CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): YACS Coding **Date:** _____

SNo	Name	SRN	Class/Section
1	M S Akshatha Laxmi	PES1201800130	5th Sem G
2	Abhishek Das	PES1201800177	5th Sem G
3	Bhargav SNV	PES1201800308	5th Sem I
4	N Sanketh Reddy	PES1201800389	5th Sem C

Introduction

Yet Another Centralized Scheduler (YACS) is a scheduling framework that can schedule tasks using 3 different scheduling algorithms – random, round robin and least loaded. The framework has one master which manages the resources of the rest of the cluster which consists of worker machines. Each worker machine has a worker process and a fixed number of slots that execute tasks and send updates about the completion of the task to the master. The framework requires a configuration file which consists of the details of each of the worker machines that is required to form a connection from the master to the worker and also schedule tasks.

For this project, all the processes, i.e., master and worker processes run on the same machine. This behaves as a simulation of the working of YACS. It can also be used in a real distributed environment by providing each worker machine's IP address in the configuration file.

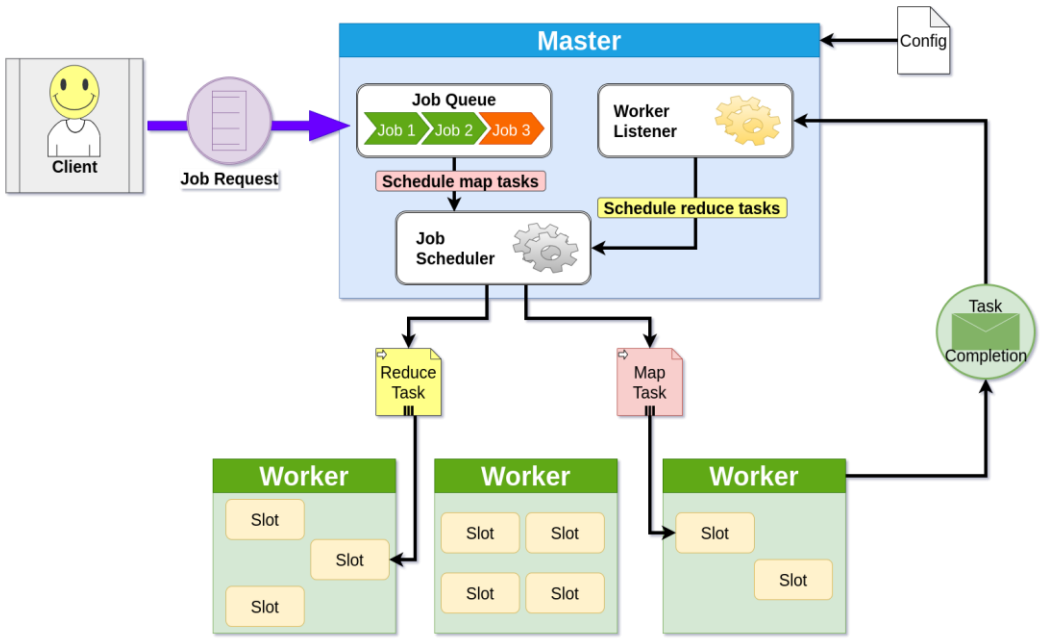
Related work

https://www.slideshare.net/Hadoop_Summit/w-525hall1shenv2 - This link contained a presentation of the architecture of YARN.

https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html - Official documentation of Apache Hadoop was used to understand more about YARN.

Design

Architecture



Master

The master process is a multithreaded process. There are 3 main threads running – client listener thread, worker listener thread and job scheduler thread. Functions of these threads are explained below:

- **Client Listener**: This runs in the master process and listens for job requests sent by our "happy" client as depicted in the diagram above. On receiving the job request, it parses the request, and adds the job to the job queue.

- **Job Scheduler**: This thread picks out jobs from the job queue and schedules the map tasks of the job. It uses the specified scheduling algorithm in the scheduling process and blocks scheduling if there are no empty slots available in the cluster. A helper function is used to schedule tasks of specified task type (map or reduce) from a job. This is done to maintain dependency integrity. Additionally, this scheduling helper function is invoked by the worker listener too, which is explained in the next point.

- **Worker Listener**: This thread listens for responses from the master (task completions). This in turn creates multiple threads to accommodate parallel communications from various workers. On receiving updates, it updates all the concerned metadata on the master. Additionally, it performs a dependency check. The dependency check is such that if a map task is completed and is the last map task of a job, the reduce tasks of that job can now be scheduled. Thus, if the dependency check passes, the scheduling helper is invoked to schedule all reduce tasks of the given job.

  In the case where the final reduce task of a job is completed, the job is then considered complete and metadata is updated accordingly.

Concurrency safety mechanisms: As both the master and worker are multi-threaded, safety mechanism are required to handle parallel execution, these are implemented using mutexes and semaphores. We maintain 4 mutexes and 2 semaphores to avoid race conditions. These are explained further in detail below:

- `task_mutex` : This is used when we read/write from the task dependency data structures

- `stats_mutex`: This is used to update metadata pertaining to workers (slots used, slots available, etc.)

