Design Decisions: Threaded vs. Non-Threaded

We chose to use multi-threading in our project for several reasons.

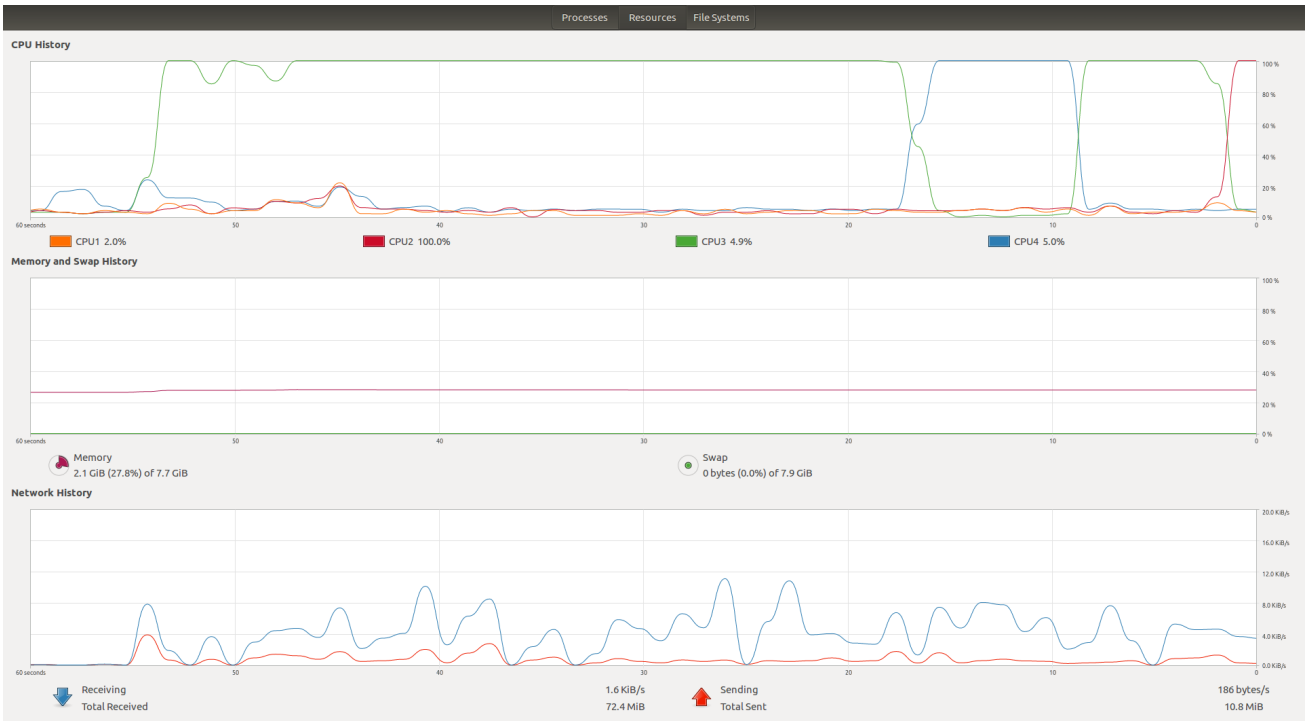
From a functional standpoint, only multi threading would allow us to simultaneously stream tweets from multiple locations. From an optimization standpoint, multi threading allows us to **evenly split up CPU usage** as well as **maximize network throughput**. To test this theory, we created both a single threaded and multi threaded version of the program. The threaded program clearly demonstrated higher throughput, as discussed previously.

Of course, by using threading, we open our program up to a few vulnerabilities. Since our project relies on a shared meter to count total hits, we need to ensure that **no race conditions are encountered** when writing to that meter. To prevent this, we used **semaphores**- otherwise known as ***locks*** in Python. `Global_meter_lock' is created by the parent, and whenever the parent or child wants to access it, the process waits its turn.

