CS322:Big Data

Final Class Project Report

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

SNo	Name	SRN	Class/Section
1	Mahesh D	PES1201800210	F
2	Khushei Meghana Meda	PES1201800416	F
3	Amrutha U	PES1201801879	F
4	Sanket N D	PES1201802057	F

Introduction

Today's world requires parallel execution of numerous jobs which are made up of several tasks, the execution of which requires many resources. There is a dire need for a job scheduler, especially in datacenters and Big Data applications.

The aim of the project is to make a centralized job scheduler which can schedule jobs across machines, with each machine having varying resources. The abstraction of resources on each machine is implemented through the concept of 'slots'. The only kind of jobs that can be scheduled and simulated are MapReduce jobs.

The job scheduler implements three scheduling algorithms-Random, Round-robin and Least-loaded, to schedule tasks of the jobs on machines.We try to analyse the scheduling of tasks on machines and the execution time for each of them, for every algorithm.

Related work

This project required us to study about many concepts in subjects like Operating Systems and Computer Networks.

We learnt the importance of mutex locks, which play a crucial role in projects like ours where race conditions are likely to happen. The usage of various functions from the threading library such as threading.Lock(), threading.Thread() etc were studied for implementation. Semaphores were used for representing the number of slots available in the workers collectively.

The communication between the master and workers was implemented using the socket library in Python. We used TCP connections between the master and the workers for reliable communication.

We also had to learn about logging in Python, which helped us a lot while debugging our code. Each file has its own logs file. All the workers and the master log the important events as they happen in real-time.

Logging reference

Threading reference

Sockets reference

Datetime module reference

Hadoop 1.0

https://www.edureka.co/community/17101/difference-between-hadoop-1-and-2

Design

The design of the system is based on the following assumptions:

1. The workers and master are fault-tolerant.
2. Each slot can accommodate task of any kind
3. All jobs have two stages- map and reduce.
4. Each task has a finite duration such that it can always end gracefully, without becoming a hanging task.

It involves four main components of this framework are:

1. **Application requests and Configuration of machines in cluster**-

- Application requests- Each application involves several jobs and each job involves several tasks. The file requests.py takes in one input parameter <*number_of_requests*> which is the number of jobs to be executed.

  The *create_job_requests* method creates jobs with a unique job_id and creates a random number of map-reduce tasks for each job with each of them having unique_ids.

  The *send_request* method sends these job requests to the master job scheduler which listens on port number 5000 (specified).

- Configuration of workers-The *config.json* file consists of a workers list that has the *worker_id, slots and port number* for each worker.

2. **Communication model**- A centralised framework model is used and demonstrated in pseudo-distributed mode. The communication of the master with the applications and the workers happen via sockets.

- The application and the master-
  - ➢ *send_request(job_request)*: This method in the requests.py file,sends job requests that it receives to the master on port number 5000.
  - ➢ *listen_for_job_requests()*: This method in the master listens to the requests.py file for job requests and adds them to the wait_queue.
- The master and the workers- The master reads the config.json file to fetch the number of workers, number of slots and the port number associated with each worker and maintains variables in the *class Worker_details*.
  - ➢ *schedule_task()*: This function in the master schedules the tasks on the workers by checking for available slots on the worker machines.

- ➢ *listen_master():* This function in the worker listens for tasks from the master on the specified port number (which is given as an input parameter), and adds the tasks to the *execution_pool*.
- ➢ *listen_for_worker_updates()*: Listens for worker updates, on port 5000, about the task completion, adds the completed task to the *task_completed* pool and increments the slot count for that worker.
- ➢ *update_master():* This function, in each worker, updates the master about task completion.

3. **Scheduling Algorithms**- The job scheduler uses 3 algorithms to select workers with available slots to schedule tasks on them. The algorithm that must be used by the scheduler is given as an input parameter to the master.

The function *get_available_workers()* checks for free slots in workers and returns a list of workers available to schedule tasks on. The function *find_worker()* uses the get_available_workers() function to find the workers and schedule tasks on them according to the algorithm specified in the input. The algorithms used are as follows-

- Random: It randomly chooses a worker from the pool of available workers.
- Round-robin: The machines are ordered based on the 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.
- Least-loaded: The algorithm is similar to the worst-fit algorithm. The machine with the highest number of available slots is chosen to schedule tasks.

4. **Logs**- Logging of events is done in both the master and the workers. The logging results are used to perform a comparative study of worker scheduling and execution time of jobs for the scheduling algorithms.

