COMP90024 Assignment 1 Report

Yaotian Wang 1503936

Zhifei Cao 1347666

**Start the Program**

1 node 1 core

#!/bin/bash

#SBATCH --nodes=1

#SBATCH --ntasks-per-node=1

#SBATCH --ntasks=1

#SBATCH --time=0-02:00:00

module load mpi4py/3.1.4

module load Python/3.10.4

time mpiexec -n 1 python3 A1.py

my-job-stats -a -n -s

1 node 8 cores

#!/bin/bash

#SBATCH --nodes=1

#SBATCH --ntasks-per-node=8

#SBATCH --ntasks=8

#SBATCH --time=0-00:30:00

module load mpi4py/3.1.4

module load Python/3.10.4

time mpiexec -n 8 python3 A1.py

my-job-stats -a -n -s

2 nodes 8 cores

#!/bin/bash

#SBATCH --nodes=2

#SBATCH --ntasks-per-node=4

#SBATCH --ntasks=8

#SBATCH --time=0-00:30:00

module load mpi4py/3.1.4

module load Python/3.10.4

time mpiexec -n 8 python3 A1.py

my-job-stats -a -n -s

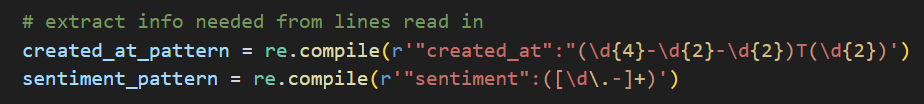
We have three simple scripts, all of them only specify node number, core number and wall time. We also time the process for further analysis. You can find all scripts under the script folder.

You can find the source code of this assignment in the file A1.py. To start this program in Spartan, run `sbatch {name of your chosen script}`.

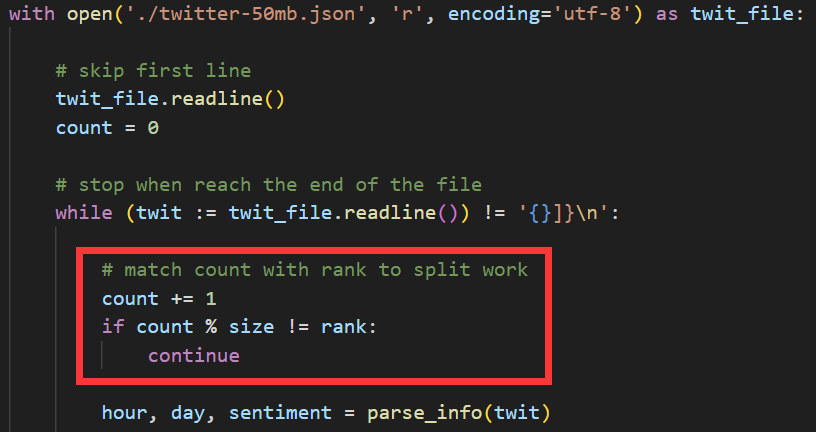
**Implementation approaches**

Our group incorporates a standard structure provided by mpi4py package for initializing MPI communication and uses built-in method gather() to achieve communication and collection of results, simplifying the implementation of data aggregation across multiple processes. In terms of data processing and storage, we use the counter collection for statistics related to activity levels, and the defaultdict collection for statistics related to sentiment to accelerate the processing speed.

Additionally, although the provided data is in JSON format, considering that the information we require occupies only a small part of each line of JSON string, we finally chose to use regular expression matching as a method to quickly parse the necessary information.



For parallelized part, our group has implemented two approaches. The first one is straightforward and intuitive, involves each core continuously reads lines from the file, assigning a number to each line read in, and then takes turns to process a line of information in a round-robin fashion based on a modulo operation.



Although this approach is accurate, with each core only needs to parse the lines assigned to it, it still presents bottleneck in I/O part, since each core still needs to read the entire file line by line. Consequently, the improvement from multi-core parallelism was not as significant as expected.

Therefore, we subsequently implemented a second approach. In the second approach, our team approximately divided the file into segments with equal byte size for each core and used the file.seek() method with the assigned pointer offset to directly jump to the corresponding section. This allowed each core to only read its allocated portion, reducing unnecessary I/O operations and thereby enhancing the speedup ratio.

total\_bytes = os.path.getsize('./twitter-100gb.json')

each\_bytes = math.ceil(total\_bytes / size)

begin\_position = rank \* each\_bytes

end\_position = (rank + 1) \* each\_bytes

with open('./twitter-100gb.json', 'r', encoding='utf-8') as twit\_file:

twit\_file.seek(begin\_position)

if twit\_file.tell() >= end\_position:

break

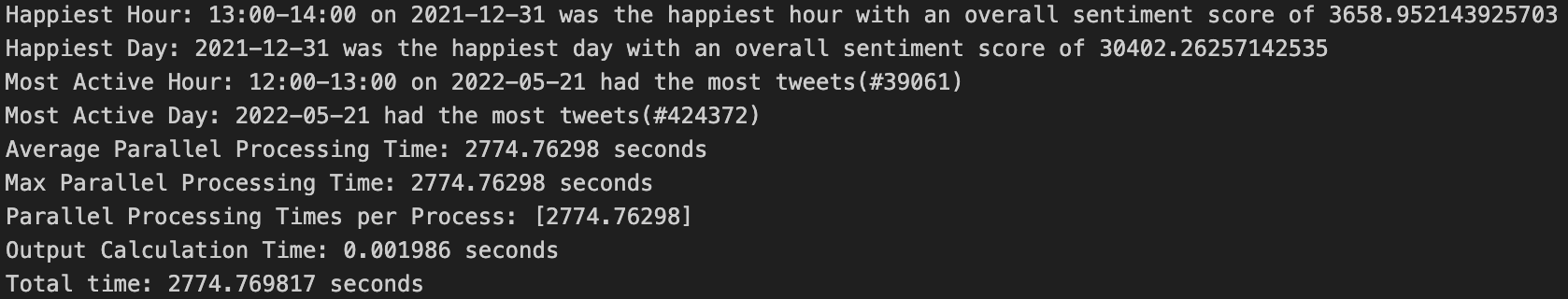
Although this approach is quicker, it still shows the evidence of I/O bottleneck due to the use of the tell() function. Since the main focus of this assignment is not on optimizing the performance, we decide to use this approach.

As you may noticed, in our approach, every core can start to work straightaway, they do not need to wait for the command from the master node. Moreover, the only part of the code cannot be parallelized is the data collection and result display. Hence, our application should have a very strong multi-core performance.

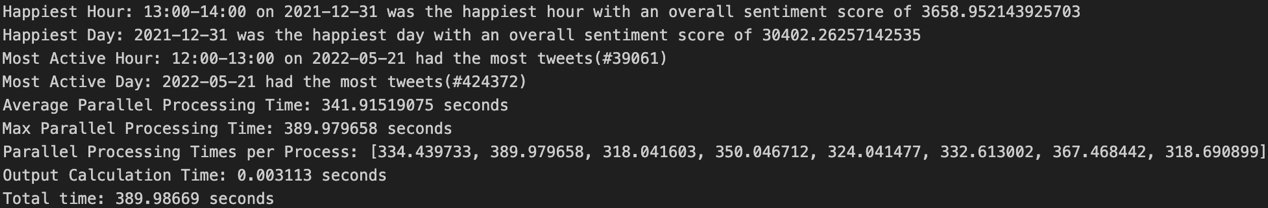
**Result**

You can find the raw results in the outputs folder.

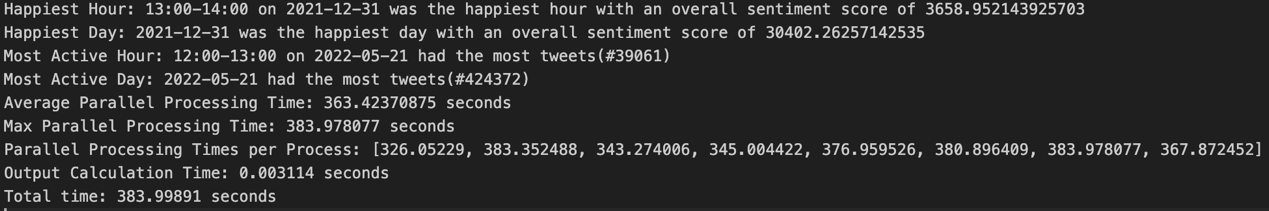
1 core 1 node



1 node 8 cores



2 node 8 cores



All of our configurations give the same result, the only difference is the time taking to get the result. This suggests we did not lose any information when using multi-core.

The chart above displays the results of our project, showing that our team achieved an approximate speedup ratio of 7.1 through parallelization with 8 cores, which is very close to the theoretical maximum of 8 calculated by Amdahl's Law. Besides, when comparing the speedup ratio achieved on a 50MB file processed locally, we observed a higher speedup ratio as we scaled up the problem size. This observation aligns with Gustafson-Barsis’s Law. However, this parallelization also introduced additional overhead, namely the time consumed in the communication part.

For optimizing the communication time, our group considered that there might be more suitable methods than comm.gather(), such as comm.reduce() or hierarchical merging. However, considering that reduce() method is not compatible with our use of dictionary collections and the performance of merging was not ideal, coupled with the fact that the time for waiting for the slowest core is also very substantial, we ultimately did not change this part of the code.

Within the provided environment, we find out that the 2 node 8 cores configuration outperforms the 1 node 8 cores configuration. If we look closer at the output, we can find out that the core ranked 1 in the 1 node 8 cores configuration takes 390 seconds to finish its job which is dramatically greater than the average processing time of 342 seconds which directly leads the 1 node 8 cores configuration need more time to finish the job. We believe this is just a coincidence instead of a trend. But we cannot test our thought due to Spartan is a shared facility. From this result, we believe our program will not benefit from cores in a single node. We suspect this is because our program will only transfer data once at the end of the process and the data is quite small, so the slightly longer transmission time is negligible.