CS553 PA4 - Project Report

Fan, Zexiang (A20313247)

Liu, Xin (A20314819)

Wang, Dingwen (A20304858)

1. Overall Program Design

1.1 Components and their functionalities

This project focus on developing a task execution framework that support dynamic scheduling. This framework is mainly composed of

- a client,
- a frontend,
- a number of local workers,
- a number of remote workers,
- a dynamic provisioner.

The remaining contents of this section describe the functionalities of each component.

a. *Client* has three functionalities:

- reading from the workload file and generating the corresponding tasks
- sending all tasks to the frontend through a socket
- spawning another thread to serve a socket, in order to receive the result of each task execution from the frontend. Notice this thread starts at the same time as sending the tasks.

b. Frontend has three functionalities:

- serving a socket to receive the tasks from the client
- sending all tasks to the local workers through a implicit memory queue, or
- sending all tasks to the remote workers through a SQS queue
- spawning another thread to check task completion by polling from the implicit memory queue or SQS queue, and send all the completed tasks back to the client through a socket. Notice this thread starts at the same time as sending the tasks.

c. *Local workers* are in charge of:

- reading the task description from the received task
- spawn an OS process to execute the description, and check if it succeed
- return the task with the result record

  (Note: the whole procedure is in a while loop.)

d. *Remote workers* are in charge of:

- poll the task from the SQS queue
- check the dynamoDB to see if the task is duplicated. If it is duplicated, return to previous step

- reading the task description from the received task
- spawn an OS process to execute the description, and check if it succeed
- send the task back to another SQS queue

  (Note: the whole procedure is in a while loop.)

e. Dynamic provisioner is in charge of:

- increasing or decreasing the number of worker according to the amount of incoming tasks dynamically in order to improve efficiency of the system.

1.2 Detailed Implementation of some components

- What **language and tools** we use for this project?

  We use Java language and Eclipse IDE with AWS Toolkit.

- How do we generate the **workload file**?

  We write a program to generate the workload file, corresponding to each benchmark request.

- How is each **task** represented?

  Each task is encapsulated into a serializable object, which contains three attributes: task ID, task description, and result.

  When the client generate each task, the task ID is a random string, the task description is the contents read from the workload file, and the result is set to “null”.

  The result will be rewrite by either local or remote task.

- How does the frontend scheduler detect **local worker’s completion**?

  Each local worker thread is encapsulated into an callable class.

  An ExecutorCompletionService object is declared at front-end to submit each callable thread. Then, the thread which is in charge of send the result back to client only need to call the take() method of this ExecutorCompletionService object, to get a completed task object.

  It is guaranteed that this take() method will return not only a completed task, but also the first completed task so far.

- How does the frontend detect **remote worker’s completion**?

Since the front-end and remote worker communicate through SQS queue, the front end periodically spawn a new thread to poll from the SQS queue, and send them back to client.

However, sometimes the queue may be empty when the front-end poll it. In order to keep the stream between the client and front-end alive, the front-end will send a “fake task” instead. The client has to check whether the received task is a real one or fake one.

- How do we implement **multi-threaded** remote worker? (5 extra credits)

  The remote worker retrieves 20 messages from SQS queue at the same time, and submit all these tasks to a thread pool contains runnable thread instances, which are the same as the threads of the local workers. At the same time, another thread is spawned to call take() method to retrieve any completed task and send them back to another SQS queue.

  This is the only extra credits components we have done in this assignment.

- How does the **remote worker start working** by itself when launched?

  We edit `/etc/rc.local` as root. Insert one line (assuming location of the runnable jar file is at `/home/ubuntu/prog4worker.jar`):

  ```
  java -jar /home/ubuntu/prog4worker.jar > /home/ubuntu/log.txt
  ```

  into this file. And create an image using this instance.

- How does the **remote worker terminate itself**?

  We have a line commented out during our experiments for convenience, which is:

  ```
  (new ProcessBuilder("shutdown", "-h", "now")).start();
  ```

  after a duration of idle time (fetch a message from task queue every half-second, until a non-empty message is obtained).

