Cloud Nonce Discovery

COMSM0010 Cloud Computing Coursework1

Tim Chen yz19086@bristol.ac.uk P1955183 1st December 2019

1. Introduction

In this project, we need to implement a Cloud Nonce Discovery system to scale out a compute-intensive problem which is embarrassingly parallelizable. As a crucial part of the Proof of Work (PoW), the problem calls the $SHA256^2$ algorithm on "block+nonce" string and judge the quantity of the consecutive zeros in the result's leftmost string, when the quantity is equal to or greater than a difficulty parameter D, then the nonce is called a golden nonce. The CND system should use Amazon Web Service (AWS) and run N instances according to direct or indirect specification, and then scale out this problem to these instances to decrease the expected time it has to spend. At the end, as soon as we find the golden nonce or complete all the work without a golden nonce, or the running time reach timeout, we should stop all the instances and tidy up the system nicely.

For actual realization, we used Python language and Boto3 framework to build a CND system which is running on a local machine and remotely invokes the Amazon Simple Queue Service (SQS) and the Elastic Compute Cloud Service (EC2) scaling out the whole task to each EC2 instances. With AWS Command Line interface, we debug the whole system as well as the remote end with specific console output of instances. We also use the CloudWatch service and the execute time gained by the SQS to do some performance analysis and then finish the indirect specification of instances' quantity with the confidence level and expected time given by users. For further extension, We choose a different way dividing the whole problem to reduce the extra expense of the different time spending on the initialization of instances. What's more, We also complete a multiprocessing component to utilize a dual-core processor more efficiently, which gains a better performance comparing to 2 single-core processors.

2. CND System

In the system, we choose to transfer a python code and specific parameters to each instance and then execute the code by defining the user data, finally letting instances do different work with a same code. Among this using the SQS service to act as a task queue in the middle and transfer the data between different components. We mainly divide the CND System into 4 parts: configuration of Boto3 framework, run and configure EC2 instances, manage and process SQS queues, and the codes of computation for nonce executed by each instance. We will give a diagram (Figure 1) for an Intuitive understanding of the user logic and then specifically discuss how to utilize these service to implement each part.

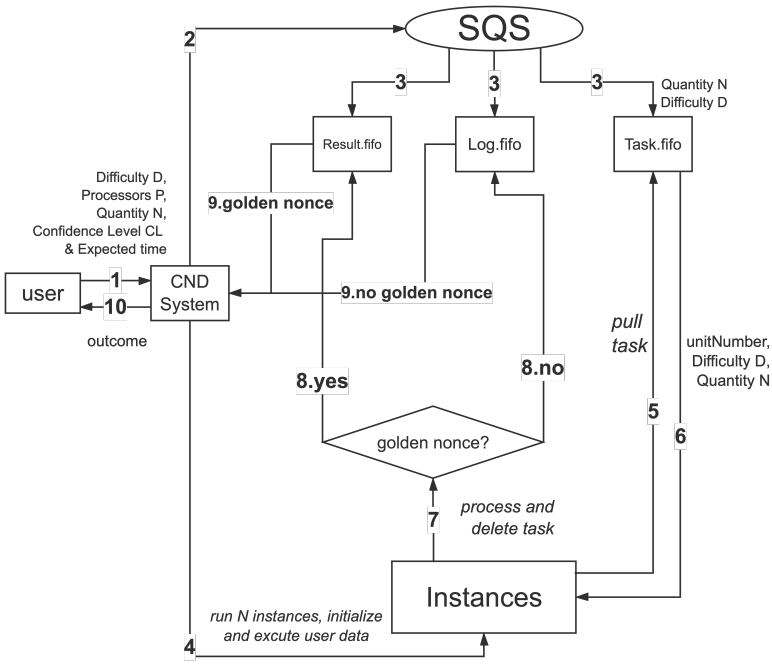


FIGURE 1

2.1.Boto3 framework

Boto3 is a sophisticated AWS SDK for python, it enables the python developers to remotely configure and manage AWS services.

For configuration, the only certificate that Boto3 uses to recognize and access our AWS user resource is "*aws_access_key_id"*,

"aws_secret_access_key" and the "aws_session_token". However, the way we gain the access key is different from the normal account's since the account we use is educate account. So we don't need to create an Identity and Access Management (IAM) user, instead, we can gain it from our educate account detail interface and copy it to the "./aws/credentials" file. The access key will expire every 3 hours or at every time we refresh the webpage, thus, every time the key has expired, copy and paste them again.

