Instituto Federal de Minas Gerais - Campus Avançado Ouro Branco
Curso: Bacharelado em Sistemas de Informação
Disciplina: Arquitetura e Organização de Computadores
Professor: Saulo Henrique Cabral Silva (saulo.cabral@ifmg.edu.br)

Trabalho Prático 2

Programação Paralela – Corrigindo Imagens

Nome: Felipe Leal Vieira RA: 0034372

João Victor Dutra Martins Silva RA: 0076873

1.Introdução:

Neste trabalho prático, foi utilizado técnicas avançadas de processamento de imagens para abordar um problema comum enfrentado por muitos ao digitalizar fotos antigas. E foi necessário utiliza a programação paralela para realizar essa tarefa de forma mais rápida, dessa forma, utilizando todos os núcleos e threads de um processador. A programação paralela por sua vez, e o verdadeiro desafio do trabalho proposto, implementá-la em um código, pode trazer grandes ganhos, ajudando ao realizar resolução de problemas complexos além de aumento na velocidade de processamento.

Nas fotos, encontramos um problema inesperado, ao dar zoom nas imagens, vemos as fotos apresentavam pequenos “ruídos” que resultam na perda de qualidade original da foto. Para resolver esse problema, vamos criar um programa que preencha esses “ruídos”, melhorando a qualidade das imagens, e ao dar zoom nas fotografias, esses “ruídos” não serão mais apresentados.

Portanto, o desafio consiste em implementar o algoritmo que preencha esses “ruídos”, e utilizar programação paralela para que a remoção dos “ruídos” de forma mais eficaz.

2. **Documentação:**

2.1. Função Ler Pixels:

Para iniciar, nessa parte, a função irá ler os pixels da imagem e convertê-la para escala de cinza. Em seguida, a função armazena os valores de intensidade de cinza dos pixels em uma matriz bidimensional e retorna essa matriz. Como vemos a função retorna uma matriz, essa matriz e onde cada elemento representa a intensidade de cinza de um pixel da imagem. A matriz tem as dimensões [largura][altura], correspondentes à largura e altura da imagem, respectivamente. Se ocorrer um erro, será retornado o valor nulo (“NULL”).

Foi utilizado a classe “ImageIO” para ler a imagem do arquivo especificado pelo caminho fornecido. A imagem é carregada em um objeto “BufferedImage”. Da mesma para obter as dimensões da imagem foi utilizado A largura e a altura da imagem são obtidas através dos métodos “getWidth()” e “getHeight()” do objeto “BufferedImage”.

Para a criação da matriz de pixels, uma matriz bidimensional de inteiros é criada com dimensões [largura][altura]. Esta matriz armazenará os valores em escala de cinza dos pixels da imagem.

A função irá percorrer cada pixel, utilizando dois laços aninhados, um para a largura e outro para a altura da imagem.Para cada pixel, a cor é extraída usando o método “getRGB(i, j)” do “BufferedImage”, e um objeto Color é criado com esse valor da cor. Já os componentes de cor vermelho (vermelho), verde (verde) e azul (azul) são extraídos a partir do objeto Color.

A média dos componentes de cor (vermelho, verde e azul) é calculada para obter o valor em escala de cinza. A fórmula utilizada é (vermelho + verde + azul) / 3. O valor resultante é convertido para um inteiro e armazenado na matriz pixels.

```
11  public class ProcessImageBlackWhite {
12
13      //Funcao para ler os pixels de uma imagem e converter para escala de cinza (Saulo)
14      public static int[][] lerPixels(String caminho) {
15          BufferedImage bufferedImage;
16          try {
17              bufferedImage = ImageIO.read(new File(caminho));
18              int largura = bufferedImage.getWidth();
19              int altura = bufferedImage.getHeight();
20
21              int[][] pixels = new int[largura][altura];
22
23              for (int i = 0; i < largura; i++) {
24                  for (int j = 0; j < altura; j++) {
25
26                      float vermelho = new Color(bufferedImage.getRGB(i, j)).getRed();
27                      float verde = new Color(bufferedImage.getRGB(i, j)).getGreen();
28                      float azul = new Color(bufferedImage.getRGB(i, j)).getBlue();
29
30                      int escalaCinza = (int) (vermelho + verde + azul) / 3;
31
32                      pixels[i][j] = escalaCinza;
33                  }
34              }
35              return pixels;
36          } catch (IOException ex) {
37
38              System.err.println("Erro no caminho indicado pela imagem");
39          }
40          return null;
41      }
```

Foi utilizado dentro do “Catch” o “IOException”, para caso ocorra uma exceção durante a leitura do arquivo, uma mensagem de erro é exibida no console, e a função retorna “null”.

