CloudKON CLONE WITH AMAZON EC2, S3, SQS & DynamoDB

INTRODUCTION:

This assignment will involve implementing a distributed task execution framework on Amazon EC2 using the SQS. The assignment will be to implement a task execution framework. Amazon EC2 is used to run the framework which should be separated into two components, client (e.g. a command line tool which submits tasks to SQS) and workers (e.g. which retrieve tasks from SQS and executes them). The SQS service is used to handle the queue of requests to load balance across multiple workers.

DESIGN:

The design part covers the description of each of the coding modules (Eg, Localworkers, Remoteworkers, Client, etc) that performs their operations on the Amazon AWS cloud

The code modules are as follows:

- LocalBackendWorkers
- RemoteBackendWorkers
- InstanceLaunch
- DynamoDB
- Client_AWS
- SQS_service
- SecureAWS

The LocalBackendWorkers.java implements the functionality of sleep tasks by periodically checking the length of the in-memory queue and performing the tasks that are inserted into the task queue.

The Client_AWS.java receives the workload file and threads as arguments from the user. User enter the Workload file and it is forwarded to the Local workers through the internal queue and to the Remote workers through the SQS queue. An handshake (Acknowledgement) is sent to the Client, once all the workloads are completed.

The Amazon AWS instances are created and launched using the code named InstanceLaunch.java. It has implementations for retrieving AWS credentials before the launch of an specific instance or image of an instance.

DynamoDB.java uses hashmap data structure to store key/value pairs of the Task and its ID. Every Worker checks the dynamoDB table for the entry of tasks. If any duplicate tasks are there, it is removed from the dynamoDB.

RemoteBackendWorkers.java has implementation to write the tasks back onto the SQS queue. When there are empty tasks or unavailable in the queue, the remote worker keeps polling (sending heartbeat messages) until the tasks are added.

SQS_service.java is a sample api from the Amazon. It uses the queue functionality that receives the tasks from the scheduler and sends to the worker nodes and clients receive the results from it. The tasks are pushed and pulled from the scheduler as and when needed.

SecureAWS.java is a sample api code for aws.amazon.com. It receives the necesary permission and creates a secruity group for launch instances.

EXTRA CREDITS:

1. Animato: Instead of sleep jobs the URL of images are given as input the scheduler. Animato creates a video of these images.

MANUAL:

LOCAL BACK END WORKERS

- Both the client and the local workers run in the same AWS instance.
- You can execute the client by executing the following command

  java -jar assignment3.jar Client_AWS -s QNAME -w <WORKLOAD_FILE>

  Now the client starts and read the tasks from the workload file and sends these tasks to the internal queue.

  You can execute the local-worker by executing the following command.

  java -jar assignment3.jar LocalBackendWorkers -t N -w <WORKLOAD_FILE>

  where N denotes the number of threads in the thread pool. It determines how many sleeps can be concurrently executed.

REMOTE WORKERS:

- The client should be running in one instance and remote workers should be working in other instances. Since you will be having many remote workers which will be executing in other instances, it is better to install pssh to parallelly start all the instances at once.

- You can execute the client by executing the following command:

  java -jar assignment3.jar Client_AWS -s QNAME -w <WORKLOAD_FILE>

  Now the client starts reading the sleep tasks from the workload file and sends these task to the SQS queue.

  You can execute the remote worker by executing the following command: IPAddrList will contain all the private IP of the Amazon instances

  pssh -i -l ecubuntu -h IPAddrList.txt -x "-oStrictHostKeyChecking=no -i <YOUR_KEY>.pem"
  java -jar assignment3.jar RemoteBackendWorkers -s QNAME -t N

