RSA concorrente

Rafaela de Carvalho Machado Pinheiro – 121038166

Gabriel Vieira do Amaral - 121069963

Relatório Final

Programação Concorrente (ICP-361) — 2022/2

1

Link para o repositório: rafaela-pinheiro/RSAconcorrente (github.com)

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?

Nosso programa para encriptação em Python recebe o nome do arquivo texto e retorna o arquivo com os blocos encriptados.

Esse programa vai 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 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 decriptada 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.

A execução principal do algoritmo concorrente de decriptação RSA se dá pela seguinte sequência:

```
for (i = nthreads; i < qntblocos; i += nthreads) {
      r = decriptar(bloco[i], n, d);
      bloco[i] = r;
}
```

Onde, temos um vetor de strings, chamado de *bloco*, que contém os blocos a serem decriptados. A quantidade de blocos é uma variável global, "facilitando a vida" das threads de decriptação. A principal diferença entre a solução sequencial e a concorrente, portanto, é que enquanto a função sequencial de decriptação faz todos os blocos de uma vez, a solução concorrente divide esse trabalho de maneira equilibrada entre as threads.

2. Projeto e implementação da solução concorrente

Inicialmente, pensamos em fazer todo o processo que envolve a Criptografia RSA, i.e., tanto a encriptação quanto a decriptação. Porém, seguindo uma sugestão da professora que ao nosso ver fez muito sentido, focamos em decriptar primeiramente e deixar completamente funcional e fazer as outras coisas se sobrar tempo.

A operação de decriptação é completamente atômica, ou seja, o cálculo de um determinado bloco de texto independe de qualquer outro bloco, posterior ou interior. Por isso, fez muito sentido paralelizar essa função.

O programa recebe um arquivo txt com os blocos já encriptados, junto com os o n (módulo público) e o d (chave privada). Os blocos são separados em um vetor de strings, e, considerando um número N de Threads, cada Thread fica responsável pelos valores pertencentes ao seu ciclo:

```
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 sejam mantidos na ordem que foram recebidos, consecutivamente garantindo a corretude, cada chamada da tarefa concorrente recebe seu "identificador", e trabalha a partir desse valor, isso é, iteram sob o array de strings começando do seu identificador e saltando "quantidade de threads".

Por fim, após a decriptação, o valor do bloco encriptado é substituído pelo bloco já decriptado. Por mais que compartilhem do mesmo espaço da memória (o vetor de strings é global, por conta da facilidade) e que mudanças no conjunto de dados sejam feitas, não é necessária nenhuma proteção contra condições de corrida, pois as threads apenas acessam as posições que tem direito, e não necessitam, muito menos utilizam, informações das outras posições.

O nosso laço principal pode ser compreendido da seguinte forma:

```
para(i de id, até número_de_posições, passo nthreads)
      //decripta e salva no array global
```

Funções presentes no código:

```
void decriptarGMP (char *bloco, char *n, char *d, mpz_t *r)
```

A decriptarGMP() é uma função que decripta um determinado bloco de texto, utilizando a biblioteca do GMP. Ela recebe o bloco, a modulo público n, a chave privada d. A Função GMP faz o trabalho sujo aqui, nos permitindo fazer potenciação de grandes números e ajudando na aritmética modular.

Criamos as variáveis da biblioteca e as inicializamos, assim podemos usar o método powm() para os cálculos de potência modular.

```
void *decriptarThread (void *arg)
```

A tarefa principal das threads. A partir dos argumentos, ela consegue manipular a variável global com os blocos. Cada thread executa essa tarefa para poder decriptar os blocos de maneira concorrente.

```
char codigoParaSimbolo (int codigo)
```

A função recebe um número que equivale a algum caractere na nossa tabela, e retorna esse caractere.

```
char *lerArquivo (char *arquivo)
```

A função recebe o nome do arquivo e abre, e retorna uma string com todo o conteúdo do arquivo.

