# Trabalho I de Computação Concorrente: Simulação de Mineração de Bitcoin

Gabriel da Fonseca Ottoboni Pinho - DRE 119043838
Rodrigo Delpreti de Siqueira - DRE 119022353

25/04/2021

# Sumário

# 1 Descrição do problema

## 1.1 Sobre a Bitcoin

Bitcoin é uma criptomoeda criada por Satoshi Nakamoto em 2009, tendo como objetivo ser uma moeda digital descentralizada. Uma parte essencial da Bitcoin é uma tecnologia chamada *Blockchain*, que consiste em armazenar todas as transações que já ocorreram em blocos interligados. Cada transação é criptograficamente assinada por seu emissor e então enviada ao resto da rede, que verifica a assinatura e a existência dos fundos.

Um ponto importante é que cada transação é verificada por todos, de modo que nenhum integrante precisa confiar em nenhum outro, cada um sendo capaz de verificar a verdade independentemente. Os integrantes da rede que coletam as transações e as colocam em blocos são os mineradores e o sistema de consenso que determina qual minerador terá o direito de criar o bloco se chama *Proof of Work* (PoW).

A ideia da PoW é que o minerador que conseguir a resposta de um desafio primeiro tem o direito de criar o bloco. Esse desafio consiste em achar um número *nonce*, tal que o *hash* SHA-256 do *header* do bloco (que contém a *nonce*) seja menor que um número chamado de dificuldade. Pelo fato do SHA-256 ser uma função *hash* criptográfica, a única forma de achar uma *nonce* correta é testando valores. Em outras palavas, não há um método fácil de achar um $x$ tal que $\text{SHA256}(x) = y$. Por outro lado, tendo um $x$, é fácil verificar se $\text{SHA256}(x) = y$. Essa propriedade é importante, pois, dessa forma, todos os integrantes da rede podem verificar facilmente se a *nonce* encontrada é uma solução válida de fato.

## 1.2 Sobre o SHA-256

O SHA-256 é uma função *hash* criptográfica, que gera uma sequência de 256 bits pseudo-aleatória para uma entrada qualquer. O cálculo dessa função é computacionalmente custoso, e é o *work* na PoW da Bitcoin. O cálculo da função consiste em 3 passos principais:

1. Pré-processamento
2. Criação do array de mensagens
3. Compressão

Durante o pré-processamento, a entrada é dividida em pedaços chamados *chunks* de 512 bits cada. Depois disso, para cada *chunk*, um array de 64 elementos de 4 bytes é preenchido com base no conteúdo do *chunk* atual. Por fim, 8 variáveis são inicializadas com valores pré-determinados que serão modificados em cada *round* do loop de compressão. O valor final do *hash* será a concatenação das 8 variáveis.

# 2 Projeto da solução concorrente

O objetivo do nosso trabalho é implementar o SHA-256 e usá-lo para simular a mineração da Bitcoin. Em outras palavras, temos que receber como entrada um número $d$ representando a dificuldade e um número $n$ de threads a serem utilizadas. A saída será uma *nonce* que resolveria um bloco com essa dificuldade e o tempo levado para calculá-la. A dificuldade será o número de vezes seguidas que o caractere `'b'` aparece no início da representação hexadecimal do *hash* calculado. Como foi explicado na sessão anterior, não é exatamente assim que a dificuldade é definida na implementação da Bitcoin, mas foi decidido simplificar essa etapa.

A simulação funcionará da seguinte forma:

1. 76 bytes aleatórios são gerados. Esses bytes representarão todos os campos do *header* do bloco, exceto a *nonce*.

2. Cada thread calculará o *hash* da concatenação desses mesmos 76 bytes com mais 4 bytes, a *nonce*. Temos assim um total de 80 bytes, que é o tamanho real do *header* de um bloco de Bitcoin.

3. Se SHA256(*header*) começar com pelo menos $d$ bytes `0x00` seguidos, conseguimos achar uma *nonce* correta, fim. Senão, repetimos as etapas 2 e 3 com uma *nonce* diferente.

