Assignment 1 report

Group members: Tianhao Liu 1105160 Bohan Su 1174197

1. **Introduction:** This Twitter analysis assignment involves three primary tasks: ranking authors based on their tweet count, ranking cities based on the number of tweets from them, and ranking authors based on a combined metric of unique cities and tweet count. The approach involves loading data using ijson, allocating data to different cores using MPI, creating dictionaries for author-tweets, city-tweets, and author-city tweets, and then ranking authors based on different metrics by gathering the dictionaries from individual cores.

2. **Assumptions:** 1)By assuming that the suburb and state are on predetermined lists, the code only checks whether the tweet's address is in the list(suburbs and state both in the list), rather than searching through all possible options. This helps to reduce the complexity of the search and improve the overall time efficiency of the program. 2)Another assumption to consider is that the time monitoring process only covers the sequential time and parallel time portion of the program and does not include the MPI.gather function.

2. **The interpretation of mpi:** To tackle large-scale problems by leveraging multiple processors, our group employs the Message Passing Interface approach along with faster algorithms for distributing data among cores and collaborating effectively.

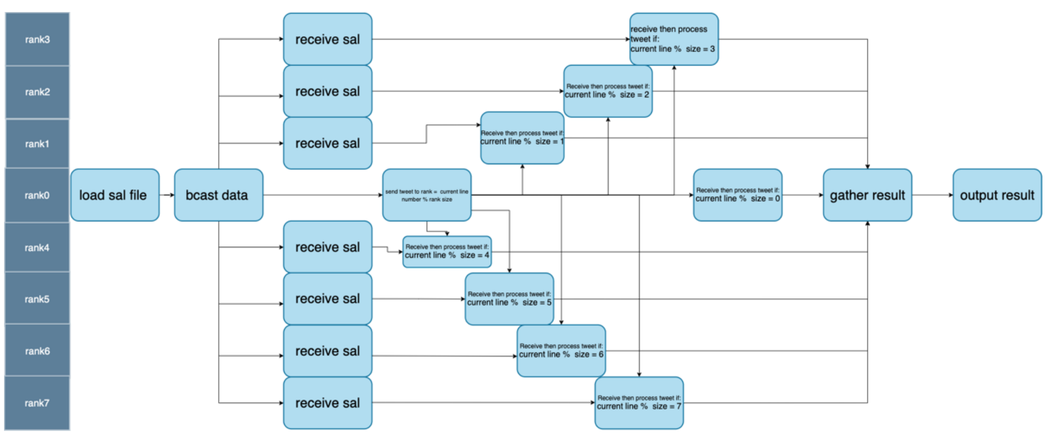


Fig1. MPI pipeline

Algorithm: Our tweet data processing algorithm in Python assigns tweets to different cores based on their index and the size of the core group. Rank 0 loads all the tweet data and uses the formula target_rank = index % size to assign each tweet to a core. This approach involves a sequential run for rank 0 and parallel runs among all other cores, making it efficient in terms of time as only rank 0 iterates through all the data.

3. **The interpretation of ijson:** To avoid high memory usage when loading a large JSON file with a size of 18.74GB, our group used the ijson method. We used rank 0 to parse the data incrementally, generating events as needed to minimize memory usage, as the standard json library reads the entire file or string at once, which can lead to high memory usage.

```python
with open(filename, 'r') as f:
    ts_loop_json_item_start = MPI.Wtime()
    tweets = ijson.items(f, 'item')
    tweet_size = 0
    chunk_size = 0
    send_info_time = 0
    # Reading data in a memory efficient way
    for index, tweet in enumerate(tweets):
```

Fig 2. Basic interpretation of ijson

5. **Submitting a job to SPARTAN:** To submit the job to spartan, specifying the resources and execution parameters(number of core and node), also include the output file if working without error and the error file, if there is something wrong required for the job. With the interpretation below

```bash
#!/bin/bash

#SBATCH --job-name=search_twitter_with_1_node_1_core
#SBATCH --output=job_1_output.txt
#SBATCH --error=job_1_error.txt
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=0-12:00:00

module load python/3.7.4
module load foss/2019b
module load mpi4py/3.0.2-timed-pingpong
source ~/virtualenv/python3.7.4/bin/activate

mpiexec -n 1 python twitterSearch.py bigTwitter.json
my-job-stats -a -n -s
```

