Cluster and Cloud Computing

Assignment I Report

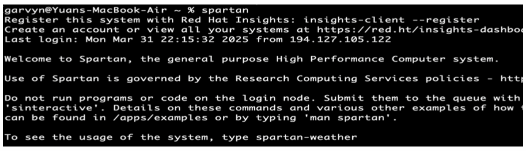
Authors

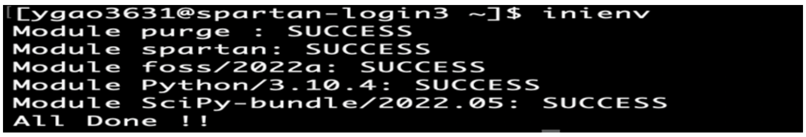
yuan.gao.2@student.unimelb.edu.au

yzhao5067@student.unimelb.edu.au

Login and Initialization

1. use **ssh-key** to login without password, copy public key to the remote host and set ssh config file and alias command locally to login by simply command 'spartan'.
2. Create shell scripts initial_env.sh and set alias command 'alias inienv='source /home/ygao3631/initial_env.sh' in ~/.bashrc 'to **conveniently initialize each time**, The function includes module purge and Load spartan & foss/2022a & Python/3.10.4 & Scipy





The slurm scripts for submitting the job

The final version:

```
#!/bin/bash
#SBATCH --job-name=mastodon_analysis
#SBATCH --output=mastodon_analysis_%j.out
#SBATCH --error=mastodon_analysis_%j.err
#SBATCH --nodes=[node_amount]
#SBATCH --ntasks-per-node=[core_amount]
#SBATCH --time=04:00:00
#SBATCH --mem=[memory per-core]

mpiexec -n 8 python mastodon_analysis.py
# or sun -n 8 python mastodon_analysis.py
```

For different Jobs, edit '--nodes='; '--ntasks-per-node=' to adjust the different circumstances.

- **2 nodes ,4 cores for each:** set '--nodes=2'; '--ntasks-per-node=4'.
- **1 node, 8 cores for each:** set '--nodes=1'; '--ntasks-per-node=8'.
- **1 node ,1 core for each:** set '--nodes=1' ; '--ntasks-per-node=1'.

```
#!/bin/bash
#SBATCH --job-name=mastodon_analysis
#SBATCH --output=mastodon_analysis_11%j.out
#SBATCH --error=mastodon_analysis_11%j.err
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=04:00:00
#SBATCH --mem=8G

mpiexec -n 1 python mastodon_analysis.py
```

```
#!/bin/bash
#SBATCH --job-name=mastodon_analysis
#SBATCH --output=mastodon_analysis_%j.out
#SBATCH --error=mastodon_analysis_%j.err
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=04:00:00
#SBATCH --mem=8G

mpiexec -n 8 python mastodon_analysis.py
```

```
#!/bin/bash
#SBATCH --job-name=mastodon_analysis
#SBATCH --output=mastodon_analysis_%j.out
#SBATCH --error=mastodon_analysis_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --time=04:00:00
#SBATCH --mem=8G

mpiexec -n 8 python mastodon_analysis.py
```

Approach to build and parallelize the code

From stream-based to File-slicing-based MPI design

- **Stream-based MPI design:** The first try uses send and recv to stream data from rank 0 to other processes. While it gives more control over memory and avoids partial line handling, it's slower because rank 0 reads the entire file alone and becomes the bottleneck for both I/O and communication. This approach runs well on 16m data but take much more time than expected on 144G data: The second version is less efficient might because rank 0 handles all the file reading, which creates a severe I/O bottleneck. Other ranks must wait for data to be sent, causing idle time. Additionally, the frequent use of send and recv introduces high communication overhead, especially when processing large files like 144GB. This serialized workflow limits parallelism and slows down the entire program.
- **File-slicing-based MPI design:** The key optimization is replacing centralized streaming with parallel file reading. Each process reads and processes data independently, and only the final results are gathered, making the whole program much faster and more scalable. The second version lets each process read its own part of the file directly using seek, so all ranks process data in parallel. This greatly improves speed by removing the central bottleneck and reducing communication overhead.
- **Optimization:** Replaced comm.send/recv with direct file slicing using f.seek() ; Let each process handle its own I/O and parsing logic ; Only final aggregation is done on Rank 0 using comm.gather().

```python
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    filename = 'large-144G.ndjson'
    chunk_size = 1024 * 1024 * 100  # 100MB

    hour_sentiment = defaultdict(float)
    user_sentiment = defaultdict(float)
```

```python
    with open(filename, 'r', encoding='utf-8') as f:
        f.seek(0, 2)
        file_size = f.tell()
        chunk_size = file_size // size
        start = rank * chunk_size
        end = start + chunk_size if rank != size -1 else file_size

        f.seek(start)
        if rank != 0:
            f.readline()

        pos = f.tell()
        while pos < end:
            line = f.readline()
            if not line:
                break
            pos = f.tell()
            created_at, sentiment, user_id, username = parse_line(line)
            if created_at and sentiment is not None and user_id and username:
                try:
                    dt = datetime.fromisoformat(created_at.replace('Z', '+00:00'))
                    hour = dt.strftime('%Y-%m-%d %H:00')
                    hour_sentiment[hour] += sentiment
                    user_sentiment[username] += sentiment
                except ValueError:
                    continue
```