É importante notar que cada tentativa é completamente independente da outra, ou seja, podemos facilmente acrescentar mais threads, com cada uma testando diferentes valores para a *nonce* até que a resposta correta seja encontrada. A estratégia utilizada foi testar valores consecutivos para a *nonce* de modo que esses números são distribuídos entre as $n$ threads de $n$ em $n$. Quando alguma das threads achar a resposta, uma variável global *flag* é setada e as outras threads terminam. A thread que encontrou a resposta retorna seu valor para a thread principal, que imprime os resultados na tela.

## 2.1 Do repositório

O repositório contém um grupo de arquivos, cuja função está brevemente descrita aqui para fins de organização. O algoritmo que calcula propriamente o hash sha-256 está contido no arquivo sha256.c, como uma função. Esta função recebe um ponteiro que aponte para o header do bloco, o tamanho do header e um ponteiro usado para retornar o resultado como string. Ela deve ser acessada pelas outras partes do programa utilizando o arquivo sha256.h disponibilizado.

O programa propriamente dito, para cumprir os objetivos acima citados, está no arquivo threadmine.c. Ele recebe como parâmetros a dificuldade do algoritmo e o número de threads que serão utilizadas. Assim, chamamos esse programa múltiplas vezes em *script.sh* para gerarmos os resultados necessários para análise posterior. Os valores utilizados na análise estão todos contidos no arquivo de texto *saida.txt*.

Por fim, os casos de teste estão separados no arquivo *testsha.c*. Há uma makefile disponibilizada, de modo que para compilar o programa basta clonar o repositório e utilizar o comando `make`. Para executar, basta chamar threadmine <dificuldade> <numero de threads>.

# 3 Casos de teste

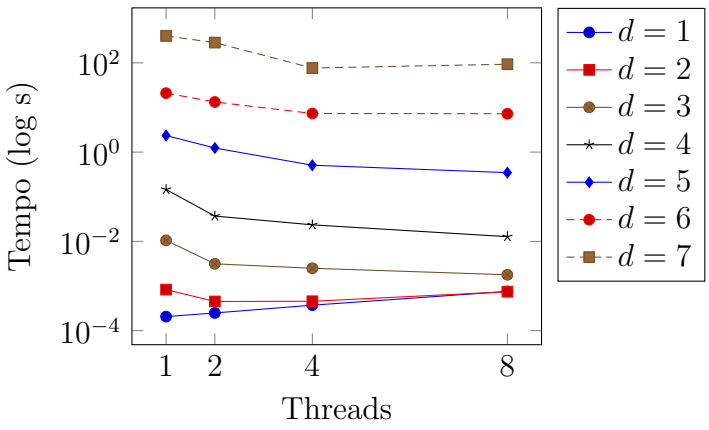
A principal função do programa que precisa ser testada é `sha256`, que recebe uma sequência de bytes e retorna uma string com o valor hexadecimal do *hash* SHA-256 da entrada. Cada teste chama essa função com valores de entrada pré-determinados e compara o *hash* retornado com um valor correto pré-calculado.

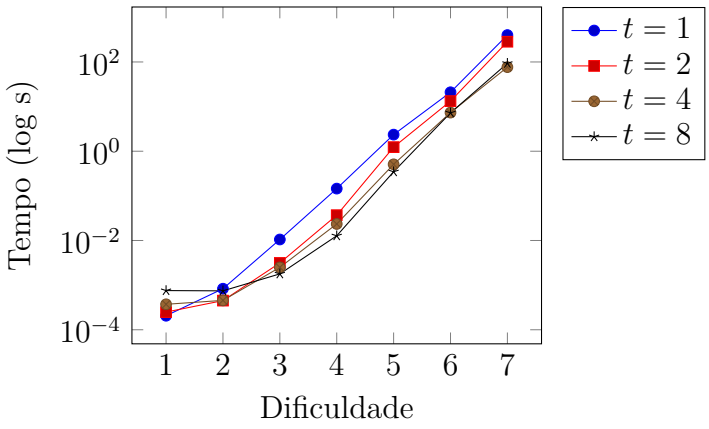
# 4 Avaliação de desempenho

Realizamos 10 execuções para cada configuração de dificuldade e thread. Devido à natureza do problema, iremos utilizar para fins comparativos a média de tempo dessas 10 execuções. Essa quantidade pode não gerar ainda grande confiabilidade estatística do resultado, porém acreditamos ser suficiente para a devida análise e escopo deste trabalho. Seria ingênuo testar execuções individuais, pois a aleatoriedade dos valores pode levar execuções individuais a apresentarem tempos muito díspares.

