Horizontal Scaling for an Embarrassingly Parallel Task: Blockchain Proof-of-Work in the Cloud

COMSM0010 - Cloud Computing

https://github.com/fznsakib/cloud-nonce-discoverer

1 Introduction

Since the introduction of Bitcoin in 2008 [5], blockchain has seen a meteoric rise in use across industries and applications. Due to the open and decentralised nature of blockchain, it is essential that it has a protocol in place in order for the network to reach consensus. In answer to this, one of the most well-known approach is the Proof-of-Work (PoW) system. This involves a computationally expensive task of finding a value referred to as the 'golden nonce' in order for the blockchain to progress.

This report aims to document the implementation, performance and challenges of a system utilising cloud infrastructure that can perform PoW to find the golden nonce. The system utilises cloud infrastructure in order to gain superior performance compared to one of local design. As such, the system will be referred to as a Cloud Nonce Discovery (CND) system throughout this report.

2 Proof of Work

For every block holding unconfirmed transactions, a random 32-bit number is appended to it, which is known as the 'nonce'. This block of data is then hashed using SHA-256 twice. The objective of PoW is to find the 'golden nonce'. This refers to the value of the nonce for which the hashed block has a number n or more consecutive leading zero bits. The number of zero bits required is known as the difficulty, D. Once the golden nonce is found, this is culminates the end of the PoW process, verifying and adding the block to the blockchain.

The difficulty is adjusted every 2016 blocks [1] and has been rising at a steady rate to compensate for increases in computational power. For the purposes of this work, the scope of difficulty will remain limited compared to what is seen in practice.

2.1 Local Implementation

A locally running nonce discovery script was first implemented before utilising any cloud utilities. It involves a simple `while` loop which iterates over all 2^{32} possible values. At every iteration, the new nonce is appended to an arbitrary data value (for this project, it is the string *'COMSM0010cloud'*) before hashing it twice with the use of Python's in-built library `hashlib` which includes the function `sha256()`. This is followed by the procedure as stated in 2.

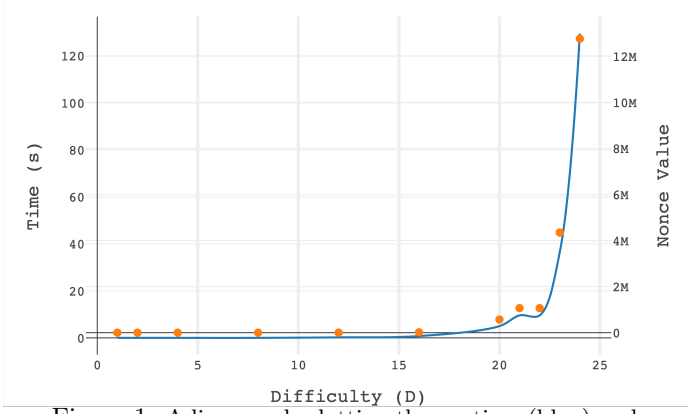


Figure 1: A line graph plotting the runtime (blue) and nonce value (orange) against difficulty, D for the local version of a nonce discoverer.

The requirement for higher numbers of leading zeroes would naturally lead to the nonce being represented by higher integers. As expected, the time for discovering a nonce increased along with the difficulty, as shown in Figure 1.

The results also show the exponential increase in time taken. A strong direct relationship can also be seen between the time taken and the value of the golden nonce, denoted by the orange points. For every increment in difficulty D, there is a doubling of the search space required to find the nonce.

From this we can say that the relationship between the difficulty and the time taken to find the

nonce is 2^D, which defines the search space. Within a difficulty of 28, the computer running the program was struggling to balance its resources and discover a nonce within an adequate time.

2.2 An Embarrassingly Parallel Task

Since the nonce is a 32-bit integer, there are up to 4,294,967,296 different values to consider. Furthermore, an increment or decrement by one in the nonce is enough to completely change the resulting hash. This makes the task difficult to optimise as there is not much that can be done to narrow down the search space. Thus, the most common approach is to carry out a brute search force from all values from 0 to 2^{32} until a suitable value is found for the nonce which matches the number of leading zeroes of difficulty.

A problem such as this can be trivially split up into n sub-problems which can then be delegated to k workers by a master entity. Once any of the workers have found a solution, it is reported back to the master before the problem is considered solved. This makes the PoW problem of the 'embarrassingly parallel' sort.

This gives us ample reasoning to go ahead and take advantage of the resources available in cloud services which can allow the parallelisation of such a problem.

3 AWS

With the demand of cloud services having skyrocketed in the last decade with around $125 billion of revenue [3], there is plenty of choice in the market for users to choose from.

Amazon Web Services (AWS) is the chosen provider of cloud services for this project. As the leading company within the domain [3], AWS offers a wealth of services with flexibility and documentation, which allows users to build systems at quickly and at scale. Due to its dominance in the market and its comparatively large resource availability, its pricing structure is very competitive, with a free tier that allows for a sufficient amount of computation for this project.

