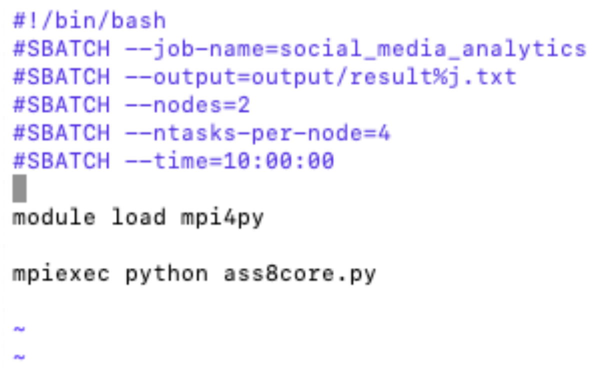
Parallelized Twitter Analytics on University of Melbourne HPC Facility SPARTAN

Gaoyuan Hao Yulin Dong

The aim of this assignment is to implement a parallelized application on SPARTAN for processing a large Twitter dataset to identify the happiest and most active hours and days. During the application, the run time for each different node and core combination will be recorded and further analysed later in this report. This report will include the script we use for submitting the job to SPARTAN, the MPI approach we took to parallelize our code, the logic of processing the JSON file, the output, and the performance on different numbers of nodes and cores.

Methodology

For parallelizing the code, the approach we use is attached below



```
#!/bin/bash
#SBATCH --job-name=social_media_analytics
#SBATCH --output=output/result%j.txt
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=10:00:00

module load mpi4py

mpiexec python ass8core.py

~
~
```

We basically define a SLURM job named “social_meida_analytics” and setting the number of nodes of cores we use by assigning different “nodes” and “ntasks-per-node” (4 core for each node and 2 nodes in total in this sample). We load the necessary MPI module (mpi4py) to enable parallel processing within the python environment.

Then, the python script “ass8core.py” will be executed by using mpiexec, which helps the parallel execution of the script across the nodes and tasks we set above. Based on this approach, we can run parallelized analytics on the Twitter dataset and utilize HPC resources efficiently to handle the computational workload.

For dividing the data, in our design, the dataset is divided among different MPI process (ranks) based on the number of available MPI processes and the rank of each process. Each MPI process reads a specific portion of the JSON file and run the analyse function on its allocated portion. Therefore, MPI process will operate independently on its portion of the data, minimize communication overhead, and enable faster processing of the dataset.

However, the 100gb data is too big to read all in memory. We cannot use the JSON index split data directly, which will cause the memory overflow. So, we decide to use seek() function to jump to the specific byte position which calculated by overall file byte size/number of available MPI process. This method reads a specific location directly from memory without traversing the entire file to get the location. But the program cannot make sure the seek position is the begin of one row. So we worked out a way to locate the JSON file position. Since all ROW data in a JSON file

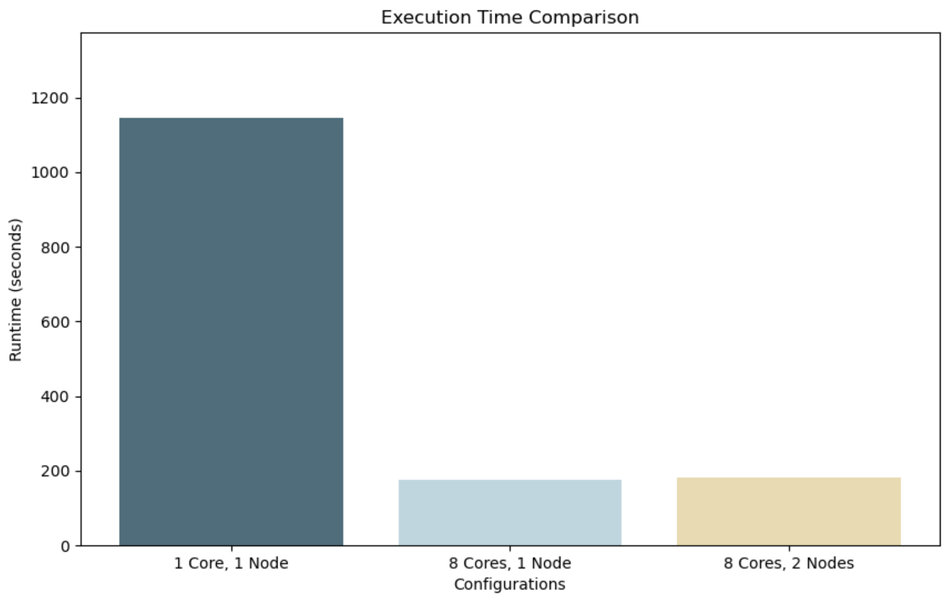
is a complete JSON structure data and is accessed on a row-by-row basis, We can skip this incomplete line directly through json.readline() function and get a complete json data instead. We use this way to read different part of the json file independently without divide the file and avoid preprocessing. Thus, we can parallelize the computation by having each core process only a small part of the file.
While the dataset was processed, for those tweets without sentiment scores or with a sentiment that is not a number (like a list or dict), we decide to ignore them. They will be added up in hours and the result will be stored in a dictionary, including both sentiment score and number of tweets in each hour. After each core completes its computation, it will send the dictionary back to the root process, which is rank 0 using MPI communication. The root will then process and identify the happiest hour, happiest day, most active hour, and most active day based on the combined dictionary. And the runtime will be count during the entire process.

