CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): **YACS coding**
Date: 8th December 2020

SNo	Name	SRN	Class/Section
1	DARSHAN D	PES1201801456	C
2	KARAN KUMAR G	PES1201801883	H
3	MAYUR PESHVE	PES1201801439	E
4	MANU M BHAT	PES1201801452	B

Introduction

YACS is a centralized scheduling framework, consisting of one Master, which runs on a dedicated machine and manages the resources of the rest of the machines in the cluster. The other machines in the cluster each have one Worker process running on them.

The Master process receives job requests, which are scheduled on multiple slots across available worker machines. The Master and the Workers communicate to share information about task completion.

In this project, we simulate the functioning of YACS. We run the Master and the Worker processes on the same machine. Task execution is simulated by decrementing times for each of the tasks.

Related work

Any background study material that you may have read and referenced

- Concepts of Socket communication: We referred a few online resources to understand the basic concepts of socket programming in Python, its working and implementation. We referred to the documentation of the **'socket'** python library for syntax to listen and write via specified ports in our program. This was necessary to simulate the communications received and sent by the Master and Workers.

  Link to documentation: https://docs.python.org/3/howto/sockets.html

- Concepts of threading: We thoroughly understood the basic concepts about threads and their execution in an Operating System. We understood the need for threads to simulate the execution of multiple operations on a single machine. We referred to the **'threading'** python library via the official documentation. We also had to figure out the prevention of race conditions using the concept of thread locks which were available in the aforementioned library itself.

  Link to documentation: https://docs.python.org/3/library/threading.html

- Concepts of logging: We discovered and utilised the **'logging'** python library to carry out the logging process. Instead of manually writing text into a file, we

made use of the functionalities provided in this library to make a clean and informative log file.

Link to documentation: https://docs.python.org/3/howto/logging.html

- Concepts of Signals in Operating System: We used the **'signal'** library in python to execute signals to successfully close the sockets and end all processes completely.

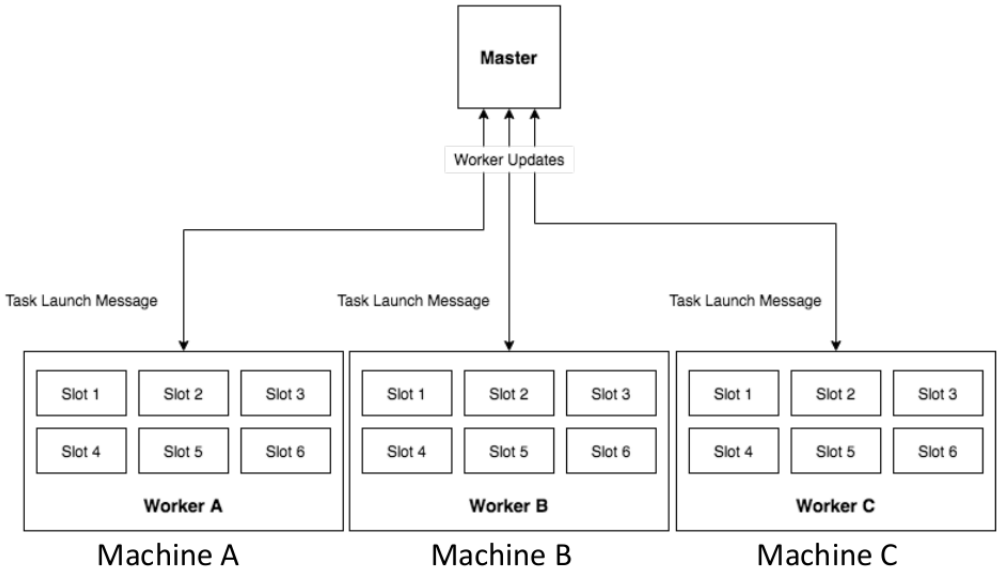
  Link to documentation: https://docs.python.org/3/library/signal.html

Apart from these python libraries, we did not have to do any other background study. The concepts of scheduling and Master/Worker flow of working was already known.

Design

Talk about the design of the system, algorithms used, and models implemented. Block diagrams are preferred wherever applicable.

