Cloud Nonce Discovery using Amazon Web Services

Benjamin Lee
University of Bristol
COMSM0010: Cloud Computing

I. Introduction

In the Blockchain ledger, Proof-of-Work is an algorithm used to verify state and add new blocks to the chain [5]. An important part of this algorithm involves finding what is known as a golden nonce. A golden nonce is a 32-bit number which, when processed through a number of cryptographic functions, results in a hash value lower than a given target value. In the case of Blockchain, this golden nonce is appended as a 32-bit binary number to the hash of the most recent block in the chain. The SHA-256^2 algorithm is then called on this new piece of binary data and compared to the target value. The target value for Blockchain is defined as 256^{256-d} where d is an integer value representing the 'difficulty' of the calculation. We will refer to this task as Cloud Nonce Discovery (CND) from this point on. This task is 'embarrassingly parallelisable', given that the search space never changes, and can thus be divided into segments and passed to distributed workers. This report details the usage of the free tier of Amazon Web Services (AWS) to implement CND.

II. Specification

This implementation of this CND system meets the following specification:

- Possible to run on a local machine using the following parameters:
  - Difficulty
  - Optional: input string
  - Optional: start of search index
  - Optional: end of search index
- Possible to run in the cloud on a number of remote workers given the following parameters:
  - Number of workers
  - Difficulty
  - Name of AMI to launch
  - EC2 key pair
  - Optional: time limit
  - Optional: input string
  - Optional: confidence of completion within time limit
  - Optional: cost limit (in dollars)
- Setup of required AMI is automated
- Includes an emergency stop for cloud implementation with Ctrl+C
  - shuts down all remote instances and deletes open queues
  - outputs information to a log file responsible for early shutdowns
- Performance analysis via logging

III. Language, Libraries and APIs

Python

Python was chosen for this task given its simplicity to write and many libraries for mathematical and cryptographic functions. There also exists a good Python API for managing Amazon Web Services (AWS) called Boto which has a large amount of documentation and support available on the web.

Amazon Web Services (AWS)

The two services used from AWS were Amazon Elastic Compute Cloud (EC2), for creating and running worker instances in the cloud, and Amazon Simple Queue Service (SQS), for passing messages between the local script and remote workers. Note that the choice of EC2 instance was limited by using the free tier of AWS. The free tier only allows usage of *t2.micro* instances, for a total of 750 hours a month.

IV. Core Algorithm

Before deploying to the cloud, the algorithm was first implemented locally to ensure proper functionality and correctness of results. The core of this algorithm is using SHA-256 to hash and compare the concatenation of the input string and a 32-bit number. Hashlib is a standard library for Python which is described as providing secure hashes and message digests and includes an implementation of SHA-256. Manipulating this to create the SHA-256^2 function was as easy as calling the function twice, passing the output of the first hash function to the second.

The difficulty in implementing the overall algorithm came from handling types in Python. The Hashlib library requires the bytes type as input to the SHA-256 function and this in turn depended on the correct encoding.

Although it was possible to implement the comparison step with bit-manipulation it was much easier to simply compare the decimal value of the hash to the target value defined by 2^{256-d} where d is the set difficulty.

The algorithm works as follows: For each integer in the given search range

1) Encode the input string in utf-8 format and pack the current integer into a 32-bit unsigned integer (big-endian)
2) Concatenate the two pieces of data from the last step and pass them as input to the SHA-256^2 function
3) Retrieve the hexadecimal output of the SHA-256^2 function and convert this to an integer value
4) Compare the resulting integer value to the value 2^{256-d} where d is the set difficulty
    - If less, then return the integer index
    - Else continue searching the space

V. Scaling for the Cloud

A. Overview

As is discussed in section I, this task is 'embarrassingly parallelisable' and therefore easy to scale up horizontally. Scaling horizontally means increasing the number of machines rather than the computational power of the single machine. Instead of searching the entire search space, the space can be divided into segments, relative to the number of workers set by the script. Each of these segments is defined by a start and an end index, both of which are passed as a single message to a remote worker via an SQS queue. The workers on the other end retrieve a message each from the queue and search the segment defined by the message. Upon calculating a result, the worker sends the result via another SQS queue back to the local distributor. The distributor takes the first message that arrives as the final result and shuts down.