- What is the dynamic **provisioning algorithm**?

  The algorithm keeps two variables: current length, and previous recorded length of task queue. When current length is longer, start a new worker. When current length is shorter, investigate if all the tasks in task queue can be consumed under current consuming rate ($\frac{previous\ length - current\ length}{time\ passed}$) in the time of starting up a new worker. If the all the tasks can be consumed in two and a half minutes (usual starting up time for

spot instance m3.medium), then don’t start a new worker. If cannot, start a new worker. Update current length, and previous length. Goto the beginning, and loop.

2. Manual

In order to get the program running, a user needs to follow the instructions below:

- Create two SQS queue, name it “Tasks” and “Results”
- Create a DynamoDB table, name it “TaskTable”, which contains primary key named “TaskID”
- Export each java package into jars;
- Save the system which contains the jar of remote worker into an Amazon image;
- Launch 3 Amazon EC2 instances, and upload the client, front-end, and provisioner jars onto each instance;
- If using local worker:
    - start the frontend scheduler by:

        ```
        java -jar prog4frontend.jar <port> -lw <nthreads>
        ```

    - start the client by:

        ```
        java -jar prog4client.jar <ip> <port> <file>
        ```

- If using remote worker:
    - [static mode] start the provisioner by:

        ```
        java -jar prog4provision.jar static <nworkers>
        ```

    - [dynamic mode] start the provisioner by:

        ```
        java -jar prog4provision.jar dynamic
        ```

    - start the frontend by:

        ```
        java -jar prog4frontend.jar <port> -rw
        ```

    - start the client by:

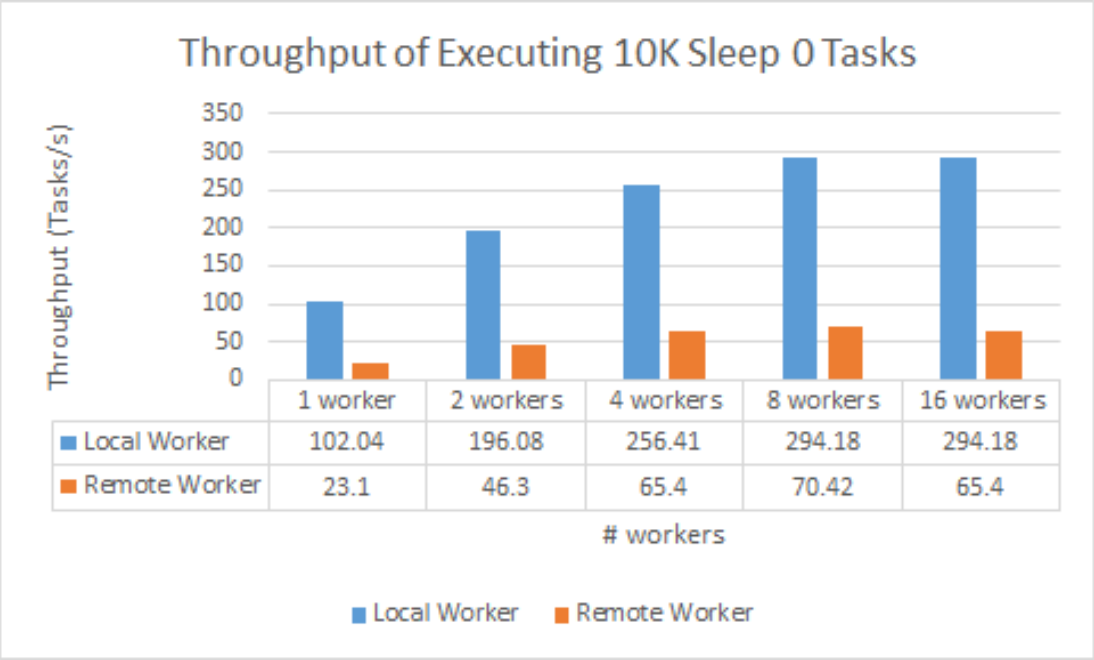
        ```
        java -jar prog4client.jar <ip> <port> <file>
        ```

3. Performance Evaluation

(Note: our experiments are all run on m3.medium instances.)