3.1 Technologies Used

These are the different AWS technologies used for the purposes of this project along with a brief description of their functionality:

- **EC2 (Elastic Compute)** : One of the main services of AWS, EC2 represents compute power available as virtual machines. Each running virtual machine is known as an instance and can be scaled and configured accordingly.
- **S3** : A cloud storage system in which any type of data (and its metadata) can be stored in globally uniquely named buckets.
- **SQS (Simple Queue Service)** : A message queuing service allowing consistent and reliant communication between system components.
- **Lambda** : A server-less computing service triggered by events with the possibility of propagating onward events to other AWS services.
- **SSM (Systems Manager)** : Allows the controlling and overseeing of the rest of the AWS infrastructure in use, such as EC2 instances.
- **CloudWatch** : A robust monitoring service which enables logging, querying, analysis and more for AWS services being used.

4 Non-Parallel CND

The first task in creating a CND was to translate the local nonce discovering system to one that operates in the cloud, with the use of only one virtual machine. The main objective of this system was to create the backbone of the system on which it will be iterated upon.

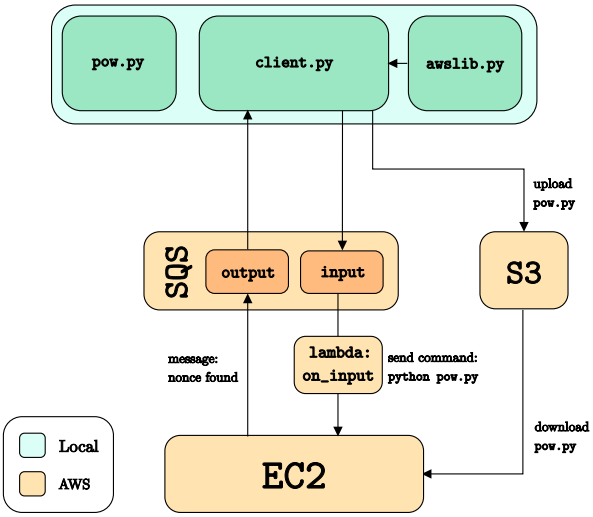
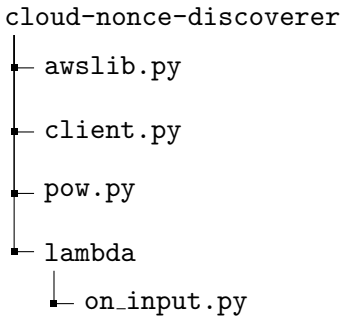


Figure 2: A diagram showing the design of the non-parallel CND system.

4.1 Implementation

Figure 2 illustrates the design used in order to maintain communication between the local system and the cloud infrastructure from the start of the program until its completion.



4.1.1 awslib.py

For this system to work, the client needs to be able to communicate with the AWS infrastructure. In order to achieve this, the AWS SDK `boto3` is used. It provides an interface for Python to the majority of services offered by AWS. For example, it allows SQS queues to be represented as a `Queue` object, which implements fields and methods applicable to SQS that can be accessed with ease. By installing the AWS command-line interface `awscli` and providing it with AWS credentials, `boto3` can seamlessly access the account's resources throughout a script.

All functionality involving `boto3` is encapsulated in `awslib.py`. These set of functions are used by `client.py`, as shown on Figure 2, ensuring abstraction of the AWS environment. These functions were written to be as generalised as possible to maintain compatibility and usability with any given program written in Python.

4.1.2 client.py

The purpose of `client.py` is to communicate with AWS and orchestrate the entire process of finding the golden nonce. It takes in an input argument D from the user, signifying the difficulty, as explained in Section 2. The design will be broken down into the different components of AWS to show how they fit together to solve this task:

- **S3 :** At the beginning, `pow.py` is uploaded to an S3 bucket. This ensures that the virtual machine has access to it, which is the script that it must run to perform the PoW task.
- **EC2 :** The client must then create an EC2 instance for which the PoW task will be performed on. Every instance is initialised based on an Amazon Machine Image (AMI), which holds the OS and software packages. It is important to note that the script that needs to be run on the instance, `pow.py` has a couple of dependencies that are not pre-installed in the default Amazon Linux AMI. The dependencies required are Python 3.6+ and `boto3`. In order to solve this problem, a created instance was accessed using `SSH` to install these packages. This particular instance can then be exported as a custom AMI, which can be used to initialise new instances including the required dependencies in the future.
- **SQS :** Two SQS queues, `input` and `output`, are created to allow the flow of information between the local system and the EC2 instance running `pow.py`. Due to the nature of this design involving one EC2 instance, a simple queue will suffice. A message is sent from the client to the `input` queue to signify the start of the process, with the message containing the user provided `difficulty`. This triggers a Lambda function, which is discussed later, leading to `pow.py` being run on an EC2 instance. At this point `client.py` will be waiting for a message to arrive at the `output` queue which will hold the golden nonce and will trigger the termination of the script and all entities within AWS. The sender of the message will be the `pow.py` script running on the EC2 instance.
- **Lambda :** As mentioned in Section 3.1, Lambda can offer event-based functionality which can make it a powerful tool in any system. As shown in Figure 2, this system uses `on_input.py` as a Lambda function which is triggered whenever a message arrives at the `input` queue. `on_input.py` parses the message for relevant information, such as the `instance_id` to send commands to the relevant instance and the `difficulty` set for the task. This is followed by the sending of two commands to the instance of `instance_id` to run:
  - Download the script `pow.py` from S3
  - Run `python3 pow.py` with the provided `difficulty`
- **SSM :** Used for reliably running `shell` commands on specified instances from within the Lambda function `on_input.py` using `boto3.ssm.send_command()`. The parameter `CloudWatchOutputConfig` allowed the SSM to track and output logs within a specified log group.

