CS322:Big Data

Final Class Project Report

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

SNo	Name	SRN	Class/Section
1	Snigdha S Chenjeri	PES1201800045	5C
2	Sakshi Shetty	PES1201800190	5C
3	Ananya V	PES1201800204	5I
4	Swanuja Maslekar	PES1201800369	5H

Introduction

The project we have chosen is YACS (Yet Another Centralised System). It's a framework simulating a master (for handling scheduling) machine and worker processes (for execution). Our project consists of one master machine facilitating jobs—which handle multiple tasks—for three workers.

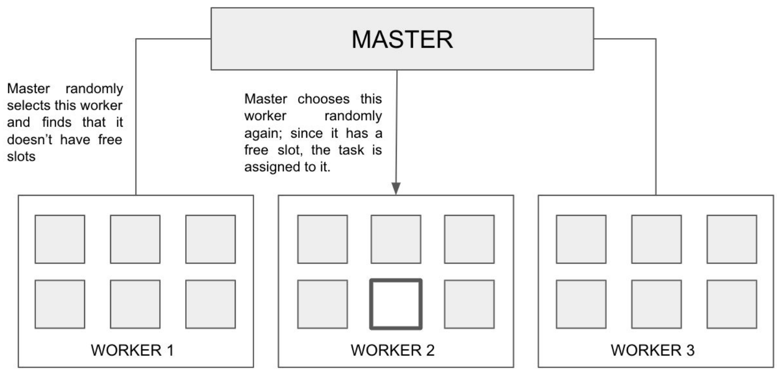
Related work

Background study material that we have read and referenced:

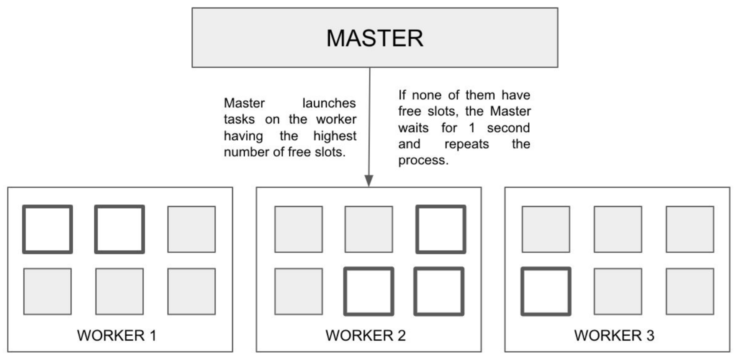
- Theoretical references
- Coding references

Design

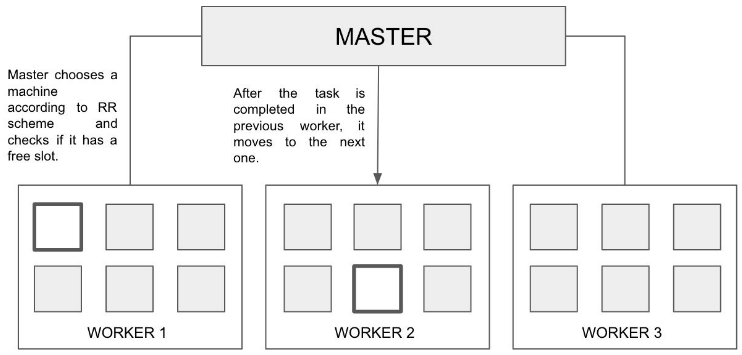
With three workers running worker processes, our Master machine allocates tasks them to based on three scheduling algorithms (as we have depicted below):


MASTER
Master randomly selects this worker and finds that it doesn't have free slots
Master chooses this worker randomly again; since it has a free slot, the task is assigned to it.
WORKER 1
WORKER 2
WORKER 3