There are two python programs - master.py and worker.py which perform the functions of the Master and a Worker machine respectively. These programs are independent of each other and can run on separate machines too if required.



master.py :

This program essentially consists of 4 separate threads to carry out the following tasks:

1. JobListener: Listens to job requests
2. JobScheduler: Starts a job and schedules its map tasks on the workers
3. ReduceTaskScheduler: Schedules the reduce tasks on the workers
4. WorkerManager: Listens to job completion updates from the workers

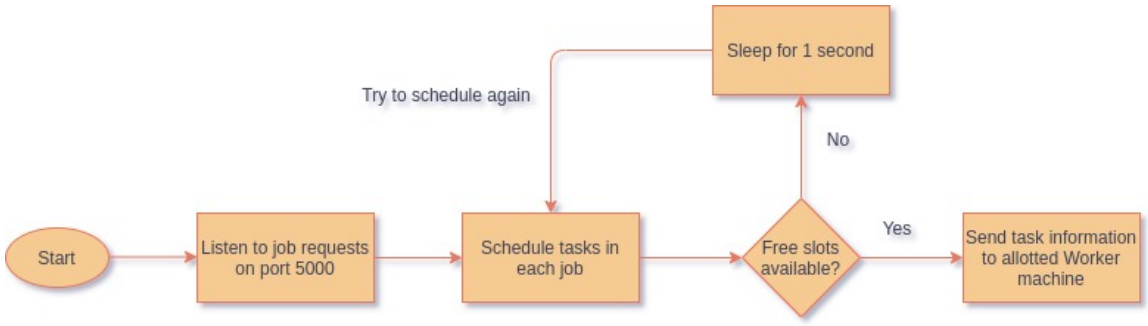
It also contains the implementation of the 3 scheduling algorithms (Random, Round-Robin and Least Loaded).

Inside the JobListener, it does the following tasks in order :

- Listen to incoming job requests through specified port (in this case, port 5000)
- Initially append only the map tasks into the Map tasks queue.

Inside the JobScheduler, it does the following tasks in order :

- Run the scheduler function to determine workers to allot map tasks to. Then send the current task information to the allotted Worker.
- Update new allotment information by calling Scheduler function again. Repeat until all map tasks are allotted to Workers.
- If there are no free slots available in any of the workers (as returned from the Scheduler function), make the Master wait for 1 second, before it again tries to find if any slots become free.



Inside the ReduceTaskScheduler, it does the following tasks in order :

- Run the scheduler function to determine workers to allot reduce tasks to. Then send the current task information to the allotted Worker.
- Update new allotment information by calling Scheduler function again. Repeat until all the reduce tasks are allotted to Workers.

- If there are no free slots available in any of the workers (as returned from the Scheduler function), make the Master wait for 1 second, before it again tries to find if any slots become free.

The WorkerManager thread does the following tasks in order :

- Listen to workers regarding updates about task completions. Update related information accordingly (increase free slot count, reduce the count of tasks to be allocated)
- If all the map tasks of a job are completed, only then append the corresponding reduce tasks of the same job to the reduce tasks queue, which can then be scheduled. This way we preserve the task dependency criteria.
- If the final reduce task is finished, then that particular job is completed and appropriately logged.

All the important events with corresponding timestamps logged are as follows :

- Binding sockets to setup communication between master and job-requests-sender, master and worker
- Scheduler type for analysis reference
- When no slots in worker machines are free
- Starting a new job
- Sending and receiving task information to and from Workers
- Ending a job

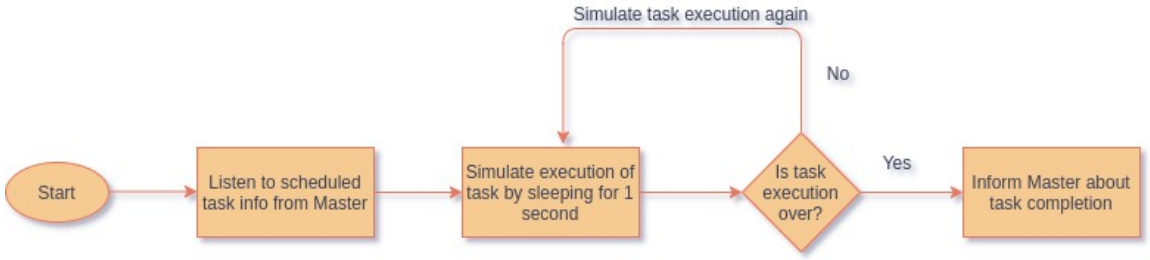
worker.py :

This program simulates the working of Worker machines (each one of them). The same program is run multiple times simultaneously to simulate the simultaneous running of multiple Worker machines.