B. Amazon Machine Images (AMI)

"An Amazon Machine Image (AMI) provides the information required to launch an instance" [1]. When using Amazon's API it is necessary to provide the ID of an AMI in order for AWS to know how to launch the instance. These AMI's can be created from a running instance or from pre-defined AMI's provided by AWS.

For this system, a custom AMI is created in order to simplify the process of initiating remote instances. This differs from the base *Amazon Linux 2 AMI (HVM), SSD Volume Type* image in that it also includes the Python script necessary for the worker to find golden nonces and communicate with the local script, and installations of both Python3 and Boto.

C. Boto

"Boto is the Amazon Web Services (AWS) SDK for Python" [2]. It enables the local Python script to manage services such as EC2 and SQS, but also to manage various credentials.

D. Architecture

The architecture takes the form of a distributor model, where the local script distributes segments of search space proportional to the number of workers in the cloud. SQS facilitates this with an input queue, which distributes the work, and an output queue, which receives results back from the workers. The architecture is illustrated below.

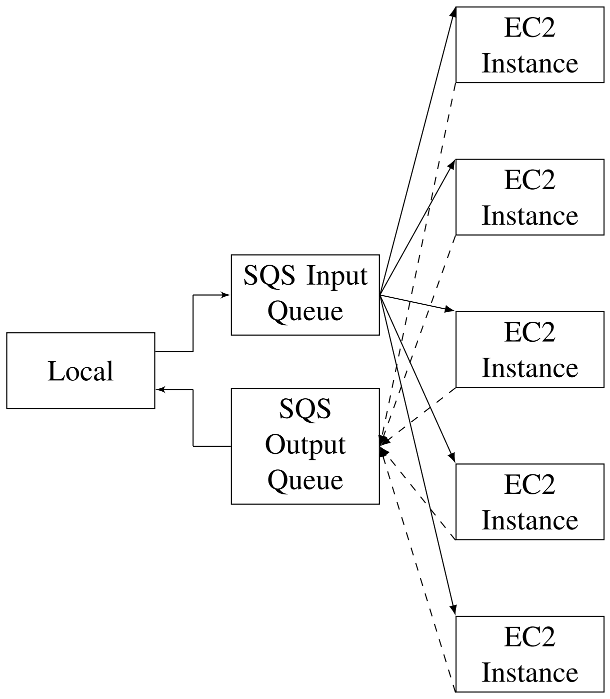

Fig. 1. Architecture diagram for five EC2 workers

E. Details of Scaled Algorithm

The local distributor on the left-hand side of Figure 1 begins the process by initialising a number of Boto variables necessary for communicating with the remote worker instances, this includes creating the SQS queues necessary for communication. After this, the local distributor processes the data in the *user_data* script. This is a short bash script which is passed to each instance at launch [6] and defines the command(s) necessary to run the CND algorithm. Next, the distributor creates instances in the cloud using the referenced AMI and the *user_data* script. After that, the distributor calculates the size of each segment to be searched and creates indices from this list of segment sizes. The distributor then pushes these to the input queue and continuously polls the output queue for a response.

Once launched, the EC2 instances on the right-hand side of Figure 1 also begin by setting up Boto variables, before checking the input queue for indices. Once received, the workers commence their search of the given space, concluding with either a golden nonce or a value of negative one which indicates a failed search. The result of the search is then pushed to the output queue and the worker script terminates. At this point the local script will check the result received and return it to the user if it is not equal to negative one.

F. Implementation of Time Limit, Confidence and Cost Limit

Aside from the standard behaviour, it is possible to run the cloud system with the following options:

1) Time limit
2) Cost limit
3) Probability of finishing within the given time limit (this is referred to as Confidence)
4) Cost limit and Confidence

Time and Cost Limits: In order to implement both time limit and cost limit it was necessary to implement a second thread for a countdown timer. In both cases, a countdown timer runs on the main thread with the standard behaviour running on a secondary thread set as a daemon. When the countdown ends the secondary thread is terminated and information is output to logs. Cost limit is implemented as a time limit and is calculated from an estimate of the price per second of running a *t2.micro* instance.

