Cloud Computing Report

Chetankumar Mistry (38523)

December 2019

0.1 Link

Please find the source code for this project here:

https://github.com/cm16161/Cloud_Computing.

0.2 Introduction

In a Block-Chain, there is typically a Proof-Of-Work (PoW)[5] function which takes a block of data, and appends an arbitrary number (called a nonce) such that, when the hash of this new block (original block + arbitrary number) is calculated, the number is deemed valid (golden nonce) if the resulting hash starts with N-many 0's (called the difficulty), where N is also an arbitrary value.
Hashing functions, such as SHA-256, have some interesting properties; such as taking any sized input, but having a fixed-sized output, or, in the case of the PoW, any input has an equal probability of being a correct output.
From this it is clear how Cloud services, such as AWS can be used to speed up the calculation of a golden nonce, since every value has an equal probability of being a golden nonce, if every value is tried, then you are guaranteed to find a golden nonce, or alternatively, there is no golden nonce for this block and difficulty.

0.3 Cloud Nonce Discovery (CND) Design

0.3.1 Variable Parameters

The CND takes in as optional command line arguments 5 different parameters:

- N - the number of machines to use
- D - the difficulty level
- C - the confidence level
- T - the maximum duration of the program
- L - specify that that a logfile should be created

These parameters are managed by the Python3 `argparse`[1] library, this also provides an additional argument:

- -h –help - display information about the above optional arguments.

```
$ python generate_in_cloud.py -h

usage: generate_in_cloud.py [-h] [-n N_THREADS] [-d DIFFICULTY]
                            [-c CONFIDENCE] [-t TIMEOUT] [-l]

optional arguments:
  -h, --help            show this help message and exit
  -n N_THREADS, --n_threads N_THREADS
                        Input the number of Virtual Machines to use
  -d DIFFICULTY, --difficulty DIFFICULTY
                        Input the difficulty value
  -c CONFIDENCE, --confidence CONFIDENCE
```

```
                      Input a confidence value such that the program will
                      complete successfully with that confidence percentage
-t TIMEOUT, --timeout TIMEOUT
                      Input a time-out value in seconds such that the whole
                      program will terminate in that given time
-l, --log             Set this to output a logfile
```

N - the number of machines

This parameter is, by default set to 1, and is used to determine how many virtual machines are to be spun up in AWS.

D - the difficulty level

This parameter is set to 32 by default (this is the initial value used by Bitcoin), it determines how many of the leading bits of the final hash need to be set to 0 for the nonce to be valid.

C - the confidence level

This parameter is set to 100 by default and represents out confidence that, within a given time, the program will, with C% confidence, find a golden nonce. This value is used by limiting the range of nonce's the program will search through to find a golden nonce. This only works due to the fact that there is an equal probability that any given nonce will be golden, by reducing the range the program searches through, it reduces the probability that a golden nonce will be found.

T - the maximum duration of the program

This parameter determines how long the program will run for, and will terminate any and all running VM's on AWS after this time-limit has been met and gracefully exit the program. This is set to 86400 seconds by default, which is equivalent to 24 hours.

L - the logfile

This parameter determines whether or not a logfile should be created by the system to output information.

0.3.2 CND

CND works by utilising the above parameters to spin up N-virtual machines to search through all numbers between 0 and $2^{32} = 4.29{*}10^9$.
This is implemented by using a for-loop to iterate over values between 0, and 2^{32}, but each VM will start at a different value, and will step by size N. This means that for 2 VM's:

- VM 1 iterates over: $0, 2, 4, 6, 8, ...2^{32}$
- VM 2 iterates over: $1, 3, 5, 6, 9....2^{32} - 1$

The VM's are in reality, individual machines on AWS, however, the program handles the spawning of multiple VM's as multiple processes using the Python3 `multiprocessing`[4] library, to create a new process which is responsible to spinning up a VM on AWS and having it do the subsequent work, this new process is then added to a list which keeps track of all of the processes being used.