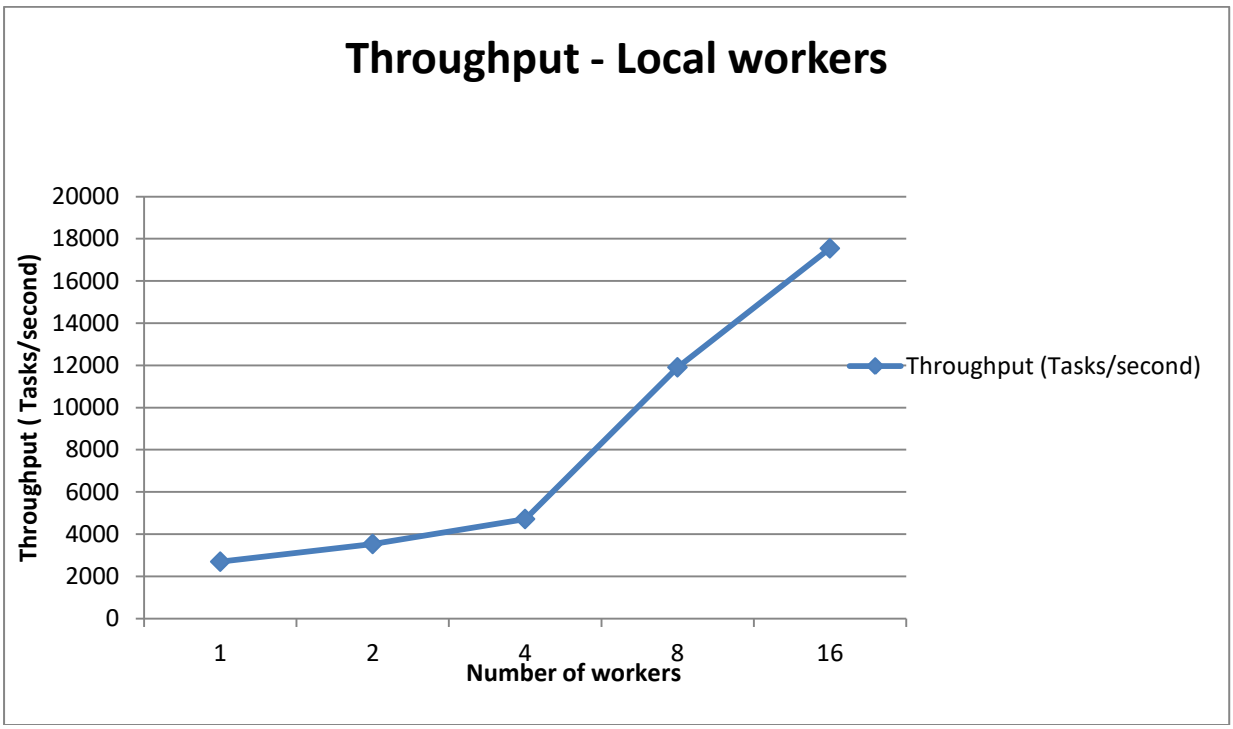
  where N represent the number of backend workers ranging from 1 client to N back-end workers.

PERFORMANCE EVALUATON:

LOCAL WORKER

Since the time taken for the execution of 10k sleep jobs is less than 10 sec only 100k sleep job is executed.

