CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): YACS Coding Date: 04/12/2020

SNo	Name	SRN	Class/Section
1	Samyak S Sarnayak	PES1201801565	5C
2	Pranav Kesavarapu	PES1201800299	5D
3	Bhargav S. Kumar	PES1201801459	5J
4	Varun P	PES1201800326	5G

Introduction

Big Data deals with processing huge amounts of data and all of this data is almost always impossible to process on a single machine, we often find the need for high levels of parallelism and efficiency and this is where our project comes in, **YACS** - Yet Another Centralized Scheduler. This is basically a framework which helps in running multiple jobs (with a number of tasks of their own) in a distributed setup for more efficiency. Going a bit in depth, the framework consists of a master and a number of workers, each running on a different machine. Master has the responsibility of accepting the job requests from the client and scheduling the different tasks to run on the available workers. In the current implementation - random, round-robin, least-loaded are the scheduling algorithms used. Each worker has a number of slots, each of which can run a task at a time. So the master has to keep track of the number of slots on each worker to schedule the tasks properly. Finally logs are written to a file and graphs are plotted to analyse the performance of the framework.

Related work

Our main reference material was the project document provided to us. For the implementation, we referred to the python documentation for sockets ([1], [2]), threading ([3]) and other general features in python such as dataclass, typing and logging. The architecture of YARN and centralized systems we learnt during the course also helped us in our project.

A major inspiration for this was a project that two of our team members had previously worked on - heiko which is a lightweight (centralized) load balancer made for managing lightweight servers.

Design

The system consists of a master which manages multiple workers. In this design, there can only be a single master for all of the workers. The architecture and implementation are detailed below.

Master

The master consists of 4 threads and 2 queues, they are:

1. **getRequestData**: receives requests from the client and adds the tasks to the **taskQueue**
2. **processTaskQueue**: takes each task from the taskQueue and using the scheduler, runs them on the appropriate worker

3. **recvFromWorker**: receives task done messages from the worker and adds them to the **workerMessages** queue
4. **processWorkerMessage**: processes messages from the worker and appropriately marks worker's free slots

The getRequestData thread opens a socket at port 5000. When a client connects, data is read in chunks of 1024 bytes until the message is completely read. The query data is stored in a class and the individual *map* tasks are added to the taskQueue. The reduce tasks that are not added to the queue are stored in a temporary jobStore.

The processTaskQueue thread gets each task in the taskQueue, runs the scheduler to determine the next worker to run on and runs the task on that node. The task can be map or reduce tasks, this thread is agnostic of task type.

The recvFromWorker thread opens a socket at port 5000. When a worker connects, the data received is deserialized from JSON, then the message is put into the workerMessages queue.

The processWorkerMessage thread gets each message from the worker and marks the task sent by the worker as done. If it's a map task, it is removed from the jobStore. When there are no more map tasks for that job, the reduce tasks are all added to the taskQueue. Thus, reduce tasks are scheduled only after all the map tasks have completed.

The master stores a representation of each worker with its ID, host, port, total number of slots and the following additional data structures:

- Slots modelled using a Bounded Semaphore with maximum value being equal to the total number of slots
- Number of free slots along with a Lock

When a task is delegated to a worker, its lock is acquired, the number of free slots is decremented, the semaphore is acquired and finally the lock is released. This avoids race conditions and makes this process thread safe.

When a task is to be marked as done, the worker's lock is acquired, the semaphore is acquired, number of free slots is incremented and the lock is released again.

Schedulers

Three different schedulers are implemented which can be selected when starting the master.

