CS322:Big Data

Final Class Project Report

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

SNo	Name	SRN	Class/Section
1	Gaurika Poplai	PES1201800374	5 I
2	Keerthana Mahadev	PES1201800768	5 E
3	Nikita J. Raj	PES1201800808	5 H
4	Rahul KR	PES1201802064	5 J

Introduction:

- The aim of this project was to simulate a centralized job scheduler with 1 master process and 3 worker process (forming a cluster). The master process runs on one machine while the other 3 workers work on 3 different machines. Each of the worker machines have fixed no. of slots depending on the machine's capacity.
- Each job is divided into multiple tasks (map and reduce tasks), such that each task can be executed on one of the slots completely. Map tasks are executed first followed by the reduce tasks.
- The master process manages the cluster by keeping track of all the slots in each worker machine. It processes the job requests and assigns the tasks to one of the workers with an available slot, based on the chosen scheduling algorithm.
- The worker process running on each worker machine is responsible for starting the execution of the tasks in their respective slots by allocating resources and freeing the resources once the task is complete. It's also responsible for sending updates to the worker once the task is completed.

Workflow:

Master has 2 threads:

1) One thread listens for job requests through port 5000
2) Another thread to listen for updates (on task completion) from workers through port 5001

Master can schedule tasks using the various scheduling algorithms:

1) Least loaded:

   Master node checks the no. of available slots in each machine and picks the one with the highest no. of free slots for task launch

2) Random:

   Master node selects a machine at random to launch a task. If the machine has no free slots, selects another machine at random.

3) Round Robin:

   Worker machines are ordered based on their worker_id. Round robin method is used to pick a worker node for task launch. If the selected worker has no free slots, the next worker in ordering is picked.

Worker has the following threads:

1) One thread to listen for task launch messages from master through port specified in the config file.
2) Another thread to start task executions in their respective slots and send updates to the master
3) The worker starts a new thread for each task that is to be run.

Related work (basic references used):

- https://realpython.com/intro-to-python-threading/#starting-a-thread To understand threading concept and implementation in python
- https://realpython.com/python-sockets/ To get a handle on socket connection in python

Design:

Priority Queue

- Implemented to store tasks - ongoing as well as pending. When a task finishes execution, it is removed from the queue.
- Initially, each job is assigned a priority of 1. The job with highest priority is picked. Provided there is a mapper task pending in the job, the mapper task will be sent to a worker for execution. Ongoing mapper tasks are stored in a list and tracked until completion.
- If there are no mapper tasks left in the job, and the mapper tasks of said job have *completed* execution, the priority of the job is increased to 2, so that the reducers of that job will be picked for execution next.
- Following the completion of all reducer tasks in a job, the job is deleted from the priority queue.

master.py

- Continuously listens for incoming job requests on port no. 5000, in a function called listen_to_requests. This function handles insertion into the priority queue.
- Each scheduling algorithm is implemented in a separate function, the algorithm function called is dependent on the command line argument passed. The algorithm then assigns tasks by calling the function send_tasks, which sends a message to the workers specifying the job number and task to be run.
- The function listen_to_workers listens on port 5001, for messages from the workers which signify the completion of a particular task.
- Implements multithreading, with two threads running in parallel: One thread listening for job requests, the other thread listening for messages from the workers.

worker.py

- The function listen_to_master continuously listens (on port number 4000, 4001,4002, for worker 1,worker 2,worker 3 respectively) to the master for tasks assigned, creating a separate thread for the execution of each task.
- Each thread calls the send_update function, where time.sleep() is called for the specified task duration. Following the simulated execution of the task, a message is sent to the master, specifying that the task is complete.

Logging

- In the listen_to_requests function, we log the timestamp upon arrival of a task.
- In the send_update function, we note the timestamp before and after the sleep function call, to track the duration of the task. These timestamps are sent as a message to master.py, which then logs these timestamps into a separate log file. This prevents overwriting by various threads and processes.

Analysis

Reading through the log file line-by-line, we collect the necessary data to plot a bar graph and heatmap.