Results

	Data/Hour	**Score**
Happiest Hour	2021-12-31T13	3658.95
Happiest Day	2021-12-31	30402.26
Most Active Hour	2022-05-21T12	39061
Most Active Day	2022-05-21	424372

According to the results of the analysis listed above, the happiest hour, day and most active hour and day has been detected. The corresponding sentiment score and number of tweets has been listed above.

Performance Analysis



Here is the bar chart showing the performance (measured by the execution time) of each different core and node configurations (1 core, 1 node; 8 core, 1 node; 8 core 2 nodes):
1 Core, 1 Node: Total runtime: 1145.30 seconds

8 Cores, 1 Node: Total runtime: 175.65 seconds
8 Cores, 2 Nodes: Total runtime: 181.23 seconds

Comparing 1 Core, 1 Node with 8 Cores, 1 Node, $speedup = \frac{1145.30}{175.65} = 6.52$

Comparing 1 Core, 1 Node with 8 Cores, 2 Nodes, $speedup = \frac{1145.30}{181.23} = 6.32$

Based on our design, the parallelizable portions of the code should be the reading and processing individual lines of the JSON file and calculating the sentiment scores and number of tweets for each hour. And the non-parallelizable portions should be merging hourly sums from different MPI processes into a single combined result, this can only be done by rank 0. Considering the large file size, merging will be a relatively small part of the overall computation compared to the reading and processing part.

According to Amdahl's Law, the maximum potential speedup of this computation should be calculated by the formula $S(N) = \frac{T(1)}{T(N)} = \frac{T}{(1-\alpha)T+\frac{\alpha T}{N}} = \frac{1}{(1-\alpha)+\frac{\alpha}{N}}$, where α is the fraction of program that can be done in parallel. In this case, since merging as the non-parallelizable portions is a relatively small part, we let α to be 0.99 and $1-\alpha$ be 0.01 (the merging part).

$$S(N) = \frac{1}{(1-0.99)+\frac{0.99}{8}} = 7.48$$

As we use 8 cores instead of a single core, which is larger than the actual speedup (6.52), we believe that's basically because of the assumption of Amdahl's Law which is the problem size is fixed. But in real life, there will be additional complexity for example the communication overhead between MPI processes, synchronization overhead, and load balancing issues.
And comparing with the 8 Cores, 2 Nodes class, its speedup is 6.32 which is also slower than the theoretical speedup, and even slower than the 8 Cores, 1 Node class. We believe it's because there is more communication overhead as the computation is distributed across multiple nodes, the communication overhead increases due to inter-node communication. It will introduce latency and overhead compare with running on a single node. Also, distributing computation across multiple nodes lead to challenges in load balancing, uneven distribution of workload can lead to lower overall speedup.

According to Gustafson-Barsis Law, $S(N) = \frac{(1-\alpha)T+\alpha TN}{T} = (1-\alpha)+\alpha N$, where T is now the runtime using n cores and α is the fraction of runtime the program spends executing code in parallel.

	1mb	50mb	100gb
1 Core, 1 Node	0.02	0.69	1145.30
8 Cores, 1 Node	0.02	0.1	175.65
8 Cores, 2 Nodes	0.08	0.19	181.23

For each different file size, the actual speedup is listed below:

	1mb	50mb	100gb
1 Core, 1 Node vs	1	6.9	6.52

8 Cores, 1 Node			
1 Core, 1 Node vs 8 Cores, 2 Nodes	1/4	3.63	6.32

We can see that there is a trend which larger file size leads to larger speedup comparing with the baseline (1 Core, 1 Node). However, in a small 1mb file, the parallelisable portion is very small, which results in (1 core 1node) being processed at essentially exactly the same speed as (8 core 1node). This is probably due to the fact that the speedup of the parallel portion is cancelled out by the time-consuming overhead of MPI transfers between cores. Meanwhile the processing speed of (8 Core 2node) is even 4 times smaller than that of 1Core 1Node. This should be because the overhead due to MPI communication between the Nodes even exceeds the runtime. And according to the Gustafson-Barsis Law formula, this indicates the performance growth of multicore programs will not grow as optimistically as amdah's Laws due to overhead. When there are fewer files, a few times the processing overhead due to multicore may even result in faster file processing than if multicore processing were not used.

In addition, when comparing the actual speedup ratio for 50 mb and 100 gb files, there is not much difference between 1 Core, 1 Node vs 8 Cores, 1 Node. However, in the case of 1 Core, 1 Node vs. 8 Cores, 2 Nodes, the speedup ratio for 100 gb files is almost double that of 50 mb. The difference between 1Node and 2Node is that 2Node consumes more communication between Nodes. And when processing 50mb files and 100gb files, the ratio of communication overhead and parallel processing time for processing 100gb files is significantly larger. Thus, we can draw a tentative conclusion that the larger the ratio of overhead time to problem size, the smaller the multicore improvement. At the same time, excessive overhead can be reduced by increasing the problem size to improve the performance of multi-core processing.

Therefore, according to the Gustafson-Barsis law and the output, an increase in the size of the dataset will increase the scalability of the application. Larger datasets will allow for a significant increase in the benefits of multi-core parallel processing. This is because it makes the parallelisable part of the code will occupy a larger proportion of the overall process, and also reduces the percentage of time that the communication overhead is running, thus making more efficient use of multi-core processing resources.

Reflection

We can consider implementing more efficient communication patterns between the cores to reduce overhead. We can use applications like MPI_Allreduce instead of point-to-point communication for the merging part.
Also, as there is a difference between theoretical speedup and real speedup, we can try to reduce inter-node communication by optimizing data partitioning and processing strategies, like redistributing the data to minimize the need for cross-node communication or change the way data is stored.