CCC report

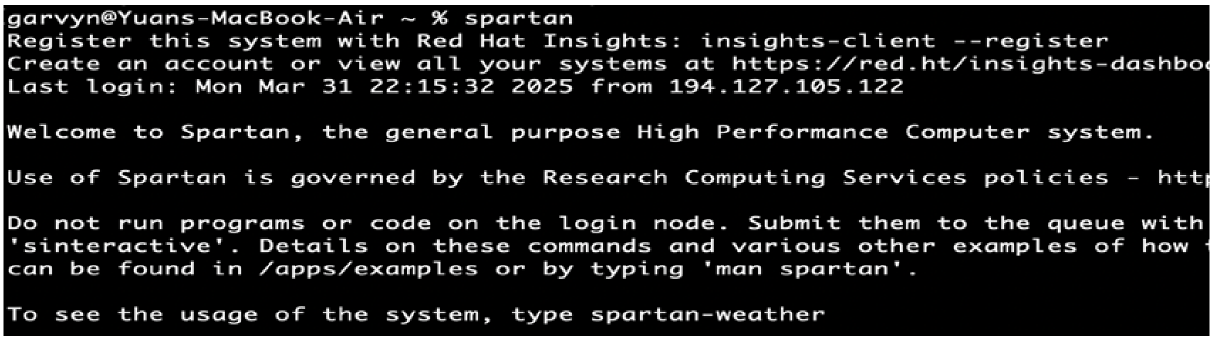
Author:

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

mailto:yzhao5067@student.unimelb.edu.au

Login and Initilization

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



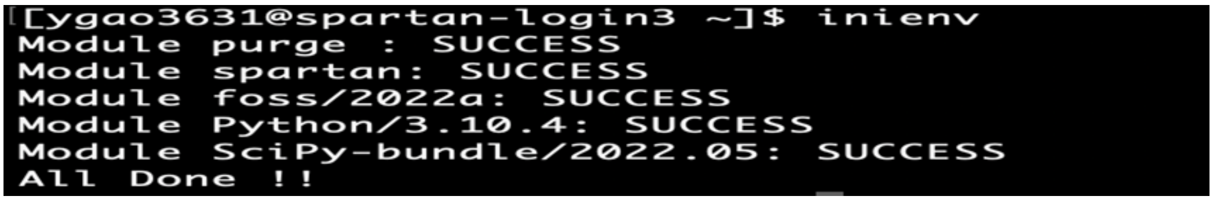
```
garvyn@Yuans-MacBook-Air ~ % spartan
Register this system with Red Hat Insights: insights-client --register
Create an account or view all your systems at https://red.ht/insights-dashbo
Last login: Mon Mar 31 22:15:32 2025 from 194.127.105.122

Welcome to Spartan, the general purpose High Performance Computer system.

Use of Spartan is governed by the Research Computing Services policies - htt

Do not run programs or code on the login node. Submit them to the queue with
'sinteractive'. Details on these commands and various other examples of how
can be found in /apps/examples or by typing 'man spartan'.

To see the usage of the system, type spartan-weather
```



```
[[ygao3631@spartan-login3 ~]$ inienv
 Module purge : SUCCESS
 Module spartan: SUCCESS
 Module foss/2022a: SUCCESS
 Module Python/3.10.4: SUCCESS
 Module SciPy-bundle/2022.05: SUCCESS
 All Done !!
```

The slurm scripts for submitting the job

The final version:

```
#!/bin/bash
#SBATCH --job-name=mastodon_analysis
#SBATCH --output=mastodon_analysis.out
#SBATCH --error=mastodon_analysis.err
```

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=04:00:00
#SBATCH --mem=8G

mpiexec -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'.

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 keep running 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().

```
with open(filename, 'r', encoding='utf-8') as f:
    # Calculate total file size and byte ranges for each process
    f.seek(0, 2)  # Move to end of file
    file_size = f.tell()
    chunk_size = file_size // size  # Divide bytes equally
    start = rank * chunk_size
    end = start + chunk_size if rank != size -1 else file_size  # Last process h
andles remaining bytes

    f.seek(start)
    # Non-root processes skip potentially truncated lines
    if rank != 0:
        f.readline()  # Key: Align to the start of a complete line
```

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

```
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
```

- **Conclusion:** Our parallelization strategy effectively leverages HPC resources through **data partitioning + local aggregation + global merging**. Key strengths:

1. **Coarse-Grained Partitioning**: Reduces inter-process communication.
2. **Dictionary Merging**: Lowers memory pressure on the root process.