2.2.EC2 Instances

In this project, we use the Ubuntu 18.04 Amazon Machine Image, and the free t2.micro instance type (use the t2.medium instances when choose multiprocessing, we will illustrate it in the further extension chapter). We also create a security group and a key pairs which enable us to access the instances with a ssh by local machine's IP address (which is fully illustrated in the lectures).

2.2.1.User data

Running script on a single instance or running multiple instances are simple, but how to launch multiple instances and run PoW on each of them is the most critical problem of this coursework. As for this part, we choose to run script through the user data. User data is an attribute of an instance, and it will be execute for only once by the cloud-init tool as the instances launch, which means if you stop the instances and start it again, it won't be execute for the second time. The user data takes effect by the cloud-init, which is a tool to initializing the environment for server (Cloud-init is innate in the Ubuntu images 18.04). Our user data is stored in the meta data of the server, and cloud-init take this data to execute. We can see this process from the instances' logs in the following directory (Figure 2).

```
/var/log/cloud–init.log
/var/log/cloud–output.log
```

FIGURE 2

Specifically, user data has many types distinguished by the format of the data, what we used is called user data script that begin with a "*#!/bin/bash*" tag followed by several commands. Here is a sample (Figure 3) of defining a user data in Python.

```
1. user_data="'#!/bin/bash
2. cd /home/ubuntu/
3. git clone https://github.com/USER/XXX.git
4. cd /home/ubuntu/XXX/
5. Python3 YOURCODE.py
6. "''
```

FIGURE 3

Every instances have an independent environment, which has an innate Python3, but this is not sufficient for the whole project, so we update and upgrade the Advanced Packaging tools (APT) and then use it to install pip package for Python3 and finally use *pip3* command install Boto3 SDK. What's more, each instance has to configure the access key for Boto3 because of the access to SQS service, by which our local machine can receive the log and result from instances. By the way, we haven't use Simple Cloud Storage Service (S3) to store our codes and resources, instead, we choose the Github since it's simpler to manage through the user data script. We clone a directory from Github which contains the python code to execute. It's worth mentioning that scripts entered as user data are executed as the root user, and any files we create will be owned by root, so when we configure the Boto3 environment, we need to create the *./aws* directory in the root directory instead of the *ubuntu* directory and configure the access key(as shown in Figure3), or we can't use AWS service by instances, which puzzles me for long.

2.2.2.CPU credits

When we define different quantity of instances work on the PoW computation, we found that if the quantity is less than 7, the performance of the t2.micro will fall to about 10% in the end according to the CloudWatch. Such a performance is surely insufficient for the instance to finish the work within the expected time (as shown in Figure4).

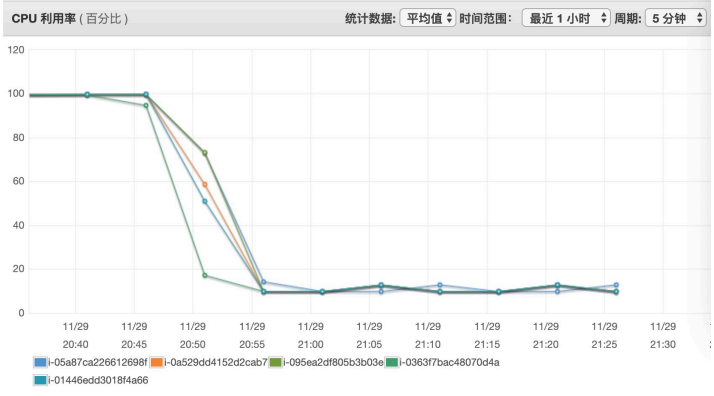


FIGURE 4 — CPU Utilization (falling before set)

Through a lot of survey, we found that the baseline level of t2.micro is 10%. We also find that T2 instances can provide a baseline level of CPU performance with the ability to burst above the baseline when needed, and whether the performance is on baseline or not is depend on a data called CPU credits. When the credits are positive, it consume the credits to let the performance burst, however, when the CPU is free or the performance falls to the baseline, it accumulates the credits (as shown in Figure 5 and 6).