3.1 Throughput Evaluation

The throughput is calculated by 10000 / execution_time. For each experiment, a workload file contains 10k “sleep 0” is past to the client. The client records the time first task object is sent to front-end, and the time it received the result of the last task.

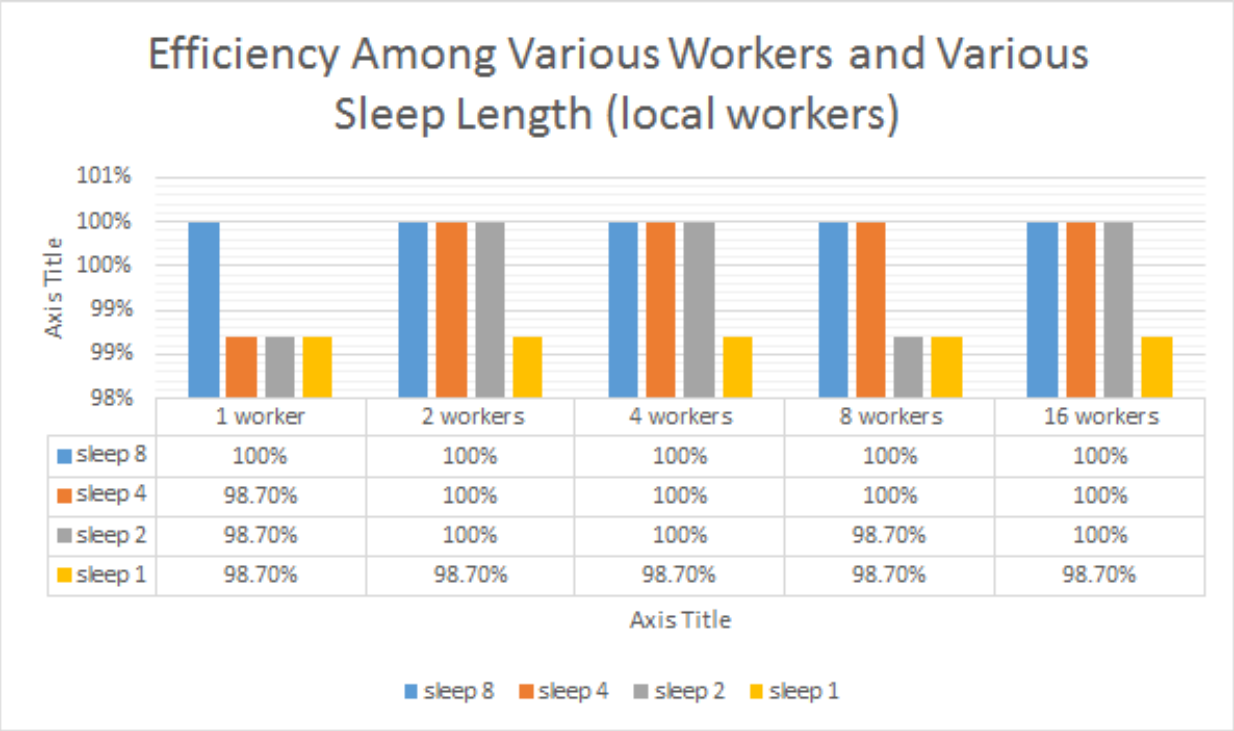


Both local worker and remote worker scale from 1-worker to 2-worker *idealy*. Their throughput improvements brought by more parallelism both stop working at 8-worker setting. For remote worker experiments, by observing dynamics of queue length at AWS SQS monitor, we find that starting at 8-worker setting, our frontend can no longer keep up with the pace the multiple workers are generating; e.g., 0 task in task queue, but more than a thousand results in result queue. Hence, the parallelism capacity of AWS services are impressive, and the throughput in our system is bottlenecked at frontend.