Confidence: As discussed in subsection VII-B, the number of workers w needed to find a golden nonce of difficulty d in time limit l with confidence p can be estimated using equation 5.

VI. Automation of Setup

The CND system relies on the existence of a prepared AMI. This AMI contains the worker script but also has Python3 and Boto installed. The creation of the AMI is automated with Terraform and Boto and can be performed by running the *setup.sh* script.

Terraform

Terraform is used to automate the creation of the initial *t2.micro* instance (with name and security groups) from which the AMI is created. The information for this instance is in the Terraform template file *main.tf*.

Boto

Boto automates the creation of the necessary security group, Identity and Acess Management (IAM) instance profiles and roles, and the creation of the AMI from the running instance.

VII. Performance Analysis

A. Logging

When either the local or distributed implementations of the CND algorithm terminate, they output information to log files. These log files may include various information such as the reason for termination, the time taken, the number of workers, and the start and end index of the given search space.

B. Local and Theoretical Performance

In theory, after accounting for the time taken to launch remote workers, the performance of the system at a given difficulty should scale linearly with the number of workers. This is because multiplying the number of workers by x means that each worker must only search $\frac{1}{x}\times$ the size of the initial space. Given that the performance of the processor should roughly remain the same without too many other processes running in the background, the time taken to find a golden nonce should roughly scale linearly.

By making the assumption that the result space of the SHA-256^2 algorithm is uniform, we can derive an equation for the number of workers necessary to find a golden nonce within a time limit l and with a probability p.

The probability of a golden nonce existing in the space is

$$p = 0.5^d \tag{1}$$

where d is the difficulty and thereby the number of leading zeros for the binary value of the nonce. The probability of not finding a nonce must therefore be

$(1-0.5^d)^n$, where n is the number of nonces tried, and we can thereby say that the probability of finding a golden nonce in the space is

$$q = 1 - (1 - 0.5^d)^n \tag{2}$$

Rearranging this equation we can state that the number of nonces n to search in order to find a golden nonce must be between $\frac{log(1-p)}{log(1-0.5^d)}$ and 2^{32}, i.e.

$$\frac{log(1-p)}{log(1-0.5^d)} \leq n \leq 2^{32} \tag{3}$$

Assuming a constant throughput r we can define the time taken t for a single instance to find a golden nonce as

$$t = \frac{n}{r} \tag{4}$$

Finally, we can calculate the number of workers w necessary to compute a golden nonce within a time limit l with probability p as

$$w = \left\lceil \frac{t}{l} \right\rceil \tag{5}$$

Equation 5 is of course only an estimate given the large number of latent variables in the real system such as communication latency and programming language choice.

Figure 2 shows how the number of nonces necessary to try before finding a golden nonce scales with difficulty for a constant probability of 0.99. The equation plotted here is $n = \frac{log(0.99)}{log(1-0.5^d)}$. This shows that in theory, the time taken should grow exponentially with difficulty. Indeed, Figure 3 shows the time taken for the locally tested CPU to search the entire space plotted against difficulty. It's clear that the time taken grows exponentially, rapidly rising around a difficulty of twenty-eight.