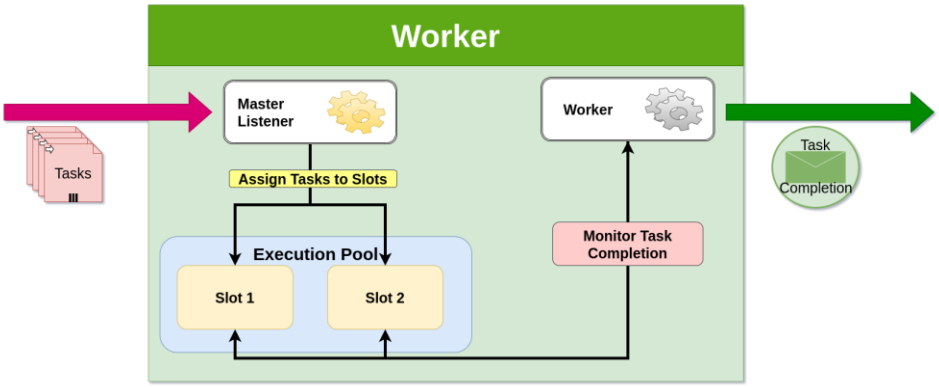
- `queue_mutex`: This is used when accessing the job queue, either while adding jobs to the queue or removing them from it.

- `reduce_mutex`: This is used to avoid redundant scheduling of reduce tasks.

- `has_empty_slots`: This is a semaphore which helps in maintaining the count of empty slots across the cluster, in the case where it is 0 (no free slots), it blocks scheduling.

- `has_jobs`: This is a semaphore which blocks execution when there are no jobs left in the job queue.

Worker



Much like the master, the worker too is a multi-threaded process. As depicted from the image above, it has 2 main threads: Master listener and Worker. A key thing to note here, is that each sot is abstracted as a thread too. As tasks come in, these threads get to work by simulating the execution of the task for a specified duration. The functions of these threads in the worker are further explained below:

- **Master Listener**: This thread constantly listens for tasks assigned to the worker by the master. On receiving a task, it schedules it on one of the slots (also threads) for execution.

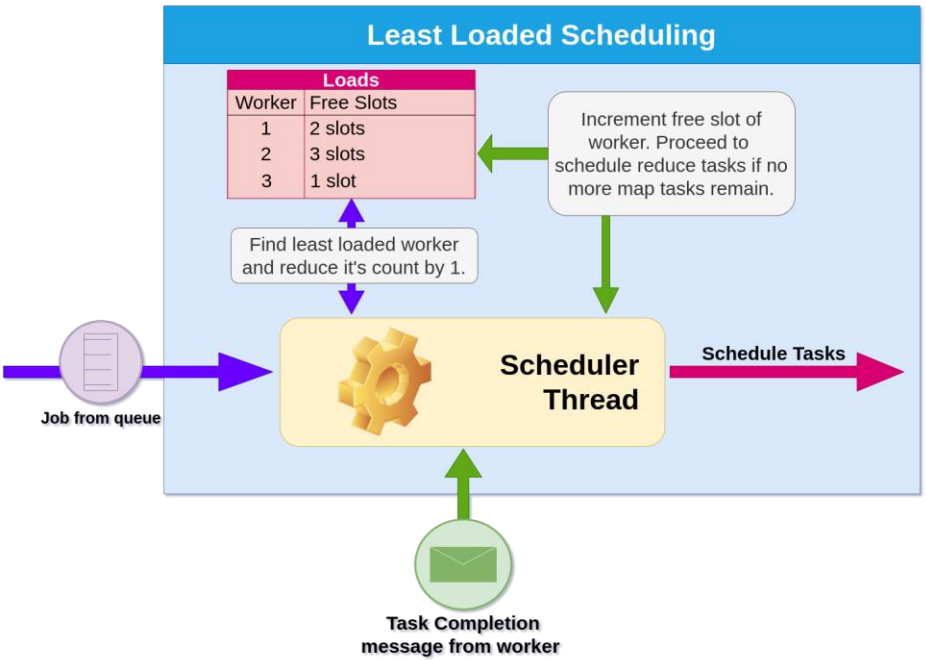
- **Slots**: These slots simulate the execution of a task for a specified duration. Once this execution has finished, task completion messages are sent to the master.

- **Worker**: This thread keeps track of the threads running as slots. It ensures none of them end abruptly and on completion, clears these threads from the assigned slot.

For enhanced functionality, workers need not be spawned individually, when the configuration file is fed to the master. The master automatically spawns child processes as workers based on the configurations provided. In case of a distributed environment, instead of creating child processes, the master can spawn processes over the network using SSH (provided SSH functionality has been set up).

Scheduler Architectures

Least Loaded Scheduling



Here, the scheduler can get inputs from two places:

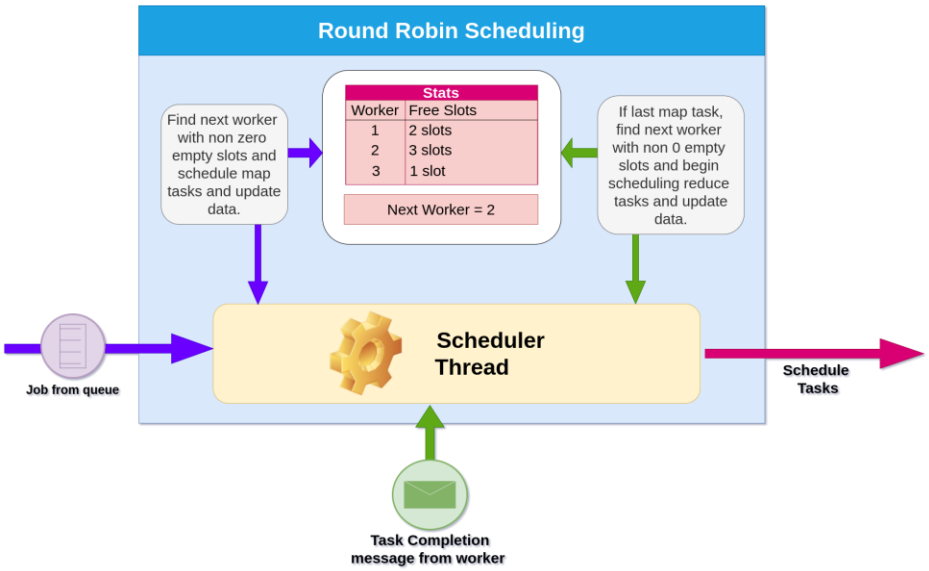
1. The job queue scheduling all map tasks of a job.
2. Response from worker, allowing reduce tasks to begin.

Job scheduled from job queue: In this case, no dependency check needs to be performed. All that is necessary, is for the scheduler to schedule all map tasks of a given job across the cluster. In order to do this, the scheduler iterates over available map tasks and finds the least loaded worker for each iteration and assigns the map task to that worker. After assigning a task, metadata is updated (free slots for workers).

Job scheduled after response: Here, a dependency check is performed by the worker listener before sending the reduce tasks to be scheduled. Scheduling is still done the same way as mentioned above. The only difference being, the scheduler now iterates over reduce tasks of a specified job rather than map tasks.

In the event where no slots are available when trying to schedule map tasks, scheduling is blocked till at least 1 empty slot exists.

Round Robin Scheduling

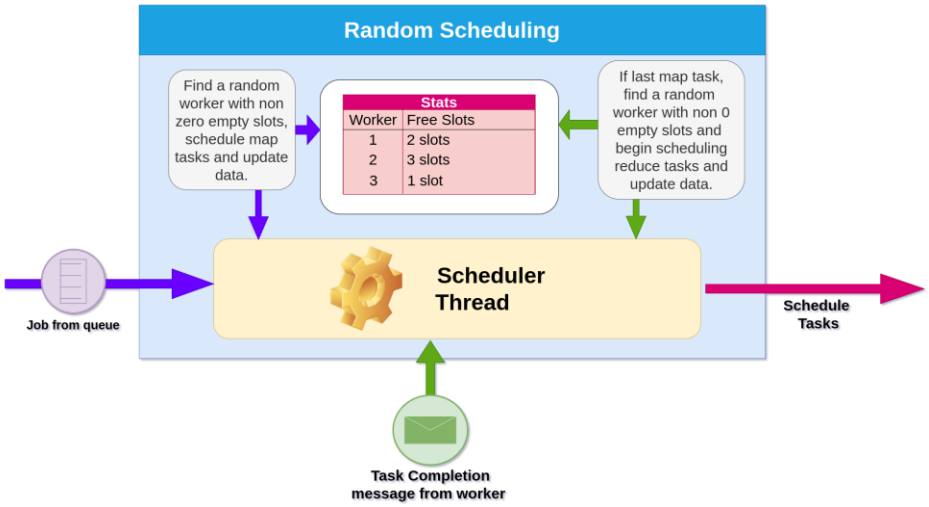


Much like the **least loaded** scheduler, **round robin** too can receive tasks to be scheduled from the same two places, from the job queue and from the worker listener. The only difference would be in the mechanism of choosing a worker to assign the task to.

Unlike **least loaded**, **round robin** is not sensitive to the load of the worker. It keeps track of the previous worker a task was scheduled to and proceeds to schedule the current task to the next immediate worker. In the event of the current worker not having any free slots, the scheduler looks for the next immediate worker with empty slots and schedules them there.

In the event of no empty slots being available in the worker, scheduling is blocked until at least 1 free slot exists.