This was helpful for debugging the `pow.py` script and ensuring that the Lambda function worked as required.

4.1.3 Identity and Access Management

By default, it is not possible to have fully functioning communication between user and AWS and inter-communication between AWS entities. This is due to an access and permission system in place to avoid unauthenticated access across AWS. Identity and Access Management (IAM) allows users to define access rules for all possible parties involved.

Firstly, in order to use AWS from outside the console (e.g. `boto3`) a *'User'* has to be created with the relevant permissions policy. In this case, a User was created with administrator access to allow maximum access across AWS. This produced an access key used for authenticating the machine running `client.py` so that `boto3` has permission to act as an interface to AWS for the respective account.

Secondly, *'Roles'* have to be assigned to the different entities of AWS to allow them to function as normal. For example, without the assignment of the *'AWSLambdaSQSQueueExecutionRole'*, the `on_input.py` Lambda function would fail to read messages from the SQS `input` queue due to insufficient access permissions. As such, the relevant policies were applied to all AWS entities used within the system so that can they behave as expected.

4.2 Performance

With only one virtual machine being utilised in this version of the CND, the performance of this is expected to be similar to the local nonce discoverer.

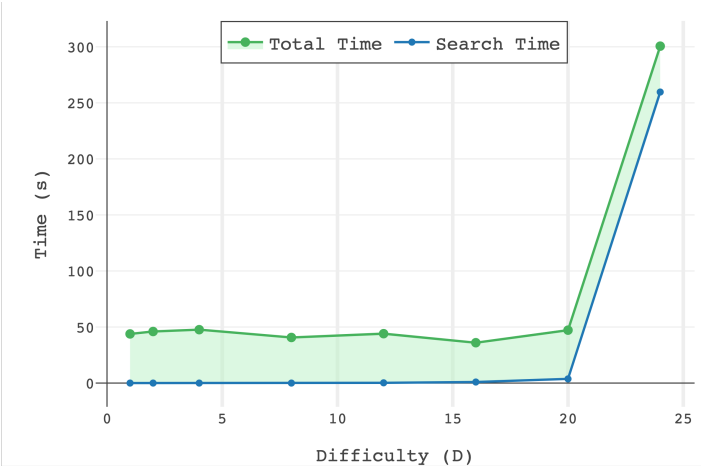


Figure 3: A line graph showing the performance of the non-parallel CND, plotting search time and the total time (including cloud overhead) against difficulty.

However, this mainly applies to the performance of sequential iteration searching for the nonce. If the entire process, from running the command on `client.py` to getting the value for the nonce, is considered, this version of the CND takes longer than a fully local nonce discoverer. It would be of interest to consider the extra time taken for the whole system to operate on the cloud. The term *'cloud overhead'* will be used to refer to this.

Accordingly, for gathering data, the *Search Time* is timed from the beginning of the search in `pow.py` until the discovery of the golden nonce. The value of this duration would then be passed back to the client. This could have been timed from `client.py`, but by doing it on the EC2 instance, there is no other inclusion of time spent outside of the machine. Meanwhile, the *total time* is timed from the running of `client.py` to reporting of the nonce on the local machine.

Furthermore, the difficulties for which runtimes were chosen to be reported include 1, 2 and the range [4...24] with increments of 4. No difficulty further 24 is used as the performance of one machine is not enough to gain a viable runtime. The mean of five samples is taken for each result.

The results reported in Figure 3 give us a better understanding of how the cloud infrastructure in this system affects the time for nonce discovery. As reported in Section 2.1, the search time for the nonce shows the same exponential behaviour as `difficulty` increases. The graph plotting the total time of the system is also consistently higher than the search time, which is also expected. However, it can be seen by the highlighting of the gap between the two line plots, that the difference is fairly constant.