Logs in workers-

1. Starting of a task :

   <filename; 'Started task jobid_M/R_taskid';algorithm; timestamp>

2. Completion of a task:

   <filename; 'Completed task jobid_M/R_taskid';algorithm; timestamp>

Logs in Master-

1. Starting of a job:

<filename;algorithm; 'Started job with job id jobid'; timestamp>

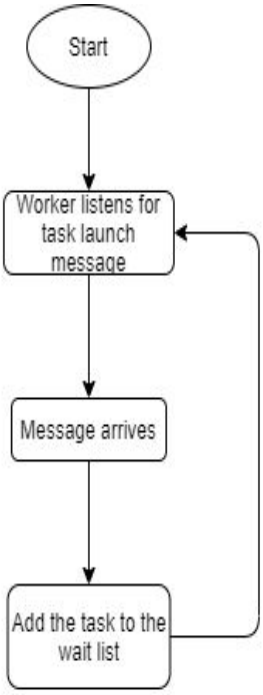
2. Completion of a task-

   <filename;algorithm;'Completed task jobid_M/R_taskid at timestamp duration time'; timestamp>

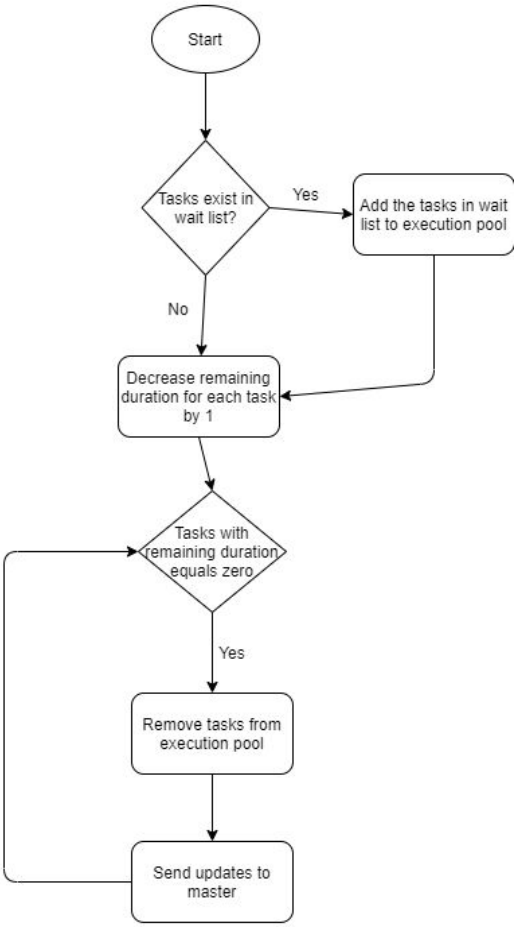
3. Completion of a job-

   <filename;algorithm;' Completed job jobid'; timestamp>

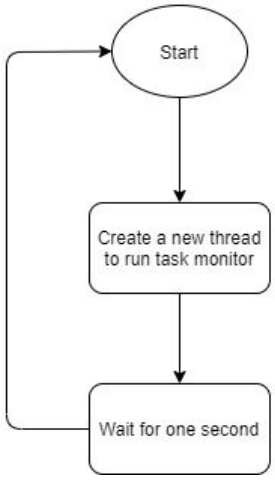
Worker:



Thread 1: Listens for task launch messages

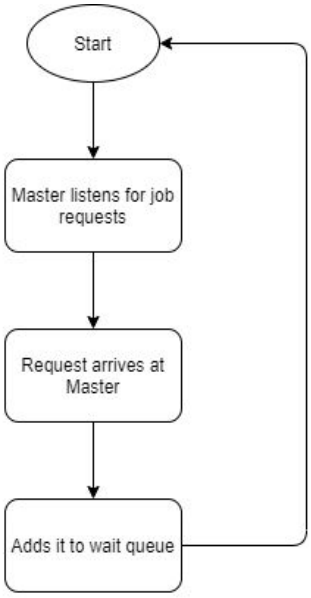


Thread 2: Task monitor thread

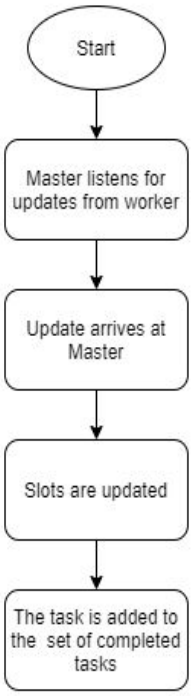


Thread 3: Thread to simulate a clock

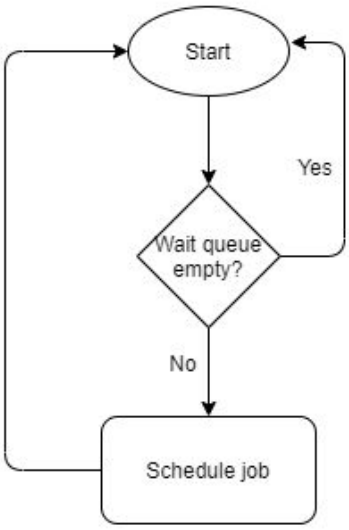
Master:



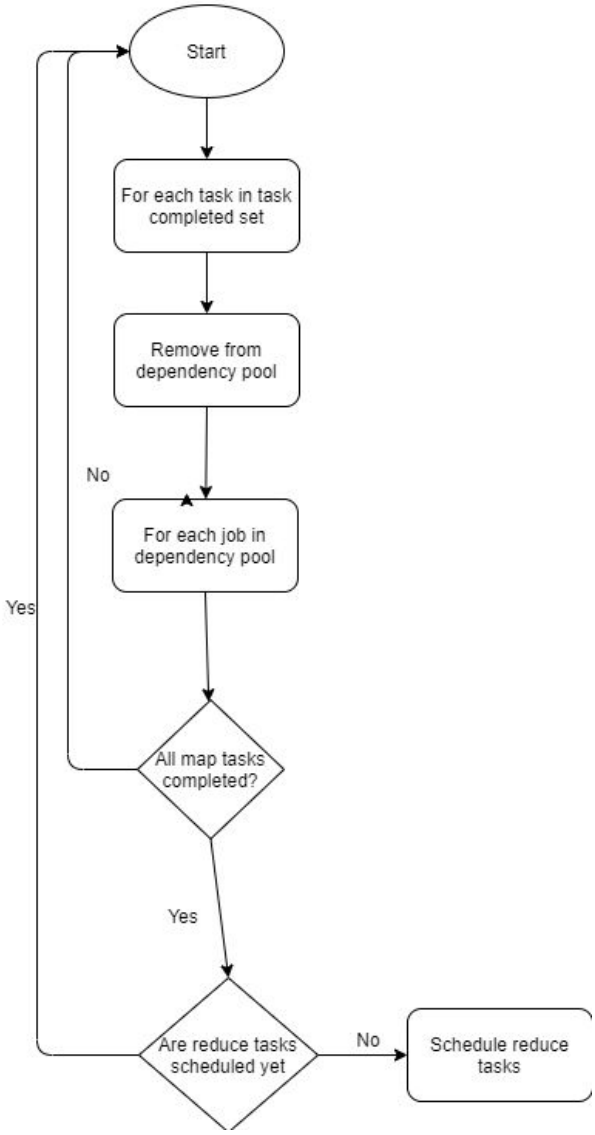
Thread 1: Thread that listens for job requests



Thread 2:Thread that listens for updates from workers



Thread 3: Thread that schedules jobs



Thread 4: Thread that updates the dependencies and schedules reduce tasks

Problems

- Our initial approach allowed a job to run its tasks only after the previous job finished running its tasks, which made the whole architecture very inefficient. This was happening because the checking for dependencies was happening in the same thread. There was a need for another thread to take care of the dependencies, and thus the dependency check and scheduling of jobs were made into two different, independent threads.

- While updating dependencies for the jobs, we ran into a problem that would reschedule reduce tasks of a job without checking if they were scheduled already. Thus we used a set called running_reduce_jobs to keep track of jobs whose reduce tasks were currently.
- Our initial approach made the task_monitor function wait for a second to simulate a clock. However, we realised that this wasn't true wall clock time. So we decided to create a new thread every second to run the task monitor. This helped us get more accurate run times.
- To give more accurate runtimes of each task, we decided to add tasks to the execution pool in the task monitor function itself.

Conclusion

We understood the resource management principles of YARN in detail. We learnt the importance of design, structure and modularization of code in scenarios that involve parallelization and synchronisation. Modelling the data in data structures was challenging. We learnt how to record and parse log files. To simulate a real clock was challenging given the variable arriving times of tasks. This project helped us understand the various intricacies that go into the design of a scheduler.

EVALUATIONS

SNo	Name	SRN	Contribution (Individual)
1	Mahesh D	PES1201800210	
2	Khushei Meghana Meda	PES1201800416	
3	Amrutha U	PES1201801879	
4	Sanket N D	PES1201802057	

(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.	