- **Fault tolerance Design:** The code include fault tolerance to deal with the dirty data(e.g. malformed JSON, missing fields) with out crashing

```python
def parse_line(line):
    try:
        data = json.loads(line)  # May raise JSONDecodeError
        doc = data.get('doc', {})  # Prevents KeyError
        created_at = doc.get('createdAt', None)  # Returns None if missing
        sentiment = doc.get('sentiment', None)
        account = doc.get('account', {})
        user_id = account.get('id', None)
        username = account.get('username', None)
        return created_at, sentiment, user_id, username
    except json.JSONDecodeError:
        return None, None, None, None  # Marks invalid records
```

- **Collect Results in rank 0**

```python
    all_hour_sentiment = comm.gather(hour_sentiment, root=0)
    all_user_sentiment = comm.gather(user_sentiment, root=0)

    if rank == 0:
        combined_hour = defaultdict(float)
        combined_user = defaultdict(float)
```

```
        for hs in all_hour_sentiment:
            for hour, sentiment in hs.items():
                combined_hour[hour] += sentiment

        for us in all_user_sentiment:
            for user, sentiment in us.items():
                combined_user[user] += sentiment
```

Performance Analysis

For 144G dataset, 1 node 8 cores task has the best performance which takes 256 seconds and 2 nodes 4 cores task takes 267 seconds, 1 node 1 core has the worst performance which is 2597 seconds. In each task, the run time on each core is almost the same and the aggregation time is less than 1 second. The reason for 1 node 8 cores job having the best performance is all 8 cores are on the same machine, so there's no inter-node communication. For 2 nodes 4 cores, (even though it's minimal due to small aggregation time). I/O bandwidth per node might also be lower if shared file system access is not as efficient across nodes. For 1node 1 core, No parallelism at all. One core processes the entire 144GB dataset sequentially, becoming the bottleneck. I/O and CPU are underutilized, leading to extremely long execution time. Finally, the data merging task is lightweight tasks and always completes quickly.