Random Scheduling

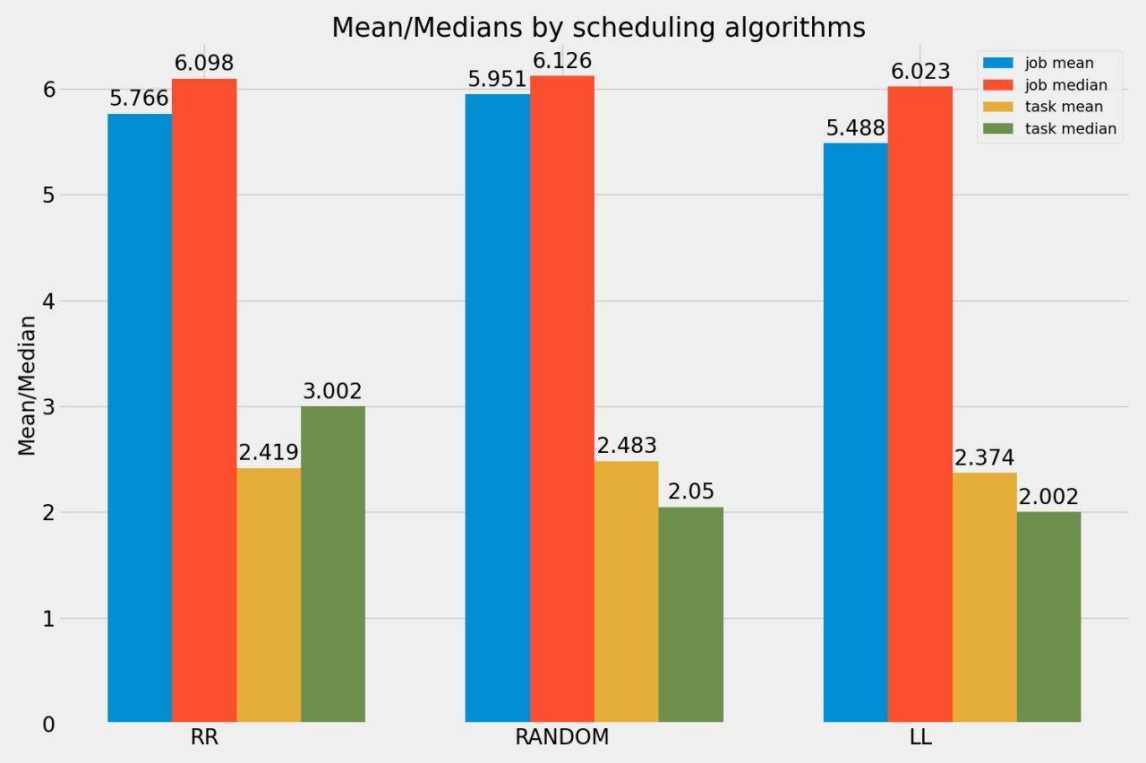


Much like the **least loaded** and **round robin, random** too can receive tasks to be scheduled from the same two places, from the job queue and from the worker listener. The only difference would be in the mechanism of choosing a worker to assign the task to.

In the case of **random** scheduling, a worker is picked at random and the given task is assigned to that worker. In case this worker has no empty slots, the random selection process is repeated but this time the worker selected previously is left out from the sample selection.

Similar to the above two scheduling algorithms, **random** too blocks scheduling in the event of no empty slots in the cluster.

Results



As depicted from the graph above, the **least loaded** algorithm performs better on average compared to **round robin** and **random**, mostly due to the "scheduling overheads" of finding a worker with empty slots which is overcome in the case of **least loaded**.

In the case of **round robin** and **random**, if the chosen worker does not have empty slots, the scheduling algorithm must start over and pick a worker that does have empty slots capable of running tasks. This is the above mentioned "overhead".

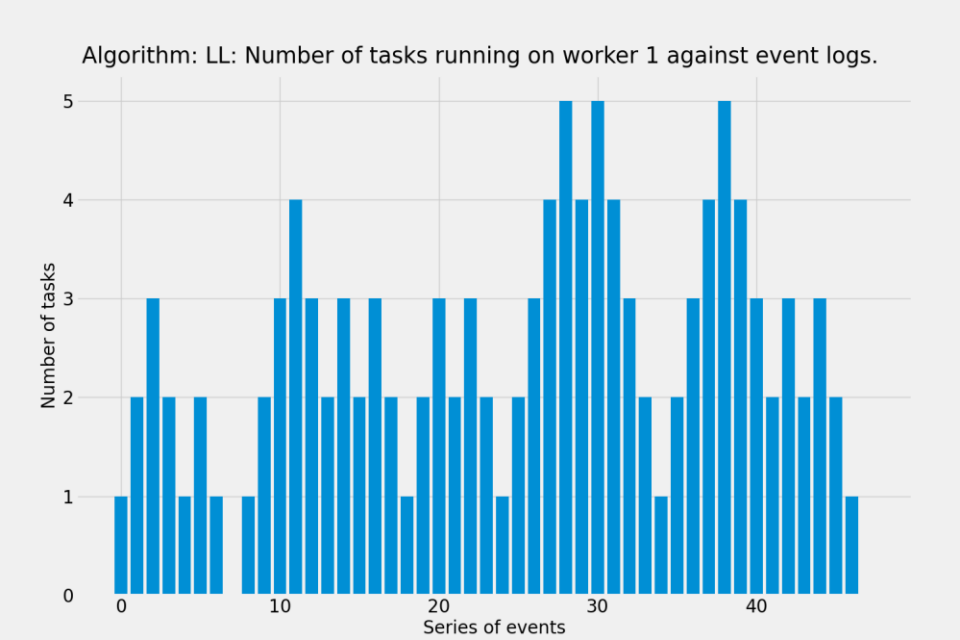
Plots: Tasks on a worker vs Event occurrence

On plotting the number of tasks running on a worker as a function of event occurrences, we obtain the below plots. Here, an event occurrence is defined as either a new task starting (signified by an increase in the bar) or a task completing execution (signified by decrease in the bar). **For the given plots below, we ran the application in a pseudo-random state to generate and work with 20 jobs.**

