CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): YACS Coding Date: 01/12/2020

SNo	Name	SRN	Class/Section
1	Adithi Satish	PES1201800150	B
2	Manah Shetty	PES1201801733	B
3	Saurav Nayak	PES1201801307	G
4	Shreya Shukla	PES1201800124	I

Introduction

The project chosen and implemented was YACS, or Yet Another Centralized Scheduler. Schedulers are of paramount importance in any big data frameworks, so as to help distribute the load and improve performance overhead of the system. The framework uses three different scheduling algorithms - Random Scheduling, Round Robin scheduling and Least Loaded scheduling, in order to determine what task to send to which worker node (here, 3 workers have been used along with one master).

The master receives a job request, consisting of several tasks. Each of these tasks is assigned to a worker based on the scheduling algorithm. Once a task is completed, the worker sends the update to the master, which then removes the task from the scheduling pool. The number of slots in each worker need not be the same. The maximum number of tasks a worker can process is equal to the number of slots on the machine.

This project deals with a simulation of this centralized scheduling framework, and also performs analysis on the performance overhead and number of tasks processed by each worker, for all three algorithms.

Related work

A few online sources were referenced for insight into socket programming:

- https://docs.python.org/3/howto/sockets.html
- https://realpython.com/python-sockets/
- https://www.geeksforgeeks.org/socket-programming-python/

Design

The implementation uses threads to introduce concurrency in the functioning to provide an accurate simulation while improving efficiency. Concurrency has been implemented with care to avoid race conditions with the shared data structures.

The master program implements four shared data structures; `task_logs`, `job_logs`, `config` and `scheduling_pool`. The first two are used to store the logs while config stores the status of the workers and scheduling_pool maintains a dictionary of all the jobs and their tasks.

The worker program implements a single shared data structure; the pool of executing tasks: `e_pool`.

Mutex locks have been used when modifying all of these structures to prevent race conditions.

The workflow of the programs is described below.

master.py:

The program spawns three threads whose functioning are as follows:

Thread 1:

- Listens for job requests on port 5000
- Adds the request to a pool
- On receipt of a request, picks an algorithm as specified to the program via the command line.
- Picks a Worker to run the task on (based on the algorithm), decrements its free slots and sends the task to the chosen worker.

Thread 2:

- Listens for task completion updates on port 5001
- Increments the free slots on the worker that ran the task
- Removes the task from the pool

Thread 3:

- Iterates through the jobs in the pool, and for each job checks the completion of all map tasks.
- If complete, launches the reduce tasks of the job.

worker.py:

The program spawns two threads whose functioning are as follows:

Thread 1:

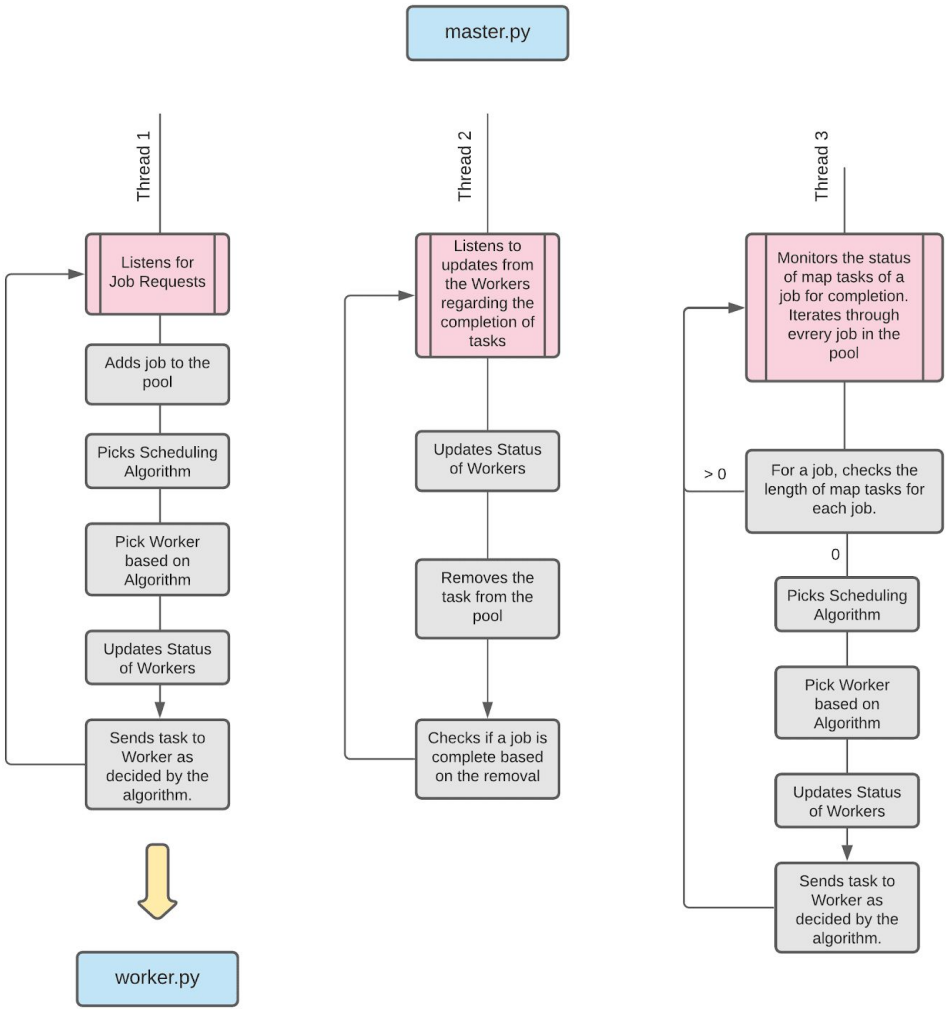
- Listens to the master for tasks on the respective port.
- Sets an end_time based on the duration of the task and adds it into the pool.

Thread 2:

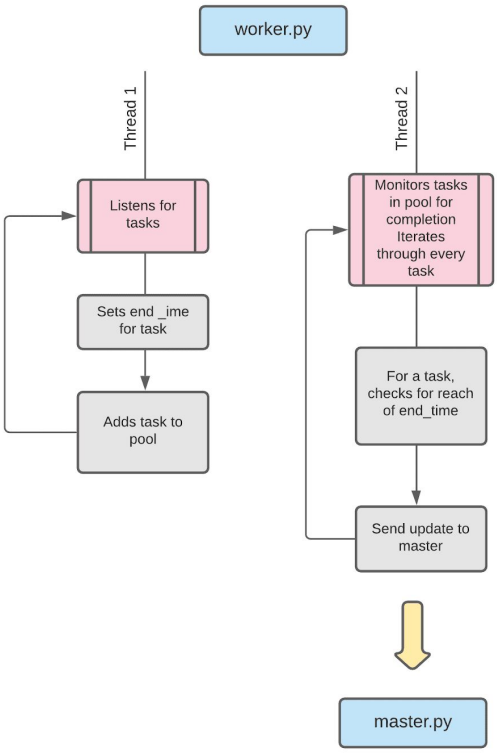
- Iterates through every task in the pool and checks if end_time has been reached.
- If yes, sends update to the master and removes it from the pool.

A detailed breakdown is depicted in the block diagrams that follow—

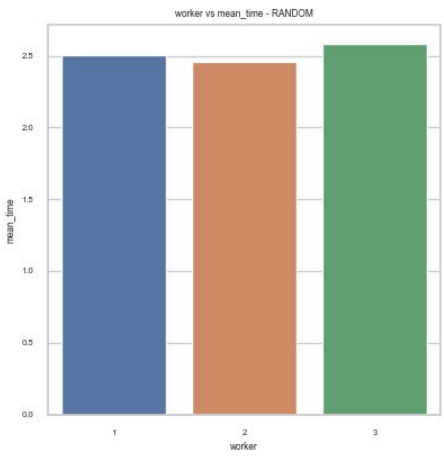
master.py:

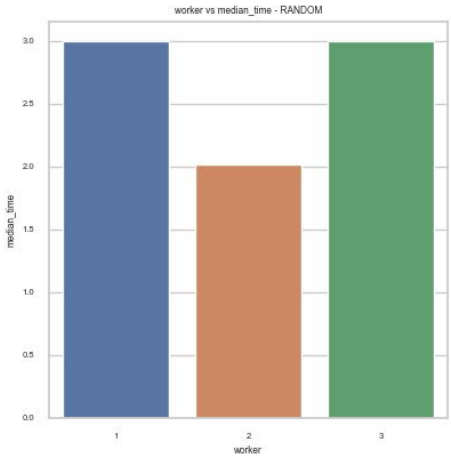


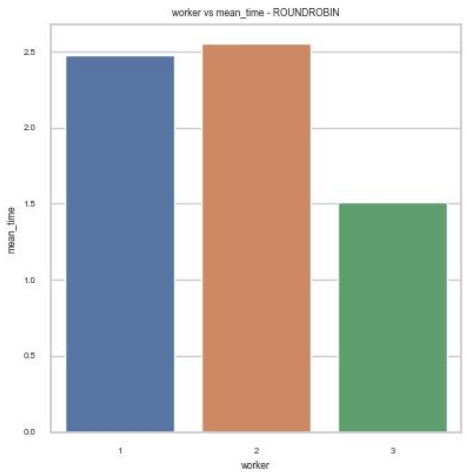
worker.py:

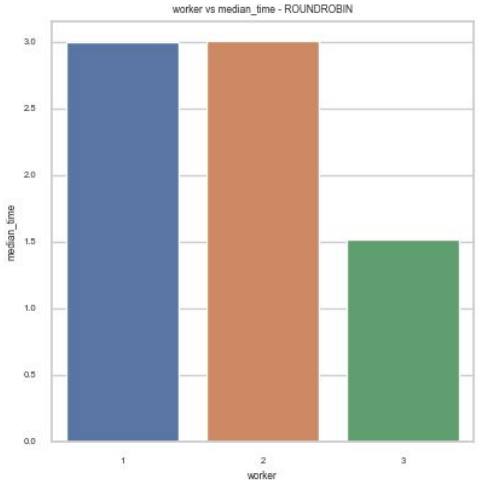


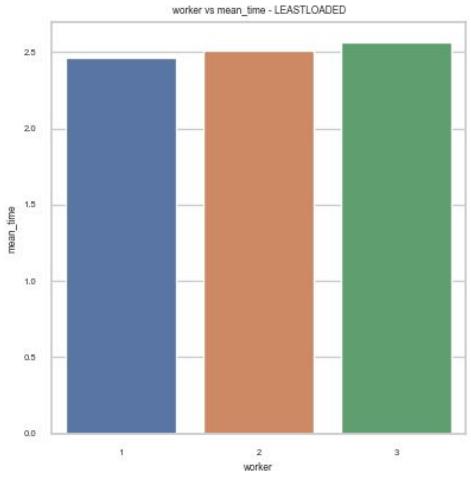
Results

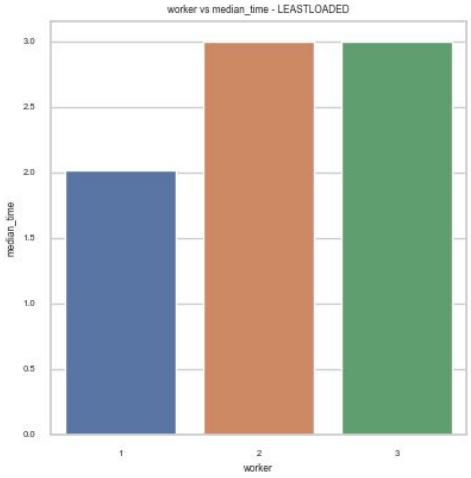




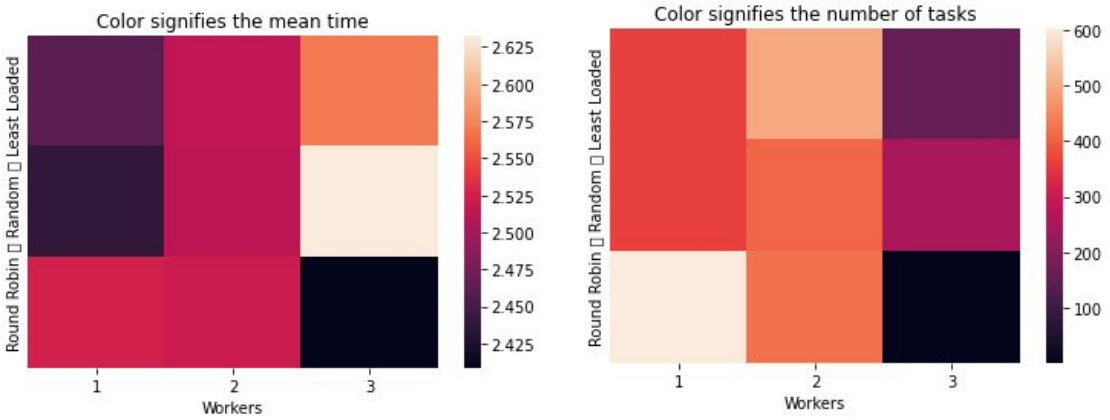








The above bar plots indicate the mean and median task completion time of the three scheduling algorithms. The mean and median values lie around the same range for all the algorithms across the worker machines.



The heatmap on the left represents the mean time of task completion on the workers for each of the scheduling algorithms. The values signify similar mean times for task completion in the range of 2.4 to 2.6 with tiny differences due to overhead. The figure on the right describes the task distribution on each of the workers. The bottom row shows an unequal distribution of tasks on the workers using the Round Robin algorithm. The first machine takes a major load of tasks but the third one gets assigned fewer tasks. However, the Random scheduling algorithm provides a more equal distribution of tasks with Least Loaded algorithm scheduling more tasks on the worker with more number of slots.

Problems

Race conditions and deadlocks are often hard to detect and sometimes come into play when implemented over a larger number of job requests where the flow also becomes harder to detect and map because of the girth of the input. The concurrency and interdependency of the programs also make it hard to trackdown the source of the failure.

The program output was first verified with a personal map for a smaller pool of requests.

Verbose print statements including information on the threads were used at every step to ensure proper flow of the tasks. Deadlocks were taken care of by ensuring that no structure remained locked in a loop, and ensuring immediate release of acquired locks.

Conclusion

In conclusion, this project proves to be an important learning experience by providing hands-on experience on how scheduling frameworks operate. We learn the nuances of how the communication between the master and worker nodes takes place in the background, how concurrency plays into all of this and how crucial the implementation of the parallelism is with respect to the avoidance of race conditions and deadlocks.

The project also gave us insight into the different scheduling algorithms that are used to schedule tasks on the worker nodes and how they affect the workload on the machines.

The project helped us gain a better understanding of the process flow that happens in the background of a Centralized Scheduler.

The performance aspect of the master-worker task flow comes into light with varying execution times on different algorithms which makes the task of distributing loads across workers a crucial decision in the system.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Adithi Satish	PES1201800150	Set-up the communication between processes; Analysis of the logs.
2	Manah Shetty	PES1201801733	Mapped the flow and structured the parallelism.
3	Saurav Nayak	PES1201801307	Set-up the communication between processes; Analysis of the logs.
4	Shreya Shukla	PES1201800124	Structured the parallelism; Concurrency checks.

(Leave this for the faculty)

Date	Evaluator	Comments	Score

CHECKLIST:

SNo	Item	Status
1.	Source code documented	
2.	Source code uploaded to GitHub – (access link for the same, to be added in status)	
3.	Instructions for building and running the code. Your code must be usable out of the box.	