The cloud overhead for each result, calculated as $total\_time - search\_time$, consistently falls within the range of 40 to 50 seconds. The two possible sources of this overhead include:

- the accumulation of time from interacting with AWS (e.g. uploading file to S3, sending messages to queues)
- the time taken for the EC2 instance to be created and ready to run

On further investigation, interactions with AWS through the use `boto3` waiting for responses would only amount to runtime on the order of milliseconds. The resulting conclusion was that it was the

latter of the two that accounted for the majority of the cloud overhead produced. Having to create a new instance, load up an image, and do consequent status checks leads to a significant amount of time spent before the CND can begin its search. Since the instances are unable to accept commands from the Lambda function `on_input.py` before these checks are done, `client.py` waits for the instances to return the required status of *running* before it proceeds.

With this, it can be said that for any *difficulty* for which the local nonce discovery system requires $<\sim 50$ seconds, the cloud overhead is large enough to make the use of the CND unjustified. However, when exploring the use of multiple instances on higher difficulties later in the report, it can be seen that the reductions in runtime is large enough to make up for the time lost initialising the EC2 instances.

Understandably, this is not the case for all cloud implementations. A containerised solution with the use of AWS Elastic Container Service has the potential to reduce the time for deployment of tasks across instances.

5 Parallel CND

This now leads to the implementation of the CND in parallel, which has the ability to delegate the PoW task to a number of N EC2 instances, before waiting for the client-side retrieval of the golden nonce. Figure 4 illustrates the design and logic sequence of the final CND. This section will discuss its implementation followed by an evaluation of the performance statistics gathered from this system.

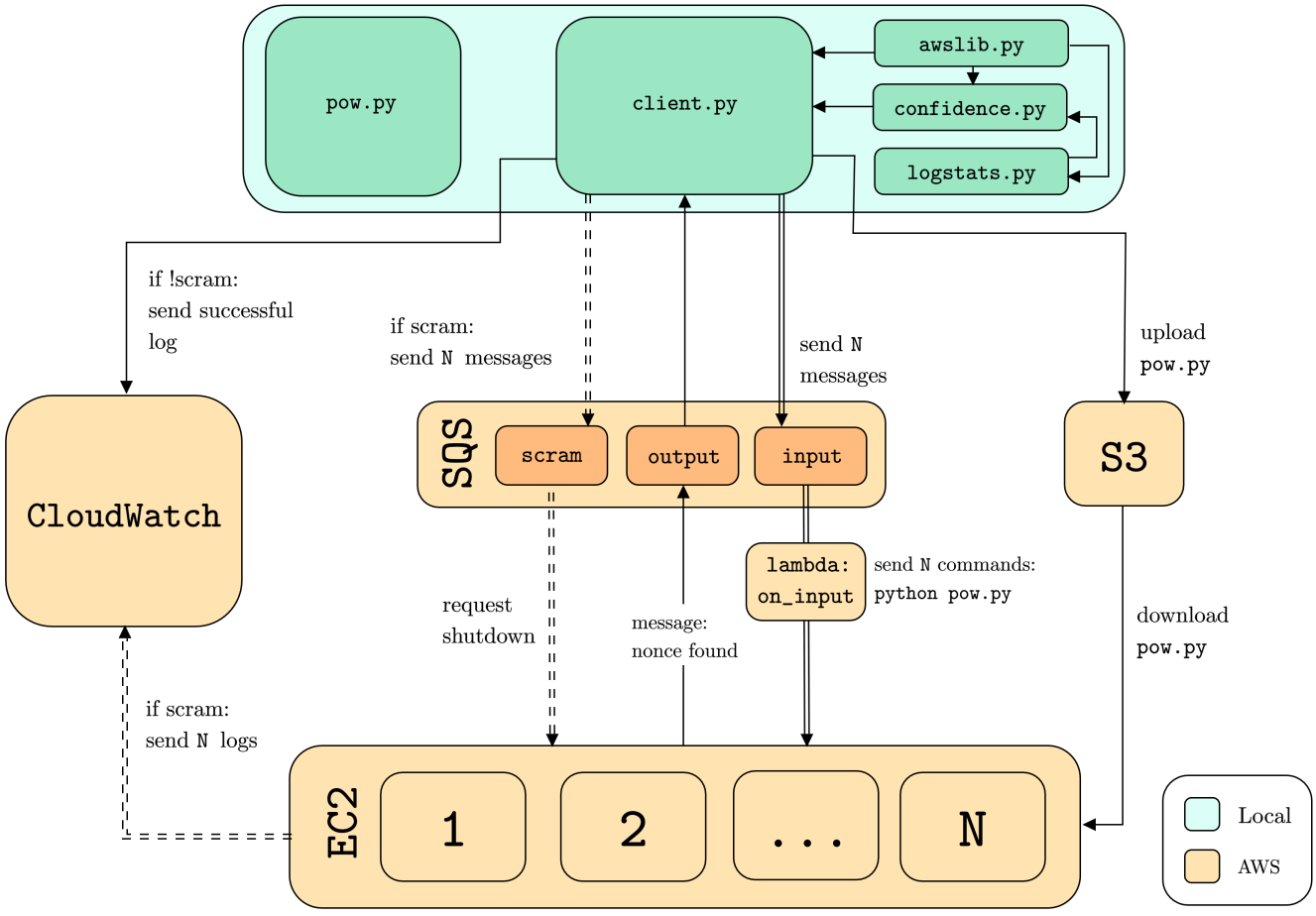


Figure 4: A diagram outlining the design and architecture for the final parallelised version of the CND. The single lined arrows represent the passing of a singular message/command; double lined arrows represent the passing of N messages; double dotted lines represent the passing of N messages given the condition of a scram initiated by the user or a timeout.