- **Random Scheduler**: this scheduler selects a worker at random, if the worker is not free, another random worker is selected. The algorithm is as follows:

    - Acquire locks of all workers - this is to ensure that the number of free slots do not change while the scheduler is computing - avoids race conditions
    - Initialise an empty set of tried workers
    - while all of the workers have not been tried (length of set is less than total number of workers)
        - Select a worker at random and add it to the set
        - If the worker has free slots, break out of the loop
    - If all of the workers have been tried and no free worker is found, release all locks, wait for 1 second and continue the loop again from the beginning.
    - Release all worker locks and return the found worker
- **Round Robin Scheduler:** this scheduler cycles through all of the workers and delegates task to the worker which is free. The scheduler keeps track of the current worker and cycles through in every invocation. The algorithm is as follows:
    - Acquire locks of all workers
    - Run a loop for (number of workers) times
    - If the current worker is not free, set the current worker to the next worker
    - If the loop exits without finding a free worker, the locks are released and the thread sleeps for 1 second
    - If a worker is found, all locks are released and the current worker is returned
- **Least Loaded Scheduler:** this scheduler selects the worker with most free slots. The algorithm is as follows:
    - Acquire locks of all workers
    - Initialise the maximum free slots seen yet to 0
    - Loop through every worker
        - If the number of the worker is greater than the max free slots, set this worker to the desired worker
    - If no free worker is found, release the locks and sleep the thread for 1 second. Then repeat from step 1
    - Else the free worker is returned

Worker

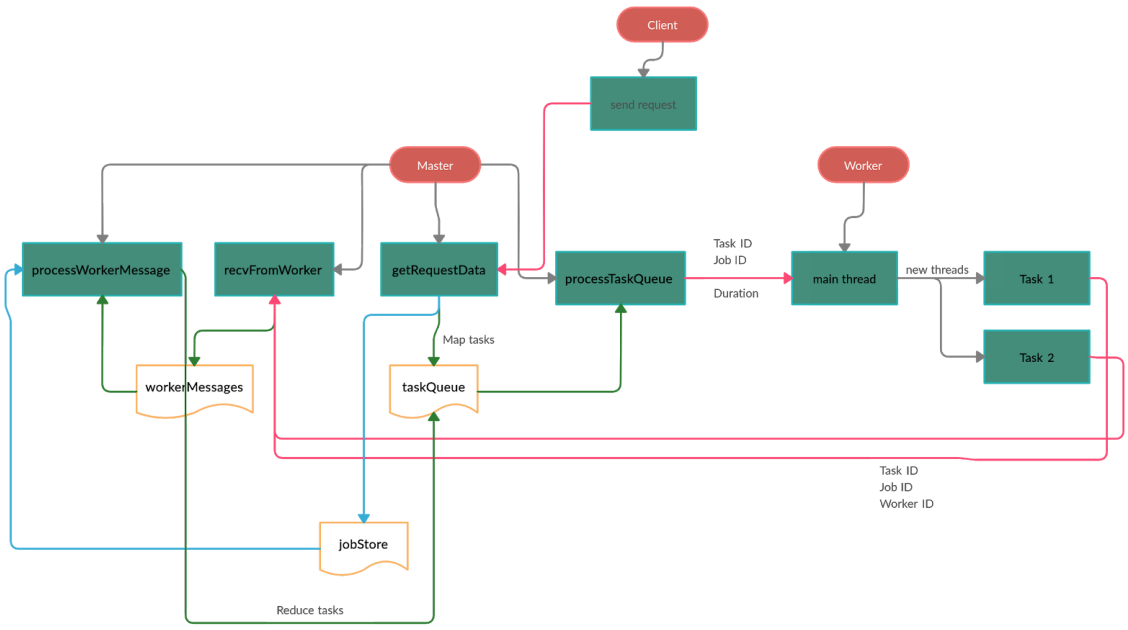
The worker consists of one main thread and a thread for every task run (i.e., upto num_slots + 1 number of threads).

The main thread opens a socket to listen for task delegations by the master. Whenever a message is received, the JSON is deserialized and the corresponding task is run on a separate thread.

The task runner thread sleeps for the duration of the task and sends a response back to the master notifying which task was done along with the job ID and worker ID.

Flow

A flowchart of the entire design is given below.



The flow is as follows:

1. Client sends a request with the job details to the master
2. Master queues the map tasks into the task queue and map + reduce tasks are stored in the job store
3. Tasks in the task queue are delegated to the worker one by one
4. The worker creates a new thread for each task. When the task is done, a response is sent back to the master
5. The master sends all worker messages to a queue from where another thread processes them
6. When map tasks are done, they are removed from the job store. If the job store has no more map tasks for that Job ID, all the reduce tasks are queued into the task queue.

