Cloud Computing and Big Data Coursework Report: Fault Tolerant Cloud Application for Mandelbrot Set

Jonah Lyu

University of Bristol, Bristol, United Kingdom

Abstract. This report is for the coursework of the unit Cloud Computing and Big Data. It introduces the implementation of a cloud application which computes Mandelbrot Set in the embarrassingly parallelisable style, including the use of Zookeeper to building a robust network among clusters of AWS EC2 nodes. Then it analyses the cost of computing across various settings. Finally, it discusses the fault-tolerant feature and evaluates the scalability of servers with multiple client connections at the same time.

Keywords: Could Computing, Fault Tolerant, Zookeeper, Iaas, Mandelbrot Set,

1 Introduction

The coursework requires to implement a fault-tolerant cloud application which computes an embarrassingly parallelisable task. With this requirement, I first did some researches on the concepts of the embarrassingly parallel task. I finally decided to pick the task of computing of Mandelbrot Set. The first step of implementing the cloud application is to build a master and worker network which allows a master to distribute a task to a worker.

Based on this basic task assignment mechanism, an election system is required to pick a master from all master candidates to ensure that there is always existing a master in the cluster. The final step is to implement a client that is aiming for the generation of tasks. Each worker's job is simply waiting for a task to be assigned and computed. In this case, A task will always include a slice of Mandelbrot Set figure.

The research will mainly focus on the fault-tolerant feature and scalability of servers. All the experiments will measure the time cost between a client send out tasks and get results back.

2 Choose of Task

The embarrassingly parallel describes a paradigm in computing where tasks run at the same time and have no dependency or communication with each other[1].

It is suitable to deploy this kind of tasks in a distributed system that utilizes the best performance of all computing resources.

There are various examples of embarrassingly parallelisable tasks. For instance, brute-force password cracker, distributed relational database queries and rendering of computer graphics. I finally decide to implement a service to calculate the Mandelbrot Set.

The Mandelbrot Set is the set of complex numbers c for which the function

$$f_c(z) = z^2 + c \tag{1}$$

does not diverge when iterated from $z = 0$ [2]. The generated set will form a figure where each pixel is calculated independently. Therefore, it is suitable to make the computation parallel by wrapping up each row of the figure into an independent task.

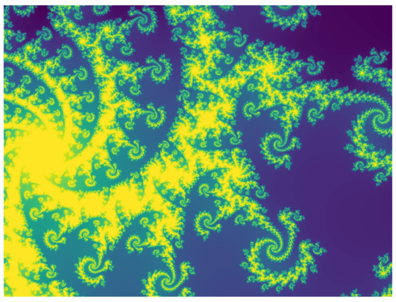

Fig. 1. Example generated figure of Mandelbrot Set with *1024*768 resolution* and *2000.0* times zoom in.

There is another benefit of choosing the Mandelbrot Set. With the different combinations of image size, zoom level and colour depth, the cost of calculation on each row can be various. The randomness on each task simulates the real connection of server and client to some extent. It avoids an ideal scenario - each task costs the same time and compute resource, which is far from reality.

3 Instance Configuration

I have launched three instances of AWS EC2[3] and built a Zookeeper[4] cluster with one leader and two followers. The tech stack is listed below:

1. *Zookeeper* - to synchronize data and status across all nodes in the cluster.
2. *Python* - the scripts for starting and stopping all servers and clients.

3. *Fabric* - to establish the SSH connection from a node.
4. *Numpy* - the core library for computation and storage.
5. *Matplotlib* - the library to plot the figure from the two-dimensional array.

From the default setting, each instance will launch one master server and two worker servers. So, it gives 3 master servers and 6 worker servers on the run. But it is easy to scale the number of instances up and down by editing the config file in the repository and manually launch or stop ones.

4 Server Topology

There are two types of servers in the cluster: master server and worker server. With the default settings, one master and two workers are running on each node. But only one master is doing the actual work of task assignment. The other masters join the election and become backups of the current master.

Master Server A master server is for assigning tasks created from client to all free worker. It also has the job of listening to any change in the status of both workers and clients. So it ensures that no blocked task in the task queue when a worker died.

Worker Server A worker server is for computing the task assigned by the master and writing the result back to the corresponding node. Each worker has its own identity and life cycle in the cluster. Therefore, no effect on other cluster components when a worker added or died.

