in tuples must be identical for a count, which is implied in the approach.

General flow of the application

Below is the description of the application flow. All processes execute the listed actions unless specified.

1. Compute sal dictionaries mentioned in Approach of filtering tweets.
2. Except process 0, a process needs to find where it should start reading the file, and communicate it to the process which has rank 1 below (i.e. rank 2 tells rank 1 about where it starts, rank 1 would know that is where it should stop reading).
 - a. Use the file size and rank to calculate from which byte the process should start reading the file (i.e. For a 8GB file, rank 1 starts at 1GB, rank 7 starts at 7GB).
 - b. Perform a `readline()` at the calculated entry byte to prevent reading corrupted data.
 - c. Keep `readline()` until it finds the first ``_id`` which is a tweet id.
 - d. Communicate the tweet id to the lower rank.
3. All process starts to read the file at the previously calculated entry byte (0 for process 0), until the received tweet id from the rank one above (until no line for the last process). Each process outputs a dictionary with `author_id` as key and count as value. All dictionaries are gathered in process 0, which would sum up counts of authors, and report tables as instructed.
 - a. Searching of the author and location information starts after the process encounters the first tweet id.
 - b. Once the search starts, each process should iteratively find an author id first, then location information with key "full_name". Therefore, every time location information is found, a count would be computed based on the previously mentioned two-dict approach, using the values stored in variables ``author_id`` and ``loc_info``.
 - c. Each count computation returns an array of length 9. The array is added to what has been stored in the value of the corresponding author id key. `Array[0]` encodes the total tweet number of an author, while `array[1:]` encode the total tweet numbers in each capital city (i.e. `array[1]` for 1gsyd, `array[2]` for 2gmel). As 9oter is not a capital city, it is not counted.
 - d. Use MPI gather to gather all count dictionaries to the root process.
 - e. The root process summarizes gathered dictionary and reports results.

Result Discussion

The performance comparison is in the first graph in the Appendix. The results of the three counting tasks are presented in the last three images respectively.

Overall, the speed difference is within the expectation. 1 node 1 core takes the longest (677 seconds), while the two 8-core settings take around 112 seconds. The 300-second 2 nodes setting will be discussed later.

The speed of 8 cores is 1/6 of 1 core, but not 1/8 due to the unparallelizable part of the application. Steps 1, 2, and part of step 3 in application flow do not become faster with more cores. The only parallelised part is in part 3 that each process performs `readline()` within the assigned chunk of the file. Moreover, by solving a simple equation $(677-x)/(112-x) = 8$, where 677 and 112 are the time taken by 1 core and 8 cores respectively, and x as the time for unparallelisable parts, x can be estimated as around 30 seconds. The parallelised part takes around 80 seconds (110-30).

Outline of unparallelisable parts in the application:

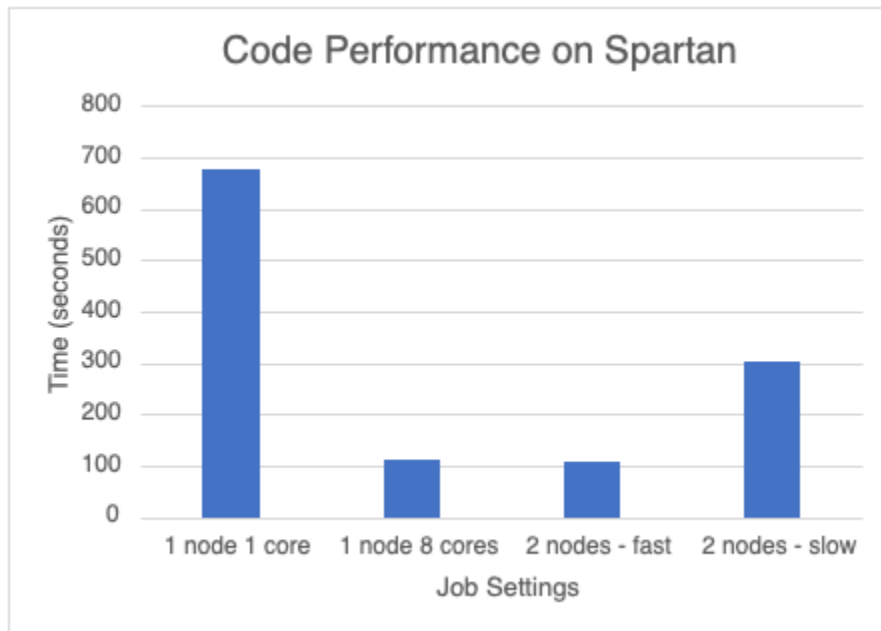
- Step 1: Every process computes sal dictionaries in Step 1. There is no parallelisation.
- Step 2: The inter-process communication is obviously not parallelisable.
- Part of step 3: The information is gathered and summarized in the root process at the end, hence not parallelisable.

