UoB COMSM0010 Cloud Computing, Academic Year 2019-20
Coursework A (summative assessment, 50% of total grade for the unit)

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

Date issued: Thursday October 3rd, 2019 (Lecture 02).
Submission deadline (on SAFE): 12:00 noon on Thursday December 6th, 2019.

1. Introduction

In the lectures on this unit we cover the difference between *vertically*- and *horizontally*-scaling system architectures. One of the advantages offered by cloud computing is elastic (smoothly varying) horizontal scaling: new virtual machines (VMs) can be created, used, and deleted as and when they are needed. In this way, as the computational demands on a system vary over time, the number of VMs used by the system can be scaled up and down as required.

Horizontal scalability is particularly attractive for situations where the task at hand is *embarrassingly parallelizable*. A computational task is embarrassingly parallelizable if, when it is spread over N machines working in parallel, the expected time for completion of the task is C/N (or less), where C is the time it takes a single machine to complete the task.

One example of an embarrassingly parallelizable task is rendering computer-generated-imagery (CGI) video frames for a movie. Digital video is a sequence of static frames, and movies are commonly produced to be displayed at 30 frames per second. If rendering a 30-second CGI movie on a single machine takes 7.5 minutes, that's 900 frames in 450 seconds of machine time. Assuming that each frame takes the same amount of time to render, that's 0.5 seconds per frame. If this rendering is done in the cloud, using VMs of the same processing capability (i.e., same clock speed, memory, etc), then using two VMs would halve the rendering time to 3.25min (450s); using ten VMs instead would reduce the render time to 0.75min (45s); and in the limit the whole movie could be rendered in 0.5s if you use an array of 900 VMs. Because VMs are paid for on a metered "power-by-the-hour" basis, if your cloud provider allows sub-second billing, and so long as you shut down each VM as soon as it has no more work to do, then the total cost will be the same regardless of how many VMs you decide to use, because 1x0.5x900=2x0.5x450=10x0.5x45=900x0.5x1= 450s of VM time.

In this coursework assessment, you are required to **work alone** to produce and submit a short written report, and associated code, describing a system that you create that uses horizontal scaling on a commercial cloud provider for an embarrassingly parallel computational task. The length of the report should be **no more than 10 pages** of A4 in 11pt text with 2cm margins on all edges, but we do not expect or require you to fill all ten pages: you could plausibly write a perfectly good report in six pages. The task we have chosen is a crucial step in the *proof of work* (PoW) stage of the Blockchain distributed ledger protocol that underlies the Bitcoin cryptocurrency: using the SHA256 cryptographic hashing function repeatedly in a brute-force search for what is referred to in the terminology of Blockchain as a *golden nonce*. More details of this task are given in the Appendix of this document, but in essence it is

just another embarrassingly parallelizable task, like CGI movie rendering, that can be sped up by horizontal scaling in the cloud.

We very strongly recommend that you write your code in Python and use the free-tier services of AWS for cloud provision. Python is an easy language for beginners to learn, and yet has also rapidly become the language of choice for scientific computing, data science, and machine learning applications. One reason for its popularity is the very wide range of open-source free-to-use code libraries. So, for example, the SHA256 cryptographic algorithm discussed in the Appendix to this document is already available in trusted implementations, and the *Boto* framework is a long-established Python interface to AWS. As this is a compute-intensive task, you can expect to use AWS's EC2 service for starting and stopping VMs (*instances* in AWS terminology), and probably also the SQS service for communicating with the VMs.

Finally, because it is easier to refer to any system by a proper noun (rather than keep saying "the system" or "this system", etc.) we'll use the name Cloud Nonce Discovery (CND) from now on.

2. CND Specification

Here we describe what the final, fully completed, version of your CND system should be capable of doing. As you will see from the Marking Scheme in Section 3, if you choose to submit work that does not match the full specification, you can still score a reasonable grade. If you are new to SHA256 and nonce discovery as proof of work, read Appendix A before you continue.