2.2. Gravar Pixels:

Na função “gravarPixels”, ele grava uma matriz de pixels em um arquivo de imagem em escala de cinza, o nome do arquivo é modificado para indicar que a imagem foi alterada. Foi criado variável do tipo “String”, como nome de “caminhoGrava” caminho onde a imagem será salva. O sufixo _modificado será adicionado ao nome do arquivo, mantendo a extensão original. A Variável “pixels” e uma matriz bidimensional representando os valores em escala de cinza dos pixels.

Dentro do funcionamento, foi feito uma modificação no nome do arquivo, Ira sobrescreve o nome do arquivo chamando “original_inicio” com o sufixo _modificado e mantendo a extensão original. No “BufferedImage” com largura e altura iguais à matriz e coloração de fundo cinza escuro com “TYPE_BYTE_GRAY”.

Para preenche os bytes de pixels, um “array” de bytes e preenchido de acordo com as intensidades da matriz. Por fim a imagem atual e salva em um novo arquivo no caminho modificado; retorna uma mensagem de sucesso, caso ocorra algum erro, e enviado uma mensagem de erro.

```
    //Função para gravar a matriz de pixels em um arquivo de imagem (Saulo)
    public static void gravarPixels(String caminhoGravar, int[][] pixels) {
        caminhoGravar = caminhoGravar
                .replace(".png", "_modificado.png")
                .replace(".jpg", "_modificado.jpg");

        int largura = pixels.length;
        int altura = pixels[0].length;

        BufferedImage imagem = new BufferedImage(largura, altura, BufferedImage.TYPE_BYTE_GRAY);

        byte[] bytesPixels = new byte[largura * altura];

        for (int x = 0; x < largura; x++) {
            for (int y = 0; y < altura; y++) {
                bytesPixels[y * largura + x] = (byte) pixels[x][y];
            }
        }

        imagem.getRaster().setDataElements(0, 0, largura, altura, bytesPixels);

        File imageFile = new File(caminhoGravar);
        try {
            ImageIO.write(imagem, "png", imageFile);
            System.out.println("Nova Imagem disponivel em: " + caminhoGravar);
        } catch (IOException e) {

            System.err.println("Erro no caminho indicado pela imagem");
        }
    }
```

2.3. Corrigir Imagem:

A função “corrigirImagem” por sua vez, ira aplica uma correção na imagem representada por uma matriz de pixels em escala de cinza. A correção é realizada utilizando uma máscara de 3x3 para ajustar os pixels pretos e brancos baseando-se em seus vizinhos. O objetivo é suavizar a imagem e corrigir possíveis erros de pixel. Como já foi dito, aplicamos uma máscara de 3x3, porém, para fins de melhor desempenho foi testado uma máscara de 3x3, 6x6 e 24x24, que será abordado melhor, na conclusão do trabalho.