5.1 Direct Specification

With the CND now being able to use multiple EC2 instances, the user is given the option to choose N instances to carry out the search. This is a form of direct specification. `client.py` will accordingly start N EC2 instances for the proceeding PoW task. This is referred to as *horizontal scaling*, which aims to scale a system and its performance by adding more machines. Due to a limit imposed on the number of simultaneous instance starts by AWS, the maximum number of EC2 instances used in this project is 14.

To delegate the task between N instances, the search space of $[0, 2^{32}]$ must be divided between them. In this case, the search space is split equally. The number of values each instance will iterate over will be calculated as $\texttt{search\_split} = 2^{32}/N$.

`client.py` iterates in the range $[0, N-1]$ inclusive. Thus, for each instance, where i is the iterator:

$$\texttt{search\_start} = \texttt{search\_split} * i$$
$$\texttt{search\_end} = \texttt{search\_split} * (i+1)$$

The EC2 instances will have to know what their search spaces are, so `search_start` and `search_end` are sent within the body of the message to the `input` queue, where it will be passed as an argument to `pow.py` by the Lambda function `on_input.py`.

The Lambda function comes to good use in this scenario as it can run concurrently. This means for N messages being passed to the `input` queue, it can process all of them simultaneously to send out the commands to the EC2 instances to run `pow.py`. At a small scale like this, performance gain will be minimal compared to sending the command to each instance form the client. However, it can help achieve a tighter range in start time for search across the N instances.

5.2 CloudWatch Logs

Before this version of the CND, there was no logging mechanism in place. A simple `JSON` object which was received as part of the message from the successful EC2 instance would be printed to `stdout`.

The CloudWatch environment as part of AWS provides a variety of powerful utilities, allowing logs to be stored and analysed for various purposes. With a task such as this involving multiple variables to measure, it proves to be a good solution for aggregating logs. CloudWatch can interpret `JSON` objects which means they can be queried by keys and values. This will prove to be useful later on in the project, enabling analysis of logs for implementing indirect specification.

When uploading logs, the name of the *log group* and the *log stream* they will be uploaded to has to be specified. A log group, as the name suggests, is a group holding up to any number of log streams. Meanwhile, a log stream is a sequence of log events, where each log event, would for example hold a `JSON` object.

Initially, the intention was to create a log group for each combination of N instances and `difficulty`, so that they are separate and easy to interpret. Alongside this, each log stream would hold all the logs for any given run, where the instance finding the golden nonce uploads its log noting it as successful and the other instances uploading unsuccessful logs with the value of the last nonce they evaluated.

However, the intentions for CloudWatch log streams are that they should each be represented by one source. This is demonstrated by the requirement of the `uploadSequenceToken` which is updated and returned whenever a log is uploaded to a stream. This token then has to be retained so it can be used when uploading a consequent log. This makes it difficult to log to one stream with ease using multiple sources, as the token would have to be passed around. Another possible workaround would be to use the `describe_log_streams()` function just before uploading a log which returns the required `uploadSequenceToken` for each log stream. However, in practice with multiple N instances, occasionally an instance may be unable to upload the log. This is because the `uploadSequenceToken` held by an instance may have been used by another instance and rendered invalid between the time it was obtained from `describe_log_streams()` and to when it was attempted to be used to upload a log.

This led to a design decision where each instance in a given run would have its own log stream. Although it was not the desired outcome of logging structure, it was later realised that the separation of log streams between instances does not resemble an issue due to the way querying works in CloudWatch.

Furthermore, it was later decided that only the successful instance would return a log, instead of

including the unsuccessful instances. This means a singular log stream for each run will suffice. This is due to the fact that an n^{th} instance should not be considered as 'unsuccessful' if another instance in N within that particular run has in fact found the golden nonce. This means that whatever configuration of N and `difficulty` was chosen has successfully returned a golden nonce. This should not then be used as an opportunity to misreport the actual result with logs labelled unsuccessful.

A successful log contains useful information that resembles the event of completion of the PoW task, such as the `instanceId`, `goldenNonce`, `goldenHash`, `searchStart`, `searchTime`, etc. All of the values mentioned above are retrieved for the instance within the message it sends to the `output` queue. However, some values such as the `totalTime` or `cloudOverhead` are calculated after the message is retrieved on `client.py`. Because of this, instead of logging from the EC2 instance, the client appends the rest of the required information to log to the message received from the EC2 instance before being uploaded to CloudWatch.