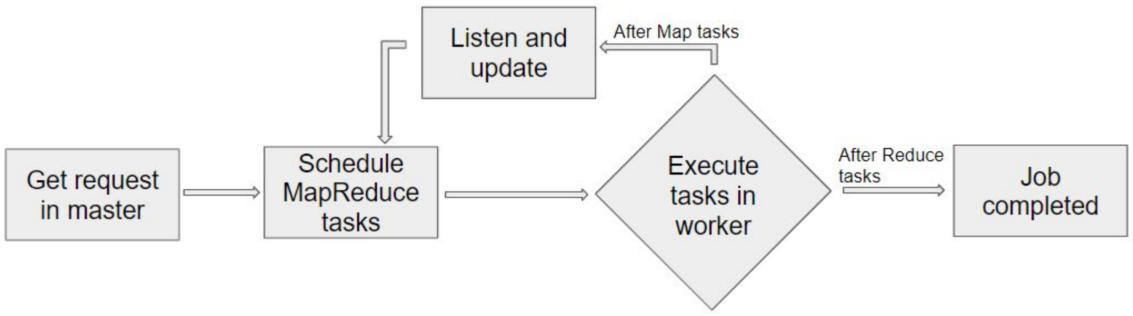
RANDOM SCHEDULING


MASTER
Master launches tasks on the worker having the highest number of free slots.
If none of them have free slots, the Master waits for 1 second and repeats the process.
WORKER 1
WORKER 2
WORKER 3

LEAST-LOADED SCHEDULING


MASTER
Master chooses a machine according to RR scheme and checks if it has a free slot.
After the task is completed in the previous worker, it moves to the next one.
WORKER 1
WORKER 2
WORKER 3

ROUND-ROBIN SCHEDULING



Flowchart

- Master.py contains three threads:
  - Thread 1 : Listens to requests from Requests.py which sends one job at a time. Allocates map tasks to workers as per the algorithm specified (LL - Least Loaded, RR - Round Robin or RANDOM - Random Scheduler)
  - Thread 2 : Listens for updates from workers regarding the completion of tasks and checks for any existing task dependency, i.e., the completion of all map tasks before launching reducer tasks.
  - Thread 3 : Schedules reduce tasks post the dependency check in Thread 2, after all map tasks of a particular job have finished executing.

- Workers.py contains two thread:
  - Thread 1 : Listens for tasks from the Master.
  - Thread 2 : Executes the task and sends updates to the Master.

- Tasks, both map and reduce, are allocated based on the algorithm chosen. The algorithm options are - RR, LL and RANDOM
  - RANDOM - Chooses a worker at random and checks if free slots are available. If yes, send the task to the worker.

    - RR (Round Robin) - Chooses the workers turn by turn and checks if the worker chosen has free slots available. If yes, send the task to the worker.
    - LL (Least Loaded) - Chooses the worker which has maximum free slots (least loaded) and allots the task to that worker.

    If all slots are full in all the workers, it waits till at least one worker has a free slot.

- Object Oriented Programming concepts of classes have been used for encapsulating functions and help avoid the use of global data structures.

  Three classes have been used, which are as follows:

  1. class Request () : This class helps create request/ job objects. These objects contain all job details - jobId, list of map tasks and reduce tasks, start time and time of the job.

     Methods within this class include - scheduling map tasks, checking if all map tasks have been executed and checking if all tasks of a job are complete
  2. class Task () : This class helps create task objects. These objects can be map or reduce tasks. They have the following attributes - jobId (the job to which they belong), t_id (task id), duration, status (to denote if task has completed execution), start time and end time.

     Methods within this class include - checking if task has completed execution and marking the task as completed once it executes.
  3. class Worker () : This class helps create worker objects. These objects contain all worker details - worker id, slots, port, busy slots (slots that have tasks executing in them)

     Methods within this class include - freeing busy slots once the task completes execution and marking the slot as busy once a task is assigned to it.

- Implementation of locks to take care of critical section problems:

  All global lists/variables which are accessed by multiple threads have been secured with locks to avoid simultaneous access amongst the different threads. Our Scheduler contains three critical sections namely -

    - reqList[] - list of all request objects - in Master.py
    - red_tasks[] - list of all reduce tasks that are ready to get scheduled - in Master.py
    - execPool[] - list of all task objects currently being run in a worker - in Worker.py

- CSV files are created to log the time taken per task and job for each of the scheduling algorithms. These log files are later used while analysing the performance of each of the scheduling algorithms.

- For analysis of tasks' and jobs' performances in Analysis.py -
    - Heat Map - Analysis.py takes an argument (LL,RR,RANDOM) to denote the algorithm being analysed and generates a heat map showing the number of tasks running on each worker at every instance of time between the start time of the first map task and end time of the last reduce task.
    - Bar Graph - Analysis.py takes an argument (ALL) and plots a bar graph depicting median job time, mean job time, median task time and mean task time for each of the scheduling algorithms. It shows a contrast to highlight which of the algorithms performs better.