Because of the number of children we anticipated having in the program, we elected to only write the **global meter** to a file every few seconds in the parent process. This was done to achieve speedup- if we were writing to the Global_meter.txt file every loop iteration of every thread, the additional file I/O overhead would significantly increase **CPU use** and **run time**. To test this, we created two extra versions of threaded_tweets.py: one with file writes every loop iteration (fileIO_test_with_writes.py), and one without (fileIO_test_without_writes.py). For N=100,000, each location took about 4.5-4.6 seconds to process the loop in parallel without writing to a file. On the other hand, with file writes, it took 86-101 seconds for each location to complete the loop (output included below). With this proven hypothesis, we decided to move global_meter read/writes to the parent process, and to happen only every second. However, we did elect to keep **local_meter** read/writes (relative to the location), which means that there was still a performance cost for file I/O, though less than for **global_meter** because 10+ threads aren't all trying to access the same variable at the same time with **local_meter**, unlike the traffic congestion than there is at the semaphores surrounding **global_meter**.

Finally, a process-kill class was added to allow the program to gracefully be killed via sigkill from the terminal. The code for this was taken from: http://stackoverflow.com/questions/18499497/how-to-process-sigterm-signal-gracefully.

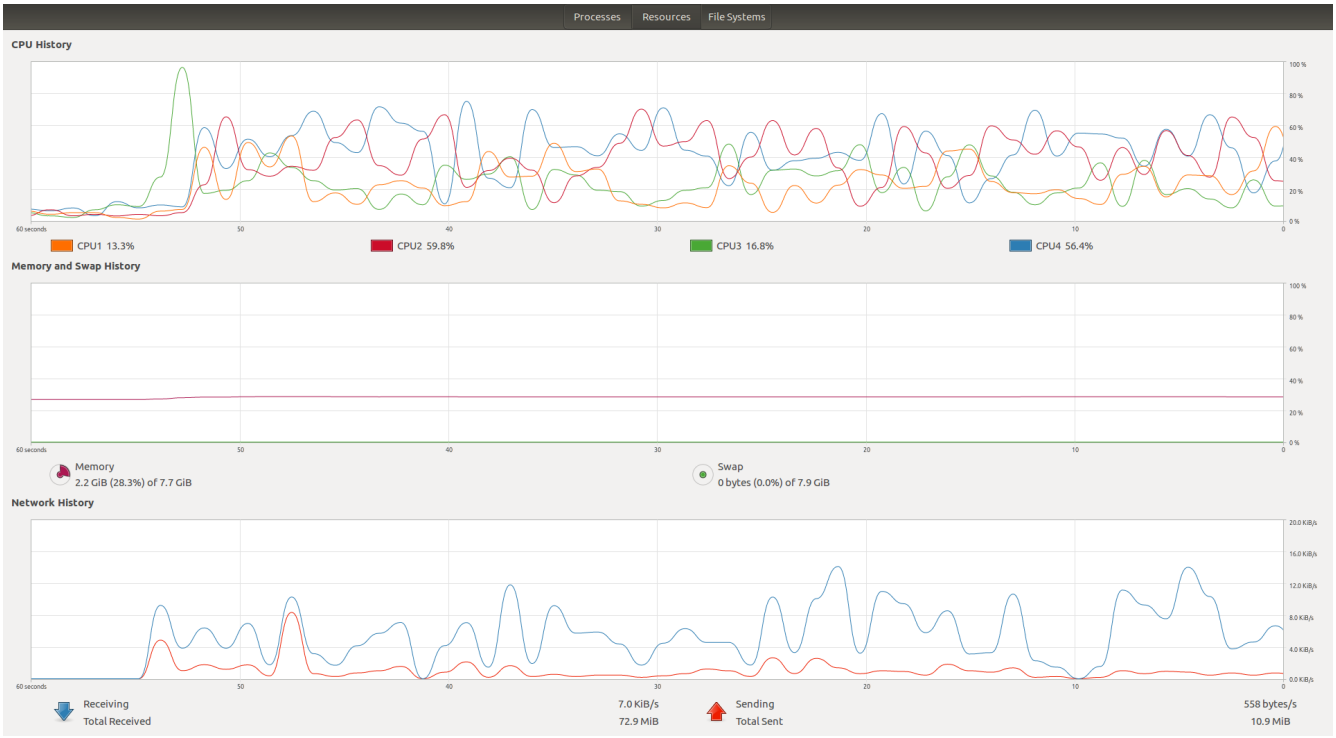
System Usage Notes



Threads = 1:

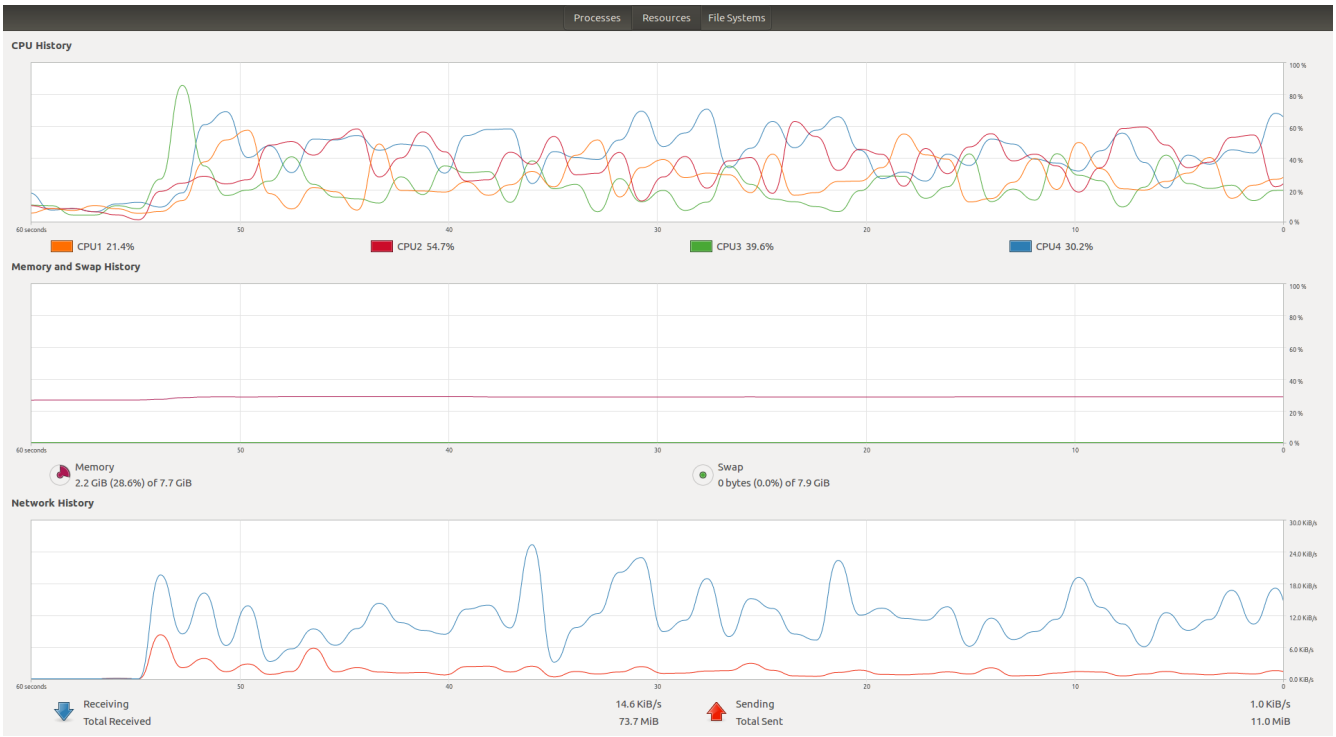
- CPU is at 100% on one core, since essentially there is only 1 highly active thread
- Averaging around 4 KiB/S network use