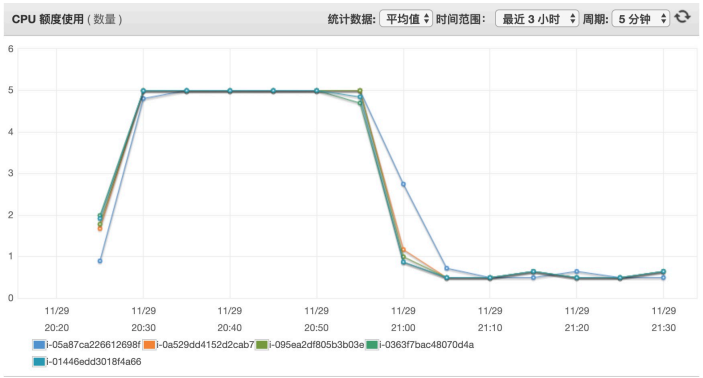


FIGURE 5 — CPU Credit Usage (before set)

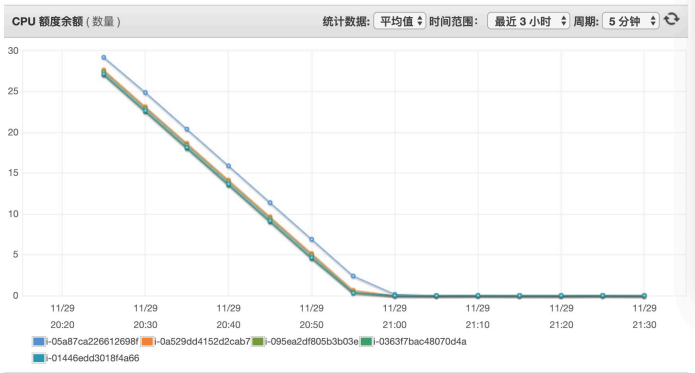


FIGURE 6 — CPU Credit Balance (before set)

To solve this problem, we have to set the attribute “CreditSpecification” when use run_instances function (as shown in Figure 7).

CreditSpecification={'CpuCredits': 'unlimited'}

FIGURE 7

In this way, though we have run out of the credits, the performance of T2 instances won’t fall (as shown in Figure 8 and 9).

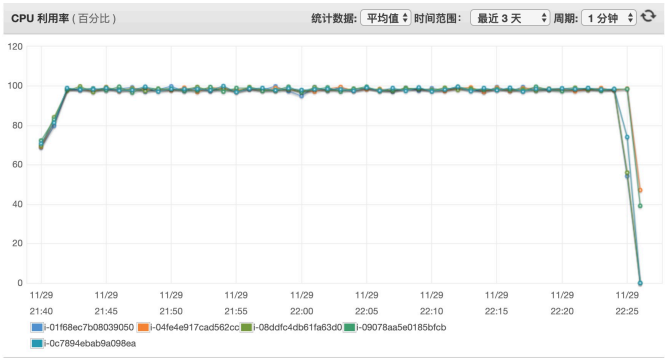


FIGURE 8 — CPU Utilization (stable after set)

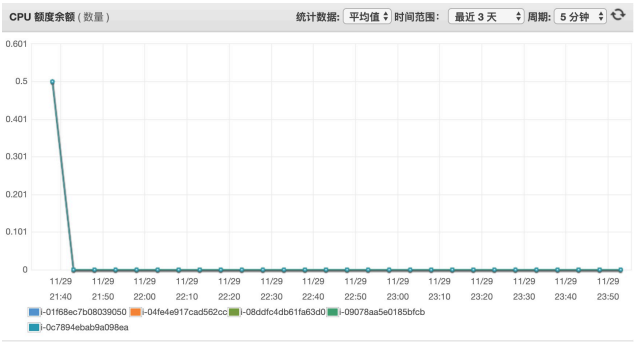


FIGURE 9 — CPU Credit Balance (after set)

2.2.3.EC2 Client and resource

In the CND system, we use EC2 client object and EC2 resource object to manage and run the instances. As soon as we find that running time reaches the timeout we set, or we fail to get a golden nonce from all the nonce, or luckily we find a golden nonce, we should terminate all the instances instantly.