Foi criado outro parâmetro “imgMat” do tipo “int”, sendo uma matriz que representa a imagem em escala de cinza, onde cada valor é um inteiro entre 0 (preto) e 255 (branco).

```
    //Funcao para corrigir a imagem aplicando uma Mascara
    public static int[][] corrigirImagem(int[][] imgMat) {
        int largura = imgMat.length; //Uma nova var para armazenar a largura da matriz de pixels da imagem
        int altura = imgMat[0].length; //Uma nova var para armazenar a altura da matriz de pixels da imagem
        int[][] novaImgMat = new int[largura][altura]; //Cria uma nova matris para armazenar a imagem corrigida e evitar manipular dados na matriz original

        //Copia os valores originais para a nova matriz
        for (int i = 0; i < largura; i++) {
            for (int j = 0; j < altura; j++) {
                novaImgMat[i][j] = imgMat[i][j];
            }
        }

        //Aplica a correção na nova matriz de pixels
        for (int i = 0; i < largura; i++) {//Comeca a percorrer a largura da matriz
            for (int j = 0; j < altura; j++) {//Comeca a percorrer a altura da matriz

                // Verifica se o pixel é preto (0) ou branco (255)
                if (imgMat[i][j] == 0 || imgMat[i][j] == 255) {
                    int somaVizinhos = 0; //Uma nova var para armazenar a soma dos valores (cores) dos pixels vizinhos
                    int contadorVizinhos = 0; //Contador de pixels vizinhos para utilizar nos calculos da media
                    int contadorPretos = 0; //Contador de pixels pretos

                    //Inicializacao da mascara percorrendo os pixels vizinhos em uma matriz 3x3
                    for (int i1 = -1; i1 <= 1; i1++) {
                        for (int j1 = -1; j1 <= 1; j1++) {
                            int x = i + i1; //Posicao x do pixel vizinho
                            int y = j + j1; //Posicao y do pixel vizinho

                            //Verifica se o pixel vizinho esta dentro dos limites da imagem
                            if (x >= 0 && x < largura && y >= 0 && y < altura) {
                                //Ignorando o pixel central para melhorar a assertividade dos calculos
                                if (!(i1 == 0 && j1 == 0)) {
                                    int valorPixelVizinho = imgMat[x][y]; //Pega o valor do pixel vizinho
                                    somaVizinhos += valorPixelVizinho; //Adiciona o valor a soma
                                    contadorVizinhos++; //Incrementa o contador de vizinhos
                                    if (valorPixelVizinho == 0) {
                                        contadorPretos++; //Incrementa o contador de pixels pretos
                                    }
                                }
                            }
                        }
                    }

                    //Se a maioria dos pixels vizinhos for preto, define o pixel central como preto
                    if (contadorPretos > contadorVizinhos / 2) {
                        novaImgMat[i][j] = 0;
                    } else {
                        //Caso contrario, calcula a media das cores dos pixels vizinhos e define a nova cor para o pixel "faltante"
                        if (contadorVizinhos > 0) {
                            novaImgMat[i][j] = somaVizinhos / contadorVizinhos;
                        }}}}}
        return novaImgMat; //Retorna a matriz de pixels corrigida
    }
```

Para o funcionamento do programa, foi criando uma matriz para corrigir a imagem, “novaImgMat”, que possui as mesmas dimensões da matriz original “imgMat”. Em seguida, todos os valores da matriz original são copiados para a nova matriz “novaImgMat”. Dessa forma deixando a matriz original intacta.

Por sua vez, a próxima parte, e é aplicar a correção nos pixels da imagem. A função percorre cada pixel da matriz, verificando se o pixel é preto (0) ou branco (255). Para esses pixels, são definidas variáveis para armazenar a soma dos valores dos pixels vizinhos, o número de vizinhos considerados e a quantidade de pixels vizinhos que são pretos.

Como já foi dito, foi utilizado utiliza uma máscara de 3x3 para corrigir, nesse ponto o programa examina os pixels ao redor do pixel central. Durante essa operação, a função assegura que apenas os pixels vizinhos dentro dos limites da imagem são considerados. O pixel central é ignorado para os cálculos para evitar que o próprio valor influencie a média. Para corrigir

Com isso, cada pixel analisado, a função soma os valores dos pixels vizinhos e conta quantos desses vizinhos são pretos. Se mais da metade dos pixels vizinhos são pretos, o pixel central é definido como preto (0). Caso contrário, a função calcula a média dos valores dos pixels vizinhos e atribui essa média ao pixel central. Finalmente, a função retorna a nova matriz “novaImgMat”, que contém os valores corrigidos da imagem.

2.4. Método main :

Por fim chegamos ao método “Main”, nesse ponto, a entrada para o programa e realiza a leitura, correção e gravação de imagens localizadas em um diretório específico. Ele usa processamento paralelo para melhorar a eficiência, aproveitando múltiplas threads para processar imagens simultaneamente.

O método inicia criando um objeto “File” que representa o diretório onde as imagens estão armazenadas. É necessário ajustar o caminho do diretório de acordo com a localização real dos arquivos na máquina do usuário.

Depois, o método obtém uma lista de arquivos de imagem no diretório especificado, utilizando o método “listFiles()” do objeto File. Em seguida, se o arquivo estiver vazio, no caso “null”, uma mensagem de erro é exibida e o método é encerrado.