This program essentially listens to the job allocation information from the master (on one thread, with a specified port number), simulates the execution of these tasks by decrementing the task duration and sleeping for 1 second (on a second thread). Meanwhile, it continues to listen to the Master (on the other thread), while each of the slots continue executing their tasks.

Each of the workers' logs events into its log file. It logs the following information :

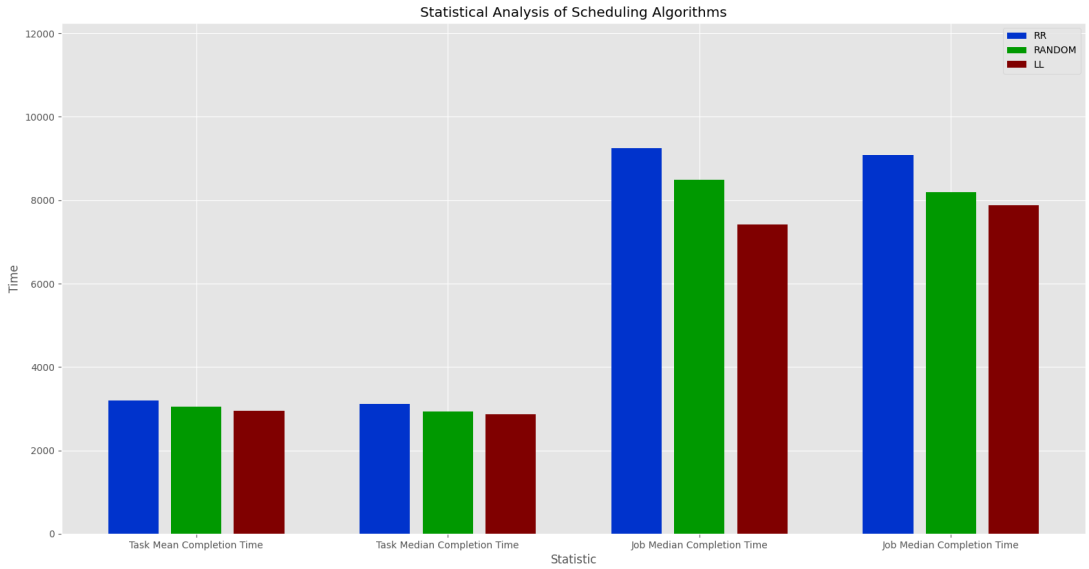
- Starting a map/reduce task and recording the timestamp
- Ending a map/reduce task and recording the timestamp



Results

Discuss the results you got. What inferences could you draw from the results? Was any result unexpected? Any fine-tuning done to parameters so that the results changed?

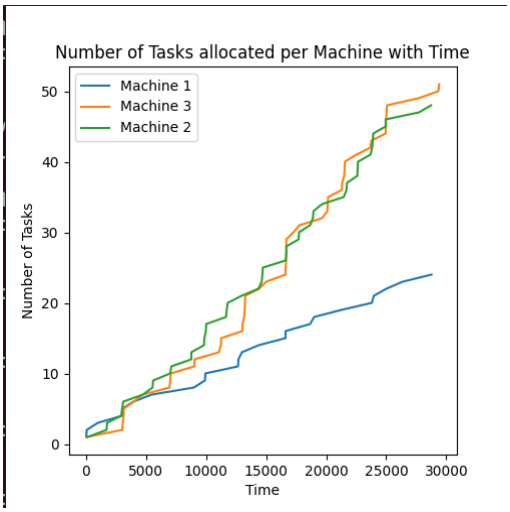
The task and job execution times shed light on the type of scheduling algorithm used. These tests were run with 3 worker machines with 3, 7 and 9 slots respectively, for 30 job requests. Below are the mean and median job and task completion times :