3. **Fault Tolerance**: Ensures long-running stability.

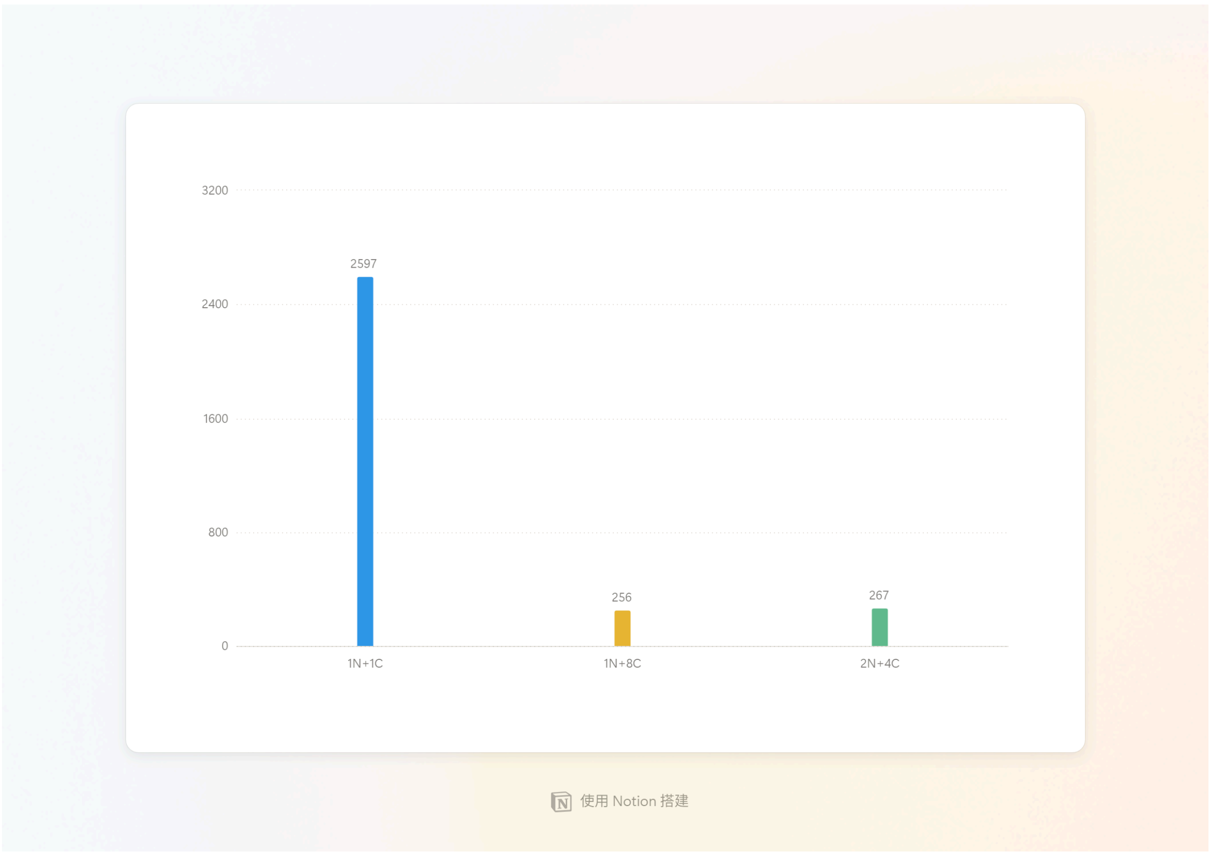
For further optimization, consider the I/O and load-balancing suggestions above.

Performance Analysis:

For 16m dataset, the execuation time is approximately the same, which is less than 1 second. The main reason is that the dataset is small enough that neither I/O nor parallel processing overhead has much impact. Even with a single core, the task completes very fast, and communication/coordination time is negligible.

However, 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.



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:

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

for 2 nodes, Still benefits from high parallelism (~99.5%), but performance loss comes from sross-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