0.3.3 Processes

Each processes will spawn a single VM on a T2-micro AWS machine by using the Python3 `boto3`[2] library, with a personalised security group and key-file. The process then waits until the VM has booted, initialised, run all the necessary checks before it can be used, and then connects to it using the key-file specified earlier by using the Python3 `paramiko`[6] library.

0.3.4 The Virtual Machines

Using `paramiko`, the file `nonce_finder.py` is transferred over to the VM, and then the following command is executed on the VM:

- `sudo yum install python3 -y > /dev/null && python3 nonce_finder.py` (+ parameters)

We can break down this command into separate components:

- `sudo yum install python3 > /dev/null`
- `python3 nonce_finder.py` (+ parameters)

The first command downloads and install the `python3` stack onto the VM so that it can be used, and redirects all `stdout` to the file `/dev/null`, which is akin to suppressing the output.
The second command executes the program `nonce_finder.py` which performs all of the calculations and hashes to determine whether or not a nonce is valid or not, returning once it has found a golden nonce (or exhausted all of the values). The parameters which are passed in to the file are either passed in directly to the system by the user (eg. -d $difficulty$), or is calculated during process generation; the other parameters are

- –start - the same as Process ID from the main system.
  When each process is created, it is done so using a for-loop, each process receives this value so that it is possible to start `nonce_finder.py` from different values.
- –step - the same as the number of VM's (N) to be spawned.
  This ensures that values are not repeated since each process/VM iterates over unique values separated by the same interval
- –end - this is the upper limit to try nonce's up to.
  This is determined by the confidence value: $end = 2^{32} * C\%$

0.4 Early Return

The `nonce_finder.py` file works by using a range-based for loop with start, stop and step values. These values dictate which values of nonce's to append to the block to compute the hash for and determine whether or not it is a golden nonce.
If a golden nonce is found, then the program simply has to print out the value of the golden nonce and then exit the program, which is an early return; Alternatively, if no golden nonce is found after exhausting all possible nonce's, then there is no output printed from the program and it will simply exit.

0.5 Waiting for Results

Once all of the processes have been created, the CND needs to wait for them to return with a value (if any) or, for the timeout duration to be met.
For timing out, at the beginning of the `main()` function, the system executes `start = time.time()` which utilises the `time` library and stores the initial time of the program. Whilst waiting for a VM to return, the system will continually get the current time and then check it with:

```
if time.time() > start + duration:
```

If this condition is true, then CND should stop waiting for a VM to return, and instead should forcibly terminate all processes and VM's like so:

```
for _p in processes:
    _p.terminate()
```

Where `processes` is a list containing all of the running processes, which will forcibly terminate all of the running processes; this is then followed by the command:

```
kill_instances.kill()
```

This runs the method `kill()` defined in the file `kill_instances.py` which gets a list of all of the EC2 instances on an AWS account and then terminate them.
Alternatively, if a VM returns because it has found a result, the corresponding process is then allowed to finish gracefully. What this means is that it is possible to check the status of all processes like so:

```
for _p in processes:
    if _p.is_alive()
```

If that process is alive, then the program can just continue to the next iteration of the loop, otherwise, that process has finished gracefully, and so we can forcibly terminate all of the other processes and VMs.

This works because of the way that `paramiko.client.exec_command()` is implemented, which is that it blocks the calling thread until the `exec_commnd()` instruction has returned. This means that it is possible to spawn multiple VM's and be confident that the process will only return early if the golden nonce has been found. As a result, when checking the status of a process, if it is still computing a result, `exec_command()` will be blocking the process and so the condition:

```
if _p.is_alive():
```

will return true, resulting in the default behaviour of continuing on in the loop.

```
if _p.is_alive():
    pass
```

0.6 Dynamic Scaling

Up to this point, the described system works for using a manually set number of VM's, it is possible to dynamically set the number of VM's to use based on a given confidence level and a timeout duration. As discussed earlier, setting the confidence is as simple as reducing the range of values to iterate over. In order to use this information to set the number of VM's to use, I wrote a python script which allowed me to estimate the time it took to calculate a single has on a T2-Micro machine. This was done by calculating 100 hashes, 10000 times, to get an average time it takes to calculate 100 hashes. This value is then divided by 100 (since it was 100 hashes) and then multiplied by 2^{32} to get the average time it would take a single T2-Micro machine to go through all of the values.