Para processar as imagens de forma eficiente, o método configura um pool de threads. Primeiro, ele obtém o número de processadores disponíveis usando o "Runtime.getRuntime()" e "availableProcessors()". Em seguida, cria um pool de threads (ExecutorService) com um número de threads igual ao número de processadores disponíveis, usando Executors.newFixedThreadPool(numberOfThreads).

O método então envia uma tarefa para o pool de threads para cada arquivo de imagem na lista. Cada tarefa realiza as seguintes operações: verifica se o arquivo de imagem existe, lê os pixels da imagem usando a função lerPixels, aplica a correção na imagem com a função corrigirImagem e grava a imagem corrigida utilizando a função gravarPixels.

```
    public static void main(String[] args) {
        //Cria um objeto File com o diretorio onde as imagens estao localizadas (Necessario trocar o local para rodar o codigo em sua maquina)
        File directory = new File("C:\\Users\\usuario\\Desktop\\projeto e arquivos para o problema de imagens\\Imagens\\modificadas");
        //Obtem uma lista de arquivos de imagem no diretorio
        File[] imageFiles = directory.listFiles();

        //Verifica se foram encontrados arquivos
        if (imageFiles == null) {
            System.err.println("Nenhum arquivo encontrado no diretorio.");
            return;
        }

        //Obtem o numero de threads disponiveis
        int numberOfThreads = Runtime.getRuntime().availableProcessors();
        //Cria um pool de threads com o numero de threads encontrados, utilizando a lib java.util.concurrent (Executor e ExecutorService)
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfThreads);

        try {
            //Envia uma tarefa para cada arquivo de imagem no pool de threads
            for (File imgFile : imageFiles) {
                executorService.submit(() -> {
                    if (imgFile.exists()) { // Verifica se o arquivo existe
                        int[][] imgMat = lerPixels(imgFile.getAbsolutePath()); //Le os pixels da imagem usando a funcao do Saulo
                        if (imgMat != null) {
                            imgMat = corrigirImagem(imgMat); //Corrige a imagem aplicando a funcao com a "mascara"
                            gravarPixels(imgFile.getAbsolutePath(), imgMat); //Grava a imagem corrigida usando a funcao do Saulo
                        }
                    } else {
                        System.err.println("Arquivo de imagem nao encontrado: " + imgFile.getAbsolutePath());
                    }
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            executorService.shutdown(); //Encerra o pool de threads inicializado anteriormente (ExecutorService)

            try {
                //Aguarda a conclusao de todas as tarefas ou forca o encerramento se o tempo limite for atingido
                if (!executorService.awaitTermination(60, java.util.concurrent.TimeUnit.SECONDS)) {
                    executorService.shutdownNow();
                }
            } catch (InterruptedException e) {
                executorService.shutdownNow(); //Forca o encerramento em caso de erro
            }
        }
    }
}
//Codigo desenvolvido por Joao Victor Dutra e Felipe Leal
```

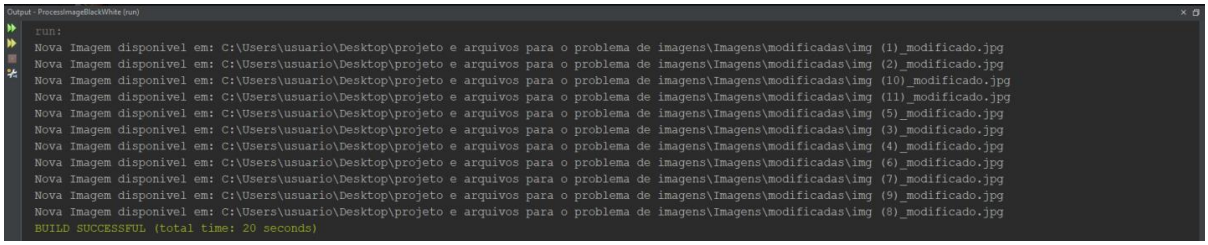
O bloco de código que envia as tarefas ao pool de threads está envolvido em um bloco try-catch para capturar e exibir quaisquer exceções que possam ocorrer durante a execução das tarefas.