1) Bar Graph (task 2a):

- Using regex, we detect whether the line is about a task arrival, starting task, or ending task. We store starting time, ending time, task arrival in dictionaries, with their id and timestamp stored as key-value pairs.
- We have a list storing the duration of each task, using these to calculate the mean and median of these runtimes.
- For job arrival, we maintain two dictionaries, one storing the job id and a list of starting times for each task in that job as a key-value pair, the other storing the job id and a list of ending times for each task in that job as a key-value pair.
- The starting time dictionary is sorted in ascending order, the ending time dictionary in descending order. For each list, we take the difference between the first value in the list, this gives us the job duration for that job. We perform this computation for each job, thereby acquiring the necessary data to plot the bar graph for job mean and median.

2) Heatmap (task 2b)

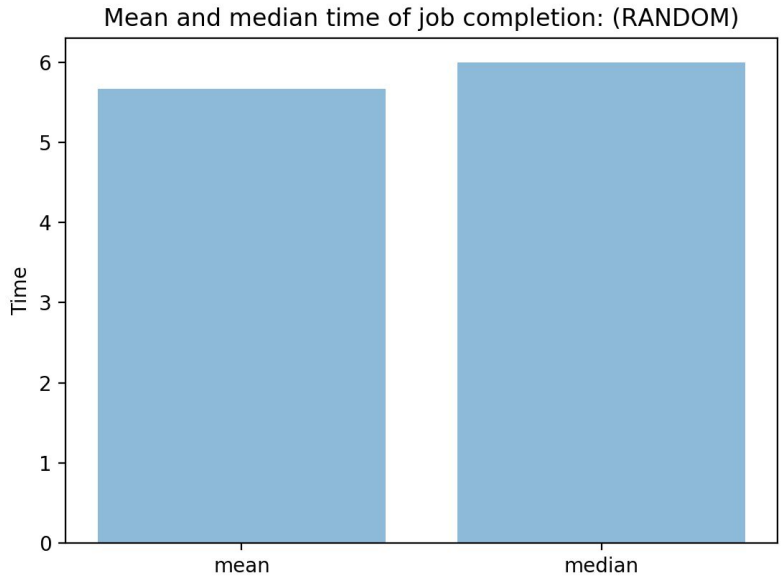
- This graph represents the number of tasks running on each worker, plotted by the runtime.
- Sorting the log file by timestamp, the heatmap data is collected by keeping counters w1,w2,w3 for each worker, to store the number of tasks running on the workers at a given time.
- Using regex, we detect whether the line is about a task arrival, starting task, or ending task.
- When the line is about a task starting, the worker id is extracted from the line, and the worker task count is incremented by 1.
- When the line is about a task ending, the worker task count is decremented by 1.
- Whenever there is a change in timestamp from the previous line, an entry is made into the dataset for heatmap, containing information about the time, worker id, and task count for each worker.
- Upon reading the entire log file, the dataset is plotted as a heatmap with time on the x-axis, workers on the y-axis, and no. of tasks represented by colour.

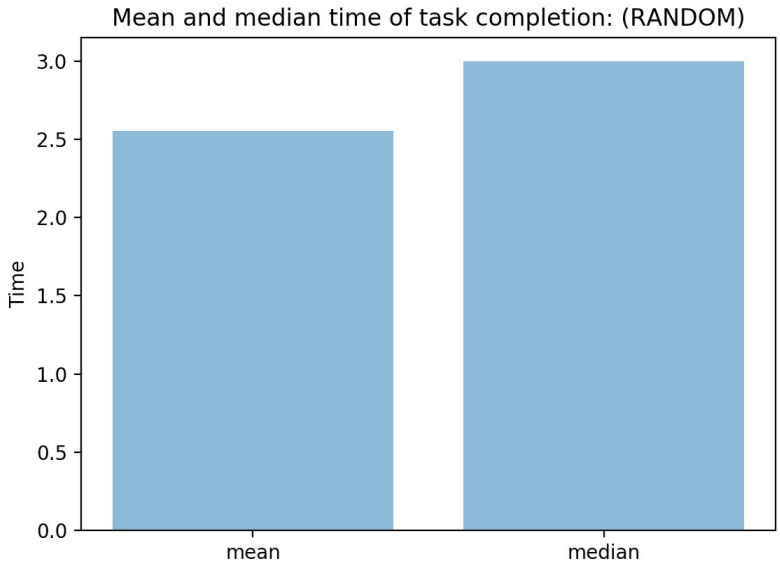
Results - Plotting and analysis:

Task 2a) Bar graphs

Random algorithm

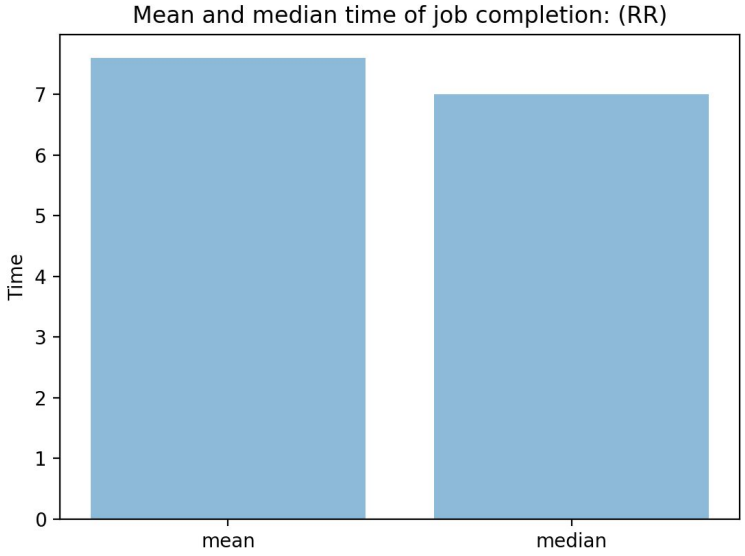
```
Number of tasks started:  9
Number of tasks ended:  9
Mean of jobs:  5.666666666666667
Median of jobs:  6.0
Mean of tasks:  2.5555555555555554
Median of tasks:  3.0
```