Results

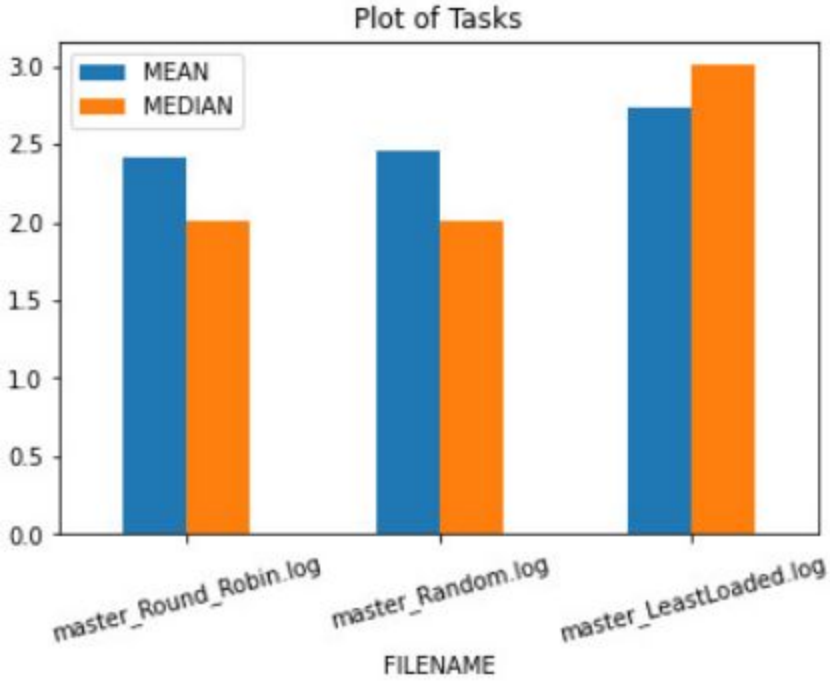
The master logs everything to a file as well as the terminal. These log files contain a wealth of information about the jobs, task scheduling as well as the workers.

After running and testing the system on multiple configurations, we parse the log files to extract information about every job, every task and every worker.

Summary of Job and Task run times

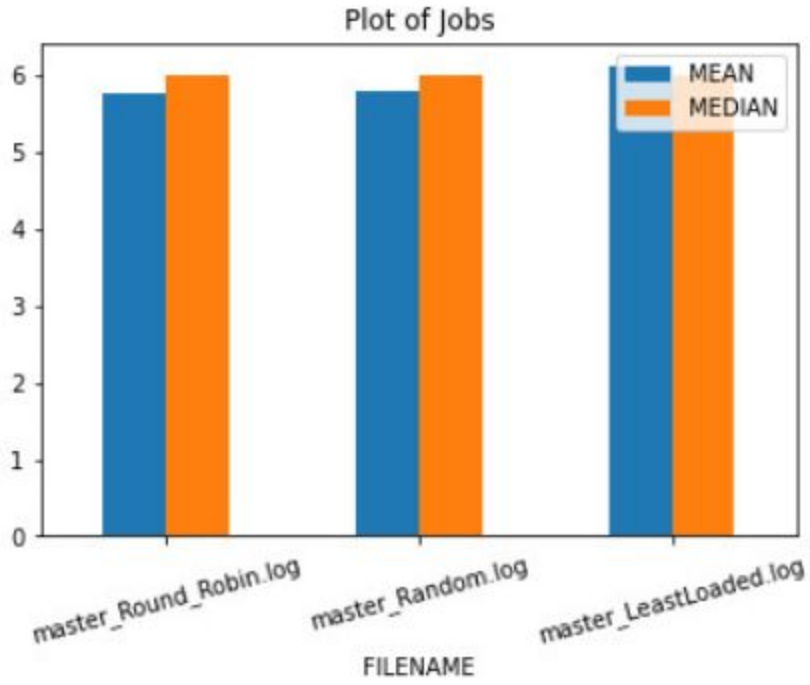
With the default configuration of 3 workers with 5, 7 and 3 slots respectively, we obtained the following results.