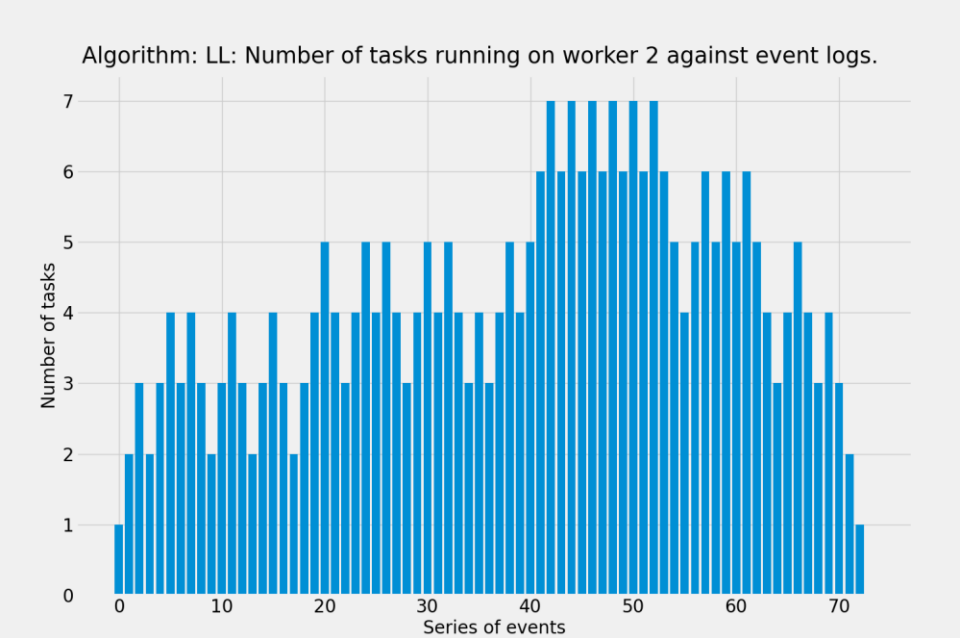
Worker Configurations	
Worker ID	**Number of Slots**
1	5
2	7
3	3

Least Loaded

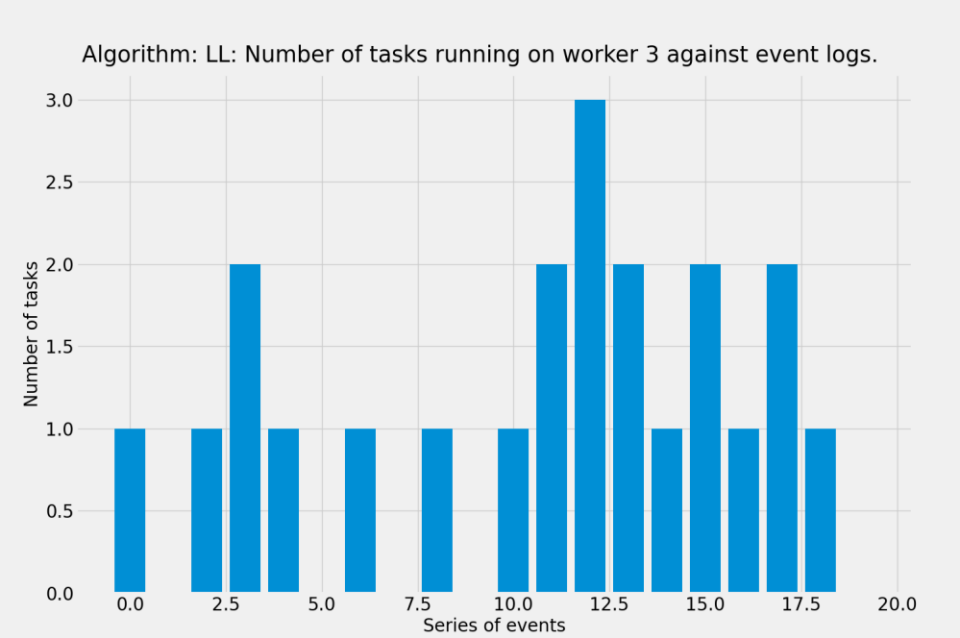
Worker 1



Worker 2



Worker 3

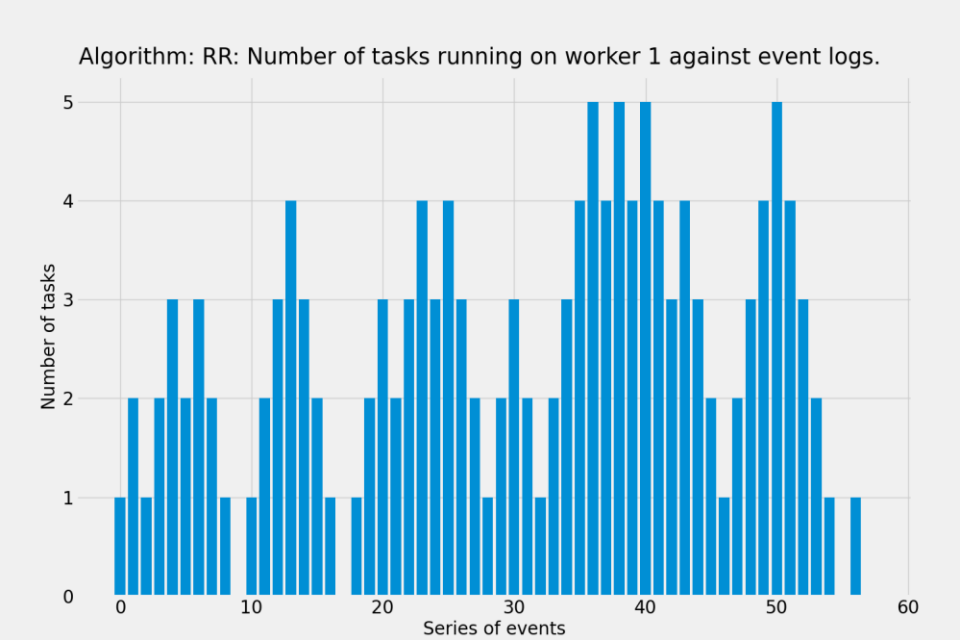


As inferable from the above plots for tasks scheduled on workers, when employing **least loaded** as the scheduling algorithm, tasks are distributed in proportion to the number of slots available on a machine. A machine capable of handling more tasks (more slots) receives more tasks which is in contrast to machines with fewer slots. This can be verified from the table given below (Usage **ratio = Number of tasks / Number of slots**).