3.2 Efficiency Evaluation

The efficiency is calculated by 80 / execution_time. For each experiment, a workload file is generated in a way such that every worker will have the average sleep time to be 80 seconds. The client records the time first task object is sent to front-end, and the time it received the result of the last task.

3.2.1 Local workers



Running local workers show great efficiency in all experiments. `sleep 1` experiments do show lower efficiency due to much more overheads with the same amount of real work.

3.2.1 Remote workers

Let's first take a detailed look at a simplest case where there is only 1 worker, and 1 task, which is `sleep 1`:

- a task is read from disk at the client's machine, wrapped into a *task* object, and sent to the frontend over a TCP connection;
- the frontend then wrap the received *task* object into a SQS message, and sends it off to the task queue;
- 1 worker fetches messages in every half-second while idle, until a non-empty message is retrieved. Once the message is retrieved, worker first send its task *id* off to task table (using DynamoDB) for consistency check. If the task *id* is proved to be valid, worker pass the task to local process to execute; if invalid, ignore and idle;
- local process `sleeps` for 1 second, and informs worker the success of the run;
- worker wraps updated *task* object (with result) in a SQS message, and sends it off to result queue, from which frontend fetches once every 10ms;

- when the frontend retrieves a non-empty message from result queue, it immediately carves out the task object within, and sends it back to client over a TCP connection;
- the client reads the resulting *task* object from TCP connection and displays it content on screen; and the user is notified that the task has been finished.

The entire process above consists of 8 transfers between 6 entities:

- client *to* frontend *to* task queue *to* worker *to* task table *back to* worker *to* result queue *back to* frontend *back to* client.

Denote the overhead in this process as Δt seconds, for 1-worker experiments, we have their total run time (in seconds) presented as follows:

- 8 sec, 10 tasks: $(8+\Delta t)\times 10 = 80 + 10\Delta t$;
- 4 sec, 20 tasks: $(4+\Delta t)\times 20 = 80 + 20\Delta t$;
- 2 sec, 40 tasks: $(2+\Delta t)\times 40 = 80 + 40\Delta t$;
- 1 sec, 80 tasks: $(1+\Delta t)\times 80 = 80 + 80\Delta t$.

How about 2-worker experiments? What becomes different? What remains the same?

First, for a single task, it goes through the exact same process as in 1-worker experiments.

When considering the process of many tasks, we divide the previously standalone overhead Δt into two parts:

- front-end overheads: Δt_{front}, and
- back-end overheads: Δt_{back}.

The reason of this partition is that, while the front-end operations (consists of the first two and the last two transfers, not to be confused with frontend, which is one of the 6 entities involved in this system) remains as sequential as in 1-worker experiments, the back-end operations (consists of the six transfers in the middle) differs in that it is now “totally” parallelized. All the operations one worker needs to to, the other worker does exactly the same. We have total run time (in seconds) of all the 2-worker experiments presented as follows:

- 8 sec, 20 tasks: $(\frac{8+\Delta t_{back}}{2} + \Delta t_{front})\times 20 = 80 + 10\Delta t_{back} + 20\Delta t_{front}$;
- 4 sec, 40 tasks: $(\frac{4+\Delta t_{back}}{2} + \Delta t_{front})\times 40 = 80 + 20\Delta t_{back} + 40\Delta t_{front}$;
- 2 sec, 80 tasks: $(\frac{2+\Delta t_{back}}{2} + \Delta t_{front})\times 80 = 80 + 40\Delta t_{back} + 80\Delta t_{front}$;
- 1 sec, 160 tasks: $(\frac{1+\Delta t_{back}}{2} + \Delta t_{front})\times 160 = 80 + 80\Delta t_{back} + 160\Delta t_{front}$.