5.3 Scram

Although this system cleanly shuts down all EC2 instances at the completion of the PoW task, it is necessary for the user to have the option to prematurely exit, initiating a scram. A scram would then proceed to shut down all running computation and resources. This is essential functionality to have for a system using cloud infrastructure, as it facilitates controlled consumption of resources that may otherwise cause unwanted financial debt.

The scram process is executed by the custom made function `scram()` found in `awslib.py`. This is held by a function `terminate()` which waits for an interrupt to be called. The scram utilises the `SSM` to cancel any running commands on the instances, before terminating those instances. By using the `signal` library in Python 3, it is possible to execute logic triggered by a specified event. This enables the system to call `terminate()` as such:

- **Ctrl + C :** This is a user-specified scram which is triggered by the interrupt signal `SIGINT`, represented by the keypress of the `Ctrl` and `C` button together. This allows the user to break from the program at any given moment while executing a scram on exit.
- **Timeout :** `client.py` takes an argument `t` which allows the user to set a time in seconds before which a scram is initiated. For this functionality, the `SIGALRM` interrupt is used, which initiates a callback to `terminate()` when `t` seconds has been reached. The timer starts after the Lambda invoking messages have been sent to the `input` queue to initiate the PoW task. This is so that the constant time to initialise the instances is not taken into account.

5.3.1 Logging on Scram

Another feature that would be useful for a user is to be able to see the status of an instance at the time of a scram. This would involve the instances logging to CloudWatch right before termination. Similar to how the `signal` library was used to execute logic at the end of a Python script, it would be assumed that this could be utilised in the case of the `pow.py` script running on the EC2 instances.

However, this attempt failed as the `SSM's cancel_command()` function seem to be forcefully killing the Python process. The would cause the script to come to an abrupt halt, bypassing the callback triggered by the `SIGTERM` interrupt.

In order to deal with logging on scram in a more graceful manner, a third SQS queue `scram` is introduced, as shown in Figure 4. When a scram is initiated, N messages are sent for each of the N instances to receive, notifying them to upload their log and shut down. Although the system has utilised queues in a similar manner before, the challenge is to interrupt the process of the script `pow.py` for the instance to act upon the message.

Essentially, `pow.py` has to continuously poll the `scram` queue to receive a message in the case of a scram. Trying to do this within the main iterating loop searching for the nonce is not viable. `boto3`'s `receive_messages()` function halts this search every time for a specified `waitTime`. Even with this parameter set to as low as possible, it noticeably hinders the time taken to find the golden nonce due to the blocking function.

Instead, the approach taken was to use Python's concurrent capabilities using the `threading` library. To achieve the wanted behaviour, one thread executes `findNonce()` which holds the logic for the search, while the second thread executes `waitForExternalNonceDiscovery()` which listens

to the `scram` queue. A `while` loop then blocks progression in `main()` until the completion of either thread. The latter thread accesses the last processed `nonce` along with other important variables to create a JSON message which is then logged to CloudWatch. This allows the user to inspect where in the search space the instance was located at the time of a scram. Although these variables are declared in the other thread `findNonce()`, any concurrency issues (e.g. race-conditions) are avoided as these variable accesses are specifically read-only.

`client.py` takes in an argument `-l` to enable logging on scram.

6 Performance

This section aims to report the performance statistics obtained for the parallelised CND. As in Section 4.2, each value is represented by a mean of five samples for the given parameters `N` number of instances and `difficulty`.

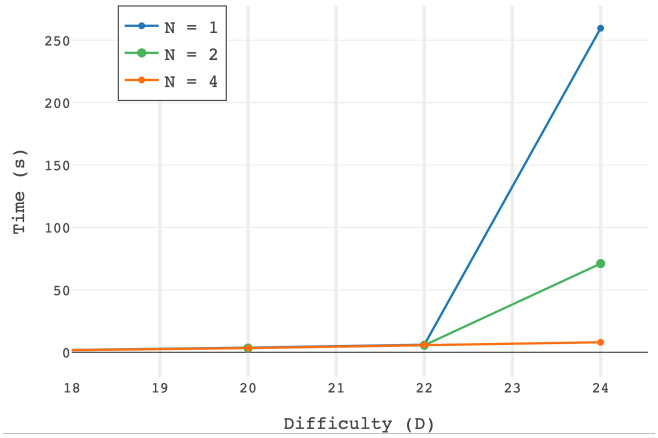


Figure 5: A bar graph showing the performance of the parallel CND for 1, 2 and 4 instances.

It has been observed in Section 2.1 that `difficulty` of up to 20 is achievable within a relatively short amount of time. As such, the increase in performance is near to negligible, as the times being compared are already of such small magnitude. Furthermore, a lower `difficulty` coincides with a lower value nonce. In this case, it is highly expected that the golden nonce will be discovered by the instance containing the first split of the search space, which contains the lowest numbers. This means the number of instances being used does not matter much, leading to results that are very similar for lower difficulties.