The function we mainly use in this system through EC2 client is “run_instances”, in which we should define the “AMI Id”, “instance type”,”key pair name”,etc. It is worth mentioning that we need to set the attributes “MinCount” and “MaxCount” to equal the quantity of the instances we want to run, instead of run a single instance in a loop (as shown in the Figure 10). Because run it in a loop will increase the difference of the time when our instances finish the initialization and begin to work, which more likely leads to a circumstance that some instances have done their work but have to wait for other instances’ completion.

```
1.  ec2_client.run_instances(
2.              ImageId=image_id,
3.              InstanceType=instance_type,
4.              KeyName=keypair_name,
5.              MinCount=instancesNumber,
6.              MaxCount=instancesNumber,
7.              SecurityGroups=security_groups,
8.              UserData= user_data,
9.              CreditSpecification={'CpuCredits': ‘unlimited'}
10.             )
```

FIGURE 10

For resource object we mainly use the function “instances.terminate()” to terminate the all instances without point out the instances ID instead of using client.terminate_instances function (as shown in Figure 11). What’s worse, it’s arduous to get all the instances ID and write a loop to terminate them one by one.

```
1. ec2 = boto3.resource('ec2')
2. ec2.instances.terminate()
```

FIGURE 11

2.3.SQS

In the CND system, we choose SQS FIFO (First In First Out) queues instead of the standard queues to implement specific functions. Because the standard queues have features of “Best-Effort Ordering” and “At-Least-Once Delivery”, which means there is no serious sequence among the standard queues messages, what’s worse, it also has the risk to send a same message for more than once. Such features severely prevent us from keeping every task unit being process for only once after we divide the whole problem into separate task unit through a task queue. Moreover, the sequence of the messages is also crucial for us to figure out the state of every instance and get the time consumed during processing the PoW. On the contrary, FIFO queues have the features of “First-In-First-out Delivery” and “Exactly-Once Processing”, which ensure a strictly preserved order among messages and a prohibition on duplicates. Although FIFO queues only support 300 message operations per second (delete, send, receive), it has met our requirements. So the FIFO queue is the only choice provided that we choose SQS to act as message queue in CND system.

We create 3 queues called Task.fifo, Result.fifo, Log.fifo to satisfy different requirements of our CND system. The following content is about how they are defined and the role they play in the system, and after this, we illustrate how we judge the outcome of execution from these queues.

2.3.1.Task.fifo

The Task.fifo queue plays as a task queue in the middle of CND system scheduling the task unit list. By user data, we can run multiple instances and let them work on the PoW, but it’s not enough. We still have to transfer some critical data to each instance like the total quantity of instances, the difficulty we need compute, and which unit one instance is in charge of. This is what Task.fifo queue do in the CND system.

In the original version of the CND system, we choose to split the whole problem into N units, which is equal to the quantity of instances (another way of dividing is illustrate in the Further Extension chapter). Then every instances will poll the Task.fifo queue and gain the task unit number for their own working field.As the codes in Figure 12 ,we set the “DelaySeconds” to equal “0” for letting the queue get messages once we send, and the same is true for it in the “sendMessage” function.

```
1. sqs.create_queue(
2.         QueueName='Task.fifo',
3.         Attributes={
4.             'DelaySeconds': '0',
5.             'MessageRetentionPeriod': '86400',
6.             'FifoQueue': 'true'
7.         }
8.     )
```

FIGURE 12

Then we send one message for every unit **containing the unit number within the “messageBody”, the total quantity of instances and the difficulty we need compute within the “messageAttributes”.** What is worth mentioning here is that we have to set the “MessageDuplicationId” and the “MessageGroupId” (as shown in Figure 13). According to the AWS Developer guide of FIFO queues, if a message with a particular message deduplication ID is sent successfully, any messages sent with the same message deduplication ID are accepted successfully but aren't delivered during the 5-minute deduplication interval. And messages that belong to the same message group are always processed one by one, in a strict order relative to the message group, however, messages that belong to different message groups might be processed out of order. So we set the “MessageDuplicationId” and the “MessageGroupId” to equal the unit number. In this way, each message with a unit number will be sent only once when we split the whole task, and each unit will be polled in an order

within a same message group. If we set the “MessageDuplicationId” to a fix string, only the first message could be sent to the queue. We’ve also tried to set the “MessageGroupId” to a fix string, only the first instance can poll the task unit.

```
for n in range(1, N+1, 1):
    sqs.send_message(
        QueueUrl= task_url,
        DelaySeconds=0,
        MessageGroupId=str(n),
        MessageDeduplicationId=str(n),
        MessageAttributes={
                …
                …
          },
        MessageBody=(
            str(n)
        )
    )
```

FIGURE 13

2.3.2.Result.fifo and Log.fifo

The codes and attributes for creating them are the same as creating Task.fifo ones.