It can be seen that the round robin scheduling algorithm had the least average task completion time. Random scheduler was in a close second while the least loaded scheduler took the most amount of time per task.



	FILENAME	MEAN	MEDIAN
0	master_Round_Robin.log	2.414910	2.007343
0	master_Random.log	2.458793	2.004407
0	master_LeastLoaded.log	2.734266	3.004223

Similar results were noticed for the average and median job completion times with round robin leading by a small margin.

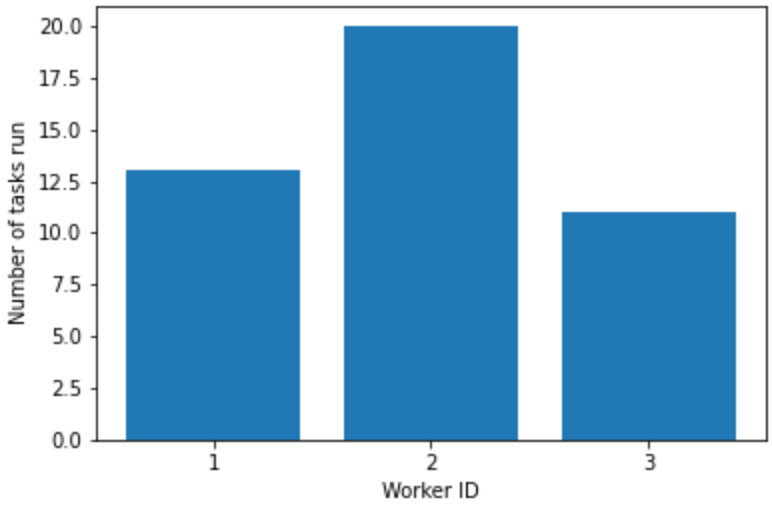


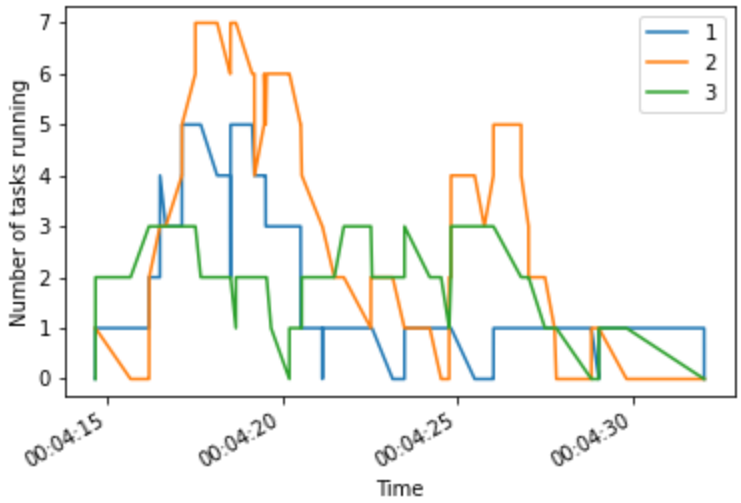
	FILENAME	MEAN	MEDIAN
0	master_Round_Robin.log	5.777026	6.016239
0	master_Random.log	5.813586	6.012842
0	master_LeastLoaded.log	6.114153	6.011900

This shows that the round robin scheduler performs the best although by a small margin.

