

CS322: Big Data

Final Class Project Report

**Project: YACS Coding**  **Date: 04th December 2020**

|  |  |  |  |
| --- | --- | --- | --- |
| SNo. | Name | SRN | Class/Section |
| 1 | Abdur Rahman Hatim | PES1201801503 | A |
| 2 | Harichandana Magapu | PES1201801041 | A |
| 3 | Samarth Mathur | PES1201800290 | A |
| 4 | Sanjna Chaturvedi | PES1201800722 | A |

## Introduction

Big data workloads, consisting of multiple jobs from different applications, are too large to run on a single machine. Therefore, they are run on clusters of interconnected machines, using a scheduling framework to manage and allocate the resources of the cluster to the different jobs in the workload. A job is made of one or more tasks. The scheduling framework receives job requests and launches the tasks in the jobs on machines in the cluster.

To launch a task, the scheduler must find a machine with available resources. Once the task is allocated resources on a machine, it executes its code. As soon as a task finishes execution, the scheduling framework is informed, and the resources are freed. The framework can thereafter assign these freed resources to other tasks.

In this project, we attempt to simulate a scheduling framework- Yet Another Centralised Scheduling framework- and demonstrate its working. The framework consists of one Master process, which runs on a dedicated machine and manages the resources of the rest of the machines in a simulated cluster. The other machines in the cluster have one Worker process running on each of them . The Master process makes scheduling decisions while the Worker processes execute the tasks and informs the Master when a task has completed its execution.

## Related work and references

1. Operating System Concepts, 9th Edition by Abraham Silberschatz, Peter B. Galvin, et al.

This book gave us a friendly introduction to processes, threads, and scheduling

1. <https://realpython.com/intro-to-python-threading/>

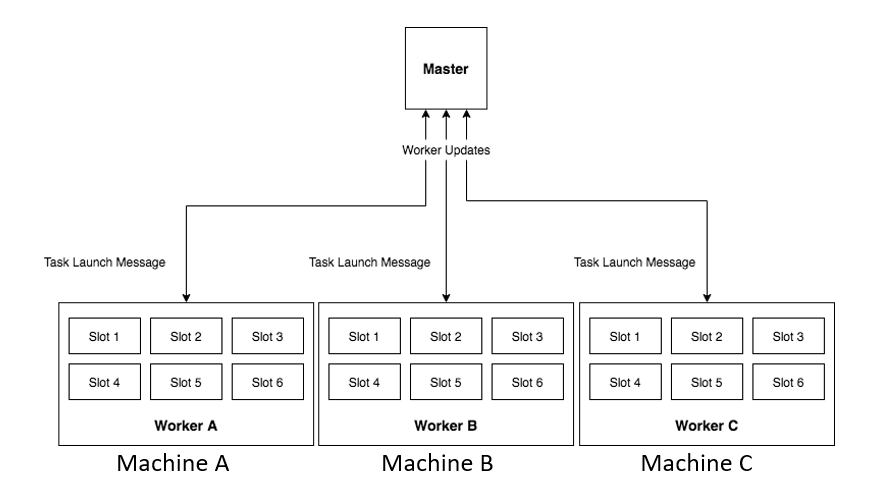
As our introduction to threading was in C, this link helped us understand threading and its capabilities in Python

1. ‘Scheduling in distributed systems: a cloud computing perspective’ by Luiz F. Bittencourta , Alfredo Goldmanb , Edmundo R. M. Madeiraa , Nelson L. S. da Fonsecaa , Rizos Sakellariouc.

This paper gave us a deeper understanding of the Master-Worker architecture and task distribution across nodes