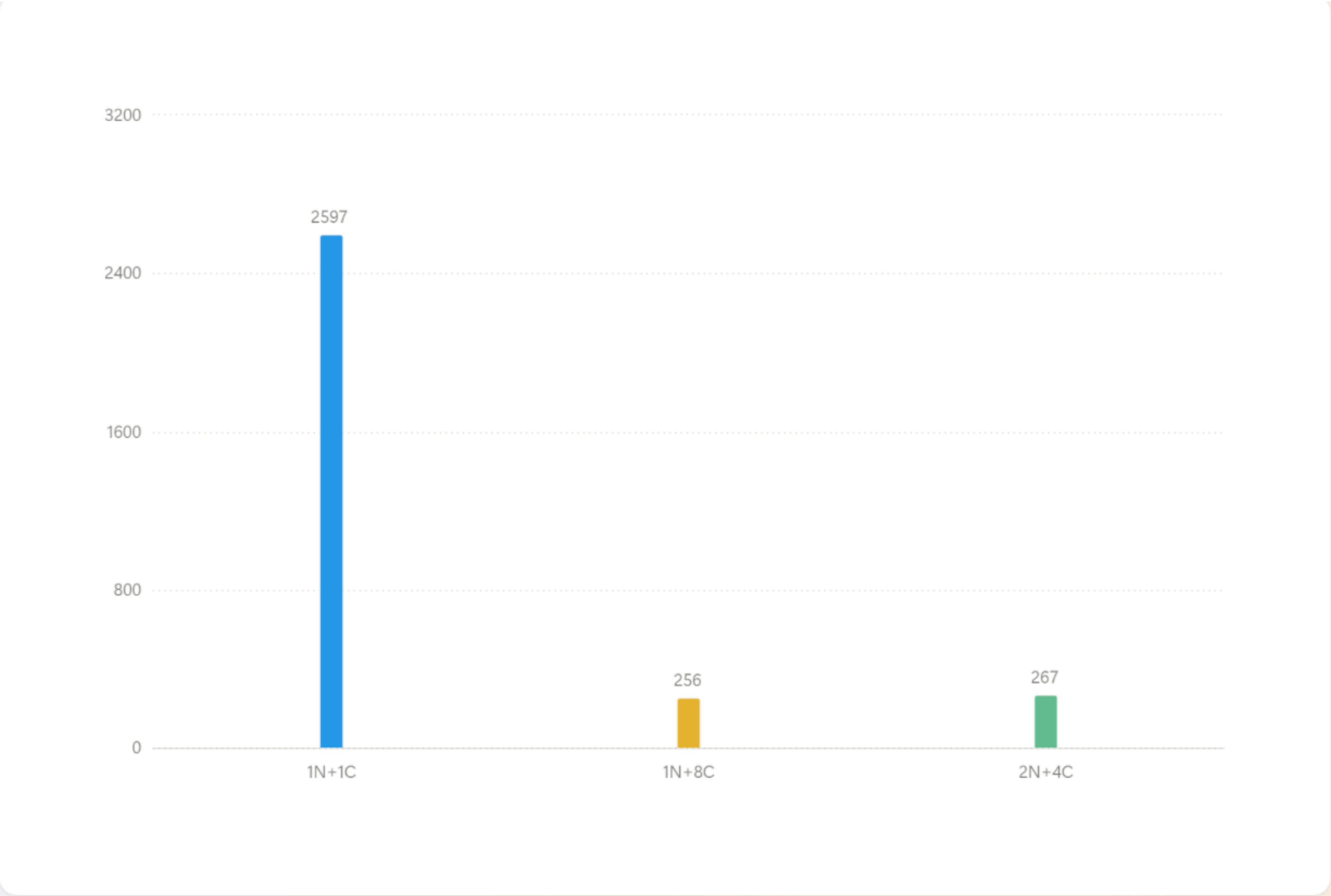
So, The key performance factors are parallelism level, I/O bandwidth, and communication overhead. Maximum performance is achieved when all cores are on the same node and can work in parallel with minimal data transfer delay. The pictures below shows the results for [1-1] [2-4] [1-8] version in order respectively.

```
Rank 0: Processing time: 2596.89 seconds

5 Happiest Hours:
2025-01-01 00:00 with sentiment score 206.15209972304362
2024-12-31 23:00 with sentiment score 187.46547663220213
2025-01-01 05:00 with sentiment score 135.92521743446807
2024-12-24 22:00 with sentiment score 117.28631361201234
2024-12-25 15:00 with sentiment score 114.60195453225161

5 Saddest Hours:
2024-11-06 07:00 with sentiment score -373.7672479738356
2024-09-11 01:00 with sentiment score -305.57606693741
2025-01-30 17:00 with sentiment score -256.5684250419835
2025-01-30 18:00 with sentiment score -226.90971970072636
2025-02-03 16:00 with sentiment score -223.8334876843535

5 Happiest Users:
gameoflife with sentiment score 9105.741942204384
EmojiAquarium with sentiment score 2605.0192646105224
TheFigen_ with sentiment score 2541.4742243891897
choochoo with sentiment score 1978.7847677692084
hnbot with sentiment score 1914.9517912924869

5 Saddest Users:
realTuckFrumper with sentiment score -9093.795561168281
uavideos with sentiment score -5901.960347558229
TheHindu with sentiment score -5710.722600659062
uutisbot with sentiment score -4066.1381701402247
MissingYou with sentiment score -3341.603338119594

Data aggregation time: 0.37 seconds
Total execution time: 2597.42 seconds
```

```
Rank 1: Processing time: 247.96 seconds
Rank 3: Processing time: 250.94 seconds
Rank 0: Processing time: 251.00 seconds
Rank 2: Processing time: 254.91 seconds
Rank 4: Processing time: 261.75 seconds
Rank 6: Processing time: 261.88 seconds
Rank 7: Processing time: 263.95 seconds
Rank 5: Processing time: 266.75 seconds

5 Happiest Hours:
2025-01-01 00:00 with sentiment score 206.15209972304456
2024-12-31 23:00 with sentiment score 187.465476632202
2025-01-01 05:00 with sentiment score 135.92521743446838
2024-12-24 22:00 with sentiment score 117.2863136120121
2024-12-25 15:00 with sentiment score 114.60195453225167

5 Saddest Hours:
2024-11-06 07:00 with sentiment score -373.76724797383497
2024-09-11 01:00 with sentiment score -305.5760669374102
2025-01-30 17:00 with sentiment score -256.5684250419817
2025-01-30 18:00 with sentiment score -226.90971970072644
2025-02-03 16:00 with sentiment score -223.83348768435087

5 Happiest Users:
gameoflife with sentiment score 9105.741942204433
EmojiAquarium with sentiment score 2605.0192646105183
TheFigen_ with sentiment score 2541.4742243891906