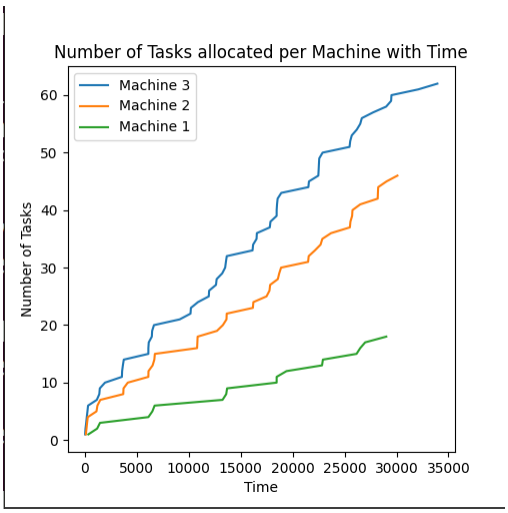
Following are the observations :

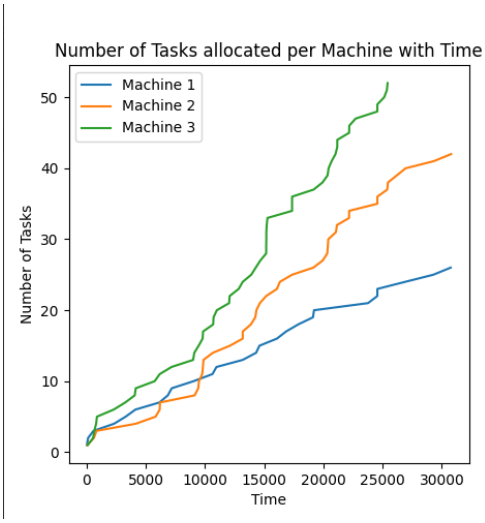
- Least Loaded scheduling algorithm has the least mean job completion time.
- The job completion times for Random scheduling vary, highlighting the random nature of this type of scheduling
- The Round Robin scheduling algorithm generally takes the longest time to finish job execution.
- Task execution times are determined in the requests.py and are as exactly as they are sent over the jobs requests. As was specified, all tasks have their execution times between 0 and 5. The mean lies in this range as can be seen in the graphs. Thus this is justified.

Random Scheduling



- Worker 3 with the highest number of slots(9) executed the most number of tasks, followed by Worker 2(7 slots) and Worker 1(3 slots)