Nevertheless, it is known that beyond the `difficulty` of 20, a single instance starts to show signs of incapability. This is recalled in Figure 5 shown above. Specifically at `difficulty` = 24, the runtime for one instance shoots up to 260s.

With the implementation of the parallel CND, the advantages of horizontal scaling can now be seen. Figure 5 is evidence showing the fact that even a doubling of the number of instances to 2, can produce a substantial improvement in performance, where a non-parallel system would struggle. With the introduction of 4 instances, the performance increases to reduce the time taken to find a golden nonce to only 8s. This signifies a 97% improvement compared to the result gained from the use of a single instance.

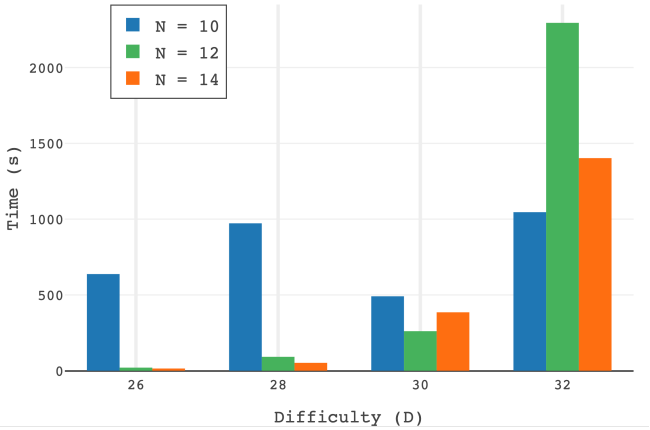


Figure 6: A line graph showing the performance of the parallel CND for higher levels of difficulty for 10, 12 and 14 instances.

Moving on, the limits of the CND are tested by increasing the `difficulty` even further. The results of `difficulty` of the range $[26, 32]$ in increments of two and inclusive are plotted against time in Figure 6.

The first pattern that can be seen on this graph is that the higher difficulties are causing a longer time for discovering the golden nonce. This is expected, as explained in Section 2.1. The exponential relationship of `difficulty` to time means a few increments in difficulty can easily render n number of instances useless for solving the task. To improve the runtimes of up to 30 minutes seen for `difficulty` = 32 and above, the CND would have to be horizontally scaled even further.

It can also be seen that, mostly, the performance for each number of instances is as expected, where the higher the number of instances, the lower the time taken to discover the golden nonce. Interest-

ingly, this is clearly not the case for `difficulty` = 32. It can be seen that for $N = 10$, the golden nonce is found quicker than both $N = 12$ and $N = 14$.

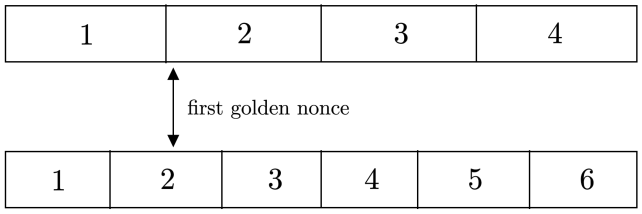


Figure 7: A diagram showing an example of the possible unwanted effect on the search space, caused by an increase of instances, from $N = 4$ to $N = 6$.

This kind of behaviour of higher instances performing less favourably than lower number of instances for the same `difficulty` has been observed elsewhere. One of the reasons to this can be that with a change in number of instances, possibly a number higher, it ends up creating a search split that is less optimal than the previous one. This is illustrated in Figure 7, showing that the golden nonce can be found close to the start of a search space for a certain instance for one difficulty. But with a change in the number of instances, this closeness of a search boundary to a golden nonce is lost, requiring a longer time to find one.

7 Indirect Specification

The CND up until this point requires direct specification where it takes a number N up to 14 as an input parameter which is used to determine the number of instances to start and delegate the task between. The next objective is to allow for the indirect specification of the number of instances used for discovering a golden nonce. In this implementation, the system will take in a a desired runtime t along with a confidence value, c. The confidence value resembles the percentage of confidence that the golden nonce will be discovered within t seconds. Using these two parameters, an appropriate number of EC2 instances will be spawned in order to meet this condition.

Given a confidence value c, for example 95%, and a sample set of data, a range of values can be determined for which there is 95% certainty of the population mean residing within it. This makes use of the fact that for large samples, the distribution of a sample tends approximately towards a normal distribution. This is also known as the Central Limit Theorem [2]. Using this, a confidence interval for a sample set can be calculated:

$$\left(\bar{x} - Z_{1-\frac{\alpha}{2}} \cdot \frac{s}{\sqrt{n}}, \bar{x} + Z_{1-\frac{\alpha}{2}} \cdot \frac{s}{\sqrt{n}}\right)$$

where $\bar{x}$ is the sample mean, α is the significance level calculated as $1 - c$, Z_q is the $q-$quantile of a normal distribution. Meanwhile, the s is the standard deviation of the sample set and n is the number of samples in the set. The standard deviation is divided by the square root of the number of samples to give us the standard error.