However, it is considered that the unparallelisable parts take very little time because a non-successful job on Spartan also takes around 30 seconds to generate a report, which is very close to the estimated time for the unparallelisable parts. This makes sense because the content in unparallelisable parts do not involve time consuming operations. The fact that the two values coincide validates that the application has managed to truly parallelise the time-consuming part, which is parsing json object in the json file one-by-one. Any code that does not parallelise properly would have x significantly higher than 30 seconds.

The speed of parallelised part can be improved. The fundamental step of processing json file in this application is `readline()`. This approach does not utilise the format of json to the fullest. It is obvious when comparing to json parsers such as `ijson` with 1 core, which takes 6 minutes. I did not use a parser because I have not managed to come up with a solution to incorporate json parsers when using byte as a entry point. An alternative is writing a json parser with regex but I consider it out of scope.

Finally, the reason why running 2 nodes gives significantly different results (110 seconds and 300 seconds) is whether the job is actually run on two machines. If it is run on two machines, the interprocess communication would need to go through the ethernet, making the whole process slower. Otherwise, it is literally equivalent to 1 node 8 cores so the two spend the same amount of time. Therefore, it could be estimated that the time for IPC to go through Spartan ethernet is around 200 seconds. Whether it depends on the network condition is beyond the scope.

Appendix



```
1 #####Task 1#####
2 Rank          Author ID  Number of Tweets Made
3   #1 1498063511204761601      68477
4   #2 1089023364973219840      28128
5   #3 826332877457481728      27718
6   #4 1250331934242123776      25350
7   #5 1423662808311287813      21034
8   #6 1183144981252280322      20765
9   #7 1270672820792508417      20503
10  #8 820431428835885059      20063
11  #9 778785859030003712      19403
12  #10 1104295492433764353      18781
13
```

#####Task 2#####

Greater Capital City Number of Tweets Made

	1gsyd	2094861
	2gmel	2252065
	3gbri	851798
	4gade	451382
	5gper	586163
	6ghob	89663
	7gdar	46297
	8acte	193655

#####Task 3#####

Rank	Author ID	Number of Unique City Locations and # Tweets
#1	1429984556451389440	8 (#1872 tweets - #10gsyd, #1832gmel, #6gbri, #2gade, #7gper, #1ghob, #1gdar, #13acte)
#2	17285408	8 (#1164 tweets - #1020gsyd, #57gmel, #40gbri, #3gade, #7gper, #11ghob, #4gdar, #22acte)
#3	702290904460169216	8 (#1050 tweets - #277gsyd, #230gmel, #197gbri, #108gade, #140gper, #36ghob, #18gdar, #44acte)
#4	87188071	8 (#382 tweets - #108gsyd, #80gmel, #64gbri, #28gade, #50gper, #14ghob, #5gdar, #33acte)
#5	1361519083	8 (#259 tweets - #11gsyd, #36gmel, #1gbri, #9gade, #1gper, #2ghob, #193gdar, #6acte)
#6	502381727	8 (#250 tweets - #2gsyd, #214gmel, #8gbri, #4gade, #3gper, #8ghob, #1gdar, #10acte)
#7	921197448885886977	8 (#197 tweets - #46gsyd, #55gmel, #37gbri, #20gade, #28gper, #4ghob, #1gdar, #6acte)
#8	601712763	8 (#146 tweets - #44gsyd, #39gmel, #11gbri, #19gade, #14gper, #8ghob, #1gdar, #10acte)
#9	2647302752	8 (#79 tweets - #13gsyd, #16gmel, #31gbri, #3gade, #4gper, #5ghob, #3gdar, #4acte)
#10	3686036533	7 (#2581 tweets - #401gsyd, #1546gmel, #91gbri, #47gade, #186gper, #84ghob, #0gdar, #226acte)