Summary of tasks run on each worker

Random scheduler

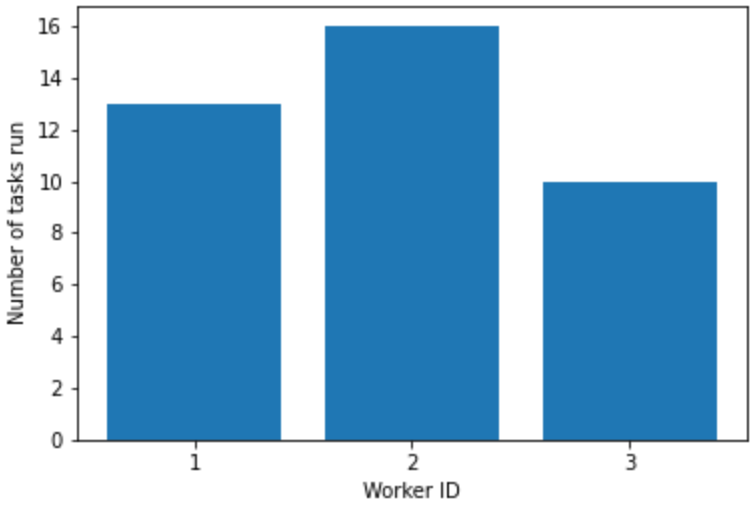


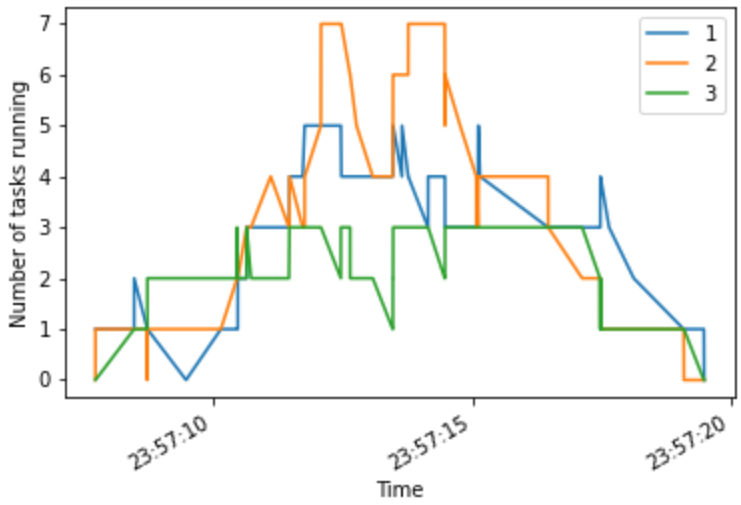


It can be noticed that the random scheduler delegates the most number of tasks to worker 2 (7 slots) followed by worker 1 (5 slots) and finally worker 3 (3 slots) which shows that the number of tasks allocated is proportional to the number of slots and this can be called a fair scheduler.

The line graph (time series) shows that the load increases sharply and reaches a point where all of the workers have their slots filled, the load then decreases gradually.