Após o envio das tarefas, o método chama shutdown() para iniciar o processo de encerramento do pool de threads, o que rejeita novas tarefas e completa as que já estão em andamento. Utiliza “awaitTermination”(60, java.util.concurrent.TimeUnit.SECONDS) para aguardar a conclusão de todas as tarefas ou forçar o encerramento após 60 segundos, se necessário. Se o tempo limite for atingido, shutdownNow() é chamado para interromper imediatamente todas as tarefas que ainda estão em execução. Em caso de interrupção durante o período de espera, shutdownNow() é novamente chamado para garantir que o pool de threads seja encerrado de forma forçada.

3.Conclusão:

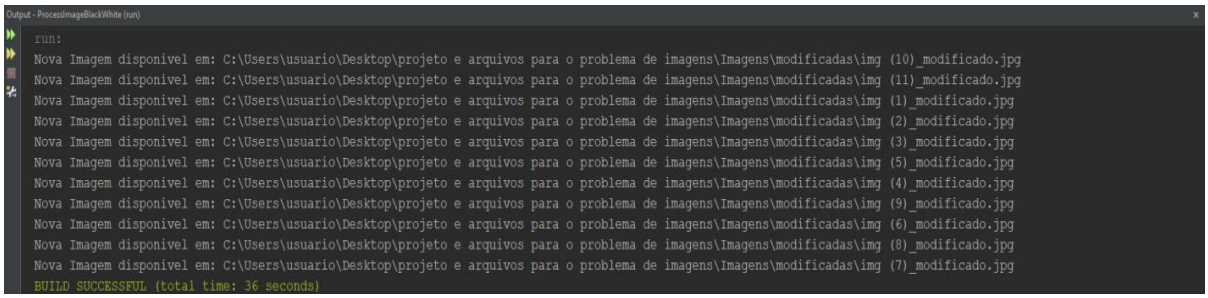
Neste trabalho prático, o objetivo foi restaurar a qualidade das imagens digitalizadas por Seu José, aplicando técnicas de processamento de imagens para remover os ruídos que comprometiam a visualização e interpretação das fotos. Uma parte muito interessante do projeto foi realizar tudo em programação paralela, que por sua vez aumentou o nível de dificuldade ao implementar o código.

Realizamos diversos testes para tentar melhorar o resultado, em questão de tempo de execução, ao realizar os testes em uma máscara de 3x3 tivemos o resultado abaixo:



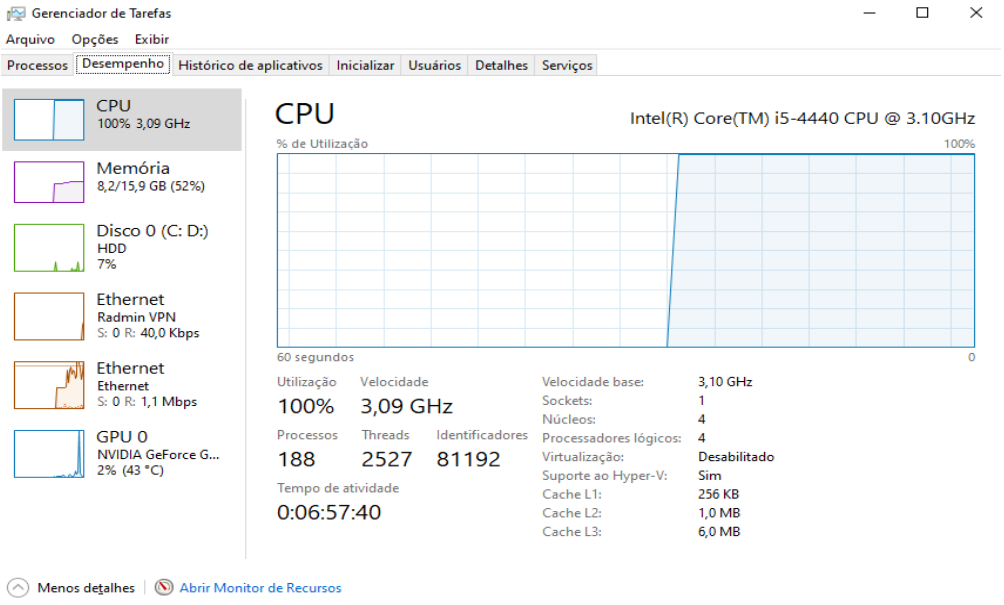
Tendo em vista, o tempo gasto de 20 segundos, ao aumentar o tamanho da máscara, para 6x6, o tempo de execução do programa aumenta.