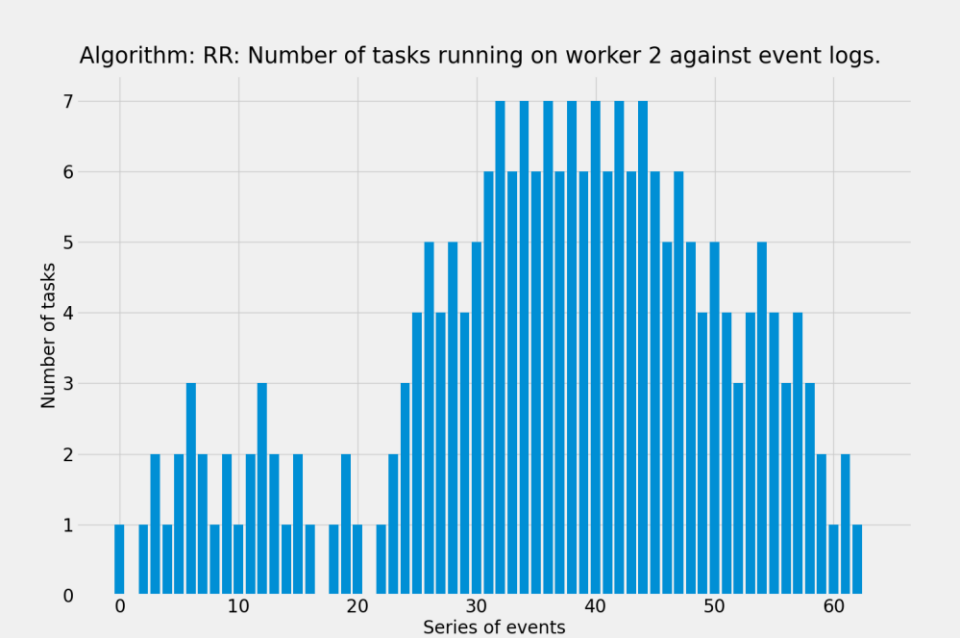
Worker ID	Number of Tasks	Number of Slots	Usage Ratio
1	24	5	4.8
2	37	7	5.2
3	10	3	3.3

