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

Sibela Chinareva

November 2019

Abstract

This document aims to outline the process of designing a cloud-based nonce discovery system and is part of the University of Bristol's Cloud Computing (COMSM0010) coursework. Its audience is assumed to be people who are generally familiar with programming, the term "Cloud Computing" and the different services that the Amazon Web Services (AWS) have to offer.

1 Introduction

I firstly begin with an explanation of the system's architecture, information about the different parameters needed to run the program, as well as reasoning about the choices of the different services used. After which I proceed with a thorough narration about the implementation process, outlining the different stages. Lastly, I present some analysis of the system's performance accompanied by relevant graphs and figures.

2 System Architecture

To begin with, the designed system was written in the Python programming language and therefore, uses the *Boto* library for interacting with the AWS services. Those services include EC2 (Elastic Compute Cloud), S3 (Simple Storage Service) and SQS (Simple Queue System). The program has been split into two main parts - a master code and a worker code, both of which have been put into separate files.

The *main.py* file contains the master code responsible for 1) spinning the virtual machines, 2) providing them with the worker file by updating the contents of the S3 bucket with the most recent version of that file, 3) gathering back results when one of the virtual machine completes execution, and 4) cleanly shutting down the instances and emptying the SQS messaging queue. The file requires three parameters for execution - a difficulty value (the number of significant zeros to be found in the hash when discovering a nonce), a yes/no value, and a varied parameter which is dependent on the second. The "yes" stands for direct specification of the number of EC2 instances to be created, in which case the user needs to provide this number as the third parameter. The "no" signifies indirect specification which will automatically infer that number based on a desired execution time (again, provided as a third parameter by the user).

The *pof.py* worker file captures all the code needed for executing the Proof of Work, i.e. discovering the nonce. In addition, it is also responsible for sending a message back to the master, reporting the value of the found nonce, the execution time and the resulting hash. Initially, the only parameter it took was just the difficulty value. After the parallel execution was implemented, the worker code also required a start and step value used in the nonce discovery (the for loop in the $findNonce$ function of the file).

2.1 Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud provides scalable computing capacity in the AWS cloud [1]. The system makes use of AWS EC2's Free Tier and therefore, all the instances are of type *t2.macro* with 8GB of storage. In order to save all the dependencies that are needed to run the worker code, an Amazon Machine Image (AMI) was created which has the relevant python and pip packages installed. This image is used to create new instances locally from the master code. All the instructions needed for the execution of the code were passed in the *user-data* parameter of the *create-instance* function from *Boto*'s EC2 object. These include 1) creating a logger file which would store logs of errors if any occur, 2) installing any additional packages needed for execution, 3) updating the worker code with its newest version stored in the bucket, 4) syncing the log folder with that on the bucket so that errors are examined later on if they occur, and finally 5) executing the worker code with the provided parameters.

2.2 Simple Storage Service (S3)

Amazon's Simple Storage Service (S3) is a scalable cloud storage service designed for online backup and archiving of data on AWS [3]. It was made use of in situations where larger data had to be transferred between the local machine and the virtual machines running on the cloud. For example, it has been used when sending the entire worker file to the worker. The code for the transfer is located in a whole separate function (*putInBucket*) in the master file and is called only when a change to the worker file has been made. Using the AWS Client this file is then copied to each virtual machine's storage on its instantiation. This method was preferred over copying the file only once and saving it when creating the AMI of the desired instance state because it allowed easily updating the script without having to create a new AMI every time. As the file was constantly being updated during the system implementation process, this design choice managed to save a significant amount of developing time.

The S3 bucket was also used in the case of an error, the logs of which are stored in a file called *user-data.log* in the */var/log/* directory on the instance's storage. The contents of this folder are synced with a folder on the bucket so that when a problem occurs, the user can see what happened by examining the stored logs in the bucket.

2.3 Simple Queue Service (SQS)

On the other hand, when wanting to communicate small amounts of data between the local and the cloud machines, the system uses Amazon's Simple Queue Service (SQS). SQS allows you to queue and then process messages [2]. A single queue has been created for the system and its incoming messages are all one-directional - from worker to master. A message is put on the queue when a worker discovers a nonce. The content of the message is information about what the nonce is, what the resulting hash was, how much time it took and which worker managed to discover it.