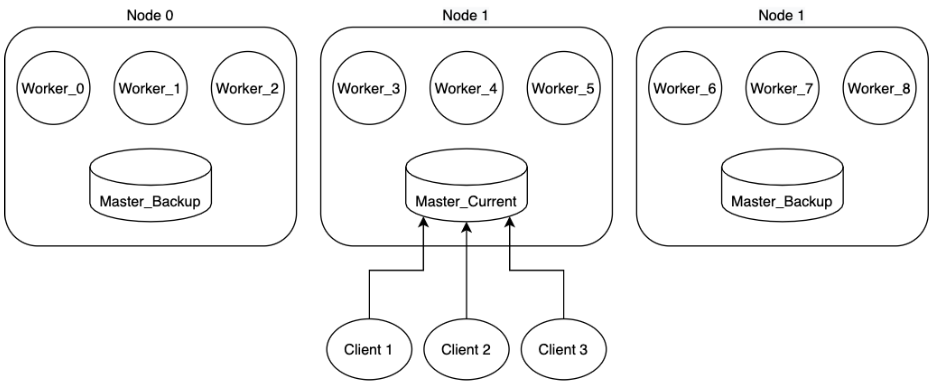

Fig. 2. Server Topology with 3 nodes running 1 master, 9 workers and 2 master backups.

Election of Master A ballot of election takes place when the whole system start or the current master dies. All backups of the master will only generate one new master. The algorithm always picks the candidate with the smallest sequence id.

Znode Design Znode is the core feature of Zookeeper used for synchronizing. The watching mechanism of Znode can help a component know the latest event taken place the cluster. All types of Znode are listed:

1. **/master/{master id}** - the children of this path are all masters' id, a child node is ephemeral with master connection. All masters will watch this path. If the current master dies, all master backups will elect a new master.
2. **/workers/{worker id}** - the children of this path are all workers' id, a child node is ephemeral with worker connection. All masters will watch this path.
3. **/status/{worker id}** - the children of this path are all workers' id with worker status inside, a child node is ephemeral with worker connection.
4. **/clients/{client id}** - the children of this path are all clients' id, a child node is ephemeral with client connection. All masters will watch this path.
5. **/tasks/{task id}** - the children of this path are all tasks' id with task status, its client id and assigned worker id inside. Each task child node is ephemeral with its client connection.
6. **/params/{task id}** - the children of this path are all tasks' id with the parameters of task inside. Each parameter child node is ephemeral with its client connection.
7. **/results/{client id}/{task id}** - the children of this path are all tasks' id with the compute result of each task inside. Each result node is persistent. All clients will watch this path.

5 Client Implementation

The main job of a client is to divide the figure into multiple sub-tasks and send them to the task queue. Each task contains a slice of the final figure of the Mandelbrot Set. A user can input the number of slices in the client to tell the client how many sub-tasks to send. For instance, a user inputs value 6 means a client will slice the figure into 6 uniform pieces horizontally. A sub-task contains multiple parameters: start row index, end row index, resolution of the output and zoom level of the Mandelbrot Set. After sending a task out, the client will watch the */result* path until all tasks have a result write back.

An example of task generation is slicing 1024*768 figure into 3 pieces with 1 zoom level. The client will generate following sub-tasks finally:

```
/tasks/0_0000000395 b'client_id'
/params/0_0000000395 b'1024:768:1.0:0:256'
/tasks/1_0000000396 b'client_id'
```

```
/params/1_0000000396 b'1024:768:1.0:256:512'
/tasks/2_0000000397 b'client_id'
/params/2_0000000397 b'1024:768:1.0:512:768'
```

Each parameter data indicates: *height:width:zoom:start_row:end_row*. The master will watch the */tasks* path at the same time. If there is a new task created, it will trigger a task assignment procedure.

6 Scalability

Scaling up or down the number of worker servers is easy by editing the variable in file *start_server.py*.

```
#edit these lines to declare number of server
MASTER_NUM_ON_EACH_NODE = 1
WORKER_NUM = 6   # deploy uniformly on all nodes

#terminal
> python start_server.py master
> python start_server.py worker
```

This script will stop all servers and redeploy them. If you want to manually add a new master or worker without stopping the existing servers, use the following commands to run in foreground.

```
#terminal
> make master   # adding a new master
> make worker   # adding a new worker
```

To test the scalability of the cloud application, the following experiment will scale up the number of worker servers, each worker processes 1 task. As recorded in the following table, the number of workers is scaled up from 1 to 6. The client will send a task with 1024*768 resolution and 100.0 zoom level.

Table 1. TIme cost vs number of workers

Number of Workers	Time cost of task in seconds
1	27.27s
2	16.15s
3	15.13s
4	14.8s
5	13.32s
6	12.26s

The table clearly shows the benefits of the scaling up the worker servers. In the case of one worker, only one node is doing the computation. Two workers bring an obvious improvement on the speed. The time decreasing is less obvious with more workers. It brings more communication between master and worker to sync each task status.