Threads = 2:



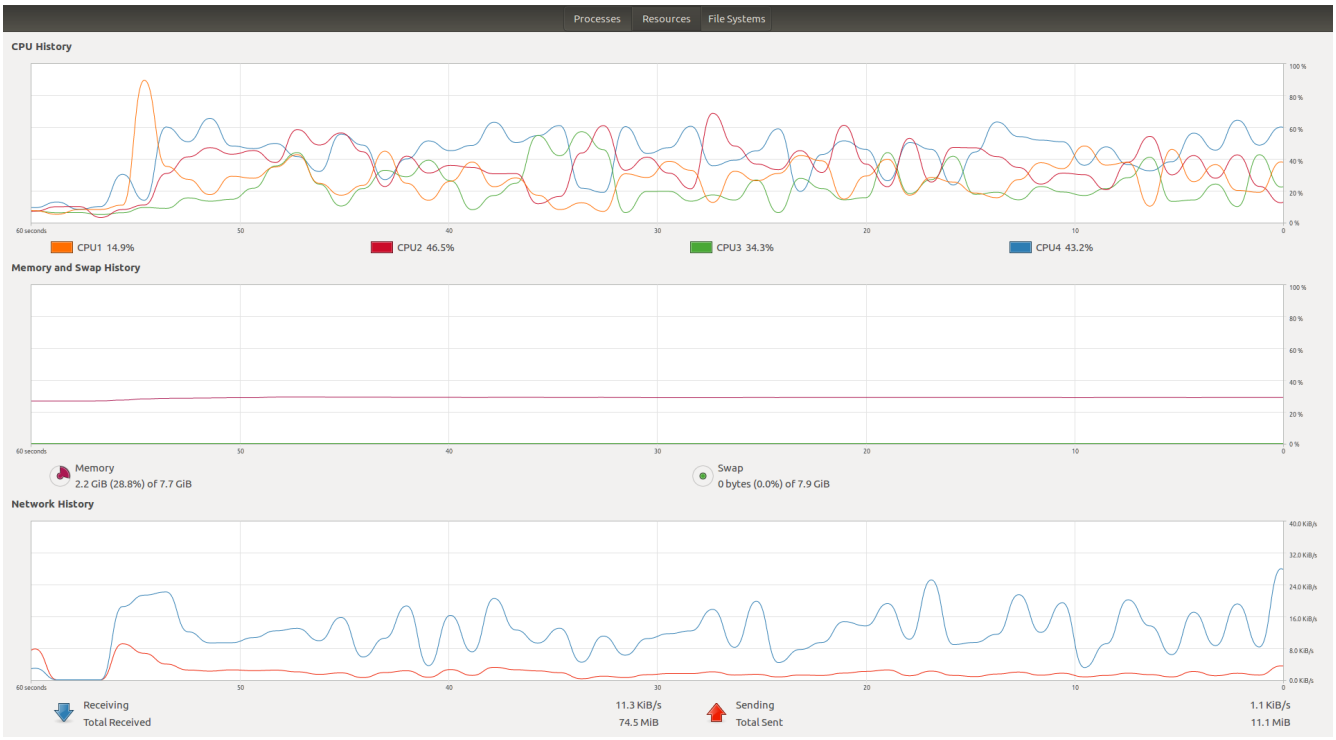
-CPU usage begins to be divvied up among 4 cores
- Network usage goes up from 1.6 KiB/s to 7 KiB/s, which is a huge increase, but probably because there's a higher amount of tweets coming from Georgia (the added second location) than Ohio (the first location) at the moment the screenshot was taken
- Network use averaging around 10 KiB/s, obviously doubling

Threads = 3:



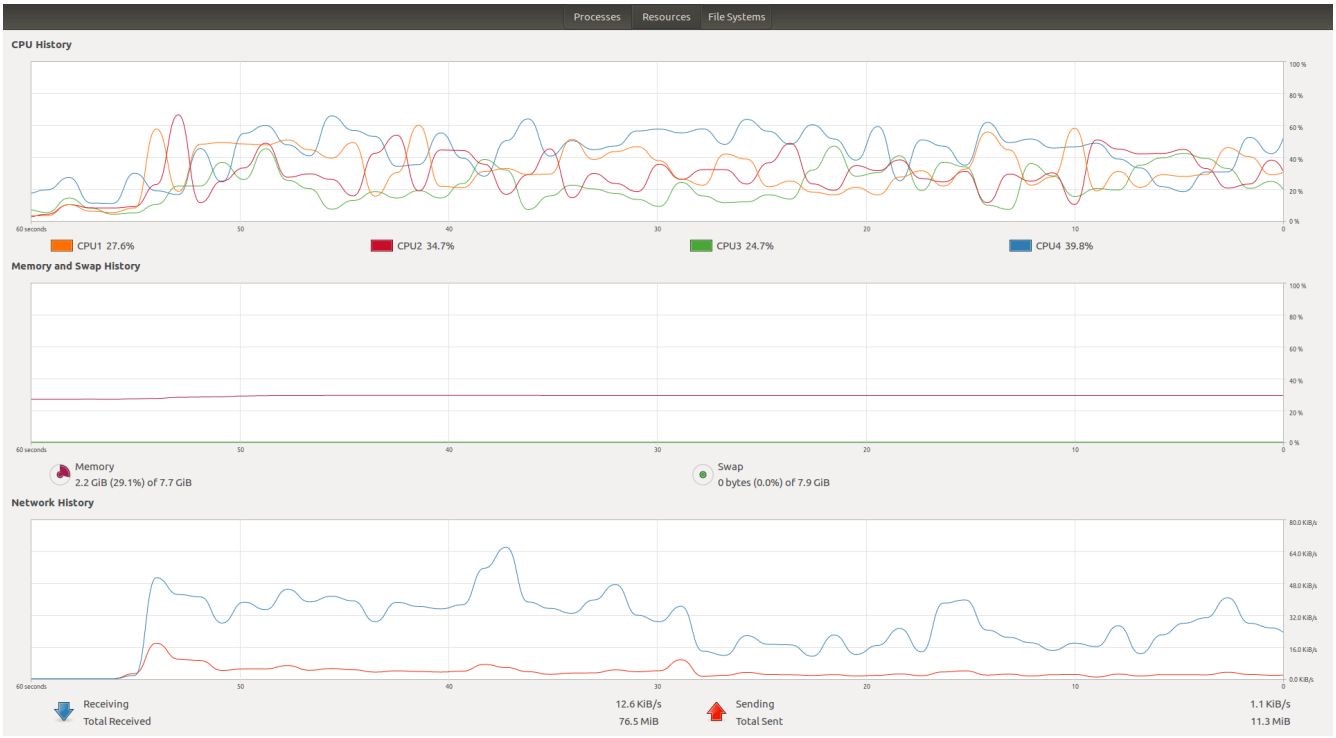
- CPU usage continues to be fairly divvied up
- Network averaging about 15 KiB/s

Threads = 4:



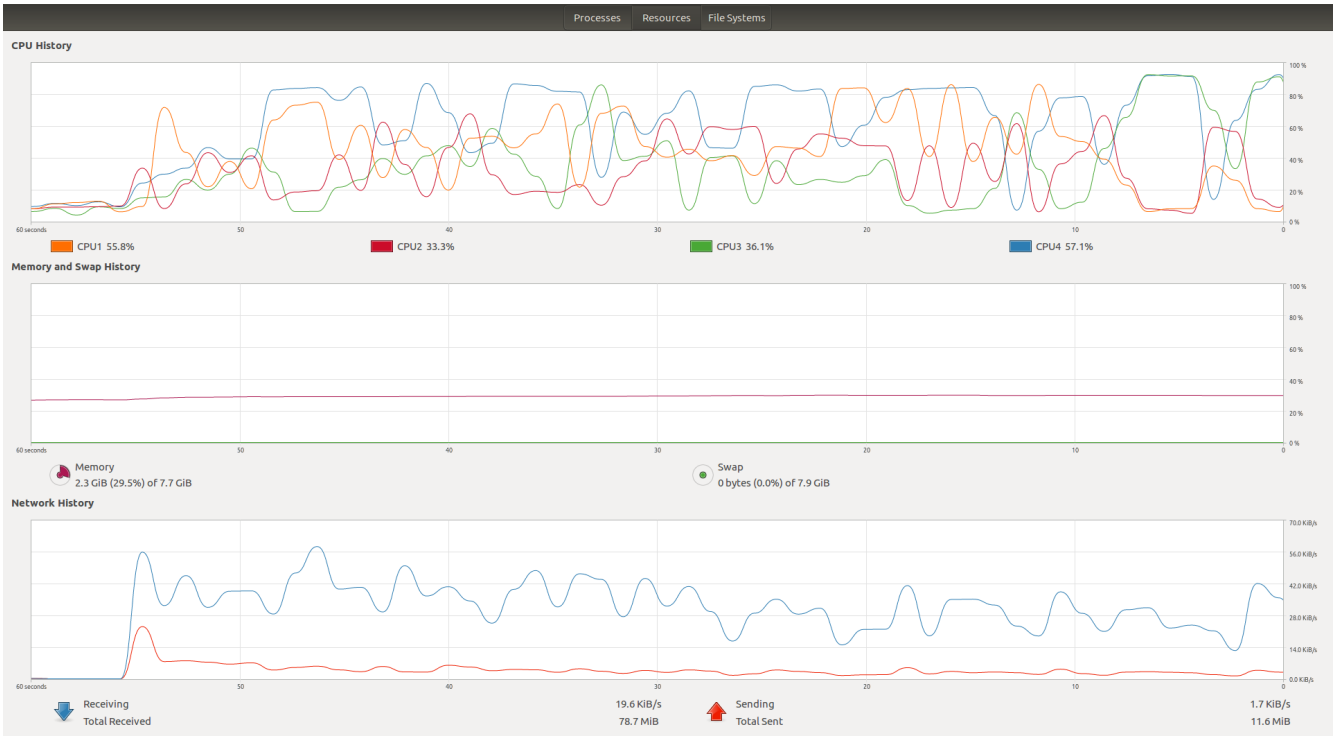
-Network usage is at 11.3 KiB/s
- averaging around 16 KiB/s

Threads = 6:



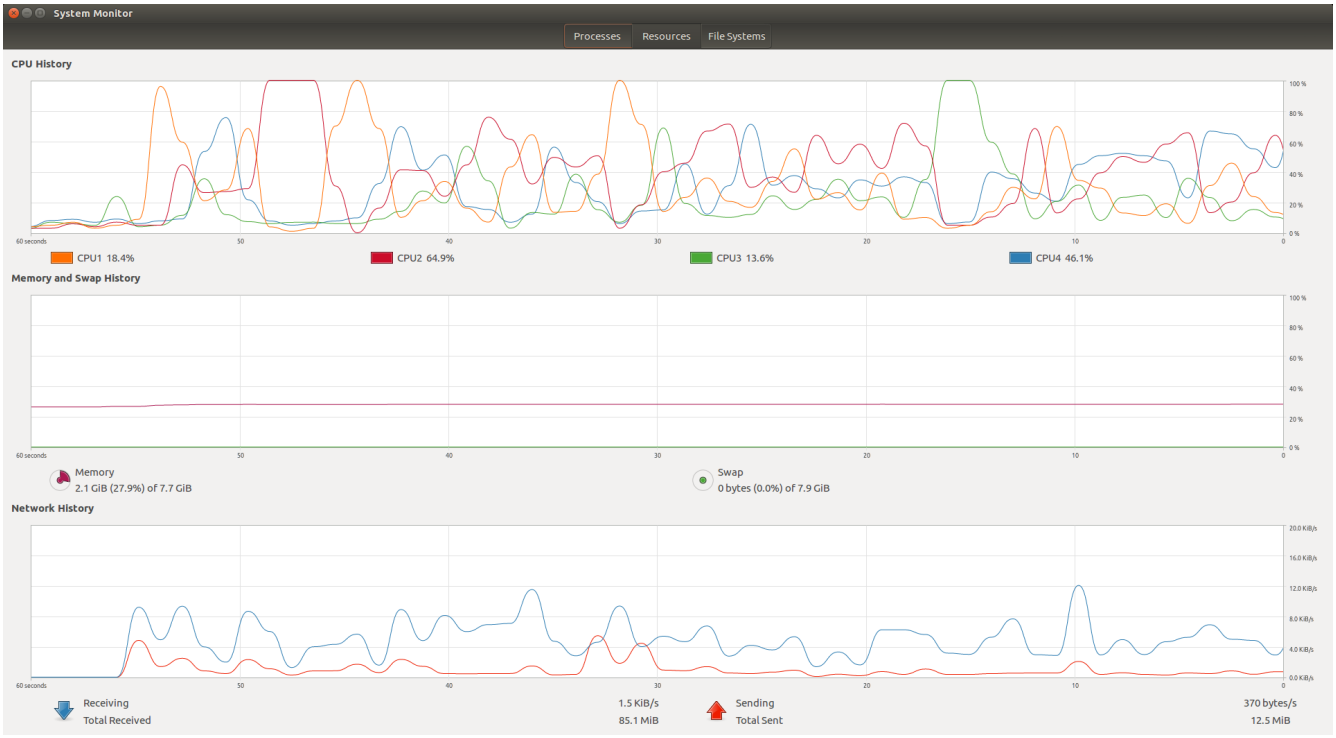
network usage at 12.6 KiB/s
averaging around 30 KiB/s

Threads = 8:



- CPU usage begins flip-flopping
-network usage is at 19.6 KiB/s
- averaging around 30 KiB/s

Threads = 10:



- CPU use begins maxing out
- Network use is averaging around 8 KiB/s

Summary

As we add more locations to be streamed, the CPU use is adequately balanced among the four cores of the tested machine due to multi-threading, even as the network usage increases with tweet throughput as more locations are adding, proving that multi threading helps optimize tweet streaming.

OUTPUT

########[without writes]############
avo@cado:~/Documents/OSF/OSFinalProject$ python fileIO_test_without_writes.py
fileIO_test_without_writes.py:78: SyntaxWarning: name 'global_meter_lock' is assigned to before global declaration
  global global_meter_lock

Location: Colorado processing time: 4.56175398827

Location: Arkansas processing time: 4.56698012352

Location: Alabama processing time: 4.58569502831

Location: Connecticut processing time: 4.59255599976

Location: Florida processing time: 4.59477591515

Location: Delaware processing time: 4.59851193428

Location: Ohio processing time: 4.6009850502

Location: Arizona processing time: 4.60762000084

Location: Georgia processing time: 4.61667895317

Location: NewYork processing time: 4.61716008186

Location: Alaska processing time: 4.61888408661

Location: California processing time: 4.61869597435
########[with writes]##########

avo@cado:~/Documents/OSF/OSFinalProject$ python fileIO_test_with_writes.py

Location: Colorado processing time: 86.6615760326

Location: Arkansas processing time: 86.9368629456

Location: Ohio processing time: 96.1166989803

Location: Florida processing time: 96.8635439873

Location: Connecticut processing time: 98.253000021

Location: NewYork processing time: 98.9788639545

Location: Delaware processing time: 99.5173130035

Location: California processing time: 99.5583689213

Location: Alaska processing time: 100.30095005

Location: Georgia processing time: 100.618832827

Location: Alabama processing time: 100.663938046

Location: Arizona processing time: 101.187855959