3 Implementation Process

3.1 Local Execution

To begin with, the system was firstly implemented locally, without being split into master and worker sections. Therefore, only a single file was needed to begin the nonce discovery process. The main function of that file (called $findNonce$) consists of a for loop that iterates through the values 0 to 2^{32} and stops when it finds a value which, appended to the data block "COMSM0010cloud", results in a hash whose number of leading zeros is larger than the selected difficulty. The SHA256 encryption is implemented using the $sha256()$ function from Python's *hashlib* library. Initially, a while loop was considered when implementing the nonce discovery. The idea was that the loop would generate random numbers from the desired intervals on every iteration. However, as we will look at later on, the for loop complied better to the system when transforming it into a parallel one.

3.2 Single VM Execution

After the nonce discovery was successfully implemented, the development process was proceeded to the next stage - executing the algorithm on a single virtual machine in the cloud. This stage began by firstly creating a separate file which would contain the master code responsible for interacting with the cloud. To obtain access to the AWS services remotely, an IAM user and role were created and attached to the designed aforementioned AMI. The access and secret key needed when using the AWS client are stored in an environment variable which is accessed by the Python master script using a module called *os*.

The first thing that the master code does after initialising the *Cloud_Pof* class, is to clear out all the messages that are currently present in the queue. This way when data is collected back from the virtual machine, the user can be sure that the received message was sent by the most recently ran instance. The next step is creating the instances themselves. This was done using Boto EC2's *create_instances* function which has been supplied with the Image-Id of the previously designed AMI, the appropriate region, instance type, key-value pair and finally, the IAM role that will be allowed access to managing the instances (by providing the Instance Profile ARN of that role in the relevant parameter of the function). Meanwhile, as this is happening, the user is instructed about the currently ongoing process of creating the instances. Once this step has finished, the user is also notified for its completion. To figure out when that happens exactly, the function *describe_instances* was used which has its *Filters* parameter set to return only the running instances. A continuous while loop is entered if that function does not return anything, and the loop does not stop until such an instance is found. The user, however, has the option to stop this process by pressing $Ctrl + C$ which will cleanly shut down all instances (both currently being created and already running).

After creating the instances, fetching the necessary data, and instructing the user of the successful completion of this step, the program progresses to the next stage - waiting for computation to finish and collecting the messages. This is done in the *collectMessages* function which firstly checks if there are any messages currently present on the queue (which we have ensured to be empty so that different executions are treated separately). It then enters another while loop which constantly checks for messages to be present on the queue (using the Boto SQS Client's *receive_messages* function) and does not proceed until a message is received. Again, the user is given the option to exit this constant search by pressing $Ctrl + C$ which would also cleanly shut down all the virtual machines and halt computation. In addition, the same exiting procedure would also be executed once a certain time limit has been reached (currently set to five minutes).

Lastly, as soon as a message is discovered, the user is notified about its content and the *clearEC2State* function is called. This function takes care of terminating all the currently running virtual machines, as well as clearing any messages currently present on the queue. This is also the end of the program.

3.3 Multiple VMs Parallel Execution

The next development stage was splitting the computation across multiple instances which are simultaneously running on the cloud, i.e. making the system parallel. Initially, this was planned to be done using the *MinCount* and *MaxCount* parameters of the *create_instances* function, setting both of those to the desired number of running instances. However, although this did spin the selected count of virtual machines, the code that they were all working on was the same, so the overall effect when executing it simultaneously was no different than the one achieved by a single instance.

Therefore, to make the whole process parallel and actually speed up computation, another technique had to be implemented. In the described system, this was achieved by sending a different starting index and a step value to every worker. The idea was that no two workers should ever look at the same nonce value, and the search space is evenly split between them. This was achieved by having a for loop (going from 0 to the desired instance count) which initialises a single instance on each iteration, sending it a start index equal to the current iteration of the loop and a step value equal to the total number of instances to be created. A diagram visualising the splitting process can be found in Figure 1 below.