```
void escreverArquivo ()
```

A função escreve os arquivos de output

```
int main (int argc, char *argv[])
```

O principal fluxo de código da nossa solução. O nome do arquivo e a quantidade de threads pela linha de comando. Comanda a abertura do arquivo,

3. Testes de corretude

Foram usados 9 arquivos do tipo txt para os testes, divididos entre 3 categorias de tamanho: pequeno, médio e grande.

Nome	Tipo	Tamanho
pequeno1.txt	Pequeno	57 KB
pequeno2.txt	Pequeno	136 KB
pequeno3.txt	Pequeno	227 KB
medio1.txt	Médio	338 KB
medio2.txt	Médio	411 KB
medio3.txt	Médio	541 KB
grande1.txt	Grande	721 KB
grande2.txt	Grande	880 KB
grande3.txt	Grande	986 KB

Cada arquivo foi testado 3 vezes com quantidades diferentes de threads, ou seja, o arquivo pequeno1.txt foi testado com 3 vezes com 1, 3 vezes com 2 threads, e assim por diante, até ser testado 3 vezes com 16 threads. Após o programa ser rodado, é comparado com o arquivo de texto original, a partir do comando *diff* para garantir que o texto é o mesmo.

No repositório do projeto, no GitHub, é possível ter acesso a planilha de testes (*testes.xlsx*), com os valores exatos em nossas máquinas.

4. Avaliação de desempenho

A avaliação de desempenho foi através de tomadas de tempo programadas no código, tanto sequencial quanto concorrente. No primeiro momento, anotamos o tempo médio (após executar 3 vezes) de cada um dos arquivos com 1, 2, 4, 8 e 16 threads e, após isso, anotamos o tempo médio sequencial para cada um dos mesmos casos de teste.

As máquinas utilizadas para as avaliações são as nossas duas máquinas pessoas. Um Ryzen 5 3600 com um Windows 10 (rodando Linux via wsl) e 16gb ram e um Intel I7 1165g7 Windows 11 (rodando Linux via wsl) também com 16gb de ram. Tentamos executar nos laboratórios do IC, para fins comparativos, mas não demos sequência.

A aceleração foi calculada da seguinte maneira:

$$\frac{T_s}{T_c}$$

onde T_s é o tempo médio sequencial e T_c é o tempo médio concorrente.

Esses tempos também incluem os tempos de entrada (de leitura do arquivo dos blocos encriptados) e de saída (escrita no arquivo textoSaida.txt). Como nos dois programas, a manipulação dos arquivos é sequencial, podemos esses tempos no tempo médio total para os cálculos.

Pode-se perceber que houve aceleração, ainda que pequena, na maioria dos casos. Depois dos testes e das tomadas de tempo, notamos que a etapa que demanda mais tempo é a de entrada, isto é, a leitura dos arquivos. Como esta é sequencial nos dois programas, o ganho de desempenho no geral não foi maior devido a ela.

Resumo dos resultados obtidos:

Tipo Arq.	Tamanho (KB)	Média Tempo Exec.	Média Aceleração
Pequeno	140	2,140250778	1,075562916
Médio	430	25,12908867	1,088579461
Grande	862	146,1656774	1,082339592

Para obter esses resultados, usamos 4 threads e executamos o programa na máquina com o processador Intel i7. A planilha completa e com detalhes de cada teste executado está no nosso repositório no GitHub.

5. Discussão

O ganho de desempenho alcançado não foi tão grande quanto esperávamos. Em um primeiro momento, a solução concorrente parecia uma saída óbvia para diminuir o tempo de processamento, pois as etapas da decriptação são naturalmente atômicas. No entanto, até a etapa de testes, não percebemos que a parte que tomava mais tempo era a leitura de arquivos, devido à grande quantidade de blocos encriptados que armazenavam.

Uma possível melhoria para o programa seria estudar a paralelização da leitura dos dados, uma ideia que não tivemos enquanto estávamos desenvolvendo o trabalho, somente na fase de tomada de tempo.

Criar a versão concorrente do RSA foi interessante porque aplicamos os conceitos vistos em aula a um outro problema e do início ao fim, desde entender essa criptografia até quais erros poderiam ocorrer ao longo da execução do programa. Também aprendemos a

utilizar outra biblioteca (a GMP), que talvez nunca conhecêssemos se não fosse pelo trabalho.

A utilização da biblioteca GMP foi uma das maiores dificuldades encontradas, pois não a conhecíamos e sua documentação não é tão simples de entender. Seu uso foi necessário para o cálculo de potências modulares com números muito grandes, essenciais para a criptografia RSA.

6. Referências bibliográficas

1. Coutinho, S. C. *Números inteiros e criptografia RSA*. IMPA, 2005.
2. Pacheco, Peter. *An Introduction to Parallel Programming*. Elsevier Science, 2011.
3. Pthread Documentation: https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
4. GMP Documentation: https://gmplib.org/