$$t_1 = t_{100}/100$$
$$t_{100} = t_{10000}/10000$$

Where t_{10000} is the runtime of the script `time_statistics.py`

This results in having the time taken to scan all 2^{32} values, which can then be scaled by using the confidence value to reduce the amount of time required to search through the full range of numbers. This overall time to calculate all of the values can the be divided by the allotted timeout duration to get a ratio between max-time and given-time, where this ratio represents the number of VM's required to do this (**NB.** this value is rounded up to the nearest integer).

$$\text{N\_VMs} = (\text{T\_One\_VM\_All\_Values} * \text{Confidence}) \ / \ \text{Timeout}$$

0.7 Logging

If the parameter -l is passed in, then at each step in the system, a logfile, called `generate_in_cloud.log` is updated so that it tells a user what is going on. Initially it outputs the different parameters to the system, i.e. number of VM's, maximum duration, confidence values, and difficulty level. Then, it outputs how the system exits, i.e. due to timing out, a VM returns successfully etc.
The logfile itself is managed by the Python3 `logging`[3] library which will periodically update the logfile with relevant information.

0.8 Results

The average time taken for a T2-micro machine to perform 100 hashes was measured at approximately 500μs. By using the equations above, it is possible to calculate the time taken for a single hash:

$$\begin{gathered} t_1 = \frac{t_{100}}{100} \\ t_{100} = 500 * 10^{-6} \\ t_1 = \frac{500*10^{-6}}{100} \\ t_1 = 5 * 10^{-6} \end{gathered}$$

Knowing the value of t_1, it is clear how to calculate the maximum runtime of a single T2-micro performing the full range of hashes:

$$\begin{gathered} t_{max} = t_1 * 2^{32} \\ t_{max} = 5 * 10^{-6} * 2^{32} \\ t_{max} = 21524 \end{gathered}$$

This means that the CND will finish within $21524s \approx 6$ hours

0.8.1 D = 10

Figure 1 shows the time it takes CND to find a golden nonce when difficulty D = 10, for a varying number of VM's (1,2 and 10).
What the results show is that the time in seconds for CND to complete does not vary much between different N's. This is due to the fact that the Nonce's returned are all computed quickly, meaning that the overhead of spinning up, and terminating a virtual machine are the majority of the time spent.

0.8.2 D = 15 and D = 20

Similarly to figure 1, figures 2 and 3 show that increasing the number of VM's N, does not speed up the overall performance of CND. This is again due to the fact that golden nonce's are computed quickly, so the majority of the time is spent spinning up, and terminating the VM's.