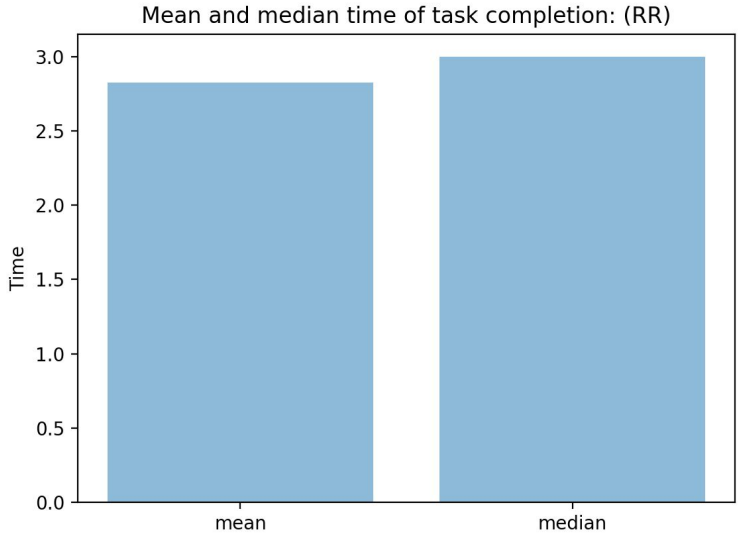




Round Robin Scheduling Algorithm

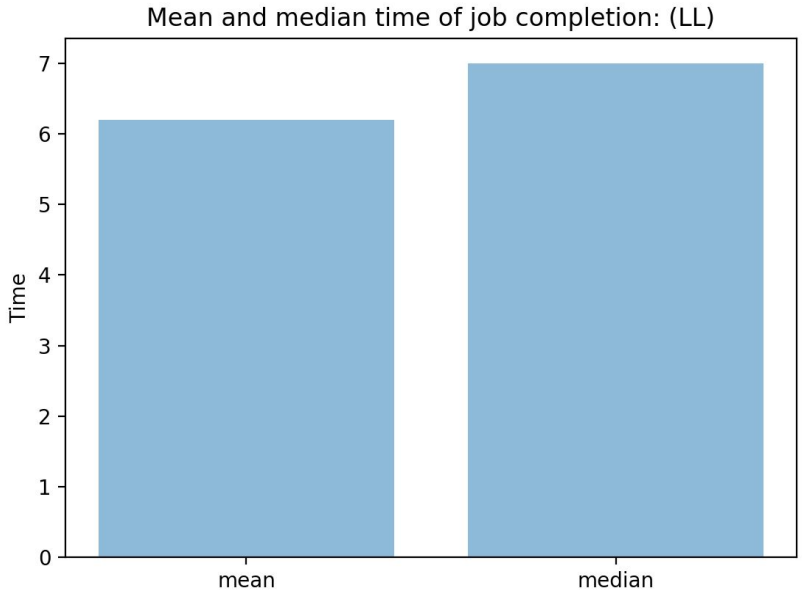
```
Number of tasks started:  23
Number of tasks ended:  23
Mean of jobs:  7.6
Median of jobs:  7.0
Mean of tasks:  2.8260869565217392
Median of tasks:  3.0
```

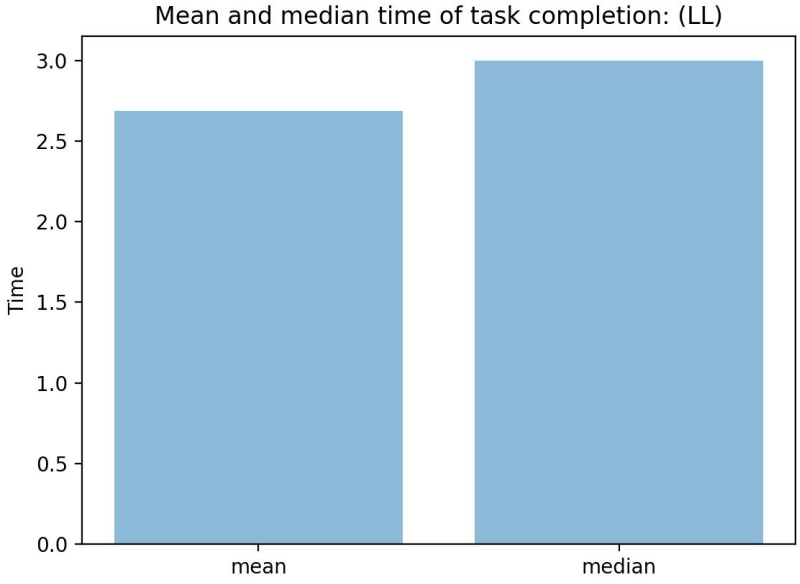




Least Loaded Scheduling Algorithm

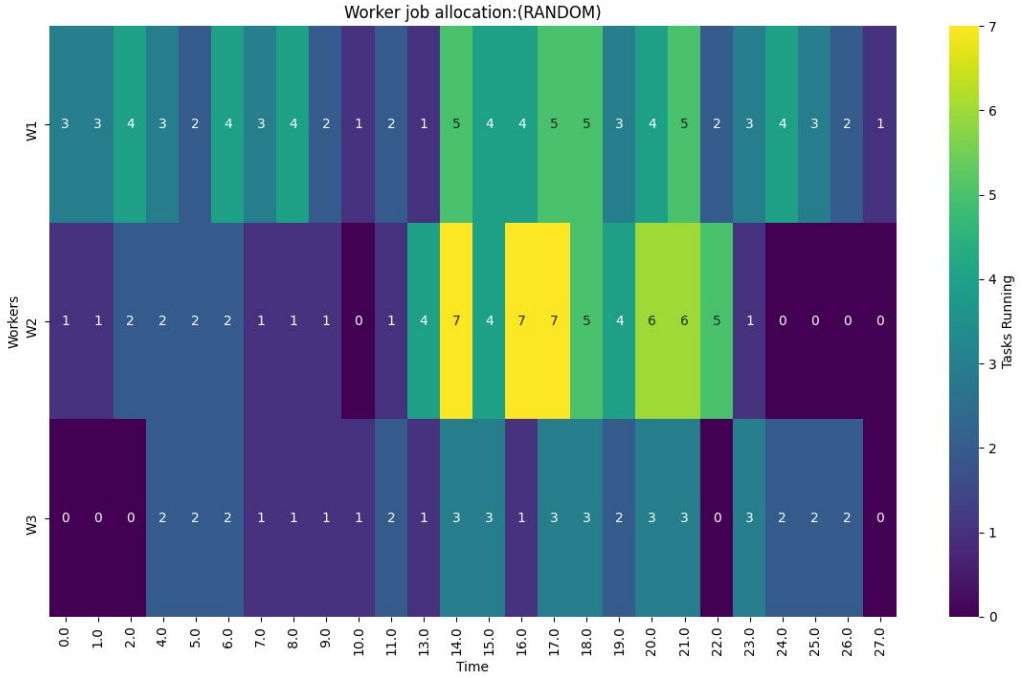
```
Number of tasks started:  16
Number of tasks ended:  16
Mean of jobs:  6.2
Median of jobs:  7.0
Mean of tasks:  2.6875
Median of tasks:  3.0
```