The Log.fifo queue is for recording the state of every instances. When each instance polls a message from Task.fifo queue, it will extracts required data and send a message containing unit number to Log.fifo to show that it starts to compute the task unit. Similarly, when it finishes the whole unit without gaining a golden nonce, it will send a message containing unit number to show that it finish the unit of task. If an instance failed to poll a task unit, which means all the units have been complete, it will send a message to show that there is no more unit after the unit it just finished. The reason for sending these messages is that we can easily know which stage goes wrong if one instance suddenly stops work. Moreover, we can get the total time spent through the SQS console checking the timestamp of first and the last message in the Log.fifo queue.

As for Result.fifo queue, it is only used for golden nonce. As soon as an instance finds one, it sends a message containing the golden nonce to Result.fifo. Meanwhile, our CND system constantly polls the Result.fifo queue, when it succeed to poll one, it will terminate all the running instances (The two queues’ function have been clearly elaborated by Figure 17).

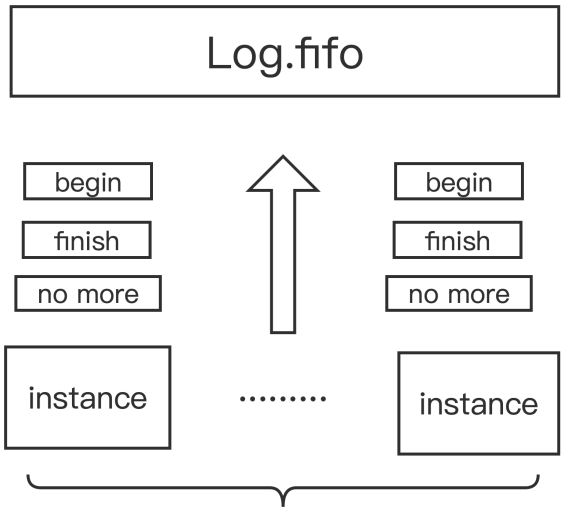
2.3.3.Judge outcome from queues

After we handing this whole problem over to the EC2 instances, we choose to poll the Result.fifo and Log.fifo queues within a “try…except” block, and the whole “try” and “except” block is within a “while” loop. In the “try” block, we poll messages from the Result.fifo queue, once we get one, it means that we have found the golden nonce (figure 14).

```
while state == 0:
    try:
        content = sqs.receive_message(
                ……
            )
    except:
        ……
```

FIGURE 14

After we failing to get one in the most of the time, in the except block, we use the SQS client function “get_queue_attributes” to get the total quantity of Log.fifo queue messages. When the number is 3 times of the total number of instances, which means every instance has send 3 messages to show “Begin”, “Finish”, “No more task unit remaining”, we can draw the conclusion that no any golden nonce under such a difficulty(as shown in figure 15). (When divided the whole problem into smaller task unit, like 1024 units, the number will be “2*unitNumber+instancesNumber”)



N instances send 3*N messages

FIGURE 15

2.4.Python codes running on instances

At beginning, we run the PoW on our local machine to test the performance and get an explicit understanding of the problem content. Here is the abstract code for computing with nonce in our local machine(Figure 16)

```
1. checkstr = “000000”
2. for nonce in range(0,2^32)
3.       result = SHA256(SHA256(block + nonce))
4.       If result[0:D] == checkstr
5.            golden nonce = nonce
6.            break
7.       print(golden nonce)
```

FIGURE 16

Through the user data, we transfer a python code to each instance, which commands every instance to poll the Task.fifo queue and then process the task unit, and meanwhile, to send the state and result to particular queues. The polling process should also be put in a “try…except” block to deal with the exceptions. There is a diagram (Figure 17) showing the structure and logic in the code:

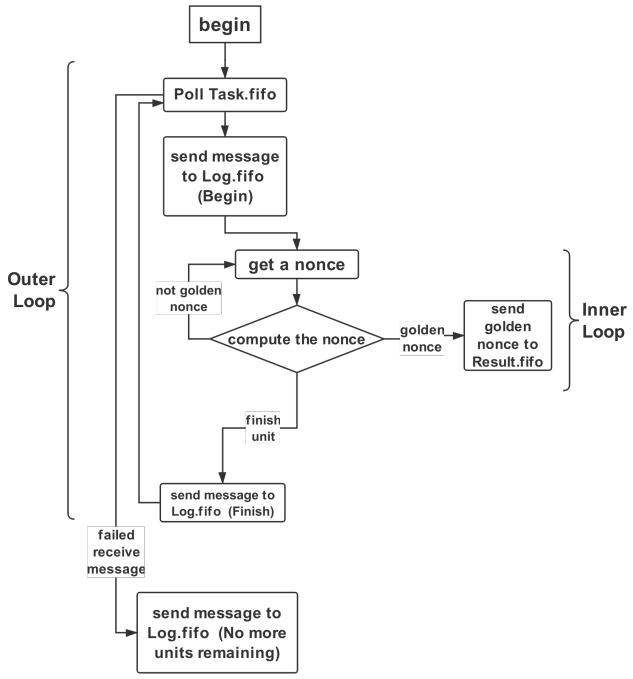


FIGURE 17

It is worth mentioning that when polling the Task.fifo queue to get a task unit, we should set the “MaxNumberOfMessages” to equal “1” in order to get only one message per operation. What’s more, to deal with the situation that many instances poll a same message, we should set the “VisibilityTimeout” to equal 43200, which is the maximum. This means when one instance succeed to get one message, this message is invisible to other instances (including the console polling). According to the SQS Developer Guide, if we know (or can reasonably estimate) how long it takes to process a message, extend the message's *visibility timeout* to the maximum time it takes to process and delete the message. It means we should set the visibility timeout to the maximum time that it takes your application to process and delete a message from the queue. So we set the attribute to 43100 ensuring each message taking effect only once (as shown in Figure 18).

```
1. sqs.receive_message(
2.          QueueUrl=task_url,
3.          AttributeNames={
4.                ……
5. },
6.          VisibilityTimeout=43100,
7.          WaitTimeSeconds=0
8.      )
```

FIGURE 18

Another thing to notice is that if we use the “range” function to fulfill a loop, we have to update the Python version of every instance to Python3. Because range in Python2 returns a list, if we use it to define a large field (eg 0~2^{32}), a memory exception will occur. However in Python3, it uses an iterator to complete this, which requires less memory. So remember to choose Python3 for range function or choose “xrange” function in Python2 for a same reason.

3. Performance Statistics

On our local machine, we have run the whole task for 13777s, and meanwhile for remotely commanding 1 instance, we got the performance of 12721s.

Through the CloudWatch service, our vCpu has a nearly 100% cpu utilization in average, so the PoW is enough to occupy a single core. **In order to test the difference of performance, we choose a difficulty that instances can’t find a golden nonce.**

The following table shows the timings from different quantity of instances. Due to there is a different time consuming between

instances initializing at every launch, the timings we get may have an exaggerated error. In some extreme situation, the first and the last instance may have a 2 minutes divergence of initializing. Even CND system continuously launches same quantity of instances for computation, there is a difference to some extent. All the timings we show in Table 2 are calculated by arithmetic mean of 5 samples, and rounded down to integral numbers. Then we calculate the standard deviation from the data we get in order to gain the confidence interval.

Theoretically, the timings we get meet standard normal distribution. So when we get the confidence level by the user, and calculate the standard deviation, we can get the confidence interval by the formula in Figure 19:

$$CI = \bar{x} \pm z^* \frac{\sigma}{\sqrt{n}}$$

CI : Confidence Interval

$\bar{x}$: sample mean

z^* : constant get from confidence level

σ : standard deviation

n : sample size

FIGURE 19

Among them, z^* can be get below in Table 1:

Confidence Level	Z^*
80%	1.28
90%	1.645(by convention)
95%	1.96
98%	2.33
99%	2.58

TABLE 1

According to the requirement, we have to yield a golden nonce **within T seconds** with L% confidence. So in the worst case, we assume that we have to compute all 2^{32} nonce to gain the golden nonce. However, the confidence interval is symmetrical, and the sample mean is in the middle. So within T seconds with L% confidence means that there is a confidence interval with 1-2(1-L) confidence level, and the larger margin is exactly smaller than the expected time.

