THE UNIVERSITY OF MELBOURNE

Cluster and Cloud Computing - Assignment 1

Twitter Data Processing Using HPC

Abhinav Sharma 1009225

Rohit Kumar Gupta 1023418

Introduction

This project aims at delivering a simple, parallelised application that leverages the power of the University of Melbourne High-Performance Computing(HPC) facility named SPARTAN. Using `TwitterGeoProcessor` package, a large dataset of geocoded twitter file can be explored and analysed for extracting relevant information such as the number of posts in individual regions and trending hashtags in each one those regions. The package has been implemented on python and designed with the concepts of MPI (Message-Passing Interface) for effectively improving the performance of processing on the HPC environment.

Task Description

This application is required to search a large geocoded Twitter dataset to identify tweet hotspots around Melbourne. Specifically, the application should:

- Order(rank) the Grid boxes(regions) based on the total number of tweets made in each box and return the total count of tweets in each box.
- Order(rank) the top 5 hashtags in each Grid boxes(regions) based on the number of occurrences of those hashtags in each box.

Implementation Approach

After implementing various possible solutions, we are able to get the best performance using line-by-line processing of data. We have used the ***mpi4py*** python package for leveraging the power of MPI in python. In this approach, we process one object at a time for extracting region and hashtags and this can be done parallelly depending upon the number of processors provided. After the required data is extracted, it is then merged and reduced to produce the desired output.

This approach can be stated as the following steps:

1. **Initiate Master-child processor:** As the application starts, we initiate processor functions for master-node and child-nodes, if the request is to run on multiple cores and data from `melbGrid.json` is extracted into parsed JSON object which is then passed to the running nodes.
2. **Read large data file iteratively:** Both master and child node threads create independent handles of the file and read the data line-by-line. We have avoided

the loading of entire JSON data in the memory since it can raise memory overflow exceptions for large datasets.

3. **Distributing data as per line index:** Each thread processes different line number, using the logic of mod function applied to the total number of parallel nodes and the rank of an individual process. Every node including the master nodes processes the data in parallel.

   if *<node_rank>* **equals** *<line_number>* **mod** *<total_number_of_node>*
   then *{ process the line }*
   else *{ ignore the line, move to next one }*

4. **Preprocessing of data**: Program preprocesses the fetched data to check if it can be parsed into a valid JSON object, avoiding unknown JSON parsing exceptions.
5. **Extracting required data:** From the parsed valid JSON object, relevant information such as id, text, coordinates and hashtags are extracted using basic selection and regular expression.
6. **Extract region:** The region of the object can be attained using grid JSON object provided from the master node.
7. **Extract hashtags:** Extract the hashtags from the tweet and add them against region key. Keep the count of hashtags by incrementing already existing entries.
8. **Collecting distributed data & reducing to desired output:** Steps 4 to 7 are processed iteratively until all the lines exhausted. Next, the master node collects the final data from child nodes. This collected data is merged and reduced using the custom algorithm of *O(n) complexity* to get desired output most efficiently.

Package Structure

```
TwitterGeoProcessor
|——— TwitterGeoProcessor                  # Main Python Package for Twitter Geodata analysis
|————— lib
|————————— mpi_geo_manager.py            # MpiGeoManager class for MPI-driven processing
|————————— utilities.py                   # Utilities class for common functions
|————— twittergeoprocessor.py             # Main package class for managing execution flow
|——— out-files
|————— slurm-8038070.out                  # 1-node-8-cores output file
|————— slurm-8038102.out                  # 2-nodes-8-cores output file
|————— slurm-8038126.out                  # 1-node-1-core output file
|——— slurm-job
|————— one_node_eight_cores_big.slurm
|————— one_node_one_core_big.slurm
|————— two_nodes_eight_cores_big.slurm
|——— tweet_crawler.py                     # Script file to use TwitterGeoProcessor package
```

Invocation

The execution of this application starts from `tweet_crawler.py` script file. This execution is triggered by submitting slurm job on SPARTAN server as per the configuration provided. We have created 3 slurm files for different configurations: 1-node-1-core, 1-node-8-cores and 2-node-8-cores (4-core/node).

Commands of invocation for 1-node-8-cores slurm file can be explained as:

```
#!/bin/bash
#SBATCH --nodes=1
```
This command defines the number of nodes for process execution
```
#SBATCH --ntasks=8
```
This command defines the number of cores for process execution
```
#SBATCH --time=0-00:02:00
```
This command defines the wall-time for process
```
#SBATCH --partition=cloud
```
This command defines a partition on which process will run
```
module load Python/3.5.2-goolf-2015a
```
This command will load the required module for running the script
```
echo "---- 1-node-8-cores/big ------"
time mpirun -n 8 python3 tweet_crawler.py -d bigTwitter.json
```
This command will start the script by taking a parameter which specifies which file to process and produce results.

Challenges Faced

Some of the challenges we faced while creating the most efficient application:

1. **<u>Avoiding memory-overflow:</u>** It was challenging to load large twitter dataset file using normal JSON loading methods since it can cause memory overflow issue due to its large size of the dataset.
2. **<u>Choosing the most efficient data structure:</u>** One of the most crucial steps was to decide the most effective data structure which can save processing time. We used hashmaps which reduced the complexity from $O(n^3)$ to $O(n)$.
3. **<u>Exploring parallelizable code sections/logic</u>:** Writing the code from parallelism perspective, in order to reduce the serial code to optimize the usage of multiple cores.

Results and Analysis

Using MPI framework for the multithreaded application we were able to achieve the output in less time. We have used the comparison parameter to be the number of seconds it took for a test case to run in different scenarios.

We ran three test case, as mentioned below.

1. **One node one core:** In this scenario, the application ran serially since we did not have multiple cores to run the program in parallel. We were able to extract the required output in 195.99 seconds.
2. **One node eight core:** In this scenario, we ran part of our application in parallel using the eight core. Parallel processing helped us in reducing the time spent by almost four times. We were able to extract the required output in 56.603 seconds.
3. **Two nodes eight core:** In this scenario, we ran part of our application in parallel using the eight-core distributed across two nodes(4 cores each). Parallel processing helped us in reducing the time spent by almost four times. We were able to extract the required output in 58.842 seconds.

We observed that the application was able to improve the efficiency by 4 when we ran it on multiple cores.
Our observation is in alignment with the Amdahl's law which states that the increase in the efficiency is proportional to the parallelizable part of the application. Hence we could not achieve efficiency by a factor of 8 even with multiple cores runnings in parallel.

Also, we noted that there was no significant difference in runtime for the case with eight cores on one node and four cores on two nodes each.