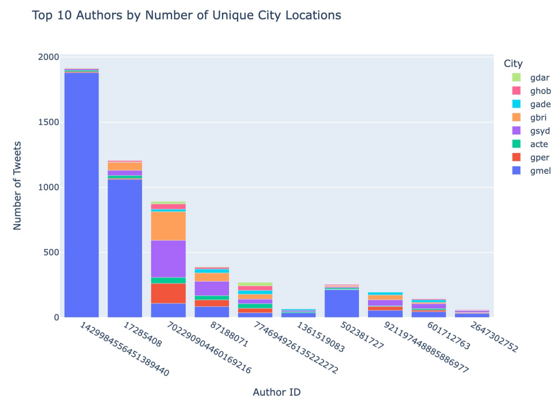
Fig3. Slurm Script

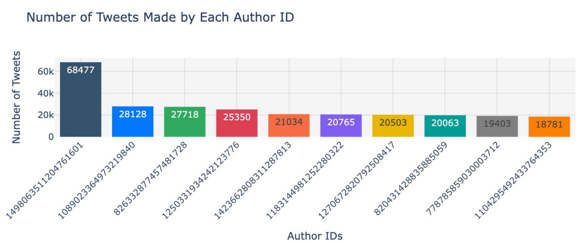
6. **Results**: The graph below shows the results of the three tasks. Author ID 1498063611204761601 has the highest tweet count at 68744, followed by author ID 1089023365973219840 with 28128 tweets. The difference between the top two authors and the rest of the top 10 is substantial. Melbourne and Sydney have significantly more tweets (2272377 and 2116618, respectively) than Brisbane and Perth. The third graph ranks authors based on unique cities and tweet count, with first and second priorities.

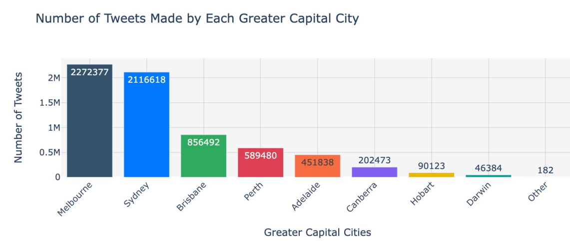
Rank	Author Id	Number of Tweets Made
#1	1498063511204761601	68477
#2	1089023364973219840	28128
#3	826332877457481728	27718
#4	1250331934242123776	25350
#5	1423662808311287813	21034
#6	1183144981252280322	20765
#7	1270672820792508417	20503
#8	820431428835885059	20063
#9	778785859030003712	19403
#10	1104295492433764353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

Greater Capital City	Number of Tweets Made
2gmel	2272377
1gsyd	2116618
3gbri	856492
5gper	589480
4gade	451838
8acte	202473
6ghob	90123
7gdar	46384
9oter	182

