Justin Beaconsfield, Daniel Ye

Report: bigTwitter

SETUP

Our application can be invoked calling the following bash script on Spartan:

```
#!/bin/bash
#SBATCH --time=0:10:00
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1

module purge
module load python/3.7.4
module load foss/2019b
module load mpi4py/3.0.2-timed-pingpong

time srun python main.py

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

The ‘node’ and ‘ntasks’ parameters can be changed depending on the number of nodes and CPUs required to run the program.

The method used to parallelise the code, was to take the twitter file and one by one send tweets across to each node to be processed. Once the master node had sent a tweet to each node, it would process one itself. We found that this was relatively efficient and resulted in relatively low idle time for each CPU (< 5%).

To process the data the tweet would be scraped for relevant details, and then a dictionary of dictionaries would be used to keep track of the count of each language in each grid.

The dictionaries are then converted to arrays so that the master node can gather the summary data and tally it up. It is then eventually converted into a CSV to display the aggregate summary data.

Justin Beaconsfield, Daniel Ye

ALTERNATIVE ATTEMPTED METHODS OF PARELLELISATION

1. A method that seemed very promising was to chunk the data into packages of 10,000 tweets and send them to each node. The master node would do this and then begin processing tweets itself. Whilst the master node was processing tweets, it would be waiting to receive a message from other nodes, to say they were ready to receive more tweets. Upon receiving this message, it would then send off more tweets to that node. We believed it may have been more efficient to send a chunk of tweets all at once, relative to sending individual tweets, as each individual send may be chewing up time. The reason we didn't implement this was we struggled to technically implement non-blocking messages. We attempted to use the 'Irecv' function, but we couldn't seem to figure out how to effectively use this function, and this eventually forced us to abandon this method

2. We decided to still have a look at the efficiency of chunking and split the tweets into blocks of 10,000 that would be continually sent out to each slave node, whilst these were being processed master node would process a chunk of its own and then go and send out another set of chunks. This seemed somewhat promising, but ultimately it was inefficient as each node spent too much time idle. We believe that this was because of inconsistencies in performance times on the various CPUs, which would result in the process becoming out of sync. If the CPUs were able to communicate (with non-blocking messages), this issue would have likely been averted.

RESULTS

We find that the quickest process is to run on 1 node with 8 CPU's, which runs in a time of 2:14. This is closely followed by 2 nodes with 4 CPU's each (8 CPU's), which runs in 2:19. Then as expected, the slowest process is to run on 1 node with 1 CPU, which runs in 5:11.

This is consistent with what we'd expect. The parallelisation of the tweet processing means that using more CPU's will significantly speed up the process. Further than this, we see that using just 1 node is quicker than splitting the load across 2 nodes, because there is less physical distance required for messages to travel back and forth between nodes.

The fact that the single CPU was only a little over double the time of the 8 CPUs suggests there was some inefficiency in our application. Amdahl's Law suggests that this could be because of a lack of parallelisation in our application, however it is also likely that in this case the time taken to send messages was severely slowing down our application (making it up to 4 times slower), meaning the single node that did not have to send messages could avoid this issue.