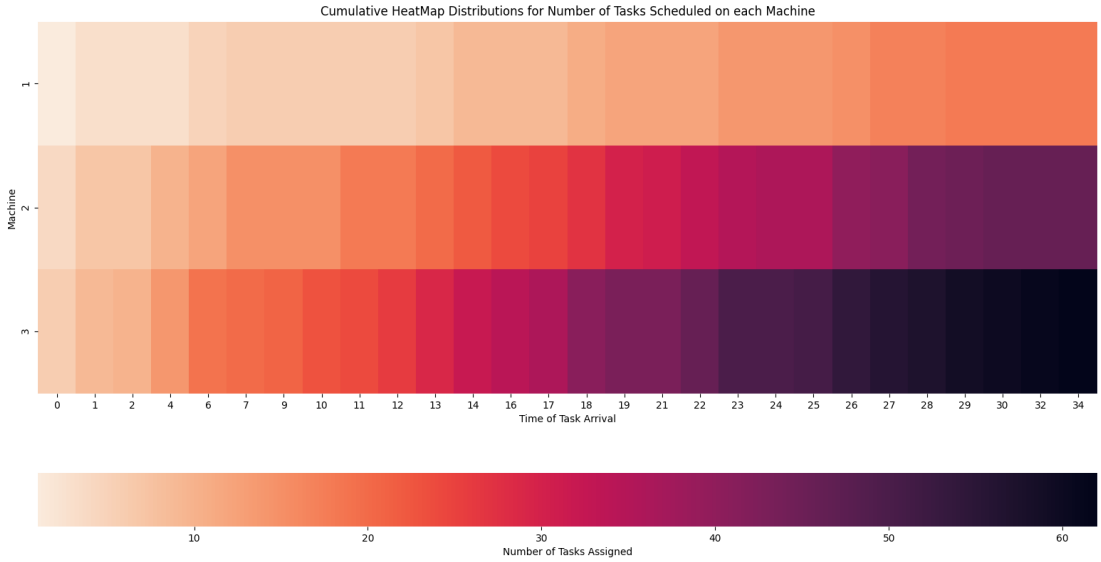


Least Loaded Round Robin

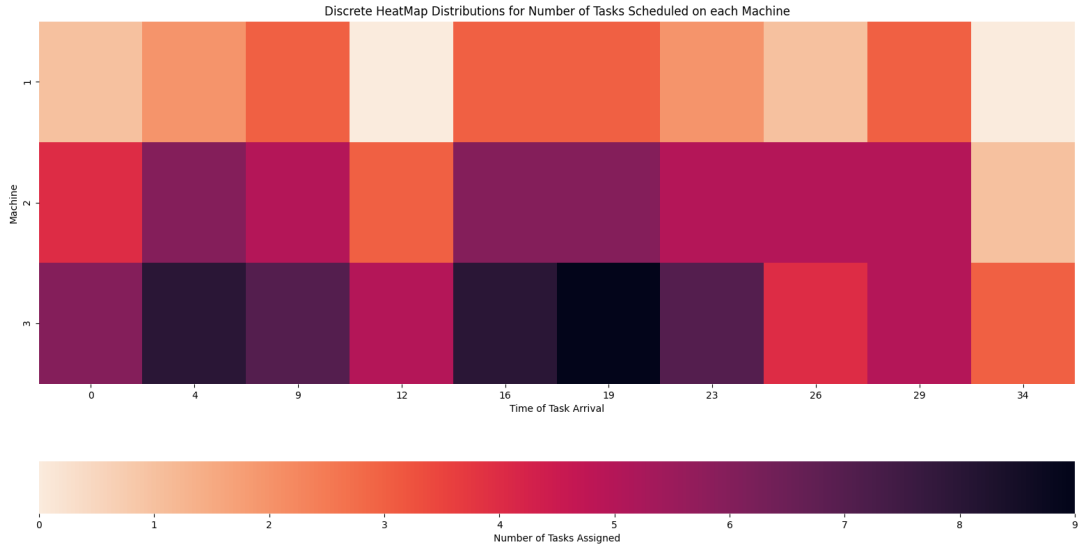
- In Least Loaded, tasks are first allotted to Worker 3, followed by Worker 2 and then Worker 1. Eventually, worker 3 has the most tasks allotted and worker 1, the least.

- The Round-Robin scheduling graph is very similar to that of Random scheduling. For a large number of tasks, Round-Robin technique and Random scheduling become very similar.

Least Loaded :