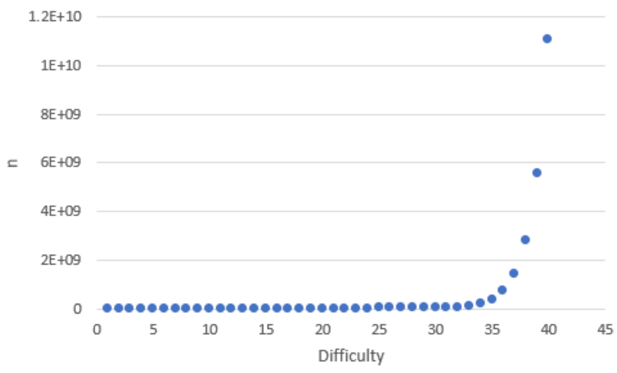

Fig. 2. Plot of n (number of nonces necessary to search to find a golden nonce) where the probability of finding a golden nonce is set to 0.99 and the difficulty d varies from 0 to 40

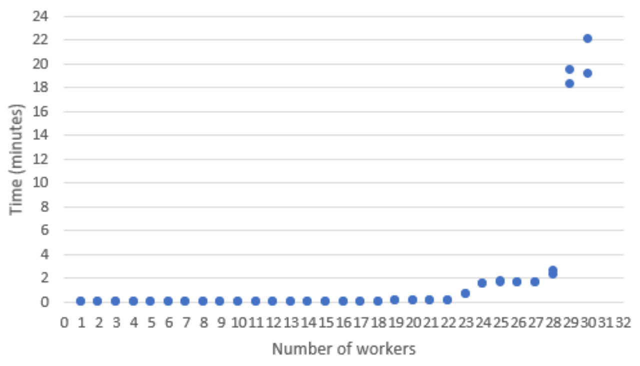

Fig. 3. Performance of locally tested CPU against difficulty

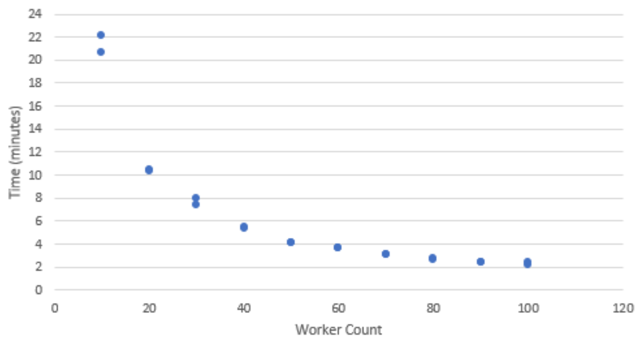

Fig. 4. Performance of locally tested CPU scaled with worker count

Figure 4 shows the performance of the locally tested CPU scaled with the number of workers. This data was gathered by first calculating the size of each segment to be searched given a number of workers. This size was then used to define a single segment for the local implementation of the CND algorithm. It appears to show that the performance scales logarithmically with worker count, not linearly as predicted. This is likely due to the latency of CPU communication [7].

C. Scaled and True Performance

In order to gather timing information from the system, two new queues were created which allow the distributor and workers to further communicate. The first queue allows the workers to send their respective start-up time, that is to say the time taken for EC2 to load the AMI and run the script, back to the distributor. The second queue allows the workers to send back running time, or in other words, the time taken to find a golden nonce.

After many runs, it was found that the start-up time for the AMI used for the task was very consistent at around thirty-nine seconds on average. The start-up timing queue was then removed given that a rough estimate of start-up time could be derived from total time and calculation time implying that the extra communication was thereby unnecessary.

Figures 5 and 6 illustrate the linear decline that section VII-B predicted. The outlier in the top right-hand corner of Figure 5 is a result of CPU throttling. This is discussed in section VII-C. Removing this outlier and thereby its influence on the plotted trend curve results in Figure 6.

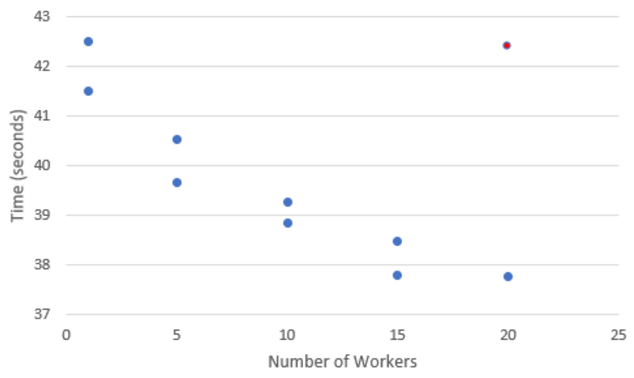

Fig. 5. Time taken for a difficulty of 15 plotted against the number of workers. Apart from the outlier in the top right-hand corner, the series shows a linear decline.