Similarly, for all the rest experiments, we have:

- 4 workers

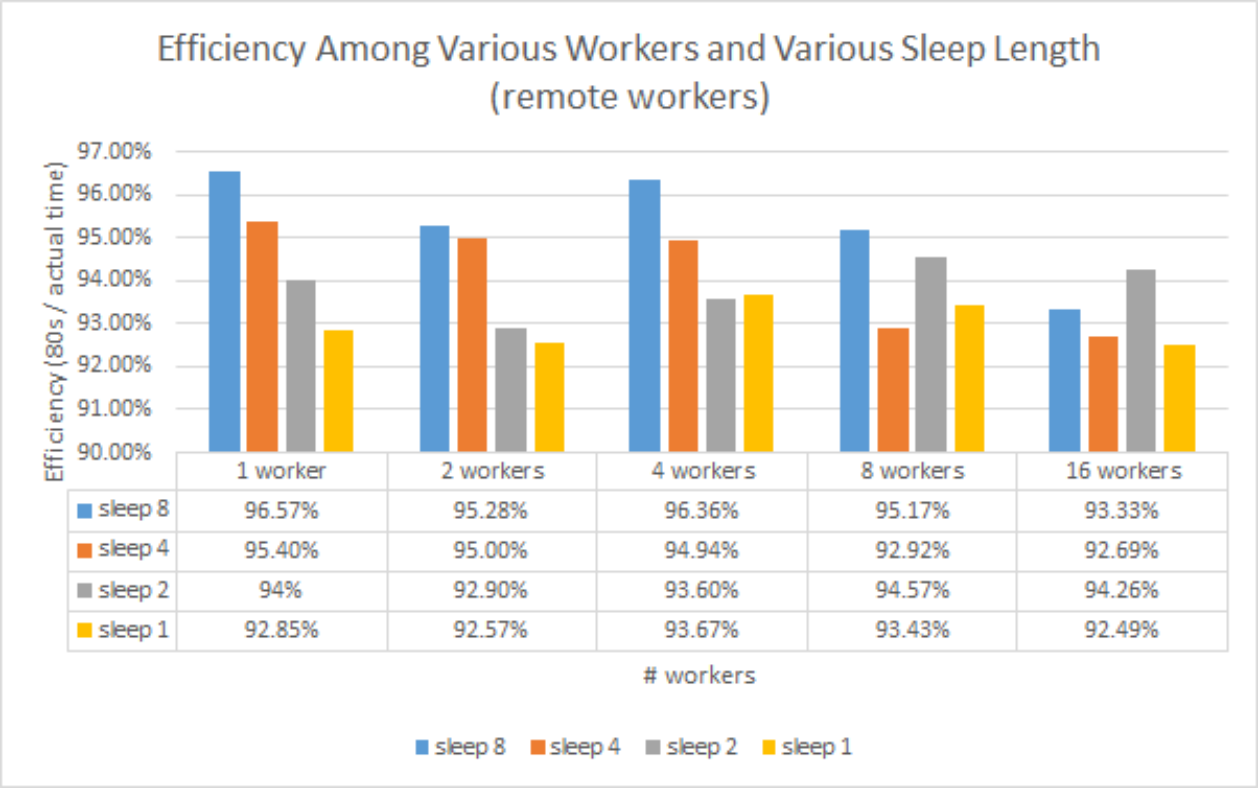
- 8 sec, 40 tasks: $(\frac{8+\Delta t_{back}}{4} + \Delta t_{front}) \times 40 = 80 + 10\Delta t_{back} + 40\Delta t_{front}$;
- 4 sec, 80 tasks: $(\frac{4+\Delta t_{back}}{4} + \Delta t_{front}) \times 80 = 80 + 20\Delta t_{back} + 80\Delta t_{front}$;
- 2 sec, 160 tasks: $(\frac{2+\Delta t_{back}}{4} + \Delta t_{front}) \times 160 = 80 + 40\Delta t_{back} + 160\Delta t_{front}$;
- 1 sec, 320 tasks: $(\frac{1+\Delta t_{back}}{4} + \Delta t_{front}) \times 320 = 80 + 80\Delta t_{back} + 320\Delta t_{front}$.