The term $Z_{1-\frac{\alpha}{2}}$ gives a Z-score symmetrical around the sample mean $\bar{x}$. This is done by ignoring half the significance level. For a confidence level c of 0.95, this would be 0.025, at each tail end of the distribution. This means the range contains 95% chance of the population mean around the sample mean, with 2.5% left out on either side of the range.

Since the objective is to limit the runtime, it would be better have the confidence interval starting at the absolute left end of the distribution to the Z_q on the right side of the distribution where $q = c$. This way, the uncertainty is isolated only to the right end of the distribution, so past the upper limit, in comparison to both sides originally. The Z value for the absolute left end of the distribution is expressed as 3. This multiplied by the margin error will account for $99.7\% \approx 100\%$ of the distribution as per the 3 standard deviation rule [4]. Rewriting the earlier formula we get:

$$\left(\bar{x} - 3 \cdot \frac{s}{\sqrt{n}}, \bar{x} + Z_c \cdot \frac{s}{\sqrt{n}}\right)$$

With this, as long as an upper limit for a confidence interval is less than the stated runtime t, it can be assumed with c confidence that the runtime will be below t.

For this method to work as intended, samples of $n > 30$ is required [2]. As such, appropriate data has to be gathered in order to obtain the statistics required to reliably calculate the required values. However, with a third variable `difficulty` in play and due to time constraints, it would not be possible to acquire a large enough sample set for every possible combination of `difficulty` and N number of instances. Therefore, 30 samples will be gathered only for each combination of `difficulty` = 24 across 1 to 12 instances. This `difficulty` will allow plentiful data collection within a viable time-frame.

It also produces a good variation of results across the different number of instances used.

Each sample set will be represented by the number of instances used for the results within it. With the appropriate data collected, the confidence ranges for each sample can now be calculated. As mentioned above, certain statistical values such as the sample mean, standard deviation and number of samples are required to calculate the range.

One of the main utilities of CloudWatch is the ability to query logs produced over a time-frame as required. The language is simple yet powerful, with the inclusion of a number of keywords allowing for a wide range of operations. As mentioned in Section 5.2, CloudWatch has the ability to interpret `JSON`, which means the logs can be queried by their keys and values. Using the query:

```
fields searchTime
| filter success = 1 and
  difficulty = 24 and
  noOfInstances = 10
| stats count(searchTime) as count,
  avg(searchTime) as mean,
  stddev(searchTime) as sd
```

CloudWatch is able to return all values required to calculate the range (sample mean, standard deviation and count of sample) for the sample where `difficulty` $= 24$ and $N = 10$, with the use of just one command. There is much more functionality including mathematical operators, sorting and parsing that could allow a user to get the most out of their logs.

This query is made by the function `getLogStats()` in `logstats.py`. The function is called by `confidence.py` as shown in 4. `confidence.py` is responsible for using these statistical values to calculate the number of instances to use to find the golden nonce given the runtime t and confidence value c.

`client.py` calls `getNoOfInstancesByRuntime()` if the argument `-c` is specified when running it. The client proceeds to ask for a confidence value and a runtime as input passed into the function mentioned above. The confidence range for each sample is accordingly calculated, before an iterating loop goes through the upper limits. The confidence range of the sample with the upper limit closest to the runtime t while being less than it is the one that is picked. This means the number of instances the sample is based on, is the number of instances N which will be used to find the golden nonce with t.

Furthermore, the user can specify to minimise the number of instances used (e.g. to save running costs). This is useful as there might be cases that within the given t, there are multiple numbers of instances that will guarantee with c confidence the golden nonce. In this case, the lowest number of instances out of the samples is chosen.

Requested runtime (s)	2.0	12.0	24.0	60.0
Instances	12	8	7	2
Actual runtime (s)	0.014	8.95	10.2	56.05

Figure 8: A table of results showing indirect specification of number of instances using the runtime requested for difficulty $= 24$ and confidence value $c = 0.95$.

8 Conclusion

This report has demonstrated the great value of cloud computing by starting with a local implementation of a blockchain PoW system before improving on it with the use of AWS infrastructure. The final version of the CND made significant performance gains using horizontal scaling, allowing for the successful execution of previously intractable tasks.

References

[1] Curran, B. Bitcoin Difficulty Target Adjusted. `https://blockonomi.com/bitcoin-difficulty-target-adjustment/`.

[2] Lisa Sullivan. {Confidence Intervals. `http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Confidence_Intervals/BS704_Confidence_Intervals_print.html`.

[3] McAfee. Custom Applications and IaaS Trends, 2017.

[4] Michael Galarnyk. Explaining the 68-95-99.7 rule for a Normal Distribution. `https://towardsdatascience.com/`.

[5] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.