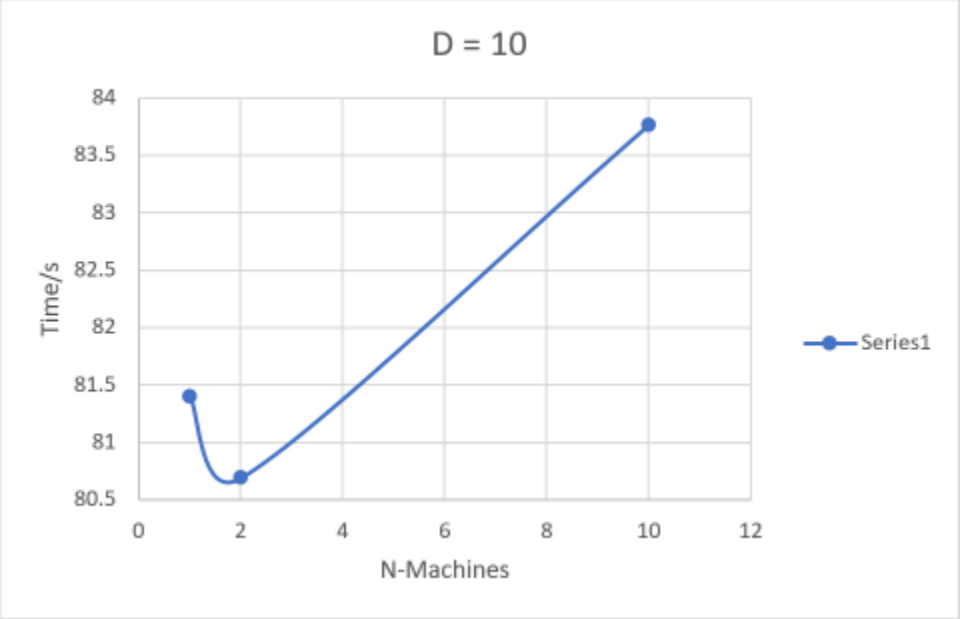

Figure 1: Times taken for D=10, N=1, N=2 and N=10

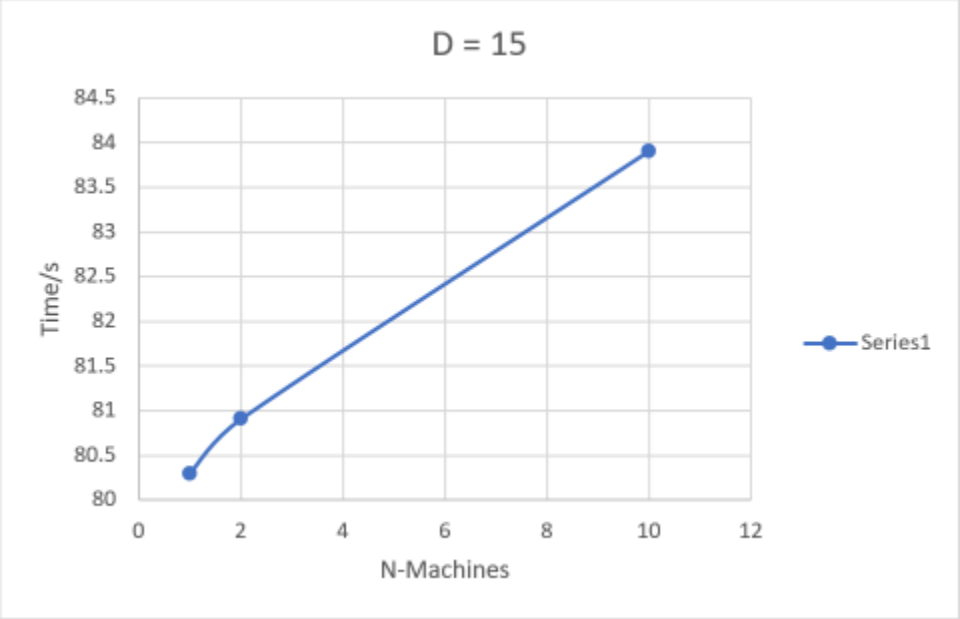

Figure 2: Times taken for D=15, N=1, N=2 and N=10

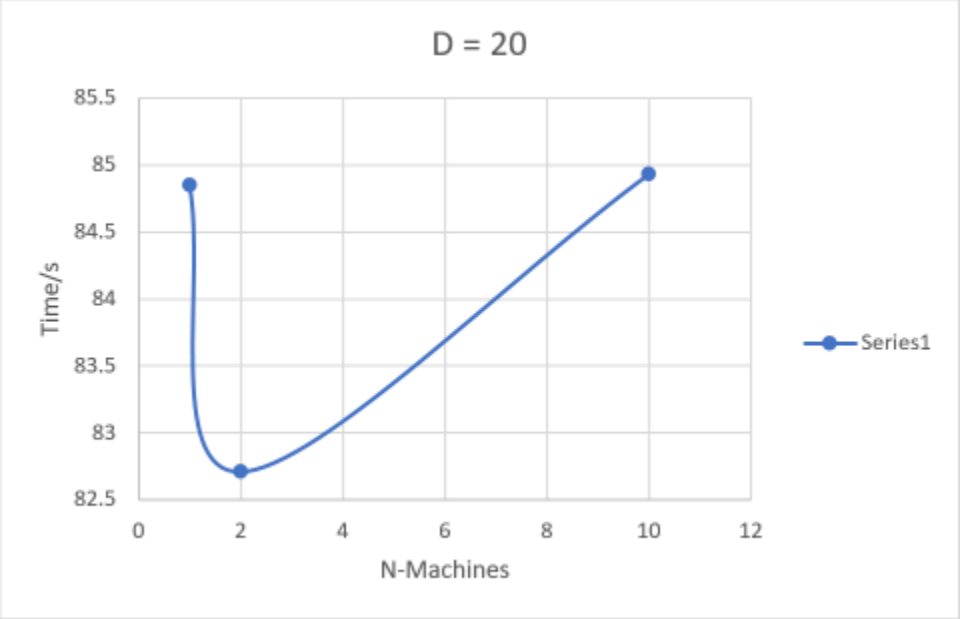

Figure 3: Times taken for D=20, N=1, N=2 and N=10

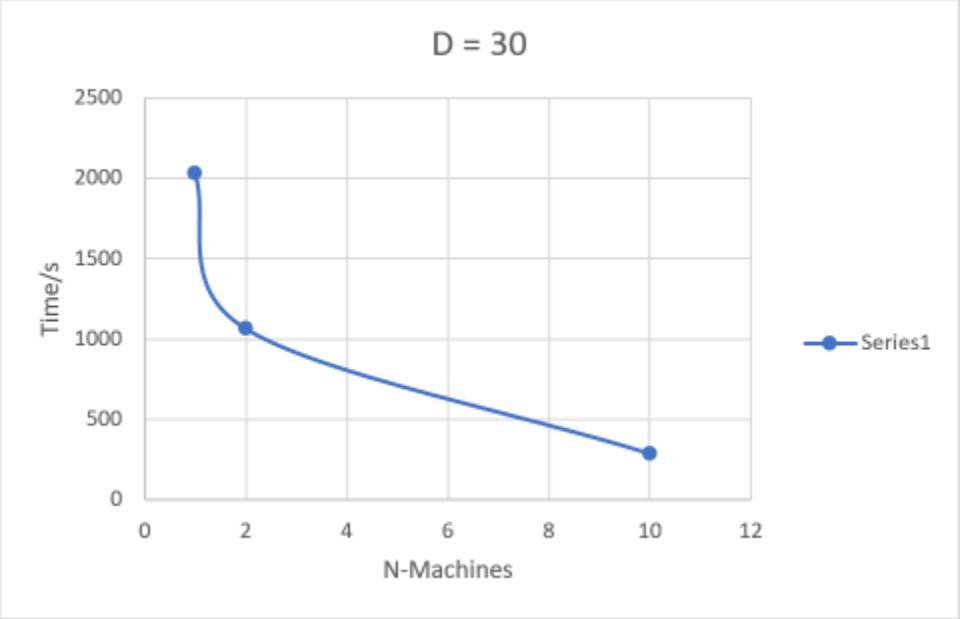

Figure 4: Times taken for D=30, N=1, N=2 and N=10

0.8.3 D = 30

Figure 4 provides more interesting information. Difficulty D is sufficiently large here to demonstrate the performance benefits of parallelising the task horizontally in the cloud. The difference it takes to generate a golden nonce for N=1 is almost double the time it takes for N=2. These values are also almost perfectly scalable by N=10, as in:

$$t_1 \approx 2 * t_2$$
$$t_2 \approx 5 * t_{10}$$
$$t_1 \approx 10 * t_{10}$$

where t_1, t_2, t_{10} are the times taken for N=1, N=2, and N=10 respectively

Bibliography

[1] *argparse — Parser for command-line options, arguments and sub-commands.* https://docs.python.org/3/library/argparse.html#action.

[2] *Boto 3 Documentation.* https://boto3.amazonaws.com/v1/documentation/api/latest/index.html#boto-3-documentation.

[3] *logging - Logging facility for Python.*

[4] *multiprocessing — Process-based "threading" interface.* https://docs.python.org/2/library/multiprocessing.html.

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

[6] *Welcome to Paramiko's documentation.* http://docs.paramiko.org/en/2.6/index.html.