Rank	Author Id	Number of Unique City Locations and #Tweets
#1	1429984556451389440	8 (#1920 tweets - 1880gmel, 7gper, 13acte, 10gsyd, 6gbri, 2gade, 1ghob, 1gdar)
#2	17285408	8 (#1209 tweets - 1061gsyd, 40gbri, 23acte, 11ghob, 7gper, 60gmel, 3gade, 4gdar)
#3	702290904460169216	8 (#1120 tweets - 110gade, 286gsyd, 219gbri, 40ghob, 248gmel, 20gdar, 151gper, 46acte)
#4	87188071	8 (#391 tweets - 84gmel, 15ghob, 109gsyd, 64gbri, 51gper, 34acte, 29gade, 5gdar)
#5	774694926135222272	8 (#272 tweets - 37gbri, 36ghob, 37gsyd, 38gmel, 34gper, 34acte, 28gade, 28gdar)
#6	1361519083	8 (#260 tweets - 193gdar, 36gmel, 12gsyd, 2ghob, 6acte, 1gper, 9gade, 1gbri)
#7	502381727	8 (#250 tweets - 214gmel, 8gbri, 8ghob, 4gade, 10acte, 3gper, 1gdar, 2gsyd)
#8	921197448885886977	8 (#199 tweets - 46gsyd, 56gmel, 20gade, 37gbri, 28gper, 4ghob, 7acte, 1gdar)
#9	601712763	8 (#146 tweets - 44gsyd, 39gmel, 14gper, 11gbri, 8ghob, 19gade, 10acte, 1gdar)
#10	2647302752	8 (#80 tweets - 32gbri, 3gade, 13gsyd, 4gper, 16gmel, 3gdar, 5ghob, 4acte)







7. **Performance of different MPI configurations:** The graph Fig.7 compares the performance of different configurations with various numbers of processes. The 1n1c, 1n8c, and 2n8c setups in the MPI framework show discrepancies in their performance. Although the total time taken by the 8-core configurations is not proportional to that of the 1-core configuration, the parallel portion of the 8-core setup is eight times faster than the 1-core configuration. The ijson is memory-efficient in processing large files and consumes less than 200MB of memory.

Configuration	1 node 1 core	1 node 8 core	2 node 8 core
Sequential Time (s)	338.62719	490.4421	499.74832
Parallel Time (mean, s)	1097.50204	140.30304	140.23316
Total Time (s)	1436.31472	633.23257	643.34862
Rank 0 Send Time (s)	0.0	146.14386	157.7002
Memory used(RAM)	153 MB	5MB	5MB
Search Rate (mean, tweets/s)	8284.516719	64804.540229	64836.833171

Fig 7. Performance on different configuration

8. **Analysis on Amdahl’ law and Gustafson-Barsis’ law:**

Configuration	1 node 1 core	1 node 8 core	2 node 8 core
Alpha (α)	0.235792	0.304081	0.308180
Number of Cores (n)	1.000000	8.000000	8.000000
Amdahl Speedup	1.000000	2.557081	2.533845
Gustafson Speedup	1.000000	5.871433	5.842742

Fig 8. Amdahl and Gustafson speedup

Amdahl's Law highlights the limitations of parallelisation in achieving the maximum possible speedup, based on the non-parallelizable portion of the code.

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

$$S(n) = \alpha + n \cdot (1 - \alpha)$$

eq1. Amdahl’s speedup eq2. The Gustafson-Barsis’s speedup

In our case, the Alpha (α) values are relatively small, indicating that the majority of the code can be parallelized. However, the Amdahl Speedup values for the 1 node 8 core and 2 nodes 8 core configurations are 2.55 and 2.53, respectively. These values are considerably lower than the ideal 8x speedup one would expect when using 8 cores. This demonstrates that the non-parallelizable portion of the code imposes a limit on the speedup, as predicted by Amdahl's Law.

Gustafson's Law, on the other hand, presents a more optimistic view of parallelization, as it considers the scenario where the problem size increases with the number of processors.

In this context, the Gustafson Speedup values for the 1 node 8 core and 2 nodes 8 core configurations are 5.87 and 5.84, respectively. Although these values do not reach the ideal 8x speedup, they suggest that as the problem size increases, the system can achieve better scaling with more processors.

9. limitation and Discussion: The study highlights that minimizing the sequential portion of a program and optimizing communication overhead are crucial for fully realizing the benefits of parallel programming. The use of ijson to parse the input JSON data presents a limitation to the program's parallelization potential because the portion of the program that reads the JSON data cannot be parallelized. This constraint contributes to the overall sequential portion of the program, limiting the potential for speed-up.

Future work should focus on improving the searching algorithm or reducing the sequential portion of the program. By optimizing these components, the program's parallel efficiency can be further enhanced, leading to greater speed-up gains in a high-performance computing environment.