In order to stay withing the free tier allowance, a limit to the number of instances has been created when using both direct and indirect specification. The user is only allowed to run between 1 and 15 instances simultaneously.

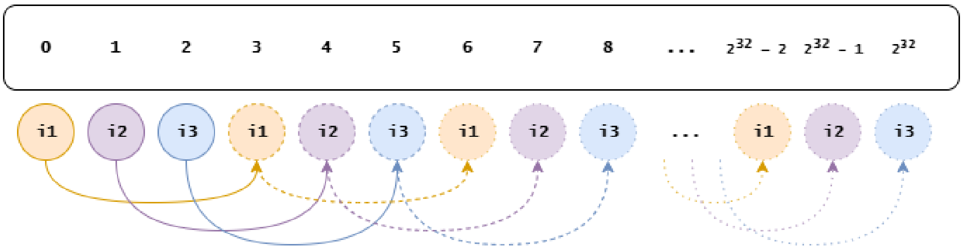

Figure 1: Splitting the search spaces across three instances (i1, i2, i3). They all start in separate consecutive positions, but their step value is all the same (equal to 3). This way each worker skips the space already searched by the other workers at previous time steps.

3.4 Indirect Specification for the Number of Instances

Last but not least, the final and possibly most exciting feature that was added to the program is the indirect specification for the number of running instances given a desired discovery time selected by the user. This was achieved by firstly collecting some data about the execution times of the program using a range of parameter values. After that, the taken metrics were examined and a pattern was sought across the different experiments. Naturally, a relationship existed between the number of instances, the difficulty level and the computation time. Therefore, proportions were used when trying to find one variable given that the others are already known. In particular, a difficulty value was divided by its computation time when ran on a single instance to produce a constant c. After which, when trying to infer the number of instances (N) which would be needed to execute the selected difficulty (D) for the desire time in seconds (t), the following formula was used:

$$N = \frac{D}{t*c}$$

A constant was computed for a range of difficulty values and a pattern was discovered between them. It seemed that the difference between every two constants was approximately equal to the difference between their difficulty values, multiplied by 2. Therefore, only one of the constants needed to be hard coded into the program (referring to a specific difficulty value) and all the other ones could easily be inferred by the following equation:

$$\text{new\_const} = \text{old\_const} + abs(\text{old\_difficulty} - \text{new\_difficulty}) * 2$$

The constant 0.45 (calculated from the difficulty value 24 and its computation time 53.378s) was used in the current version of the program. This choice had no particular underlying reason, and every other alternative was proven to lead to the same results.

Finally, this technique was tested with various difficulty values and speed requirements and its results seemed to only differ by +/-0.2 seconds when compared to the desired execution times selected by the user (provided that the latter was in a reasonable time range similar to the results of the system's performance). When an illogical time is selected for a specific difficulty, the system does its best trying to replicate that time by selecting either the lower bound (when that time period is too long) or the upper bound (when it is too short) of its allowed instance count.

4 Performance Evaluation and Comparison

4.1 Local Execution

In the beginning the nonce discovery was executed locally on a single core and although the program managed to find a nonce rather quickly when smaller values for the difficulty were selected (e.g. 4, 8, 16), it struggled when that number was higher (e.g. 20, 24). The computation times of the sequential nonce discovery on the local machine are shown in Table 1 for a range of difficulty values. As we can see from there, the performance decreases drastically for larger difficulty values, with an entire ~ 91 seconds difference between 20 and 24 number of leading zeros.

Difficulty	Time(seconds)
4	0.0002
8	0.0019
16	0.1155
20	3.7474
24	94.7829

Table 1: The execution of nonce discovery on a local machines for different difficulty values.

4.2 Single VM Execution

As only the free instance types that Amazon offers were used (which are not very computationally powerful), no noticeable difference was expected in the execution time when running the program on a single virtual machine in the cloud. In fact, the process of creating the instances, sending all the information and executing the appropriate instructions turned out to be quite computationally heavy and therefore, the overall time to run the program actually increased.

