Criptografia RSA concorrente

Rafaela de Carvalho Machado Pinheiro - 121038166

Gabriel Vieira do Amaral - 121069963

Relatório Parcial

Programação Concorrente (ICP-361) - 2022/2

1

1. Descrição do problema

Dentre as opções de criptografia, temos a RSA, que por mais que tenha caído em desuso por ser potencialmente "quebrável", é uma maneira interessante de estudar a Teoria dos Números. Para funcionar, o RSA deve ter os seguintes dados:

- Um texto para ser encriptado
- Dois números primos grandes, p e q
- Um n tal que $n = p \cdot q$
- Um ϕ tal que $\phi = (p - 1) \cdot (n - 1)$

Como funciona?

O programa vai receber o texto e gerar dois números primos grandes, com isso, também vão ser gerados os expoentes para a encriptação e a decriptação.

Após receber o texto, o programa terá que converter cada caractere para um valor numérico correspondente, para isso uma tabela de correspondência que deverá ser criada. Isso é necessário pois todas as demais operações são cálculos com valores numéricos, não com letras.

A encriptação é basicamente fazer a seguinte operação:

$$mensagemEncriptada = bloco^{e} (mod\ n)$$

E a decriptação se dá como:

$$mensagemDescriptada = bloco^{d} (mod\ n)$$

Após a mensagem ser desencriptada, é necessário fazer o caminho inverso e reverter cada número para uma letra.

Ambas as operações são atômicas, por isso podemos explorar uma solução concorrente para o problema. Diante de grandes textos, nota-se uma lentidão nessas operações, e a nossa principal motivação é tentar diminuir o máximo de tempo distribuindo a operação de encriptação e decriptação.

2. Projeto da solução concorrente

Tanto as operações de encriptação quanto a de decriptação são completamente atômicas, i.e., o cálculo de determinado bloco de texto independe de qualquer outro bloco, posterior ou anterior. Por isso, podemos calcular as operações de potenciação de maneira concorrente.

Por exemplo, dentre um número N de threads, cada uma ficaria responsável por encriptar/decriptar somente a sua posição, não precisando do valor das outras. A nossa solução também balanceará a carga de trabalho entre as threads de maneira que nenhuma trabalhe muito mais que a outra. Isso será feito da mesma forma que fizemos no exercício de incrementar elementos de um vetor: cada thread é responsável por manipular (para encriptação e decriptação)

o bloco da posição i,
o bloco da posição $i + quantidadeThreads$,
o bloco em $i + quantidadeThreads + quantidadeThreads$,

e assim por diante. Para garantir que os blocos serão mantidos na ordem em que foram recebidos (caso contrário, o texto retornado não será igual ao original), cada chamada da função para decriptar recebe o bloco encriptado e seu "identificador" no conjunto de blocos de entrada, ou seja, em qual posição no vetor de blocos ele estava. Ao final da função, o bloco decriptado será armazenado em um vetor global criado para guardar cada bloco já processado em uma posição correspondente à sua posição original. Dessa maneira, mesmo que a segunda thread decripte o 2º bloco antes que a primeira thread consiga decriptar o 1º bloco, o bloco 2 será colocado na posição $blocosDecriptados[1]$ e o bloco 1, na posição $blocosDecriptados[0]$, preservando a ordem original.

3. Casos de teste de corretude e desempenho

O programa pode realizar a encriptação e a decriptação. Para a primeira operação, as entradas para teste são textos, de tamanhos variados, i.e., com com quantidades diferentes de palavras. Analogamente, para a decriptação o programa recebe os blocos de texto encriptados e a chave pública. A quantidade e tamanho dos blocos estão diretamente relacionados ao tamanho do texto original que foi encriptado, então essa entrada para a decriptação também varia em tamanho.

Para avaliar a corretude da encriptação, basta inserirmos as mesmas entradas no código completamente sequencial (com as funções de encriptar e decriptar de maneira sequencial) e na sua versão concorrente. Ou seja, o código sequencial deve

receber o mesmo texto, o módulo n e o expoente e - gerados pela versão concorrente e retornados para que sejam usados no teste. Os blocos encriptados gerados pelas duas versões devem ser iguais, assim saberemos que a versão concorrente está correta. No entanto, para testar a decriptação, podemos utilizar somente o programa concorrente. Isso porque basta que este retorne o texto original para garantirmos sua corretude, então só precisamos informar os blocos encriptados que foram retornados da etapa de encriptação.

Para testar o ganho de desempenho, basta inserirmos as mesmas entradas no código completamente sequencial (com as funções de encriptar e decriptar de maneira sequencial) e na sua versão concorrente e avaliarmos a diferença entre os tempos de execução. No caso da encriptação, informamos só o texto original e na decriptação, os blocos já encriptados, sempre variando o tamanho do texto de entrada. Considerando que a versão sequencial está correta, pois já passou pelos testes de corretude, os retornos das chamadas para encriptação e decriptação devem ser iguais àqueles da versão sequencial.

Tanto para os testes de corretude, quanto para a avaliação de desempenho, para cada tamanho de texto, devemos variar o número de threads em cada chamada. Por exemplo, para um texto de 100 caracteres, vamos testar com 1, 2, 3, ..., 8 threads; para um texto de 1 milhão de caracteres, testaremos também com 1 a 8 threads, e o mesmo deverá acontecer para outros tamanhos de textos.

4. Referências bibliográficas

1. Coutinho, S. C. Números inteiros e criptografia RSA. IMPA, 2005.
2. Pacheco, Peter. An Introduction to Parallel Programming. Elsevier Science, 2011.