Results

Configuration details:

Number of jobs executed = 75

Number of workers = 3

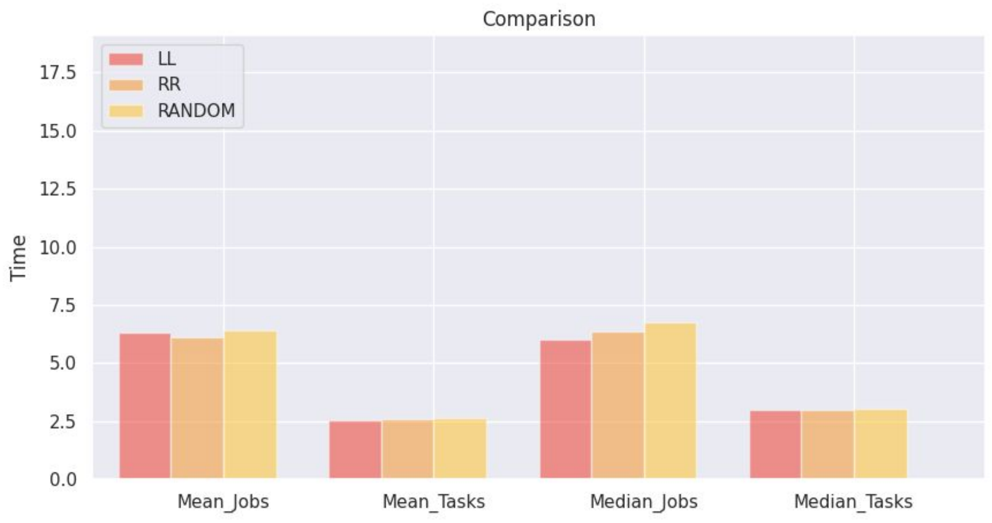
Number of slots on each worker machine:

Worker1 = 5

Worker2 = 7

Worker3 = 3

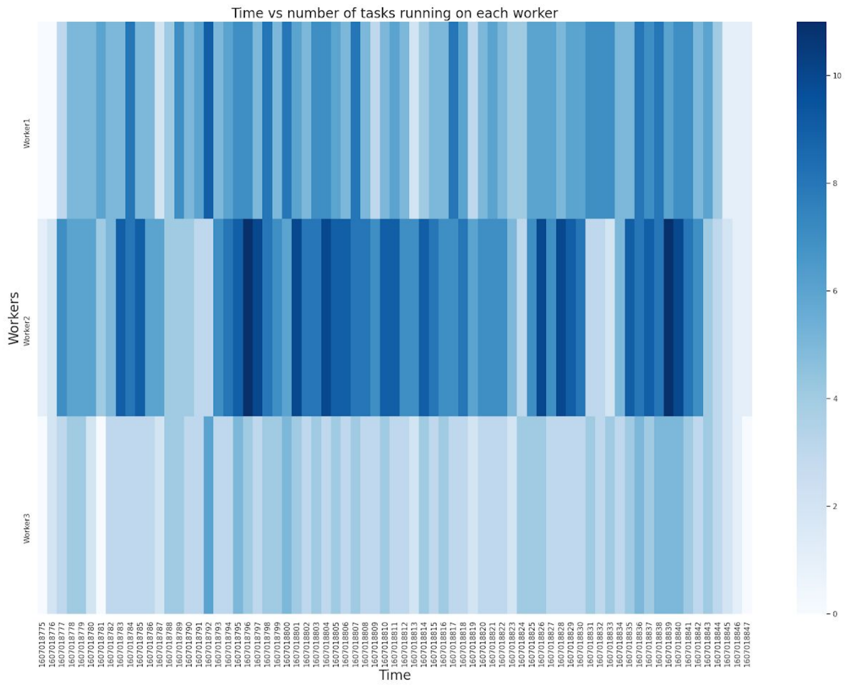
The results of programs gave us the following insights:



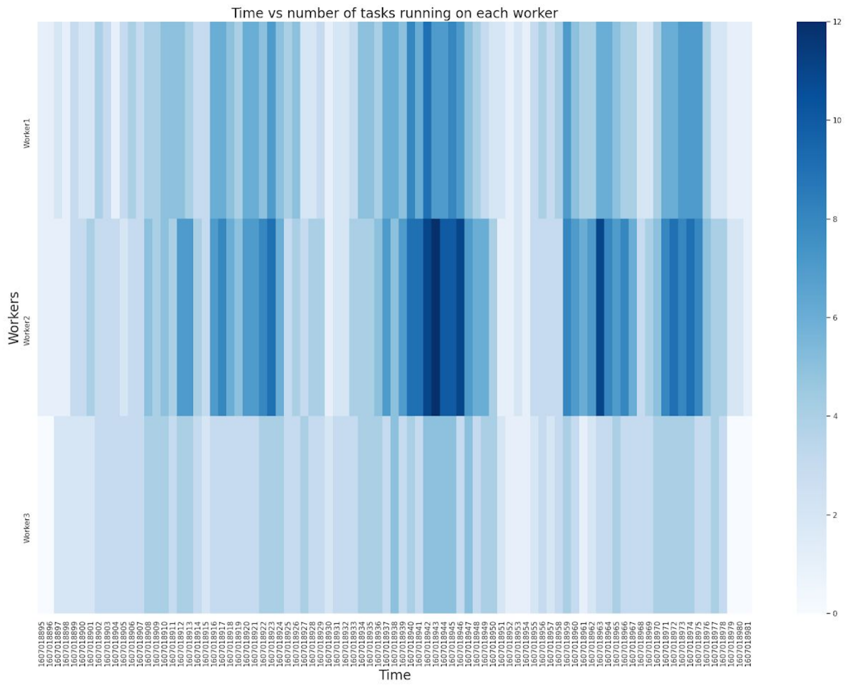
According to the graph, the least-loaded scheduling algorithm seems to run for the least mean time for jobs and tasks, and least median time for tasks as well. Random scheduling seems to be the slowest.

Here are some heat maps showing the number of tasks running on each worker from start to end time:

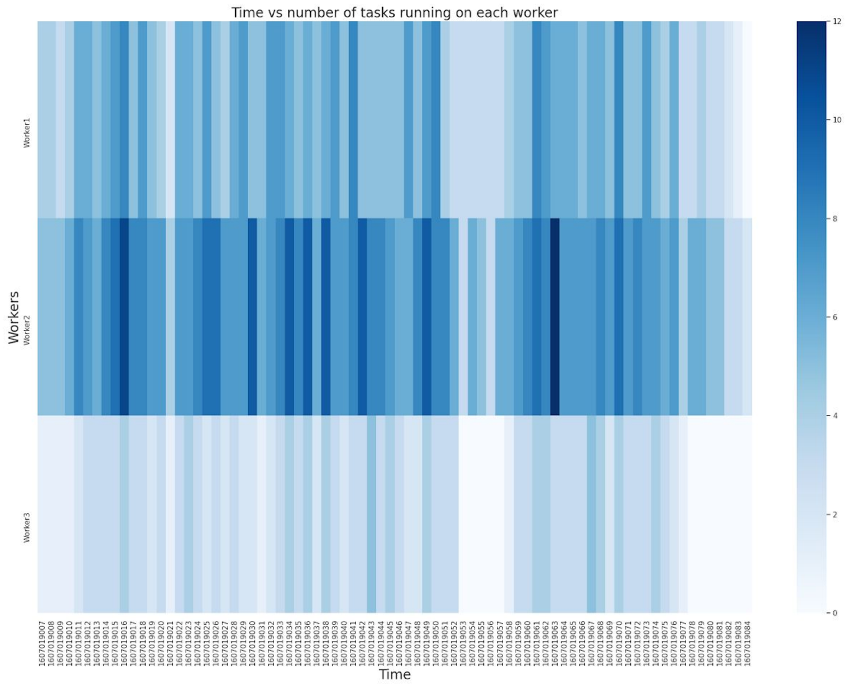
Random:



Round robin:



Least-loaded:



Problems

- Distributing the work appropriately among threads
- Ensuring parallel distribution of work across mappers and the reducer
- Time intervals and delays (and keeping a log of these times).

Conclusion

From this project, we learned how a centralised scheduling framework actually translates from paper to code. Although the problem statement itself is based on Big Data's concepts, the actual coding required core OS concepts and computer networking, which were added bonuses for our learning component in this project. By integrating object oriented programming, space, efficiency and design have been neatly optimised. Scheduling as a whole is a lot clearer and more structured for us to understand now.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Snigdha S Chenjeri	PES1201800045	Object oriented framework, dependency check
2	Sakshi Shetty	PES1201800190	Socket communication, report
3	Ananya V	PES1201800204	Scheduling algorithms, task execution
4	Swanuja Maslekar	PES1201800369	Logs and analysis, report

(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.	