Round Robin scheduler

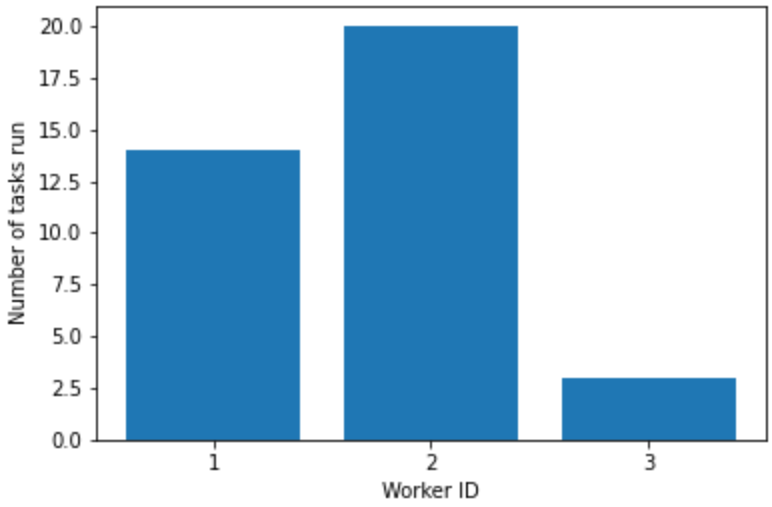


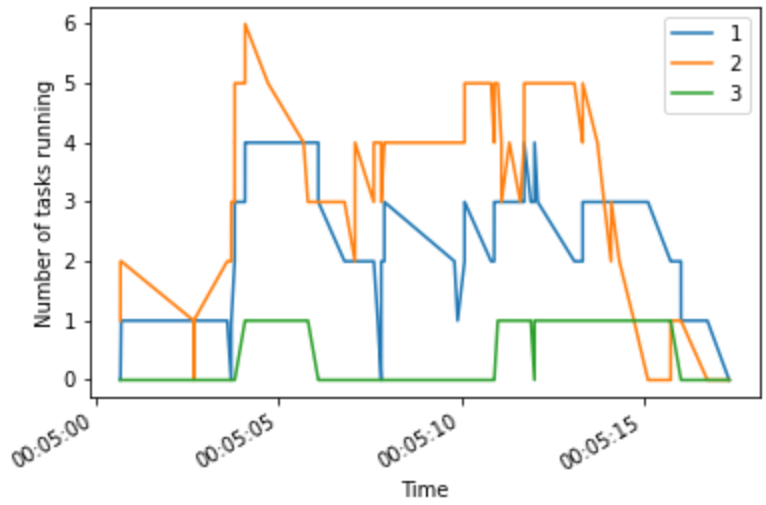


From the bar graph, it can be seen that the round robin scheduler tries uniformly schedule equal number of tasks to each worker but is limited by the number of slots i.e., the workers with lower number of slots are kept more busy.

This can be confirmed from the time series plot - worker 3 (with 3 slots) has all of its slots filled for an extended period of time while worker 1 has slots filled for a lesser amount time and worker 2 only has its slots filled for small periods of time.

Least Loaded scheduler





The least loaded scheduler disproportionately delegates tasks to workers with higher number of slots (worker 2 and 1) while at the same time, none of the workers have their slots filled at any point of time - this is most likely the cause for the least loaded scheduler performing poorly wrt average task and job times.

Benchmarking

The pipeline was benchmarked with empty tasks (of duration 0) to test the latency of the system. The following results were obtained:

- 20 workers, 20 empty tasks: time taken from receiving Job to completion of last task = **72ms**

- 5 workers, 5 empty tasks: time taken from receiving Job to completion of last task = **38ms**
- 1 worker, 1 empty task: time taken to complete last task from receiving job = **2.4ms**

This shows that the system has sub-0.1s latency even when there are 20 workers to handle and can achieve sub-5ms latency for a single task and single worker.

Problems

The first problem we faced was separating the map and reduce tasks. We were initially perplexed about whether this would be handled by the master or the worker who has been assigned the job. We then concluded that it would be better to keep the workers as nodes that just execute the given tasks, whereas the master would handle the coordination of the map and reduce tasks. The worker would finish the task and send a message to the master, the master would keep track of all the task state for each worker as well as a job store to keep track of tasks contained in a job. As soon as all the map tasks are done, it would then remove them from the job queue and then add the reduce tasks from the job store.

The second issue we faced was finding out the number of slots that are free in a worker at a given time. To allocate tasks, we were using a BoundedSemaphore on the number of slots. This would increment and decrement each time a task was done and executing respectively. The dilemma here was that the library didn't allow us to check the number of slots that are being used, or the current slots that are free, it would just check if the value of the semaphores wasn't 0. The way we solved this was by creating a variable that would be updated whenever a slot was occupied. This would result in race conditions since there would be many slots being occupied and freed in parallel resulting in race conditions. To prevent this we also associated a lock with this variable so that only one process/thread updates it at a time.

Conclusion

After this project, we got a slightly clearer understanding of how YARN manages nodes. We also learnt about schedulers, threading, logging and thread safety using semaphores and locks.

We could have done a few things differently such as adding backup nodes for masters - to make the system robust to failure of master, allow workers to manage splitting of map and reduce tasks, or instead of sleeping when executing tasks print the duration left for the task. We could also have the workers in separate machines or even containers to simulate the real environment better. Note that our design does not limit this to be used with workers on different machines in any way.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Pranav Kesavarapu	PES1201800299	Utilities, Layout
2	Bhargav S Kumar	PES1201801459	Worker
3	Samyak S Sarnayak	PES1201801565	Scheduler, Layout, Design
4	Varun P	PES1201800326	Analysis

(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.	