Programming Assignment-3
Design Document
(Kiran Ramamurthy and Priya Patel)

Overview of the Assignment

- Assignment focuses on using Amazon ec2 instances.
- Major objective it to test the performance of the instances dynamically scaled in relative to the manually scaled instances.
- Depending on the workload, the instances are scaled up and down using the cloud watch
- Client reads the tasks in the file and sends it to a scheduler.
- The scheduler takes the task and places in a in-memory queue(in case of local workers) and SQS in case of remote workers.
- The local workers are the threads which take up the task from the in-memory queue and executes it
- The remote workers takes up the task from the SQS and executes it.
- Result is written back to the in-memory queue in case of local workers and a result queue on aws in case of remote workers

The client

- The client reads a file from the local and sends it to the server task by task.
- Client is run using the command "java -cp pa4.jar Client <<Server Name>> <<Port No>>"
- Server name is the name of server to which data is sent to. Localhost is given to say that the server is in the current machine.
- Port number specifies the port on which the server is listening to.
- **Extra credit**
    1. The results are batched and sent to the server.
    2. Each task is separated by the % sysmbol.
    3. The results are put together on the result queue which is an in-memory queue with local workers and SQS in case of remote workers.
    4. The results are then batched together and sent to client.
- A hash map is implemented in order to associate a task with ID.

The server

- The server listens on the port it specifies.
- The tasks are either sent in batches or one-by-one.
- If the 'lw' switch is used, the tasks are written to a in-memory queue which is implemented using a linked list.
- If the 'rw' switch is used, the tasks are written to the SQS on AWS. The queue is called the 'taskQueue'
- If the switch is 'lw', then number of threads can be specified.
- A thread executor service is run which pops and item from the queue and submits it to a thread pool.
- The thread which is free takes up the task and executes it

The Local backend workers

- The number of threads specified in the scheduler defines the number of local workers

- A local worker method will take the task provided and sleeps for the specified amount of time.
- As the thread completes running the task, the result is written to another queue.
- If the thread fails an exception is raised and the return value is written into the result queue.

Remote Backend Workers

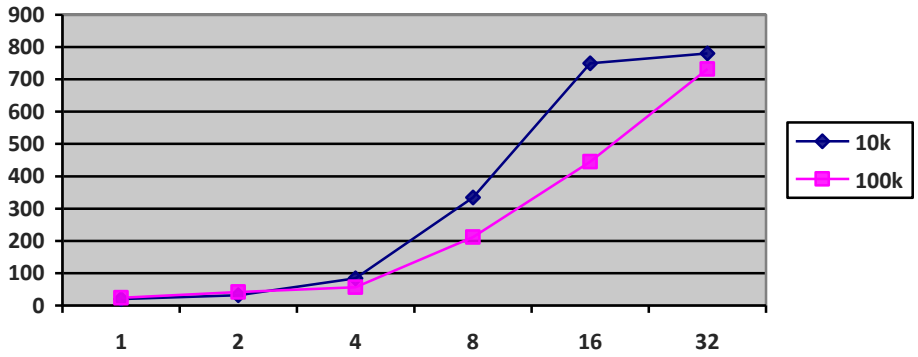
- When the 'rw' switch is used, the tasks are written into SQS queue on AWS.
- A custom AMI is created holding the jar file and a crontab file.
- The crontab file runs the command on the system boot.
- The remote worker polls the SQS queue.
- If there is a task, it is retrieved and executed.
- If there are no tasks, it polls after a few minutes.
- If it is idle for the long time, the remote worker is terminated.
- **Extra Credit**
    1. Threads are implemented to run multiple tasks in one local worker
    2. The number of threads are specified with the command line argument in the crontab file.
    3. Each remote worker takes up the number of tasks equivalent to the number of threads.
    4. The tasks are run concurrently in each remote worker as each remote worker again implements a thread executor service.

Dynamic Provisioning

- A cloud watch is implemented in this method.
- The cloud watch polls the SQS queue every 1 minute.
- If the number of tasks is 0 initially the cloud watch does not start any instance.
- If there are 1 or more and less than 500, cloud watch instantiates 16 instances.
- Each instance is the same AMI the manual remote workers use and hence the task execution starts as soon as the instance starts.
- A cloud watch is implemented to get the number of messages in the SQS, the number of instances running.
- Based on the available information, the scaling up of instances are performed.

Throughput 10k vs 100k with manual provisioning of workers

	1	2	4	8	16	32
10k	20	31.25	84	334	750	780
100k	23.8	42	56	212	445	732



Observation

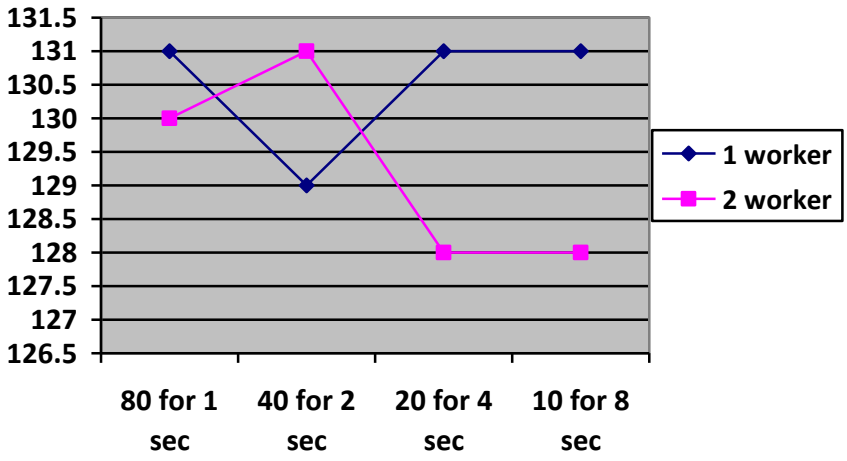
- As the number of workers increases, the throughput increases accordingly.
- Based on the throughput information, the 10k tasks were finished within 10s with 16 and 32 workers.
- The throughput w.r.t 100k tasks vary significantly in comparison with the 10k tasks.

Comparison with Falkon

- Falkon being tested on 100 tasks with sleep 0 has a throughput of 500 tasks per sec
- Our experiment does not take more than 9secs to run 10k tasks of sleep 0 and hence the throughput is more than 750 tasks.
- Running the experiment on bigger data set provides more accurate results which is depicted in 10k tasks as well as 100k tasks.
- As the Falkon graph shows the throughput would be constant after certain number of workers.
- Our experiment does not implement more than 32 workers and hence the throughput increases.

Efficiency evaluation

	80 1 sec tasks	40 for 2 sec	20 for 4 sec	10 for 8 sec
1 worker	131	129	131	131
2 worker	130	131	128	128



Observation

- As the graphs shows the time for executing the tasks on single worker by varying the time and number of tasks per worker is around 130 seconds.
- Increasing the number of workers does not affect the timing in anyway as the number of tasks also increases on the whole for the workers.

Dynamic Provisioning

- The experiment was run for only 3 stages
- In the first stage 500 sleep 0 tasks were passed and 16 instances were created. The tasks were run in 4 seconds.
- After the completion of the 1st stage, another 1000 tasks are passed from the client to the server. Now the number of instances scales up to 32 and it took 4 seconds.
- In the 3rd stage, 500 tasks were sent to the scheduler which is run using the 32 instances present and it approximately takes 3 seconds to complete the tasks.