## Design

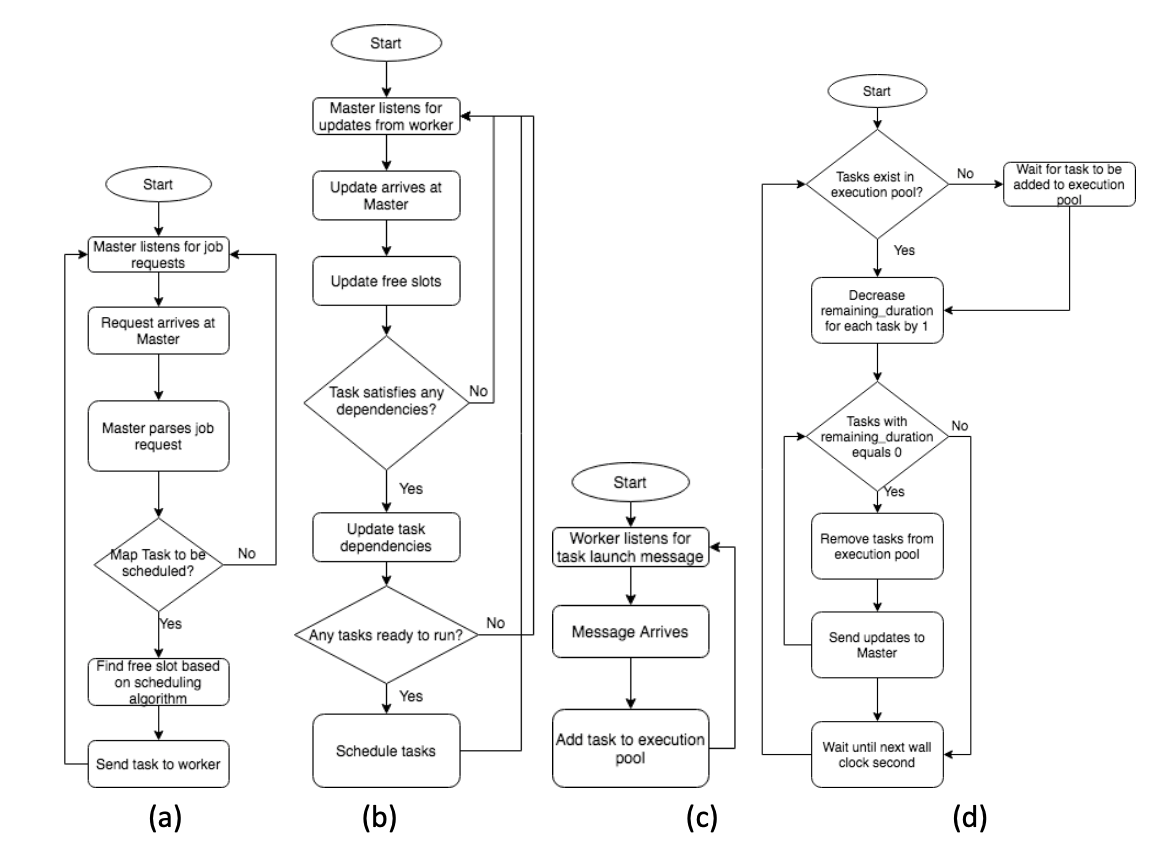
The design of our scheduler is as shown in the diagram:



Every job is broken up into tasks, which are launched by threads. The Worker process listens for task launch message from the Master. When it receives a task launch message, it adds the task to its execution pool. The Worker process simulates the running of the task in the execution pool. Once a task has completed execution, the Worker removes the task from its execution pool and reports to the Master that the task has completed its execution.

All jobs have 2 stages - The first stage consists of map tasks and the second stage consists of reduce tasks. The reduce tasks in a job can only execute after all the map tasks in the job have finished executing. All map tasks can run in parallel, and all reduce tasks can run in parallel (there is no ordering within reduce tasks, or within map tasks). The framework takes this map-reduce dependency in the jobs into account - Master checks if it satisfies the dependencies of any reduce tasks and whether the reduce tasks can now be launched. A job is said to have completed execution only when all the tasks in the job have finished executing.

The Master process has 2 threads- one to listen for job requests (a), and another to listen for updates from Workers (b). Each Worker has 2 threads-one for listening for task launch messages from Master (c), and another to simulate the execution of the tasks and to send updates to the Master (d). The workflow involving all four threads can be depicted as:

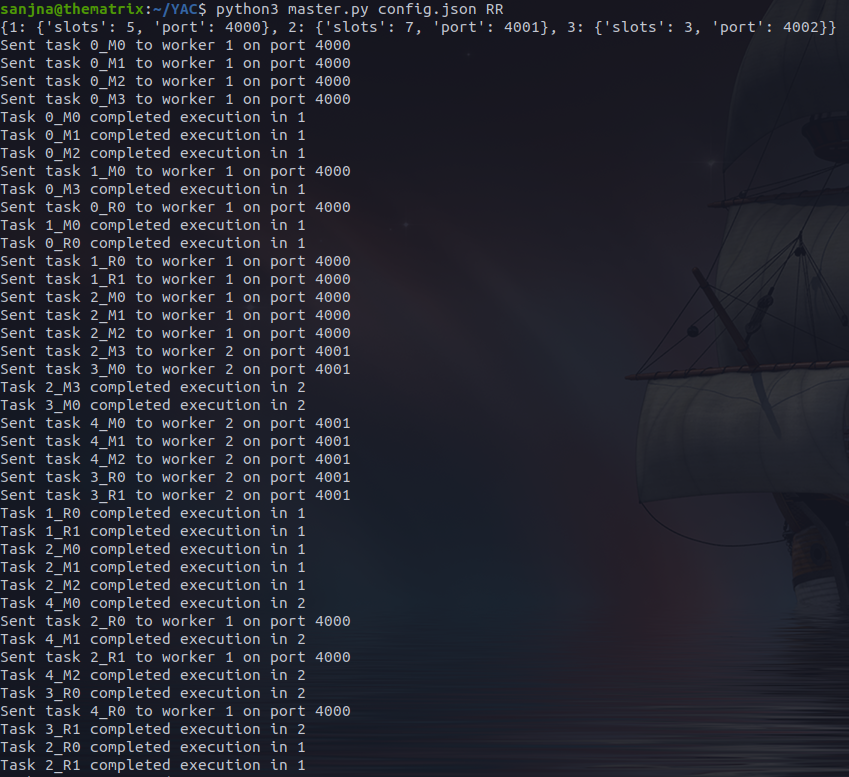


The framework uses three scheduling algorithms:

1. Random: The Master chooses a machine at random. It then checks if the machine has free slots available. If yes, it launches the task on the machine. Else, it chooses another machine at random. This process continues until a free slot is found.
2. Round-robin: The machines are ordered based on worker\_id of the Worker running on the machine. The Master picks a machine in round-robin fashion. If the machine does not have a free slot, the Master moves on to the next worker\_id in the ordering. This process continues until a free slot is found.
3. Least-loaded: The Master looks at the state of all the machines and checks which machine has highest number of free slots. It then launches the task on that machine. If none of the machines have free slots available, the Master waits for 1 second and repeats the process. This process continues until a free slot is found.