Round Robin

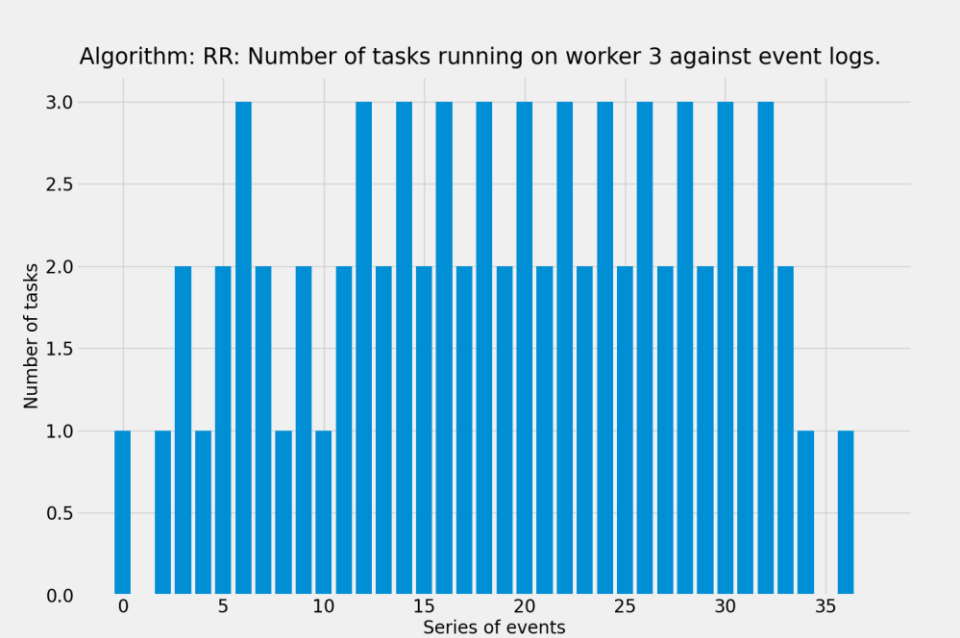
Worker 1



Worker 2



Worker 3



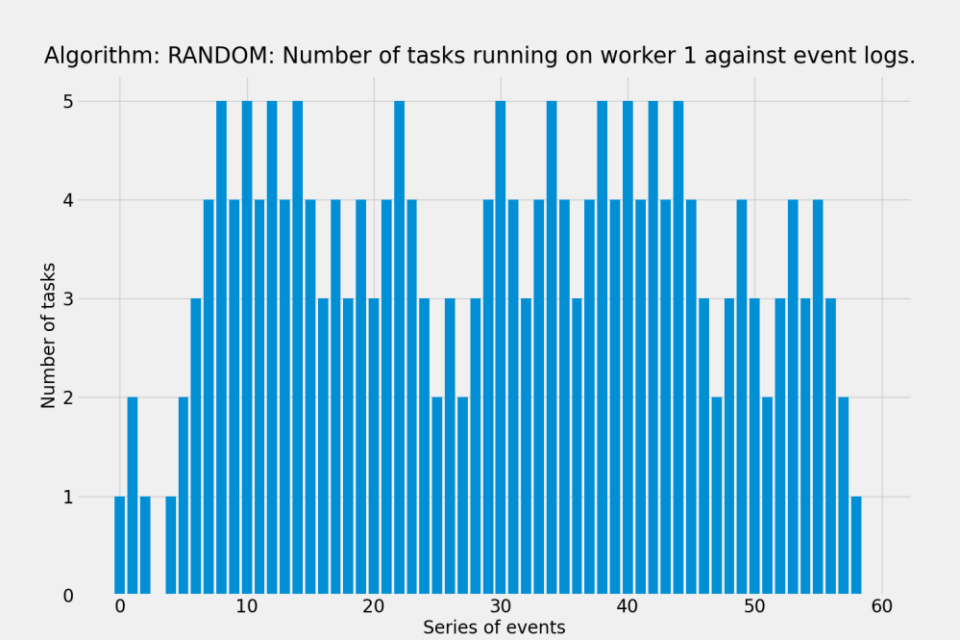
As deducible from the above bar graphs, **round robin** algorithm does not have any sort of bias, unlike **least loaded** algorithm. It schedules a task on a worker even if it has the least number of free slots. It tries to equally distribute tasks among all available workers, irrespective of the amount of resources available at the workers.

As **round robin** aims to distribute tasks equally among all workers, workers with fewer slots tend to have higher usage ratios too.

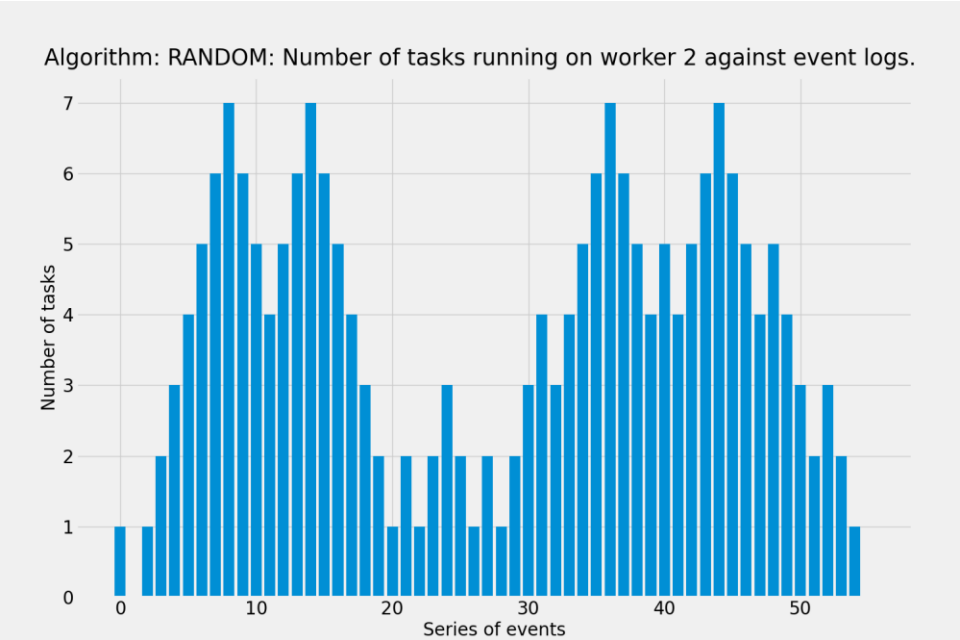
Worker ID	Number of Tasks	Number of Slots	Usage Ratio
1	29	5	5.8
2	32	7	4.5
3	19	3	6.3

Random

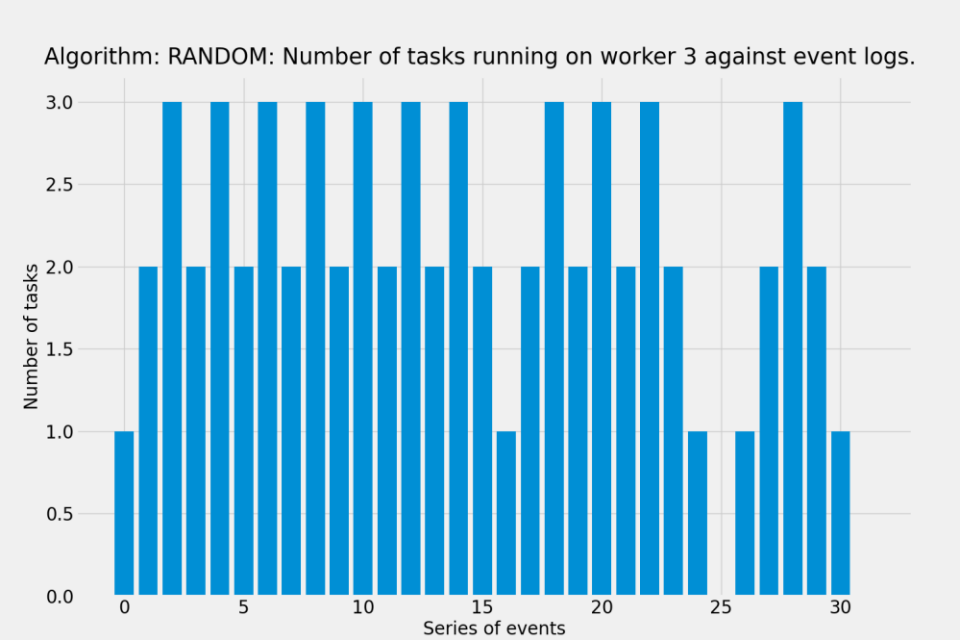
Worker 1



Worker 2



Worker 3



No trend can be inferred in this case since the scheduling is random.