- 8 workers
  - 8 sec, 80 tasks: $(\frac{8+\Delta t_{back}}{8} + \Delta t_{front}) \times 80 = 80 + 10\Delta t_{back} + 80\Delta t_{front}$;
  - 4 sec, 160 tasks: $(\frac{4+\Delta t_{back}}{8} + \Delta t_{front}) \times 160 = 80 + 20\Delta t_{back} + 160\Delta t_{front}$;
  - 2 sec, 320 tasks: $(\frac{2+\Delta t_{back}}{8} + \Delta t_{front}) \times 320 = 80 + 40\Delta t_{back} + 320\Delta t_{front}$;
  - 1 sec, 640 tasks: $(\frac{1+\Delta t_{back}}{8} + \Delta t_{front}) \times 640 = 80 + 80\Delta t_{back} + 640\Delta t_{front}$.

- 16 workers
  - 8 sec, 160 tasks: $(\frac{8+\Delta t_{back}}{16} + \Delta t_{front}) \times 160 = 80 + 10\Delta t_{back} + 160\Delta t_{front}$;
  - 4 sec, 320 tasks: $(\frac{4+\Delta t_{back}}{16} + \Delta t_{front}) \times 320 = 80 + 20\Delta t_{back} + 320\Delta t_{front}$;
  - 2 sec, 640 tasks: $(\frac{2+\Delta t_{back}}{16} + \Delta t_{front}) \times 640 = 80 + 40\Delta t_{back} + 640\Delta t_{front}$;
  - 1 sec, 1280 tasks: $(\frac{1+\Delta t_{back}}{16} + \Delta t_{front}) \times 1280 = 80 + 80\Delta t_{back} + 1280\Delta t_{front}$.

Is the analysis above totally correct? Or is there something wrong? Does it have certain assumptions? Are they reasonable? Most importantly, do real results of runs verify the analysis?

After running the 4×5 experiments, each with 3 runs, and get average run time, we can plot the following chart:



	1 worker	2 workers	4 workers	8 workers	16 workers
sleep 8	96.57%	95.28%	96.36%	95.17%	93.33%
sleep 4	95.40%	95.00%	94.94%	92.92%	92.69%
sleep 2	94%	92.90%	93.60%	94.57%	94.26%
sleep 1	92.85%	92.57%	93.67%	93.43%	92.49%

Aside from these finals results in chart, we have 2 major observations during experiments:

- in all experiments, frontend feeds task queue much faster than workers consume;
- in all experiments, client receives many “fake” results (utilized as heartbeat in order to keep TCP connections alive) from frontend, which means that the front-end retrieves results much faster than workers produces.

According to the two observations, we conclude that front-end overheads Δt_{front} doesn’t play big role in total overheads. And the total run time is largely dominated by 80 sec sleep time plus back-end overheads.

Thus, we have total run time for experiments with various number of workers:

- 8 sec experiments: $80 + 10\Delta t_{back}$;
- 4 sec experiments: $80 + 20\Delta t_{back}$;
- 2 sec experiments: $80 + 40\Delta t_{back}$;
- 1 sec experiments: $80 + 80\Delta t_{back}$.

The chart confirms this, to certain degree. Experiment results within one cluster show a trend that efficiency decreases as sleep time shrinks (or, the total run time increases as sleep time

shrinks). While experiment results across clusters for a given sleep time show a degree of evenness, suggesting that the elimination of front-end overheads in our analysis above is reasonable, and demonstrates scalability of AWS SQS service and DynamoDB service.

3.3 Dynamic Provisioning Evaluation

Five runs of dynamic provisioning experiment (one `sleep 1` task with no worker beforehand):

- 135,066ms
- 198,935ms
- 124,454ms
- 142,970ms
- 154,219ms
- **Average: 151,129ms, approximately 2min 30sec.**

Five runs of static provisioning experiment (one `sleep 1` task with worker ready):

- 1,980ms
- 3,135ms
- 1,846ms
- 5,262ms
- 4,058ms
- **Average: 3,256ms, approximately 3sec.**

The dynamic provisioning experiment basically measures the start up time for our experiment instance type m3.medium. Hence, the latency is huge when compared with static provisioning, where the worker instance is already up-and-running. The experiment doesn't exploit the advantage of dynamic provisioning through.