Report

Student ID: 1189869, 1023137

Note: The submission package includes this report, as well as slurm scripts to run 3 tasks(xnodexcore.slurm), slurm output files(slurm_xnodexcore.out) and code script search.py, to run code in spartan, follow command in slurm script, to run locally, use the command (for example 4 cores):

```
mpiexec -n 4 python3 search.py --twitter_path='your_twitter_path' --grid_path='your_grid_path'
```

Designing and implementing a simple parallelized application to read huge data sets is a very effective way to take advantage of the efficient performance of computers to improve program efficiency.

In this report, we further explain the method we use to process and analyze the usage of twitters in Sydney by describing the code and running results in detail.

In the beginning, we process the Twitter.json file first. Extract the ID of each twitter in the file, using language and location (location includes longitude and latitude). When reading the twitter.json file, our approach is to use MPI for parallel processing.

Parallel Processing is a computing method that can execute two or more processors simultaneously in a computer system. Processors can work on different aspects of the same program at the same time. The main purpose of using this method is to save time in solving large and complex problems. In order to use parallel processing, you first need to parallelize the program, that is, distribute the parts of the work to different processors. Since parallelism is an interdependence issue, it cannot be implemented automatically.
The basic idea of our solution is to distribute accessing big JSON file and matching tweets with grids to multiple processes. Each process keeps its own version of 16 grids (a Python dictionary in our implementation) with each containing the number of tweets, languages used and geographical location. These information will be filled with the chunk of tweets from the bigTwitter.json specifically read by that process. Finally the master process (the one with rank 0) will gather grids information from all other process through collective communication, and combine and sort to get the final view of grids data describing the desired information of the whole file. More details follow.

We first need to calculate the total number of lines in the twitter json file. We aim to allocate the file reading task by each process only reading part of file. For example, total_line = 1000, if running 4 cores, process 0 only need to read 1-250, process 1 need to read 251-500…and so on. We let the master process opens the file and count the total number of lines and transmit that value to

other processes through point-to-point communication. Upon receiving the number of lines, each process calculate the range of lines it needs to read and opens file to start reading (same for master process). To read the file we use with open (file_name) as f such that it does not load the whole file into memory but gives a pointer to iterate through. The drawback of this approach we think can be the fact that when process 0 is counting the lines, all other process is idle and waiting. The process 0 need to do the IO twice and the point-to-point communication takes some time as well. However, we still prefer this approach in that we think other processes do not necessarily have to do the counting, which saves some computing resources. If the total number of lines given in the first line of Twitter file is correct that would be a huge relief since no counting is needed. We also discard the method of hardcoding number of lines for the reusability and portability of our code.
The purpose of using a parallel system is to improve CPU utilization and avoid wasting resources. And improve the running speed of multithreaded programs.

In the second step, we let each process process the sydgrid.json file. By reading the coordinates in the json file, the specific locations of the 16 areas are determined. Each grid also stores the latitude and longitude information of the grid. Each process then determine whether each twitter belongs to 16 grids by its position, get the total number of twitters for each grid, and update language information (a Python dictionary of {language: count})

The third step is process 0 gathering all the grids from other processes by comm.gather, combine them into one finalized grids and sort the language used in each grid. And present the result as a table (this is from slurm output file)

cell	#Total Tweets	#Number of Languages used	#Top 10 Languages & #Tweets
23	969	22	('en', 'ht', 'und', 'ja', 'in', 'ro', 'es', 'et', 'it', 'tl')
22	470	11	('en', 'ja', 'pt', 'it', 'in', 'de', 'es', 'tr', 'nl', 'fr')
21	85	4	('en', 'tr', 'fr', 'es')
20	162	8	('en', 'und', 'es', 'et', 'pt', 'th', 'tl', 'ca')
19	4522	31	('en', 'ja', 'es', 'und', 'in', 'ca', 'pt', 'zh', 'tl', 'de')
18	642	13	('en', 'zh', 'ja', 'in', 'ro', 'es', 'pt', 'ca', 'tl', 'und')
17	11	3	('en', 'th', 'in')
16	33	2	('en', 'es')
15	97	7	('en', 'in', 'es', 'und', 'zh', 'ro', 'fr')
14	508	13	('en', 'zh', 'und', 'it', 'ja', 'de', 'tl', 'pt', 'fi', 'es')
13	23	3	('en', 'ja', 'und')
12	98	7	('en', 'pt', 'cy', 'und', 'es', 'ca', 'lt')
11	47	5	('en', 'fr', 'de', 'und', 'ht')
10	184	5	('en', 'tl', 'da', 'ca', 'ht')
9	21	2	('en', 'fr')
24	3	1	('en',)

After that we put the code on spartan to run. The Slurm job scheduling tool was used. It provides a framework for starting, executing and monitoring work (usually parallel jobs such as MPI) on a set of assigned nodes. Slurm can do job analysis, including periodic sampling of per-task CPU usage, memory usage, power consumption, network and filesystem usage. It can help us understand the operation of the cpu in more detail.

