COMP90024 Cluster and Cloud Computing

Assignment 1 2018 Semester 1

Application Report – HPC Instagram Geoprocessing

Ivan Ken Weng Chee

736901

Objectives of the Project

To implement a parallelized application to parse and analyse a large Instagram dataset to determine usage across locations in Melbourne.

To learn about the different functions and uses of Message Passing Interface (MPI).

To familiarize myself with working on High Performance Computing (HPC) Systems such as SPARTAN.

Approaches I Took to Parallelize My Code

There are multiple ways of approaching this problem and arriving at a conclusion. I have written and tested four different approaches in four different languages. Below are short descriptions of these:

Approach 1 (Python)

Master reads in melbGrid coordinates and sends this to all workers. Master then reads bigInstagram line by line, converting them to json format (python dictionary), and extracting point coordinates. These coordinates are sent to workers in an alternating manner. Workers checks and counts coordinates using three dictionaries. Upon master reaching the end of file, these dictionaries are reduced into master. Master finally sorts and prints the output.

What I learnt: How to make processes communicate via blocking point-to-point communication.

Approach 2 (Java)

This program works in a similar manner to Approach 1, except that master sends entire lines to workers instead of extracting coordinates. Workers also extract point coordinates from lines via string matching instead of JSON libraries. However, while this program works on mediumInstagram, it does not seem to terminate while processing bigInstagram, as sending each line essentially means copying the entire file through MPI calls.

What I learnt: Differences in regex vs. JSON parsing. The importance of minimizing MPI overhead.

Approach 3 (C)

All processes read in melbGrid, removing the need to rely on supply from master. Master reads bigInstagram line by line and extracts coordinates, storing them in an array, which is then scattered to all workers to process. Although only storing doubles, the array grows as large as the number of posts. The coordinate conversion process is done quickly but dirtily using strtok to delimit the line until (y,x) coordinates are found. The rest of the line is ignored. Upon counting points in corresponding grids, all processes reduce a single integer array into master. Master then derives row and column counts from this array, sorts them, and prints the final output.

What I learnt: Collective communication functions. How memory usage can speed up processing.

Approach 4 (C++)

All processes read in melbGrid. Each process opens bigInstagram and queries the system to obtain the size of the file in bytes. The total size of the file is then divided by the number of processes, whereby the last chunk size is modified to cater for uneven division. Each process, using their rank, calculates and seeks to the position of their chunk in the file. Processing begins from this point, taking care not to cross over into the chunk of the next process. Point coordinates are obtained using the same manner as Approach 3. Once processing is done, each process tallies the points and combines their results into master, again using the same reduce method as Approach 3. Master then determines row and column counts, sorts, and prints the final output.

What I learnt: How to achieve more optimal parallelisation. How data can be lost through splitting.

Comparison Between Approaches

Approach	Advantages	Disadvantages	Speed	Results
1 (Python)	-Work is distributed evenly among processes -Detects valid JSON structured posts -Gets the job done	-Workers may be blocking when master tries to send -For 2 processes, master passes everything to worker	Slow	1n1c: 154.860s 1n8c: 153.445s 2n4c: 173.362s
2 (Java)	-Identifies more posts due to string matching -Faster on single core compared to Python	-Extremely slow -Essentially copying the entire file through MPI calls	Very slow	1n1c: 034.367s 1n8c: N/A 2n4c: N/A
3 (C)	-Single scatter more efficient than calling send for each line -Ignoring the rest of line after encountering coordinates greatly speeds up processing	-Workers idle while master reads file -Does not make much use of parallelisation -Storage of coordinates may exceed memory for much larger files	Fast	1n1c: 013.367s 1n8c: 012.140s 2n4c: 014.064s
4 (C++)	-Processes are very independent with only one call to reduce results -Makes full use of parallelisation	-Reading at arbitrary locations in the file may cause some lines to be cut and potentially lost	Very fast	1n1c: 014.838s 1n8c: 002.083s 2n4c: 002.091s

Difficulties Faced

- Size of the dataset which inhibits reading into memory before processing
- Irregularity of JSON formatting of lines in dataset
- Missing coordinates or coordinates of [NULL, NULL] in some Instagram posts
- Differences between function arguments between different programming languages
- Limitations in terms of resources available to test and run on SPARTAN
- Limited local testing and different outcomes for local vs. SPARTAN environment
- Subtle data differences between tiny, medium, and big Instagram files
- Dealing with overhead when communicating between processes
- Debugging multiple, non-sequential running processes
- Partitioning of dataset may lead to irregular cuts and data loss

Variations in its Performance on Different Numbers of Nodes and Cores

```
1n1c
// Rank by Unit
C2: 175969 posts
B2: 22797 posts
C3: 18293 posts
B3: 6420 posts
C4: 4234 posts
B1: 3311 posts
C5: 2638 posts
D3: 2467 posts
D4: 1923 posts
C1: 1595 posts
B4: 1069 posts
D5: 783 posts
A3: 497 posts
A2: 479 posts
A1: 262 posts
A4: 133 posts
// Rank by Row
C-Row: 202729 posts
B-Row: 33597 posts
A-Row: 6720 posts
D-Row: 5173 posts
// Rank by Column
Column 2: 199245 posts
Column 3: 27677 posts
Column 1: 10517 posts
Column 4: 7359 posts
Column 5: 3421 posts
Time: 14.73s

real    0m14.838s
user    0m13.418s
sys     0m1.378s
```

```
1n8c
// Rank by Unit
C2: 176055 posts
B2: 22806 posts
C3: 18301 posts
B3: 6423 posts
C4: 4236 posts
B1: 3311 posts
C5: 2638 posts
D3: 2467 posts
D4: 1924 posts
C1: 1595 posts
B4: 1069 posts
D5: 786 posts
A3: 497 posts
A2: 479 posts
A1: 262 posts
A4: 133 posts
// Rank by Row
C-Row: 202825 posts
B-Row: 33609 posts
A-Row: 6758 posts
D-Row: 5177 posts
// Rank by Column
Column 2: 199340 posts
Column 3: 27688 posts
Column 1: 10555 posts
Column 4: 7362 posts
Column 5: 3424 posts
Time: 1.9s

real    0m2.083s
user    0m13.545s
sys     0m1.657s
```

```
2n4c
// Rank by Unit
C2: 176055 posts
B2: 22806 posts
C3: 18301 posts
B3: 6423 posts
C4: 4236 posts
B1: 3311 posts
C5: 2638 posts
D3: 2467 posts
D4: 1924 posts
C1: 1595 posts
B4: 1069 posts
D5: 786 posts
A3: 497 posts
A2: 479 posts
A1: 262 posts
A4: 133 posts
// Rank by Row
C-Row: 202825 posts
B-Row: 33609 posts
A-Row: 6763 posts
D-Row: 5177 posts
// Rank by Column
Column 2: 199340 posts
Column 3: 27688 posts
Column 1: 10560 posts
Column 4: 7362 posts
Column 5: 3424 posts
Time: 1.82s

real    0m2.091s
user    0m6.653s
sys     0m0.706s
```

The different post counts for 1n1c and 1n8c/2n4c are due to the nature of splitting the file at random positions. I am aware these may cause a line to split between processors, which may cause a loss in point coordinates. Executing with different processor counts may produce different results. 2n4c is slightly slower than 1n8c as message passing between different nodes is generally slower due to the systems being physically apart. Running on the cloud partition is also slower compared to physical.

Bar Chart Showing Times for Execution vs. Numbers of Nodes and Cores