A4: Truck Factor

Tyler Brown

Feb. 21, 2019

This assignment is not currently up to standards because I spent too much time focusing on how to run my code in parallel using a distributed system. Given that I cannot meet the requirements of A4 at the agreed upon time, I can detail my efforts. I think that missing this deadline will be an important lesson for me in terms of project management and work performance going forward. I was able to get tests passing for a mocked up truck factor analysis. My efforts since A2 are described in sections on understanding the problem, distributed systems, and implementation of the truck factor using mocked data, and issues going forward.

Understanding the Problem

Downloading GitHub repos means retrieving a number of relatively small files via HTTP. This problem does not need to be solved sequentially. We can most efficiently solve this problem by distributing the workload across multiple machines solving sub-problems in parallel. More formally, the problem is

$$g(x) = \sum_{i=0}^{N} x_i$$

where $g(x)$ refers to downloading all GitHub repos in vector x. Each sub-problem x_i must be solved when computing $g(x)$. We do not have to solve this problem sequentially.

Distributed Systems

Employees at Databricks (https://databricks.com/) have been very generous in answering my questions about Apache Spark. There are many things Spark does well but handling a lot of small files is not one of them. Using Spark for collecting GitHub repos is not appropriate. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. Using containers to process batches of the GitHub repos bypasses the small files issue.

Kubernetes can be used to run applications or jobs. The design pattern for our job workflow is called "Fine Parallel Processing Using a Work Queue" (https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/). In Kubernetes, multiple pods, containers, can exist on a node. For the design pattern, each pod is created, it picks up one unit of work from a task queue, processes it, and repeats until the end of the queue is reached.

Using Kubernetes

Minikube (https://kubernetes.io/docs/setup/minikube/) is the recommended way to run Kubernetes locally. I first tried to run Kubernetes locally using a `t2.medium` AWS instance with a `Amazon Linux 2` operating system. I was unable to install minikube because a Kubernetes utility, `kubectl` depends on `systemctl` which is not present in the Amazon Linux distribution. I then switched to `Ubuntu 18.04 Server` on AWS. I was not able to reliably start minikube using Docker as the `--vm-provider`. Trying to switch to other providers such as VirtualBox or KVM (KVM2) was not available on AWS. I then tried to bypass the issues with minikube by solving a bigger problem.

AWS EKS (https://aws.amazon.com/eks) is a managed Kubernetes service. I was able to start this cluster, install the `kubectl` and amazon authentication utilities, my ssh key was previously configured on `IAM`. However, I was not able to appropriately define permissions. I scheduled an office hour with the instructor to resolve the AWS permissions issue.

Using a Fine Parallel Processing Work Queue

I wrote and released a Python package, `okra` ("pip install okra"), to handle the work queue. I used code from their example, a Redis backend, and added some utilities. If you install `okra` and open multiple terminals then you can currently run the work queue. Right now it just writes out the job assigned to a log file.

```
(okra) bash-3.2$ okra -h
usage: okra [-h] [--redis REDIS] [--load_redis LOAD_REDIS]
            [--logfile LOGFILE]

optional arguments:
  -h, --help            show this help message and exit
  --redis REDIS         job name for redis loader/worker
  --load_redis LOAD_REDIS
                        file path to load redis queue
  --logfile LOGFILE     filepath to log file
```

An example that will retrieve all items from work queue `job2` is

```
(okra) bash-3.2$ okra --redis job2 --logfile task.log
```

the log file is optional. I still need to install the `okra` Python package into a docker container. Kubernetes has some specifics about how it interacts with Docker containers. It would be helpful for me to have Kubernetes working first so I can ensure compatibility when building the `okra` Docker container.

Truck Factor using Mocked Data

I was only able to implement the Truck Factor using unit tests. I did not get my whole system working in time to complete A4. Tests for the Truck Factor are located under `tests/test_assn4.py` within the `okra` working directory. You can run all my tests with `python setup.py test`. A documentation website is available locally under the `build/sphinx/html` folder.

Summary

Here's what I've done:

1. Wrote a python package to handle github data
2. Researched parallel solutions for the Truck Factor
3. Implemented some components of this parallel solution

Here's my TODO list:

1. Office hours to troubleshoot AWS EKS
2. Complete Truck Factor using 1000 repositories by finishing my 'book keeping' code in `okra`.
3. Build Docker container for `okra` and set up small-scale parallel process
4. Visualize my findings
5. Create a map of GitHub repos and try to scale up the Truck Factor