7 Fault Tolerant

The previous implementation section shows how the whole system is established. In this section, it introduces multiple experiments to test the fault-tolerant feature. There are different fault scenarios. I will label each case in a short form where *M* represents *Master*, *W* represents *Worker* and *C* represents *Client*. Four cases are introduced:

1. *2M:2W:1C* to *1M:2W:1C* - test 1 master fails
2. *2M:2W:1C* to *2M:1W:1C* - test 1 worker fails
3. *2M:2W:1C* to *2M:0W:1C* - test all worker fails

7.1 Test 1 master fails - *2M:2W:1C* to *1M:2W:1C*

This experiment will test the fail of the current master when a client is connecting to it. An interrupt signal will stop the current master while it is on the halfway of a client task.

```
----------
python ./app/master.py   # master 1
...
My id is:  id_0000000034
Master is: /master/master_current/id_0000000034
I am master now
...
#Interrupt signal
----------
python ./app/master.py.  # master 2
...
My id is:  id_0000000035
Master is: /master/master_current/id_0000000034
I am master backup now
Master just died, new master election start...
Master is: /master/master_current/id_0000000035
I am master now
----------
```

The result shows that the master can transfer to a new one when the old one is killed by an interrupt signal. In this case, all workers can get a task from master and clients can get the correct result in the end. So it tolerates the current master fault.

7.2 Test 1 worker fails - *2M:2W:1C* to *2M:1W:1C*

This experiment will test the fail of a worker when it is processing a client's task. An interrupt signal will stop one of two workers.

```
----------
python ./client.py   # client 1
2 workers
1 masters
create: /results/client_3c161852-d05f-46db-8f84-790dd0d7623a
enter number of slices to deploy: 10
...
task progress: 1/10
task progress: 2/10
1 workers. # one of the worker dies
task progress: 3/10. # the task is still going running
task progress: 4/10
...
----------
python ./app/master,oy.  # master
...
Worker died 1e2fe38f with status: 1_0000000429
Task free: 1_0000000429.   # free the task from died worker
...
Try to distribute 1_0000000429 to worker: 9a928e11
...
----------
```

The output shows there is no effect when one of the workers dies. This is because the master will free the assigned task from the died worker and assign it to a new worker.

7.3 Test all worker fails - *2M:2W:1C* to *2M:0W:1C*

This experiment will test the fail of all workers when they are processing a client's task. All workers are stopped by an interrupt signal. Then I will launch a new worker to see if a client task can be recovered.

```
----------
python ./client.py   #client 1
2 workers
1 masters
create: /results/client_3c161852-d05f-46db-8f84-790dd0d7623a
enter number of slices to deploy: 10
...
task progress: 0/10
task progress: 1/10
```

```
0 workers.   # all worker fails, client stucks here
...
1 workers.   # relaunch a new worker, task is recovered
task progress: 2/10
2 workers
task progress: 3/10
...
----------
```

The output shows a client will wait if all workers die and it can be recovered when at least one worker relaunched.

All experiments have proved that this cloud application is resilient to various fault scenarios. With the larger scale of server deployments, the service will become more stable and highly available. There are three highlights in the design of the system to ensure its fault-tolerant feature:

1. Election - to ensure there is always a master working.
2. Unblocked Task - a task will always free from a died worker.
3. Independent Client Status - each client has an ephemeral node to store its status.
4. Persistent Result - all client result is stored persistently and can only be accessed by the master.

8 Conclusion

This cloud application can compute the Mandelbrot Set in multiple embarrassingly parallelisable tasks. It is resilient to various fault scenarios and easy to scale up or down. It also supports numbers of clients to connect at the same time. At the end of the task, a beautiful figure of Mandelbrot Set will be generated.

References

1. Wikipedia contributors, "Embarrassingly parallel," Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Embarrassingly_parallel&oldid=988058646` (accessed December 4, 2020).
2. Wikipedia contributors, "Mandelbrot set," Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=991934444` (accessed December 4, 2020).
3. Wikipedia contributors, "Amazon Elastic Compute Cloud," Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Amazon_Elastic_Compute_Cloud&oldid=991810523` (accessed December 6, 2020).
4. "Zookeeper" Apache ZooKeeper `https://zookeeper.apache.org/` (accessed December 6, 2020).

Appendix

GitHub Repository: `https://github.com/ccdb-uob/CW20-20`