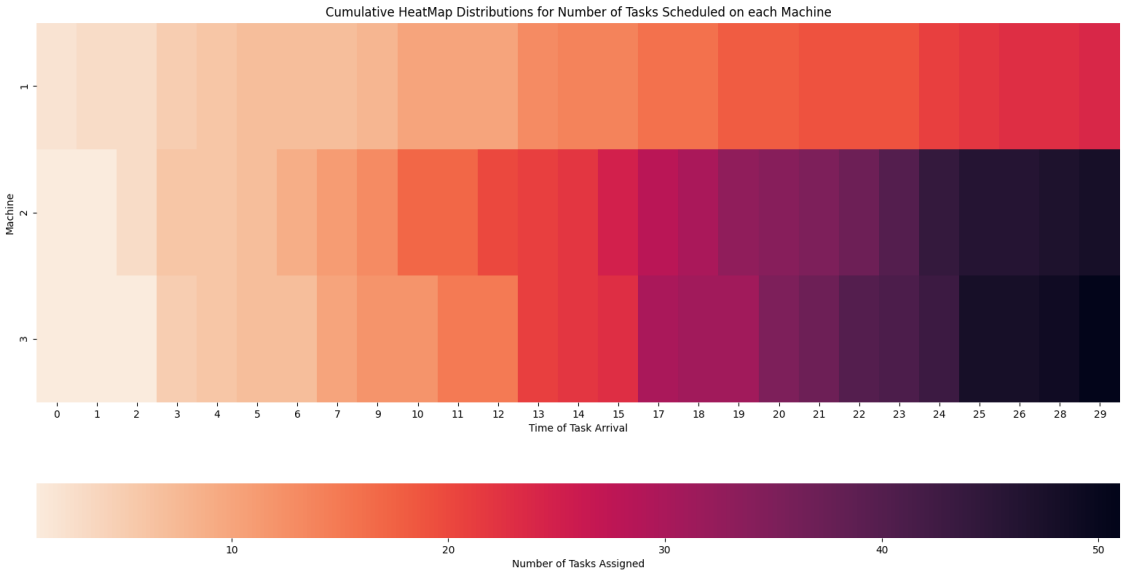


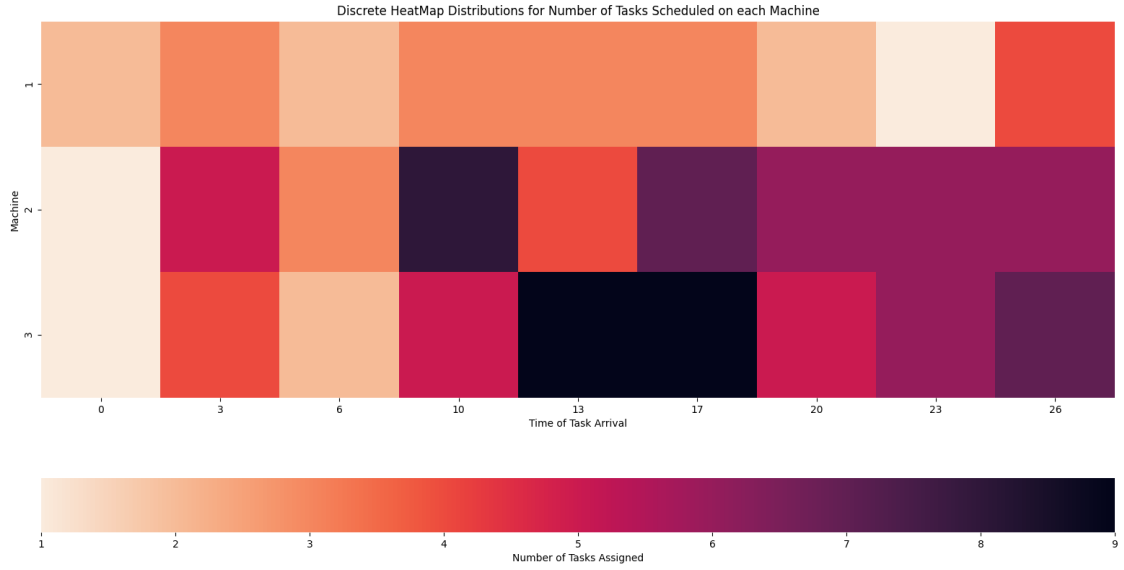
Observation: Machine 3 becomes darker first then 2 followed by 1. The time at which 2 starts becoming darker is when 3 and 2 have the same number of tasks.



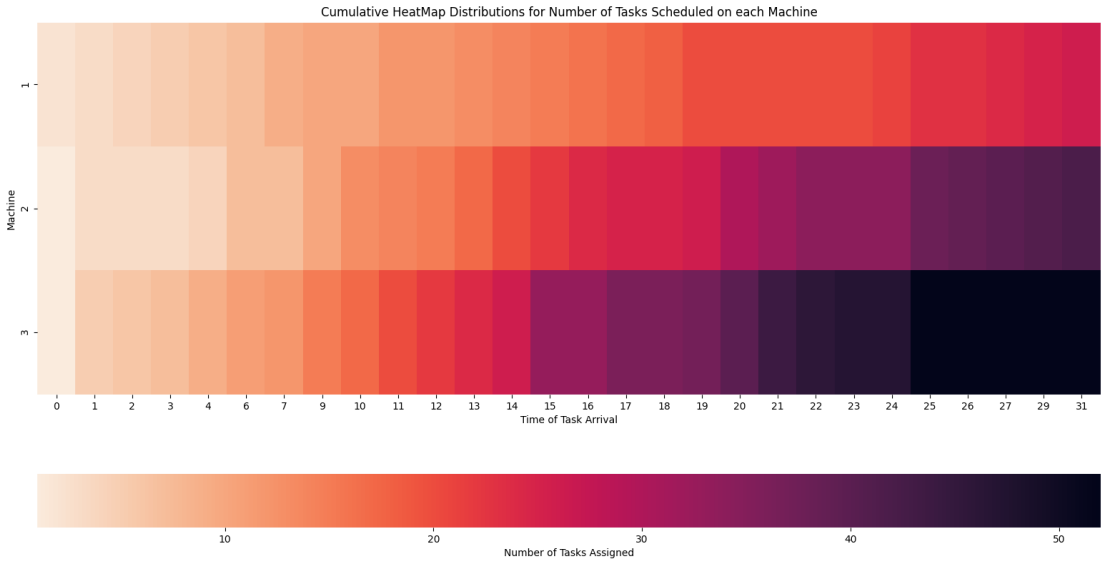
The shades are usually always descending with respect to the darkness of the shade. Few exceptions are because of logging and plotting at discrete time values and not continuous values.