The algorithm to be used is provided as a parameter (along with the worker config file) when executing the master file.

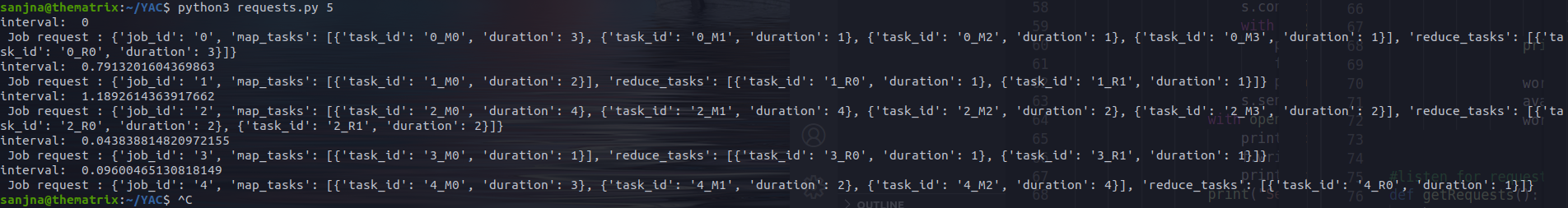
## Results



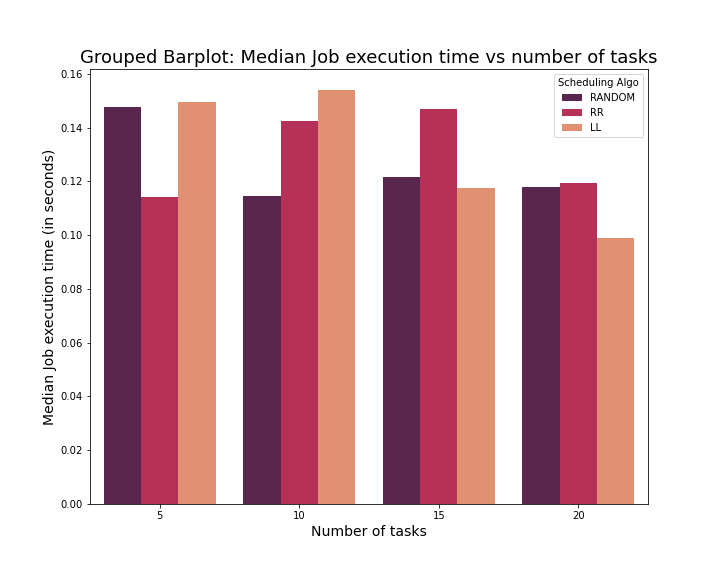












## 

## 

## 

## 

## Problems

1. Port errors: Socket programming was fairly new to us, and we had some trouble connecting the master to the workers without ‘connection refused’ errors. We solved these by tackling them one at a time, removing any errors we encountered
2. Finding a regex to extract timestamps and other relevant information and putting it into a format suitable for analysis was quite challenging and extremely time consuming.
3. Resolving deadlocks: Multithreaded programming is challenging for any programmer, and as beginners, we found it tough to resolve race conditions and deadlocks. We had to figure out where exactly to put the locks to avoid race conditions.
4. Datetime library: this one is more trivial, but we spent a good amount of time trying to figure out the right import statement to use in order to use the now() function from the library
5. Working as a team virtually: Due to the virtual nature of this semester and everyone’s timings being out of sync, it was a challenge to find a time that worked for everyone to discuss the project and work together.

## Conclusion

1. Least Loaded scheduling becomes more efficient as the number of tasks approaches the number of slots
2. Round Robin scheduling appears to follow a normal distribution
3. Aside from Random scheduling, there are instances where workers 2 and 3 are not used at all (slots in worker 1 are sufficient) – this was surprising

## EVALUATIONS:

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | SRN | Contribution (Individual) |
| 1 | Abdur Rahman Hatim | PES1201801503 | Analytics, regex, debugging and assistance with master and worker files |
| 2 | Harichandana Magapu | PES1201801041 | Analytics, regex, debugging and assistance with master and worker files |
| 3 | Samarth Mathur | PES1201800290 | Master.py, Worker.py, debugging and assistance with analytics |
| 4 | Sanjna Chaturvedi | PES1201800722 | Worker.py, Master.py, debugging and assistance with analytics |

## (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. |  |