Como vemos o tempo de execução foi ainda maior, indo para 26 segundos, já com uma máscara 24x24, o tempo de execução ficou aumentou, chegando a 36 segundos. Vemos que aumentando a máscara, faz com que o processamento seja mais demorado. Tendo em vista que os pixels a serem comparados aumentam.



Por outro lado, ao realizar essas operações com máscara maiores, a qualidade da imagem fica ainda melhor, em questão da proporção de pixels a serem comparados serem comparados. Outro ponto que mesmo usando a máscara 3x3, teve um ótimo resultado, entregando o que foi proposto, em um menor tempo de execução. Logo optamos por usar 3x3, como vemos abaixo, o exemplo do antes e depois das imagens.

Como consideração final, vemos que a programação paralela e um recurso de extrema importância para realizar cálculos, entregando um ótimo desempenho, outro ponto e que a programação paralela traz grandes ganhos, ajudando na resolução do problema proposto além do ganho na velocidade de processamento. Como podemos ver logo após o início da execução do programa o CPU vai para 100% e a memória “RAM” vai para 52% de uso.



Outro ponto a ser destacado e o antes e o depois das imagens, como vemos a imagem abaixo, e a imagem disponibilizada para testes (Imagem 1), está com alguns "ruídos", já a outra imagem (Imagem 2), o programa já preencheu os pixels, no caso os "ruídos", assim fazendo com o o programa tenha rodado da forma correta.

Imagem 1

Imagem 2

Programação Paralela – Quebrando Senha

1.Introdução:

Neste trabalho prático, fomos desafiados a quebrar a senha de 5 arquivos ".zip", de forma bruta, isso quer dizer que, termos que tentar quebrar a senha, rodando um algoritmo. Durante o processo, será necessário basear-se na tabela ASCII para testar diversos caracteres possíveis, que incluem letras maiúsculas e minúsculas, números e caracteres especiais. O uso da tabela ASCII permite que cada caractere seja mapeado a um valor inteiro, facilitando a experimentação das possíveis combinações de senhas.

Por sua vez, este trabalho é um exercício técnico que oferece uma oportunidade valiosa para aplicar e aprimorar habilidades de segurança digital e criptografia. Ao enfrentar esse desafio, você ganhará experiência prática em técnicas de quebra de senha e em métodos de desencriptação de arquivos protegidos ao mesmo tempo que utilizamos a programação paralela, para realizar o processo.

Os pontos que podemos ter dificuldades, e principalmente ao tentar quebrar a senha usando a tabela ASCII e utilizar a programação paralela para que o tempo de desencriptarão seja menor, caso seja feita de forma erronia pode ocasionar uma maior demora do processo, ou tento um ganho pouco significativo de tempo, porém isso será melhor abordado mais à frente.

2.1. Início do Programa:

Neste trecho do programa, define as constantes e as estruturas de dados necessárias para a operação de um programa que tenta encontrar as senhas de vários arquivos ZIP utilizando uma abordagem de força bruta. O caminho dos arquivos, o intervalo de caracteres ASCII a serem utilizados para a geração das senhas, bem como listas para armazenar as senhas encontradas e flags de controle, são todos definidos aqui. Essas configurações são fundamentais para que o programa possa iterar sobre os arquivos ZIP e tentar diferentes combinações de senhas de forma organizada e eficiente.

Foi criado uma variável global, do tipo “String” como nome “CAMINHO” que define o caminho absoluto da pasta onde os arquivos ZIP estão localizados. A outra variável global “arquivo_zip” define uma lista com os caminhos completos dos arquivos ZIP que serão testados para encontrar as senhas.