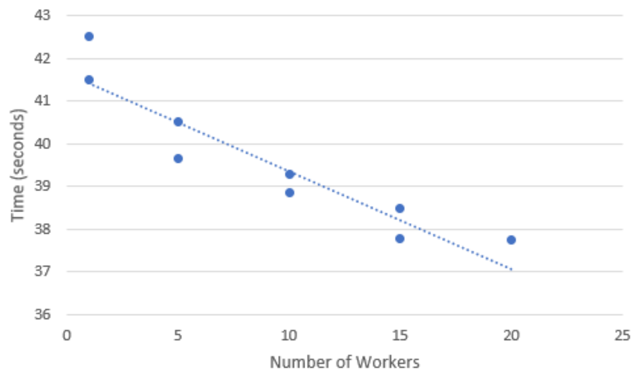

Fig. 6. Time taken for a difficulty of 15 plotted against the number of workers (removal of outlier and addition of trend line)

T2 Micro Throttling

Performance for higher numbers of workers and for higher difficulties was significantly hindered by throttling. Burstable instances are defined as being able to rapidly increase CPU performance level from a baseline level (for *t2.micro* instances this level is only 10%) [3][4]. AWS CPU credits are earned by burstable instances which use ”fewer CPU resources than is required for baseline performance (such as when it is idle)” [4]. This entails that unless burstable instances frequently under-utilise CPU resources, CPU credits will dwindle and eventually run out. One CPU credit allows full CPU utilisation for one instance for one minute, these credits can be shared amongst instances however, CPU utilisation percentage is then also shared [4]. What this implies is that unless the CPU credit balance is high, running a large number of instances does not result in an increase of computational throughput. Instead, CPU utilisation is shared and additional communication latency is added to overall running time. This effect on running time, where using large numbers of instances does not result in linear scaling due to throttling, can clearly be seen in Figures 7, 8, 9 and 10. Figures 7 and 10 in particular shows the extent to which throttling affects performance. The trend curve for Figure 7 appears to be polynomial, with the time decreasing to a minimum at a difficulty of fifteen before increasing for higher numbers of workers. On the other hand, Figure 10 appears to show a linear increase in time with number of workers. It is predicted that these particular timings were taken at a time of very low CPU credit. A clear implication of throttling is that equation 5 (the equation for calculating the necessary number of workers) cannot be relied upon when CPU credit is low.