Num.	Timing (s)	standard deviation	$z^* \frac{\sigma}{\sqrt{n}}$ (95%)	Larger Margin (95%)	$z^* \frac{\sigma}{\sqrt{n}}$ (99%)	Larger Margin (99%)
Local	13777					
1	12721	4.5	3.30	12724.30	4.68	12725.68
2	6678	5.7	4.19	6682.19	5.93	6683.93
3	5273	4.8	3.53	5276.53	4.99	5277.99
4	3569	5.9	4.33	3573.33	6.14	3575.14
5	2491	5.3	3.89	2494.89	5.51	2496.51
6	2251	8.4	6.17	2257.17	8.74	2259.74
7	2067	6.5	4.77	2071.77	6.76	2073.76
8	1849	6.4	4.70	1853.70	6.66	1855.66
9	1675	7.5	5.51	1680.51	7.80	1682.80
10	1547	8.9	6.54	1553.54	9.26	1556.26
11	1429	8.4	6.17	1435.17	8.74	1437.74
12	1347	7.9	5.80	1352.80	8.22	1355.22
13	1208	7.5	5.51	1213.51	7.80	1215.80
14	1079	8.9	6.54	1085.54	9.26	1088.26
15	929	11.3	8.30	937.30	11.75	940.75
16	876	13.2	9.69	885.69	13.73	889.73
17	849	16.4	12.04	861.04	17.06	866.06
18	817	17.5	12.85	829.85	18.20	835.20
19	799	14.4	10.58	809.58	14.98	813.98
20	769	14.8	10.87	779.87	15.39	784.39
21	745	15.9	11.68	756.68	16.54	761.54
22	697	17	12.48	709.48	17.68	714.68
23	637	19.6	14.39	651.39	20.39	657.39
24	620	17.5	12.85	632.85	18.20	638.20
25	615	14.7	10.80	625.80	15.29	630.29
26	588	18.5	13.59	601.59	19.24	607.24
27	544	19.4	14.25	558.25	20.18	564.18
28	537	21.5	15.79	552.79	22.36	559.36
29	524	20.4	14.98	538.98	21.22	545.22
30	506	21.7	15.94	521.94	22.57	528.57

TABLE 2

In this project, we only tried the input confidence level of 95% and 99%, which

actually means a confidence interval with 90% and 98% of confidence level. From what we have mentioned above, we can caculate the $z * \frac{\sigma}{\sqrt{n}}$ and then calculate the Larger Margin of the confidence interval.

Next, we choose to compare the expected time and the larger interval to get the suitable quantity of instances, and finally let the CND system run them.

4. Analysis and Further Extension

Through the analysis of the performance timings, we find that a lot of time has been waste during instances waiting for the slowest instance because instances spend different time on PoW and initializing. However, in some cases, a particular instance may be slower a lot on initializing than others. The reason may be some exceptions of the AWS service, or that in peak hours, instances may have a higher delay than the off-peak time, which causes that the particular instance may have to wait for system resources to initialize. Exceptions of network may also cause this, to be more specific, we have update and download some packages through network, so if there is a bad network condition, some instances may be delayed for long.

So we choose to use two ways to relief this, and to elevate the performance at the same time. The first is changing the way dividing the embarrassingly parallelizable problem, and the second way is implementing multiprocessing.

4.1.Another way of dividing the problem

In the original version of the CND system, we choose to give each instance 1 unit to divide the whole problem. In this way, the instances that have finished their work have to wait without anything to do. So the whole running time will be influenced a lot by the accidents we mentioned above. So we find another way to divide the problem.

Roughly, the whole problem has got 2^{32} nonce to be computed, if we divide these 2^{32} nonce into 2^{10} units, using Task.fifo queue to store these units' information as we have elaborated above, we will reduce the influence caused by such accidents. Because instances still have work to do after they completing one unit.

4.1.1.Technical challenges

Similar to the concept of integral, theoretically, the more units we divide the whole problem into, the less divergence between instances. However we have to consider the time spent on sending and receiving messages. And this requires us to get a lot of statistics choosing a units quantity. Because the number of nonce is multiple of 2, so we have tried to divide these nonce into 2^9, 2^{10}, 2^{11} units and then find that the "2^{10}" will provide a better performance.