Random :

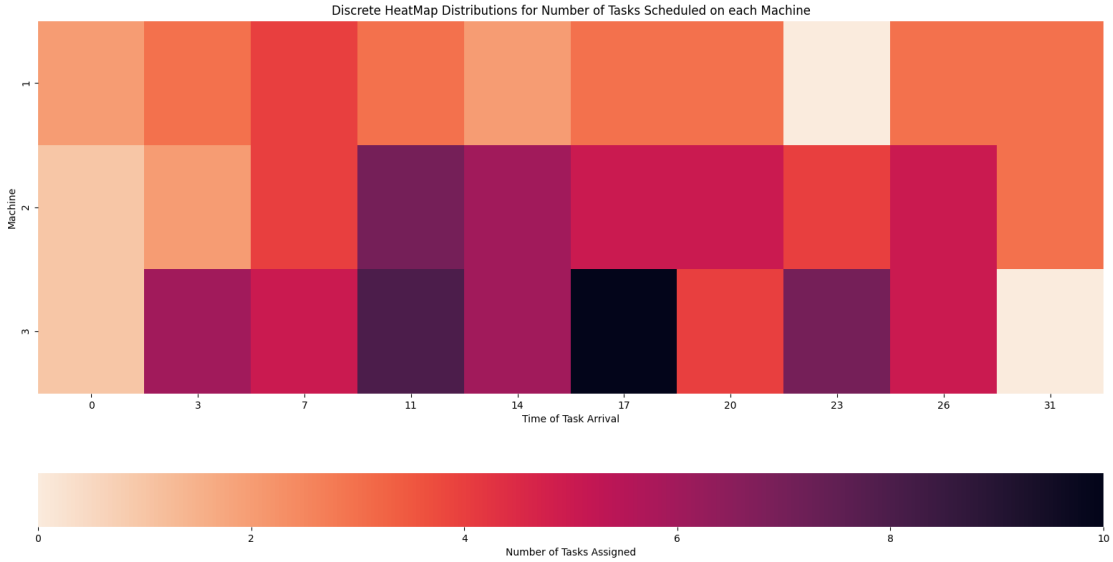




Round Robin :



Observation: All machines become darker simultaneously at the same pace but later differ due to differing number of slots



Usually, all colour shades are similar as all machines get an equal chance. the difference is because of the different maximum number of slots

Problems

Mention problems faced and how were they solved

- We faced some issues with respect to the design of our model. Initially, we implemented just two threads to implement the master.py. However later we realised the requirement for 4 different threads to separately carry out different tasks so as to increase the efficiency and the throughput of our design.
- We had to put thread-locks specifically for only those variables which were shared between threads, and not entire functions implementing the threads.
- Sockets by default would allow only one simultaneous connection while listening. We had to increase this to match the number of workers and slots we had and ensure no information or packets are lost due to bounded buffering.

Conclusion

What was your main learning from this project?

- Scheduling algorithms
- Working with Sockets and Threads
- Handling the critical section problem while working with multiple threads
- Importance of distribution of workflow into multiple machines, especially in cases of handling very large quantities of data and the necessity of a Master-Worker architecture to implement this execution.
- Nature of job handling by the Master and ways to efficiently implement this.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	DARSHAN D	PES1201801456	10
2	KARAN KUMAR G	PES1201801883	10
3	MAYUR PESHVE	PES1201801439	10
4	MANU M BHAT	PES1201801452	10

(Leave this for the faculty)

Date	Evaluator	Comments	Score

CHECKLIST:

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