CND should allow the user to specify (directly or indirectly) how a SHA256-based nonce discovery process should be divided over one or more VMs running in the cloud. Direct specification would involve the user supplying a value of N, the number of VMs to run. Indirect specification of N could involve (for example) the user specifying a desired time by which there is a 95% or 99% (or other level) of confidence that a golden nonce will have been discovered, and then setting N accordingly. Another form of indirect specification would be for the user to specify a maximum hourly rate of expenditure on cloud services, and for the value of N to be set accordingly. CND should automatically start up N VMs in the cloud, distribute the brute-force search for a golden nonce over the N VMs, and then when one of the VMs does find the golden nonce, CND should cleanly close down all VMs and report the nonce value back to the user. There should also be user-specified options for initiating a scram (i.e., an immediate emergency stop and deletion of all VMs), for timeout (i.e., run only until some maximum time has elapsed, then scram) and for spendout (i.e., run only until some maximum expenditure limit has been reached, then scram). Scrams should tidy things up nicely and give an option of returning log files that say what happened in the run up until the shutdown.

Your implementation of CND should allow the user to also set the difficulty-level D for nonce discovery, how many leading bits in the hash should be zero: for early-stage development and testing, use a low value of D; then watch what happens when D is set to the kind of values seen in practice.

For simplicity, the "block" of data that the nonce is appended to can just be the string *COMSM0010cloud*, with each character converted to binary via an appropriate encoding such as ASCII or UTF-n.

3. Marking Scheme

Your written report will be evaluated on four criteria: scope; clarity/quality of writing; presentation (use of graphs, diagrams, etc); and evaluation (which we expect to be quantitative, with discussion of performance testing etc). The *scope* criteria is explained further below.

3.1 For a mark in the range 0-39%, create a version of CND that meets the specification given in Section 2, but which runs entirely on a local machine (e.g. your laptop or desktop computer, possibly using multi-threading to split the processing across multiple on-board processors), and present a performance analysis (e.g. how run-times vary as you alter user-specified system parameters). Remember, this coursework is for a unit in cloud computing, so if you stop here you won't get a pass grade for the coursework. if your coursework doesn't attempt do *anything* in the cloud, you can't reasonably expect to get a pass-grade.

3.2 For a mark in the range 40-54%, implement CND as described above, but arrange it so that when a the CND program/script/app is launched on a local machine, it automatically spins up a single VM in the cloud and runs the computation remotely, on that VM, reporting back to the local machine when the computation is completed.

3.3 For a mark in the range 55-69%, implement CND as described in Section 3.2, but arrange it so that the user also directly specifies *N*, the number of VMs to be used in parallel: your version of CND should then remotely spin up *N* VMs, run the PoW on each of them, and report back with details of the golden nonce as soon one of the VMs has found one. When the first of the *N* VMs reports a success back to the client, your system should then immediately cleanly shut down all the VMs and exit the cloud service provider.

3.4 For a mark in the range 70-80%, implement CND as described in Section 3.3, but extend it by gathering appropriate performance statistics so that the user can specify a desired run-time T and a desired confidence level *L,* and the system automatically chooses a value of N that will yield a golden nonce within T seconds with *L*% confidence.

3.5 For a mark in the range 80-100%, implement CND as described in Section 3.4, but use your own ingenuity and creativity to extend it in interesting and useful ways. In your report, explain clearly why the extensions you have introduced are interesting and useful, and discuss the technical challenges that had to be addressed in the extended implementation.

Appendix: Nonce Discovery as Proof of Work (PoW)

One major problem with electronic currency is the need to avoid *double-spending*: if I actually give you a £1 coin, the transfer of the physical metal coin from me to you prevents me from spending it after I've given it to you; but if I electronically give you £1 via purely digital means, if I am really sneaky I might find a way to quickly also give the same electronic £1 to someone else, maybe many times to many other people, and in this way I could defraud the system.

The popular cryptocurrency Bitcoin relies on a specific distributed ledger technology (DLT) called Blockchain for its record-keeping, which by clever means greatly reduces the chances of anyone being able to double-spend their bitcoins. A *ledger* is just a record of transaction details such as who transferred what money to which recipients, and when, and how much money the giver (payer) and the receiver (payee) have in their bank-balances after the transaction is completed. All traditional banks maintain ledgers, in their central records, but if you don't trust the bank, there is a possibility that the main ledger's record of electronic currency transactions could be altered to hide double-spending. A DLT is one that is distributed across multiple nodes, with each node holding a copy of all (or a large part) of the total ledger, so that if any one node was to alter its copy, all (or many) of the other nodes would notice the discrepancy when they compare ledgers.