Some statistics are given below (Table 3).All the timings we show are calculated by arithmetic mean of 5 samples, and rounded down to integral numbers.

Instances quantity	1 units per instance (s)	1024 units in total (s)
5	2491	2132
10	1547	1333
15	929	875
20	769	742

TABLE 3

4.2.Multiprocessing

Because multiple instances may have an accident that a particular instance may delay more time during initializing, so reduce the number of instances may reduce the influence cause by this. Then we try to implement multiprocessing for every instances, and at the same time, we need tochange the instances type from t2.micro to t2.medium to keep the performance of processor. Implementing multiprocessing is also for improving the performance and reducing the running time we spent. This time we only implement 2 process on dual-core processor of t2.medium.

4.2.1.Technical challenges

The most obvious challenge of implementing the multiprocessing is to know

the concept of multithreading and multiprocessing of Python language. Python has a Global Interpreter Lock (GIL) which is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. So we can't use multithreading taking full advantage of the processor to elevate the performance. However, we can rely on the multiprocessing to let multiple processes execute the Python bytecodes concurrently.

The other challenge is how to allocate task to a process, letting each process poll the task queue separately or letting processes only cooperate for the PoW after receiving a task. In order to manage them conveniently, we choose the latter one. For the specific implementation, we choose to encapsulate the computation part into a function, and then use "args" parameter to distinguish their own work (as shown in figure 20).

```
1. P1 = mp.Process(target=compute, args=
2.         (unitBegin, unitBegin + int(0.5 * unitLength), D, checkstr))
3. P2 = mp.Process(target=compute, args=
4.         (unitBegin + int(0.5 * unitLength), unitStop, D, checkstr))
5. P1.start()
6. P2.start()
7. P1.join()
8. P2.join()
```

FIGURE 20

Next we have checked the CloudWatch to see the CPU utilization, which shows that we have taken full advantage of these 2-vCpu instances (figure 21).

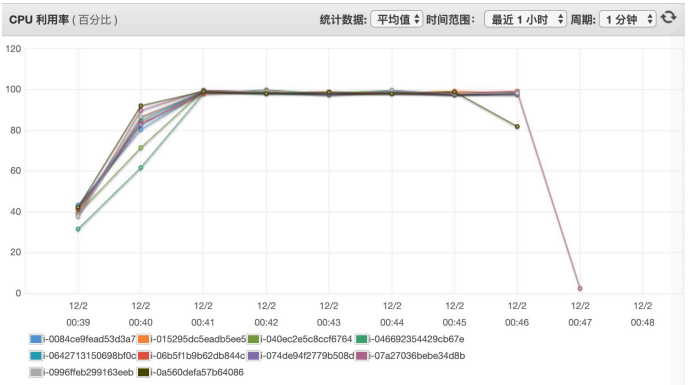


FIGURE 21

After using the t2.medium instance with 2 processes, we gained a better performance of below (Table 4). All the timings we show are calculated by arithmetic mean of 5 samples, and rounded down to integral numbers.

t2.medium (/instance)	timings(s)	T2.micro (/instance)	timing(s)
1	5673	2	6678
5	1394	10	1547
10	644	20	769
15	433	30	506

TABLE 4

Through our analysis, the extra performance we get may be from less time waiting for each other (same amount vCpu with less instances) and from a slight higher performance that t2.medium have. Moreover, t2.medium has an obviously higher memory which may cause a lot difference in performance.

5. Conclusion

We have complete the CND system to scale out the whole problem, and then we do a lot performance statistics for indirect specification of instances' quantity from the confidence interval we calculate. What's more, the CND system can terminate all the instances when system timeout or when we succeed or fail to find a golden nonce. Finally, for a better performance and reducing embarrassing errors caused by EC2, we try a different way of dividing the whole problem and implement multiprocessing. In a word, we succeed to finish the coursework.

This coursework gives us a wide range of freedom to fulfill the horizontally scaling. And through exploring the ways to implement this, we have comprehended the multiple service AWS provide and their features, which is a substantial journey of studying Cloud Computing. I'm looking forward a higher stage of my application of Cloud Computing in the future.