4.3 Multiple VMs Execution

A significant improvement in execution times was achieved when switching to parallel execution of the algorithm on multiple virtual machines running in the cloud. As expected for an embarrassingly parallel task, the time for running the program on N machines was C/N, where C is the time it takes for it to run on a single one. Figure 2 proves this by displaying the discovery time which is five times less when switching from a single instance to five, and halved when going from five to ten instances running in parallel.

The pink bar in the chart (Management Time) encapsulates all the time spent managing the instances - creating and stopping them, as well collecting and printing the message once it is on the queue. This part of the execution stays relatively constant across the different number of instances. The overall time of the program (green bar in Figure 2) therefore only improves because of the large difference in the discovery times across multiple instances.

On the other hand, smaller difficulty values lead to less noticeable performance improvement. As we can see from Figure 3, discovery time does not change much when going from one to five instances. Even though more significant improvement is noticed when going from five to ten running virtual machines (it is halved), the discovery time values are still all relatively small for a difficulty of 16. Due to this fact, the overall execution time of the program seems to be invariant to the number of instances. Therefore, it was concluded that using the cloud services contributed to the overall

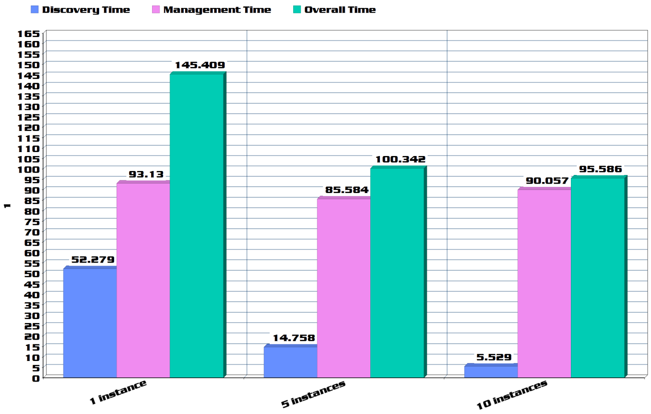

Figure 2: The discovery, management and overall time of parallel execution on the cloud with difficulty 24.

performance of the program only if the computation alone was incredibly time consuming, such that it would exceed the time spent dealing with the cloud management itself.

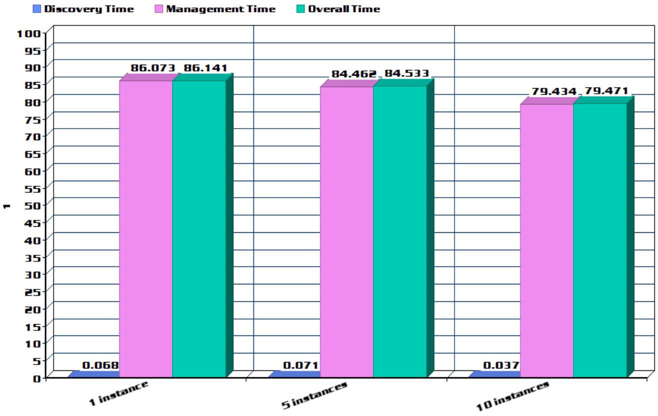

Figure 3: The discovery, management and overall time of parallel execution on the cloud with difficulty 16.

5 Conclusion

All things considered, the designed system offers a scalable and reliable computation on the cloud using the Amazon Web Services. The user has the option to either specify a set number of instances to be ran, or if this cannot be known in advance, to select a desired discovery time in which case the program would automatically infer the number of instances needed to achieve it. Moreover, it grants the user with full control throughout all of the execution stages by providing the option to halt computation and shut down all instances.

References

[1] AWS Documentation. Amazon ec2, user guide for linux instances, what is amazon ec2? `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html`.

[2] Boto3 Documentation. A sample tutorial. sqs. `https://boto3.amazonaws.com/v1/documentation/api/latest/guide/sqs.html`.

[3] Margaret Rouse. Aws analytics tools help make sense of big data. amazon s3., August, 2015. `https://searchaws.techtarget.com/definition/Amazon-Simple-Storage-Service-Amazon-S3`.