Throughput for 100K SLEEP 0 task

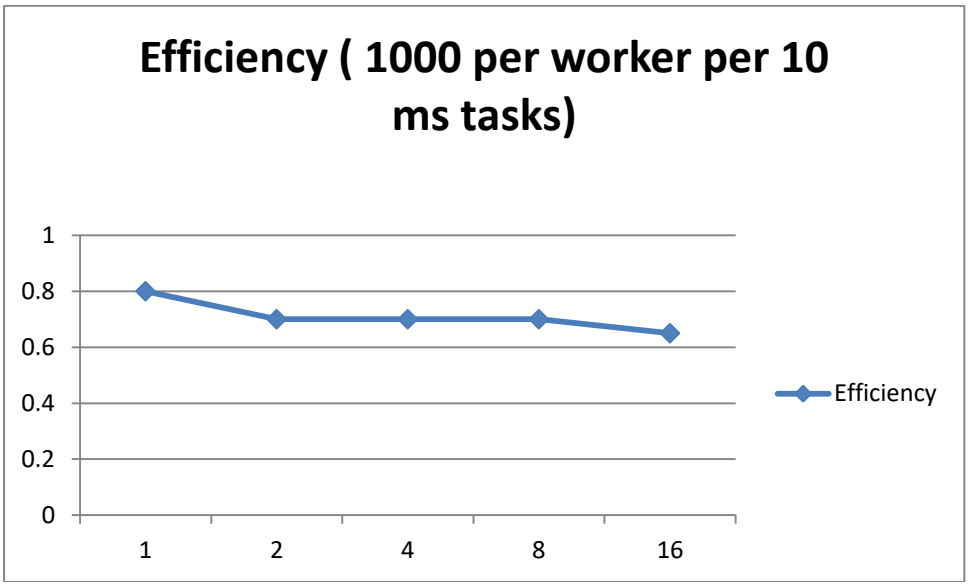


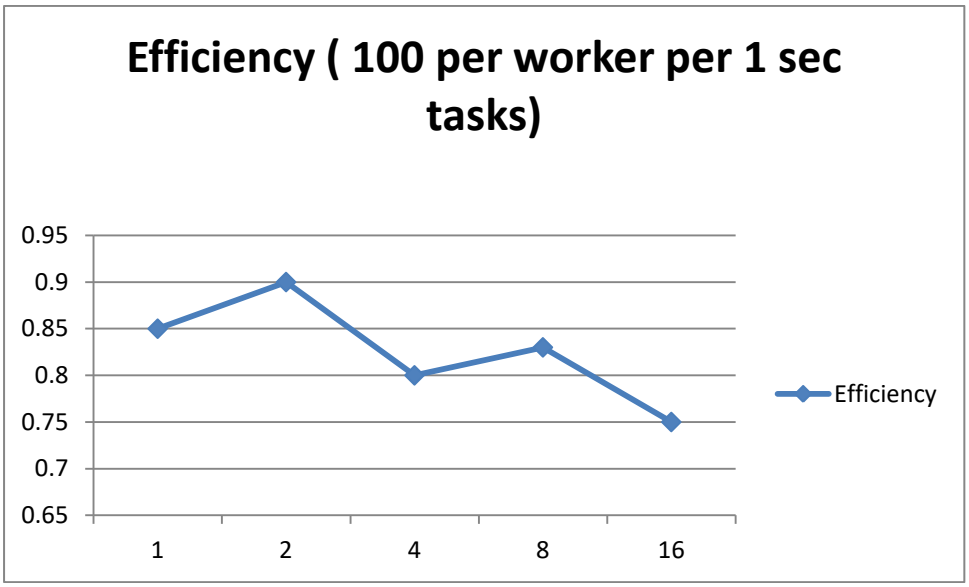
Local
Throughput

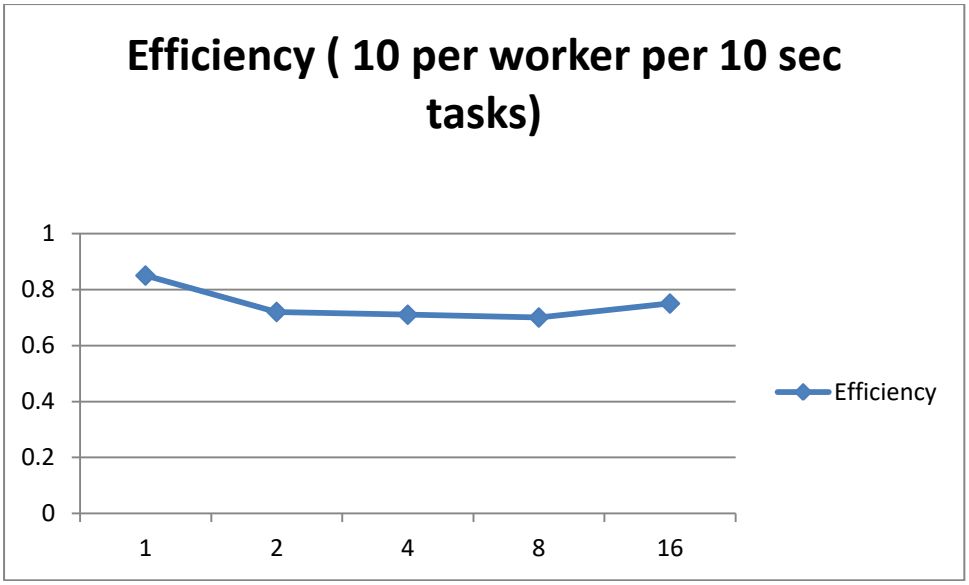
Workers	Time (seconds)	Throughput (Tasks/second)
1	37.15	2691.79004
2	28.32	3531.073446
4	21.21	4714.75719
8	8.4	11904.7619
16	5.7	17543.85965