In Blockchain, the most recently-announced transactions are grouped together into a new *block*, and all the transactions in that block are then jointly verified. As part of the verification process, each block has a *256-bit hash* of the previous block appended to it. Hashes are explained in more detail below, but the key thing is that the hash of a block is a unique short one-way encoding of all the data in that block. Because the N^{th} block to be verified has an encoding of its predecessor (the $(N-1)^{th}$ block) appended to it, and in turn the $(N-1)^{th}$ block had an encoding of the $(N-2)^{th}$ block appended to it, and so on, each block can trace a linear path through encodings of all previous transactions back to the first ever block (known as the *Genesis Block*); and that's what gives Blockchain its name.

A *hash algorithm* takes as input a chunk of data of any size, and always (i.e., for any input, of any size) returns a fixed-size output which is commonly referred to as the *hash* of the input. The size of the input and of the hash are usually measured in bits (binary digits) so some people use *length* as a synonym for size. Hash algorithms are deterministic, i.e. they always produce the same output for any one specific input. Crucially, for a hashing algorithm to be considered good (or "strong") if the input is altered by only one bit, then the hash of the altered input should be wildly and unpredictably different from the hash of the original input; hence, for a strong hashing algorithm, attempts at analysing the output hash should tell you almost nothing about what the original input data was from which the hash was generated.

Good hashing algorithms are used a lot in various aspects of computer security because they act as a form of one-way encoding: if I show you a good hash of some input, if you don't already know the input then you'll never be able to guess what it was; but if you do know the input, you can tell me it and the hash algorithm would demonstrate whether you're correct or not. A safe way of storing computer passwords for a group of user-accounts is as hashes. Anyone who sees the hashed passwords should never be able to reverse-engineer the hashes back to the actual passwords. But when a user logs in, the characters they enter as a password can be used as input to the hash function: if the output hash of what the user enters matches the stored hash of the password, then we know the user has entered the correct password.

Let's illustrate this with two hypothetical extremely simplistic hash algorithms, HBAD and HGUD. Both of these algorithms produce fixed-length 8-bit hashes, which is too short to be of practical interest but long enough to be useful for illustration. Here are some examples of binary input-output pairs for HBAD:

```
HBAD(00) = 1000 0000
HBAD(01) = 1000 0001
```

```
HBAD(0011) = 1000 0011
HBAD(0000 0000) = 1000 0000
HBAD(0010 0010 1000) = 1010 1000
```

As you can see, HBAD always produces an 8-bit hash, but with only a little bit of analysis you should be able to work out that it is a very simple algorithm that could be expressed in pseudo code as:

```
define HBAD(inp): return( OR(1000 0000, LSB(inp,7) )
```

Where `OR(x,y)` is the two-argument Boolean Or-operator, and `LSB(b,r)`gives the sequence of `r` least significant (right-most) bits of the bit-string `b`. Once you've worked this out, anytime you see a hash from HBAD you can work out a fair amount about what the input was. That's not very secure at all. The clue is in the name.

Now let's look at the outputs HGUD produces for the same set of inputs

```
HGUD(00) = 0101 0111
HGUD(01) = 1011 0101
HGUD(0011) = 1111 0010
HGUD(0000 0000) = 1010 1000
HGUD(0010 0010 1000) = 0000 1000
```

Because HGUD is a strong hashing algorithm, there is no discernible correlation between the input values and the hashes of those inputs produced by HGUD. In fact, HGUD is so good that we are keeping details of the algorithm secret right now.

HGUD and HBAD both produce 8-bit hashes, which mean they each have only 2^8 (=256) possible distinct outputs. Given that they each accept any of a potentially infinite number of possible inputs, it won't take long to find two inputs that give the same output hash (referred to as a *collision*). You can see a collision in the examples given for HBAD, where the two distinct input strings of 00 and 0000 0000 both result in the same hash. For this reason, in reality people are generally interested in hashing algorithms that produce much longer hashes. For example, if a hashing algorithm produces 256-bit outputs then the number of possible distinct outputs rises to 2^{256}, which is roughly 10^{77}, a really very very very big space of possible outputs (for comparison, the number of atoms in our solar system is roughly 10^{57}). Nevertheless, 2^{256} is still a finite number, so a badly-designed 256-bit hashing algorithm could suffer from collisions; but it is big enough for there to be a reasonable hope of creating a 256-bit hasher that has, for practical purposes, a 1:1 mapping from inputs to outputs, for any input.