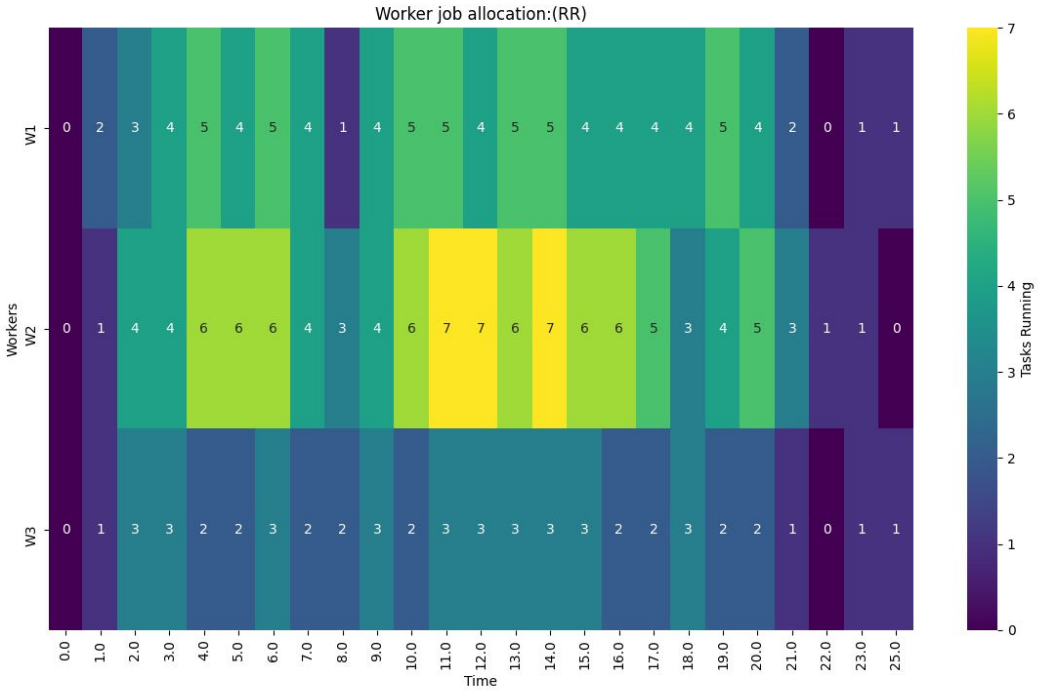


Task 2b) Heatmaps:

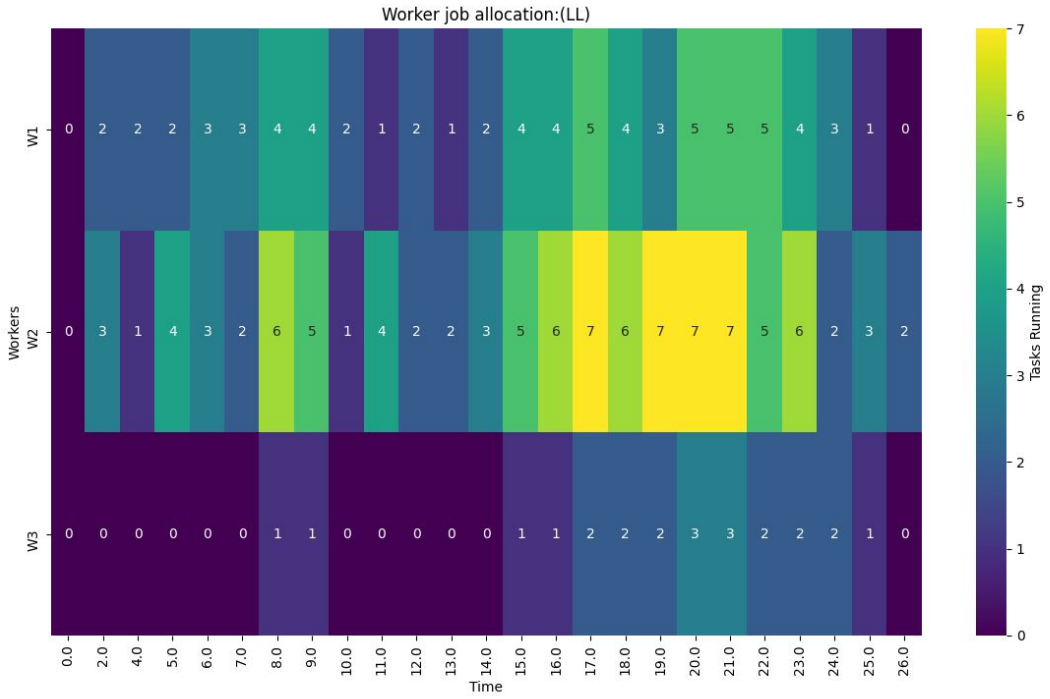
Random Scheduling Algorithm



Round Robin Scheduling Algorithm



Least Loaded Scheduling Algorithm



Observations :

The bar graphs demonstrate the efficiency of the workers per task and job.

- Due to the uniform distribution of **Random** scheduling, task allocation appears to often find a free worker and therefore completion time is marginally quicker.
- In **Round Robin** scheduling, the third worker, with the default configuration, has the fewest number of slots and therefore is loaded very quickly, leaving two free workers for a good portion of the runtime. This could potentially be contributing to a slightly longer overall runtime.
- **Least Loaded** scheduling works faster than RR, for the given test set. Particularly in the initial stages, the tasks will be assigned to a free worker and will run at a smooth pace. No worker will be completely filled early on, owing to the design of this algorithm.

The heatmaps above depict accurate working of the respective scheduling algorithms.

- **Random** scheduling has resulted in a more-or-less uniform distribution of tasks, for the given test set of requests.
- In the case of **Round Robin** scheduling, the number of tasks per worker holds steady until such a time as one or more of the workers have completely occupied slots, in which case the free workers get assigned more tasks.
- The graph for **Least Loaded** scheduling clearly demonstrates the working of the algorithm, with worker 2 being assigned the most tasks, having the most number of slots in it. In keeping with this, worker 3 receives very few tasks, as it has the fewest number of slots and is often less free than workers 1 and 2.

Problems:

- Deciding on an efficient data structure for requests, as well as tasks
- Encountering connection reset by peer error, immediately after each worker completed a task. To solve this, we decided to run each task on a separate thread, and implemented a separate send_message function to keep track of sockets.
- Thinking along the lines of multithreading, monitoring the activities of each thread to ensure no overwriting of data.
- Logging into a single file, while preventing communication delay due to sockets.
- Handling deadlock while modifying priority queue.

Conclusion:

Main learning from this project:

- Understood master-worker architecture and its implementation.
- Improved understanding of socket connections.
- Proper implementation of multithreading, giving insight into its widespread applications.
- Comparing the working of different job scheduling algorithms, their pros & cons.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Gaurika Poplai	PES1201800374	worker.py, logging, data collection and plotting of bar graph, listening for requests.
2	Keerthana Mahadev	PES1201800768	multithreading implementation, handling socket connections, worker.py
3	Nikita J. Raj	PES1201800808	data structures for requests & priority queue, implementation of algorithms, data collection,plotting of heatmap
4	Rahul KR	PES1201802064	master.py - locking, implementation of algorithms, priority queue

(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.	