We first write the entire program into a script, and submit it to the computing node for execution through the sbatch command;

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
module load foss/2019b
module load python/3.7.4
module load numpy/1.18.0-python-3.7.4
module load mpi4py/3.0.2-timed-pingpong
srun -N 1 -n 1 python3 search.py --twitter_path='bigTwitter.json' --grid_path='sydGrid.json'

##DO NOT ADD/EDIT BEYOND THIS LINE##
##Job monitor command to list the resource usage
my-job-stats -a -n -s
```

SBATCH --nodes=1 means that the number of nodes applied for is 1 node;
SBATCH –ntasks-per-node=1 means the number of cores used by each node is 1
Here we use srun -N 1 -n 1 to allocate nodes.

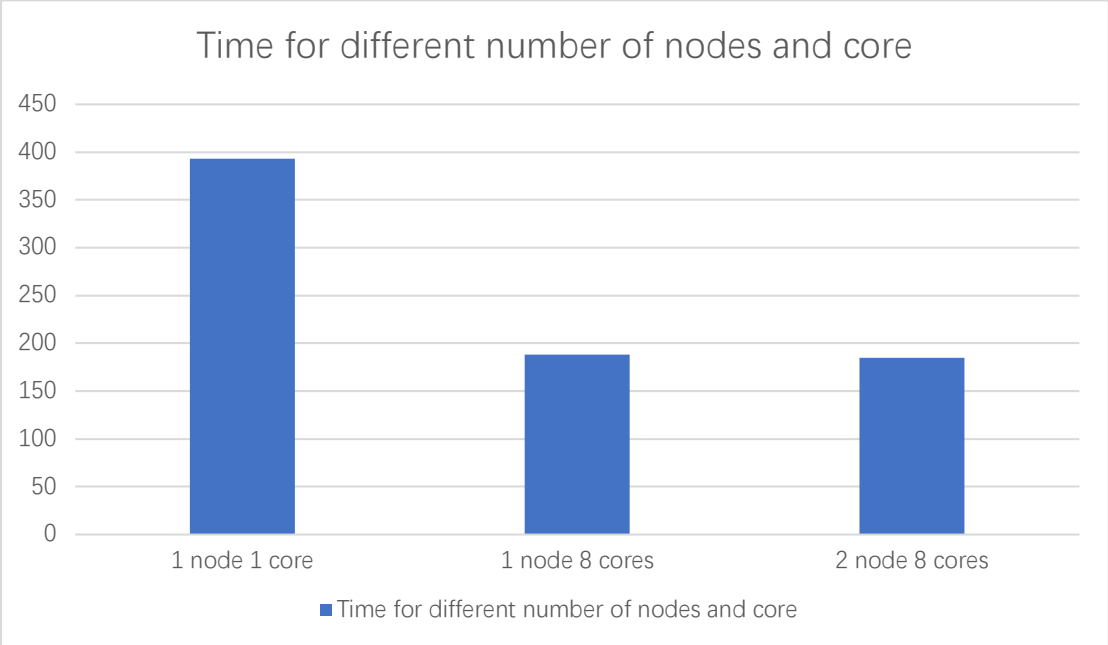
Observe the computer's running time and cpu usage by changing the number of nodes and cores. The cases we tested were 1node1core, 1node8cores and 2node8cores.
From a programming point of view, a multi-core CPU can increase the computing power of a PC without increasing the complexity of the program. However, due to the data consistency and master-slave relationship control among multiple CPUs, the complexity of program implementation, debugging and operation is relatively high.
From the perspective of program operation, the cooperation between multiple CPU runtime threads needs to be completed through the network or the motherboard bus, which is inefficient and affects the overall performance, while multi-core CPUs can be completed through the shared cache and main memory, and the cooperation efficiency is higher. However, when running multiple large programs at the same time, multiple processes will use the multi-core CPU in time-sharing, and the program switching overhead is relatively large.

The usage rates of nodes and cores for our jobs under different numbers are:
1node1core: node#1 core#1---95.6%
1node8cores: node#1 core#1---93.5%; core#2---80.0%; core#3---75.6%; core#4---83.9%; core#5---89.7%; core#6---94.9%; core#7---93.2%; core#8---93.0%
When running 2node8core, the slurm code we used is srun -N 2 -n 4, but the running result is that the utilization rate of four cores are very low, less than 25%. At this time, it was discovered that 8 cores were not allocated to our program. So we changed the code to srun -n 8, and the final usage rate of 2node8core is:
2node8core: node#1 core#1---93.3%; core#2---77.4%; core#3---86.4%; core#4---73.6%;

node#2 core#1---90.9%; core#2---94.9%; core#3---98.3%; core#4---98.2%;

CPU usage is above 50%. In the output given by slurm, there is also user time, which indicates the time that the CPU executes the user process. The higher the user space CPU the better. system time, which represents the time the CPU is running in the kernel. Higher system CPU usage indicates a bottleneck in some part of the system. The lower the value is better. idle time, indicating that the system is in an idle period, waiting for a process to run.

The running times of our jobs with different numbers of nodes and cores are:
1node1core: 397 seconds;
1node8cores: 188 seconds;
2node8cores: 185 seconds;



According to the bar chart, it can be clearly seen that when the number of nodes remains the same and the number of cores increases to 8 times, our program runs twice as fast as before. However, when the number of nodes changed from 1 to 2, the speed did not improve significantly. We think this may be because 1node reduces node-to-node communication than 2node. And the dataset we need to process is not very large. So when using 2node to run the program, there is not much improvement.

This program stimulates the ability to write code in python and more accurately understand how computers work. Through this program, we carefully discussed what we learned in class and successfully reduced the time to process big data using parallelization.