Problems

- **Concurrency issues** – In our design, we have separate threads in the master, for each worker, that listens for updates. Upon receiving an update about the completion of a task, we update the list of tasks to be executed. Following which, we check if it is a map task and also if there are any remaining map tasks. If it was the last map task for the job, we call the scheduler function to schedule remaining reduce tasks. A problem occurs when `2` map tasks of the same job running on `2` different workers complete at the same time. Both threads will infer that map tasks are complete and thus both call the scheduler function for reduce tasks. Now, reduce tasks would be scheduled twice.

  Solution – We introduced an additional data structure to keep track of scheduled tasks. If all map tasks have been scheduled, we check said data structure to see if corresponding reduce tasks have been previously scheduled. If one of the two threads scheduled the reduce tasks, the other does not attempt to schedule them again.

- **Fans go "brrrr"** - Initially, we were busy waiting if slots of the worker were occupied (no empty slots). We were also using an empty while loop in the worker process that waited for the execution pool to have at least one task. This made the CPU overwork due to the sheer number of times the loops were executed, thus making the fan spin very fast and make the acclaimed "brrrr" noise.

  Solution – We replaced the empty loops with semaphores that would block the execution of further instructions till the semaphore had a value greater than `0`. The semaphore provides the same functionality as an empty while loop, but without overworking the CPU (as it works on signals rather than busy waiting). This brought in some optimizations to our design.

- **Tasks not "executing" for specified duration** - Our primitive idea to simulate the execution of tasks was to put these tasks to be executed in a list called `execution_pool`, and decrement the duration for all the tasks in the `execution_pool` by `1` and sleep for `1` second after. If a newly assigned task arrives at the same time at which the tasks' duration was being decremented, without a mutex lock over the data structure, a race condition is inevitable, which is why a mutex lock was necessary. If the sleep occurred when the `execution_pool` was locked, it severely affects the performance of the worker. But, if the sleep occurs after the `execution_pool` is unlocked or before the `execution_pool` is locked, a task that arrives in the middle of the sleep gets added to the `execution_pool`. So, the next time all of the tasks' duration is decremented, the newly arrived task gets decremented before it sleeps for `1` full second. Thus, when the duration of this task becomes `0`, it would not have slept for the entirety of the duration that it was required to, thus not truly simulating the execution of tasks.

Solution – We refactored our entire implementation for the simulation of tasks. We created a new thread for each newly arriving task (depicting a slot). We did not need any mutex locks or semaphores for this. The thread would repeatedly decrement the duration and then sleep for `1` second until the duration became `0`. As soon as the duration became `0`, an update was sent to the master and the thread would complete execution. Now, the tasks would run for the specified duration. This also felt like a much cleaner way of achieving simulation of tasks.

Conclusion

Scheduling - We learnt about a few scheduling algorithms that can efficiently schedule tasks in a distributed environment. Some scheduling algorithms perform better in some cases and some algorithms in other. Random scheduling algorithm works decently for a very few number of tasks. Round robin works well yet provides some amount of overhead due to the case where it must find a worker with non-zero free slots, thus performing the best in the cases where the number of tasks to be scheduled is not too high. Least loaded performs better than the other two scheduling algorithms when the number of tasks to be scheduled is high. The overhead of finding the worker with free slots is much lesser here than in the other two scheduling algorithms.

Practical knowledge of parallelism - Due to the multithreaded nature of this project, we gained a lot of practical knowledge of working with threads and parallelly running processes.

Concurrency models - As a result of the previous learning, we also learnt a great deal about the concurrency models available in python because it was important to prevent race conditions in many cases.

Gained experience with socket programming - Since the project involved inter process communication, we gathered some experience with the concept of sockets and socket programming in python.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	M S Akshatha Laxmi	PES1201800130	Master, Analysis (25%)
2	Abhishek Das	PES1201800177	Worker, Logging (25%)
3	Bhargav SNV	PES1201800308	Master, Analysis (25%)
4	N Sanketh Reddy	PES1201800389	Worker, Logging (25%)

(Leave this for the faculty)

Date	Evaluator	Comments	Score

CHECKLIST:

SNo	Item	Status
1.	Source code documented	Yes
2.	Source code uploaded to GitHub – (access link for the same, to be added in status)	https://github.com/Gituser143/YACS

3.	Instructions for building and running the code. Your code must be usable out of the box.	***On a shell run:***
		`cd YACS/src`
		`python3 master.py <config> <algo>`
		
		NOTE: Master spawns worker processes automatically, DO NOT run workers again manually!
		
		On a new shell, run:
		`python3 requests.py <num_requests>`
		
		To export logs, run:
		`./export_logs.sh <sched_algo>`
		
		NOTE: Export logs after each run! Logs are overwritten for each run. To save them, move the generated 'logs/' folder to a different location after running the export script to avoid loss.