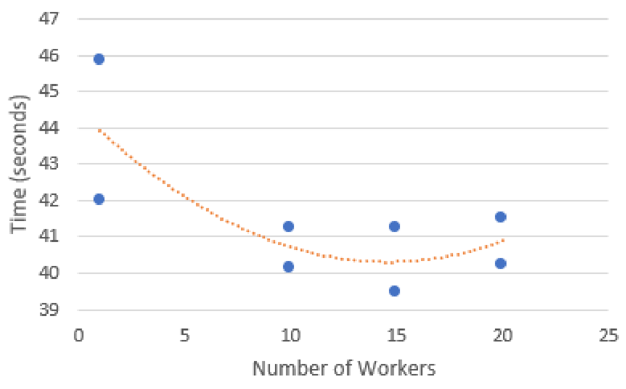

Fig. 7. The effect of throttling on varying numbers of workers for $d = 20$

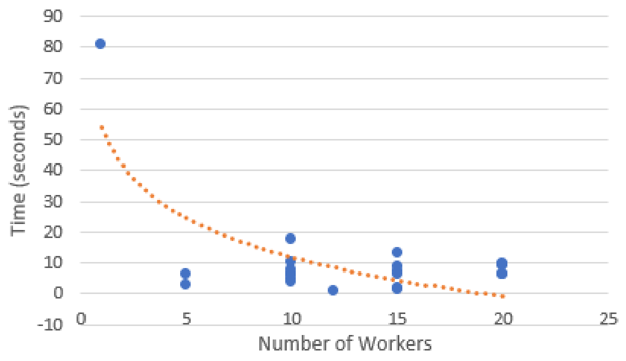

Fig. 8. The effect of throttling on varying numbers of workers for $d = 24$

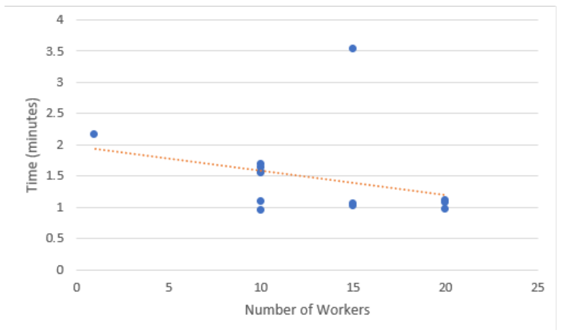

Fig. 9. The effect of throttling on varying numbers of workers for $d = 26$

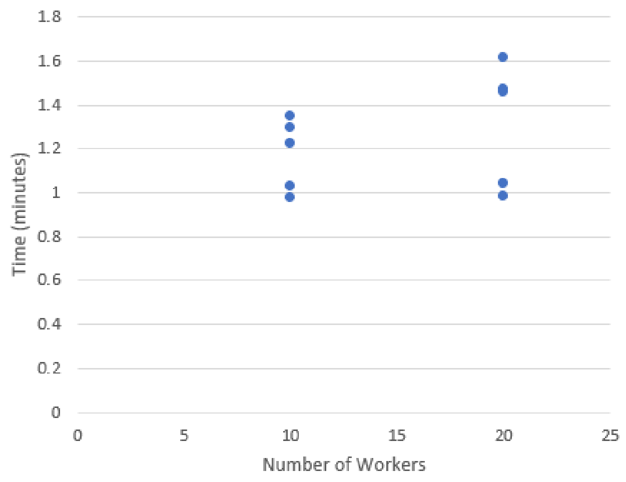

Fig. 10. The effect of throttling on varying numbers of workers for $d = 28$

VIII. Docker

It was decided that Docker would not be used for this system at the initial stages of planning. Whichever way workers are launched in a system that uses Docker, each worker must perform a *docker pull* from Amazon Elastic Container Service (ECS) every time. This lengthens the start-up time and was therefore not used. Had the system been designed to launch and then remain running without terminating, Docker would almost certainly have been used. The most significant benefit Docker would bring would be making changes to the remote worker scripts far easier. With Docker this is as easy as a *docker build* and a *docker push*. This would be more convenient than the current deployment method using Terraform and Boto.

IX. Kubernetes

Given the small scale of the system due to the imposition of Amazon's free tier limit of twenty EC2 instances, and also given the relatively short lifespan of the system, Kubernetes was deemed an unnecessary addition to the system.

X. Conclusion

In this report a system which uses EC2 and SQS to produce a distributed CND algorithm in the cloud is presented. Due to the effect of CPU credits on AWS free tier EC2 instances, the performance suffers with higher numbers of workers. Where one might expect to see a linear performance increase, a logarithmic performance increase is observed at best. For low difficulties the communication overhead outweights the increase in speed from having multiple workers and so for difficulties less than around twenty-five it is faster to run the CND algorithm locally. It would be very interesting to see how the system performs on a higher number of remote machines and machines without CPU throttling. The theory presented in section VII-B would suggest a linear increase in speed for the same difficulty once communication latency is accounted for.

References

[1] Amazon machine images (ami). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html.

[2] Boto 3 documentation. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[3] Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[4] Cpu credits and baseline performance for burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html.

[5] Proof-of-work, explained. https://cointelegraph.com/explained/proof-of-work-explained.

[6] Running commands on your linux instance at launch. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html.

[7] Performance comparison: linear search vs binary search. https://dirtyhandscoding.wordpress.com/2017/08/25/performance-comparison-linear-search-vs-binary-search/, 2017.