A tabela abaixo apresenta o tempo médio das 10 execuções, para 7 dificuldades em ordem crescente, com 1, 2, 4 e 8 threads.

| Threads | Dificuldade | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 0,000206 | 0,000831 | 0,010532 | 0,145358 | 2,354084 | 20,96 | 400,6 |
| 2 | 0,000249 | 0,000450 | 0,003139 | 0,036904 | 1,235175 | 13,27 | 284,26 |
| 4 | 0,000372 | 0,000455 | 0,002497 | 0,023467 | 0,508354 | 7,34 | 76,56 |
| 8 | 0,000755 | 0,000744 | 0,001785 | 0,012743 | 0,346573 | 7,24 | 93,16 |






Podemos observar diretamente que a utilização de threads não foi vantajosa para as dificuldades 1 e 2. O tempo gasto com overhead de criação das threads foi superior ao ganho de velocidade.

Outro resultado muito interessante do experimento é que o tempo se torna mais consistente com um número maior de threads. Quanto mais valores são testados para um mesmo intervalo de tempo, maiores as chances de obter tempos similares entre as execuções. Olharemos para a variança dos resultados em dificuldade 5 como exemplo:

| | Threads | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| 1 | 1,81866 | 2,25835 | 0,12451 | 0,84497 |
| 2 | 0,7952 | 2,07361 | 0,19844 | 0,14135 |
| 3 | 1,36829 | 0,12525 | 0,24746 | 0,34731 |
| 4 | 0,34141 | 0,80404 | 1,90943 | 0,36996 |
| 5 | 0,64075 | 4,27544 | 0,37336 | 0,03728 |
| 6 | 0,2022 | 0,17607 | 0,88296 | 0,10144 |
| 7 | 6,07897 | 0,75382 | 0,50713 | 0,56717 |
| 8 | 4,3207 | 0,63343 | 0,31499 | 0,4524 |
| 9 | 5,05224 | 1,1545 | 0,39751 | 0,10372 |
| 10 | 2,92242 | 0,09724 | 0,12775 | 0,50013 |
| Média | 2,354084 | 1,235175 | 0,508354 | 0,346573 |
| Variança | 4,065096858 | 1,54277901 | 0,262755251 | 0,058923561 |

Na tabela acima, vemos que a redução na variança é mais acentuada que a redução do tempo médio de execução.

# 5 Discussão

Apesar do problema possuir diversas variáveis de cunho aleatório, podemos estimar razoavelmente o tempo de execução a partir da probabilidade de se encontrar um *hash* com determinada dificuldade. Conforme especificado no código, para que a execução termine adequadamente, os $d$ primeiros caracteres do *hash* gerado como resultado da `sha256` devem ser iguais a `'b'`, onde $d$ é o valor da dificuldade passado como parâmetro na chamada de `threadmine`. Existem 16 caracteres possíveis para cada posição, pois o algoritmo retorna como string os valores em hexadecimal. Supondo que a função de *hash* apresente uma distribuição uniforme de valores, a probabilidade de acerto será de $1/16^n$.

Este resultado nos indica que o tempo médio deverá aumentar exponencialmente com a dificuldade. Essa estimativa foi verificada na prática, como podemos observar no gráfico Tempo x Dificuldade (note que o tempo está em escala logarítmica). Por tal razão, os testes foram realizados apenas até a dificuldade 7: para dificulade 8, o tempo estimado de execução para uma única thread seria superior a uma hora.