Efficiency (X axis -> Number of workers , Y axis -> Efficiency)

Efficiency						
Workers	1000 per worker per 10ms tasks	Efficiency	100 per worker per 1 sec tasks	Efficiency	10 per worker per 10 sec tasks	Efficiency
1	8	0.8	85	0.85	85	0.85
2	7	0.7	90	0.9	72	0.72
4	7	0.7	80	0.8	71	0.71
8	7	0.7	83	0.83	70	0.7
16	6.5	0.65	75	0.75	75	0.75

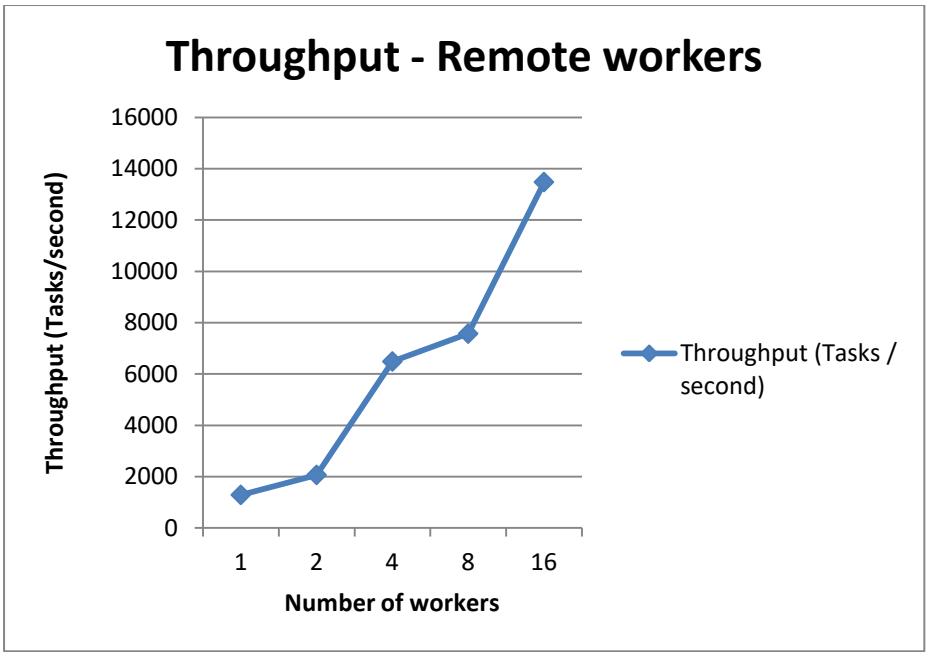






From the above graph, we can infer that, The throughput of local workers are increasing, as we scale up the system by including more threads. The efficiency ranges between 60 - 80 percent for the above tasks mentioned. I could achieve a maximum throughput of 17543 Tasks per second for 16 threads.

Remote Worker



Throughput		
Remote		
Workers	Time (seconds)	Throughput (Tasks / second)
1	77.75	1286.173633
2	48.43	2064.835846
4	15.42	6485.084306
8	13.21	7570.02271
16	7.42	13477.08895

From the above graph, we can infer that, The throughput of remote workers are increasing, as we scale up the system by including more worker nodes (From 1 remote node to 16 remote node).

REFERENCES:

1) http://docs.aws.amazon.com/AWSToolkitEclipse/latest/ug/

2)http://docs.aws.amazon.com/amazondynamodb/latest/gettingstartedguide/GettingStarted.Java.html

3) https://aws.amazon.com/code/Java/1173s