choochoo with sentiment score 1978.7847677692096
hnbot with sentiment score 1914.9517912927743

5 Saddest Users:
realTuckFrumper with sentiment score -9093.795561168006
uavideos with sentiment score -5901.960347560852
TheHindu with sentiment score -5710.722600655697
uutisbot with sentiment score -4066.138170138108
MissingYou with sentiment score -3341.603338119541

Data aggregation time: 0.69 seconds
Total execution time: 267.79 seconds
```

```
Rank 2: Processing time: 252.50 seconds
Rank 5: Processing time: 252.56 seconds
Rank 6: Processing time: 252.67 seconds
Rank 0: Processing time: 254.38 seconds
Rank 7: Processing time: 254.97 seconds

5 Happiest Hours:
2025-01-01 00:00 with sentiment score 206.15209972304456
2024-12-31 23:00 with sentiment score 187.465476632202
2025-01-01 05:00 with sentiment score 135.92521743446838
2024-12-24 22:00 with sentiment score 117.2863136120121
2024-12-25 15:00 with sentiment score 114.60195453225167

5 Saddest Hours:
2024-11-06 07:00 with sentiment score −373.76724797383497
2024-09-11 01:00 with sentiment score −305.5760669374102
2025-01-30 17:00 with sentiment score −256.5684250419817
2025-01-30 18:00 with sentiment score −226.90971970072644
2025-02-03 16:00 with sentiment score −223.83348768435087

5 Happiest Users:
gameoflife with sentiment score 9105.741942204433
EmojiAquarium with sentiment score 2605.0192646105183
TheFigen_ with sentiment score 2541.4742243891906
choochoo with sentiment score 1978.7847677692096
hnbot with sentiment score 1914.9517912927743

5 Saddest Users:
realTuckFrumper with sentiment score −9093.795561168006
uavideos with sentiment score −5901.960347560852
TheHindu with sentiment score −5710.722600655697
uutisbot with sentiment score −4066.138170138108
MissingYou with sentiment score −3341.603338119541

Data aggregation time: 0.78 seconds
Total execution time: 256.00 seconds
```

- **Diagram of Results**



Amdahl'S Law Application

Observed speedup ≈ 2597 / 256 ≈ **10.14x ; 2597 / 267** ≈ 9.72x **then** Using Amdahl's Law to estimate parallel portion P. One node eight cores & Two nodes four cores:

$$10.14 = 1/((1-p) + (p/8)) \rightarrow P \approx 0.995 \tag{1}$$

for 2 nodes, Still benefits from high parallelism (~99.5%), but performance loss comes from cross-node communication overhead and shared file system I/O contention.

Mode	Time	Speed Up	Performance
1node,1core	2597	1x	No parallelism, CPU and I/O are bottlenecks
1node,8cores	256	10.14x	Near-optimal scaling, minimal communication
2node,4cores	267	9.7x	Effective parallelism, slight inter-node cost

❗ **Tips** -- More information about *CI/CD* process can be seen our Github website https://github.-com/GarvynY/Cluster_parallel_computing_spartan.git