In principle, any hashing algorithm can be inverted (i.e., starting with the hash, find an input which leads to that hash being generated as output) by brute-force, stepping through every conceivable input of all lengths (i.e., starting with 0, then 1, then 00, then 01, then 10, then 11, then 000, and so on) until the target hash is produced. If you have a guaranteed-collision-free hashing algorithm, and a computer that can process one billion hashes per second, it will take less than one millionth of a second to invert an 8-bit hash (because you will either find an inversion on the 256^{th} attempt, or sooner). In contrast, using the same computer to brute-force invert a 256-bit hash will take roughly 10^{68} seconds, or 3×10^{60} years (for comparison, our planet Earth first formed roughly 4.5×10^{9} years ago).

In 2001 the United States National Security Agency (NSA) designed and published a set ("family") of related very strong hash algorithms called Secure Hash Algorithm 2 (SHA-2). The algorithms in the SHA-2 family are distinguished by their hash-length, and one popular member of SHA-2 is widely referred to as SHA256, because it produces 256-bit hashes. Blockchain uses SHA256 for PoW. In fact, it uses SHA256 called on the output of SHA256 of an input block of data B, i.e. it computes SHA256(SHA256(B)): so the input B is hashed once to one-way encode it, and then that one-way encoding is itself one-way encoded by a second call to SHA256. This nested-call is referred to as *SHA256-squared*, and written SHA256^2.

The PoW step in Bitcoin involves packaging up data concerning some (small) number of as-yet-unconfirmed transactions into a data "block"; appending an arbitrary 32-bit number (i.e., in decimal, a number between 0 and 4,294,967,296) referred to as a "nonce", to the block; and then calling SHA256^2 on the combined block-and-nonce. If the resultant 256-bit hash of the block-and-nonce has some number Z of consecutive zeros in its most significant (leftmost) string of bits, and Z is equal to or greater than some difficulty-number D, then the nonce is referred to as *golden*, and the block of data is considered confirmed and added to the blockchain. The Bitcoin protocol periodically increases the value of D to keep the rate of block-confirmations approximately constant: initially D=32 was used, but after a succession of increments by early 2019 the value of D had reached 74.

So, to summarise, PoW in Bitcoin works by requiring the Bitcoin miner to use brute-force trial-and-error search to find a nonce-value, some 32-bit number. When the nonce is appended to the block, and the two are subjected to SHA256^2, if the resulting 256-bit hash has at least D leading zeros then the nonce is golden, and once the miner has found a golden nonce she has proved that she's done enough work. Of course, extremely rarely a miner will just happen to get lucky and the very first nonce that they try will turn out to be golden, so they don't have to do much actual work to get the PoW done; but the combinatorics tell us that events like that will be very rare indeed.

Hopefully it will be obvious by now that Bitcoin PoW is embarrassingly parallelizable: you could use a single VM to first try SHA256^2 with `nonce=0`, then `nonce=1`, then `nonce=00`, and so on continuing until the hash that comes back has at least D leading zeros in it. Or in principle you could use 200 VMs in parallel, so that you first evaluate nonces `0` to `011000111` (i.e., 0 to 199 in decimal) all being tested at the same time, then nonces `011001000` to `110001111` (i.e., 200 to 399 in decimal), and so on: all being well, with this method on average you'll find a golden nonce 200 times faster than if you were using a single machine.

There was a time when real-life professional Bitcoin miners used to employ this kind of machine-scale parallelism, but a technology arms race among miners led to the development of mining algorithms that run on multicore chips such as GPUs, and then to the development of special-purpose silicon chips (called *ASICS*: Application-Specific Integrated Circuits) designed specifically to mine bitcoins at much faster rates. No current miner would be likely to use a commercial cloud provider, but this context for the coursework does serve to familiarise you with what in the past would have been a realistic application of elastic horizontal scaling in the cloud.