A variável global “String arquivo_final”, define o caminho completo do arquivo ZIP final que será extraído após encontrar as senhas dos arquivos ZIP listados. Já os parâmetros “inicio_ascii” e “fim_ascii” define o intervalo de valores dos caracteres ASCII que serão utilizados para gerar as senhas a serem testadas.

```
13  public class TesteAbrirFilePassword {
14
15      //Caminho absoluto da pasta onde os arquivos estao localizados
16      private static final String CAMINHO = "C:\\Users\\usuario\\Desktop\\senha\\arquivosTP\\";
17
18      //Lista com os caminhos dos arquivos ZIP que serao testados
19      private static final String[] arquivos_zip = {
20          CAMINHO + "doc1.zip",
21          CAMINHO + "doc2.zip",
22          CAMINHO + "doc3.zip",
23          CAMINHO + "doc4.zip",
24      };
25
26      //Caminho do arquivo ZIP final que sera extraido apos encontrar as senhas dos arquivos anteriores
27      private static final String arquivo_final = CAMINHO + "final.zip";
28
29      //Valores dos caracteres ASCII que serao utilizados para gerar as senhas (32 a 127)
30      private static final int inicio_ascii = 32; // 32 e o codigo ASCII de '[SPACE]'
31      private static final int fim_ascii = 127; // 127 e o codigo ASCII de 'DEL'
32
33      //Criacao de uma lista para armazenar as senhas encontradas para os arquivos ZIP
34      private static final List<String> senhas_encontradas = new ArrayList<>();
35
36      //Criacao de uma lista de controladores (flags) indicando se a senha de um arquivo ZIP foi encontrada
37      private static final List<Boolean> senha_encontrada = new ArrayList<>();
38
```

Por fim a “senhas_encontradas” Cria uma lista para armazenar as senhas que foram encontradas para os arquivos ZIP e “senha_encontrada” Cria uma lista de controladores (flags) para indicar se a senha de cada arquivo ZIP foi encontrada. Cada elemento da lista corresponde a um arquivo ZIP na mesma ordem da lista.

2.2. forcaBrutaAgoraVai:

Para iniciar, foi criado uma função “forcaBrutaAgoraVai” implementa uma abordagem de força bruta para gerar e testar todas as possíveis combinações de senhas utilizando caracteres ASCII. A função usa um “ExecutorService” para distribuir o trabalho de tentativa de senhas entre múltiplas threads, melhorando a eficiência e a velocidade do processo. Por outro lado, vemos que ocorre um pequeno problema nessa parte, que será mais bem explicado na conclusão.

A função “forcaBrutaAgoraVai” itera sobre cada arquivo ZIP definido na lista “arquivos_zip”. Para cada arquivo, a variável “arquivoIndex” é definida para manter o índice atual do arquivo na lista.Em seguida, a função gera todas as combinações possíveis de senhas com três caracteres ASCII. Três loops aninhados iteram sobre os valores ASCII de “inicio_ascii” a “fim_ascii”, criando combinações de três caracteres. Para cada combinação de caracteres, uma “string” senha é formada concatenando caractere1, caractere2 e caractere3.Para cada combinação de senha gerada, uma tarefa é submetida ao pool de threads (executor). A tarefa submetida chama a função “tentarSenha”, passando o índice do arquivo (“arquivoIndex”) e a senha gerada (senha).

```
65      //Funcao de forca bruta que gera todas as possiveis combinacoes de senhas com o uso da pool de threads criada anteriormente
66      private static void forcaBrutaAgoraVai(ExecutorService executor) {
67          //Itera sobre cada arquivo ZIP
68          for (int index = 0; index < arquivos_zip.length; index++) {
69              final int arquivoIndex = index;
70
71              //Gera todas as combinacoes de tres caracteres ASCII
72              for (int i = inicio_ascii; i <= fim_ascii; i++) {
73                  for (int j = inicio_ascii; j <= fim_ascii; j++) {
74                      for (int k = inicio_ascii; k <= fim_ascii; k++) {
75                          final char caractere1 = (char) i;
76                          final char caractere2 = (char) j;
77                          final char caractere3 = (char) k;
78
79                          String senha = "" + caractere1 + caractere2 + caractere3;
80
81                          //Submete uma tarefa ao pool de threads para testar a senha no arquivo ZIP
82                          executor.submit(() -> tentarSenha(arquivoIndex, senha));
83                      }
84                  }
85              }
86          }
87      }
88
```

2.3. EncerramentoDaPoolDeThreads:

Foi necessário cria uma função “encerramentoDaPoolDeThreads” é responsável por garantir que todas as tarefas submetidas ao pool de threads (“ExecutorService”) sejam concluídas de forma ordenada antes de encerrar o executor, evitando problemas de concorrência. Com isso função inicia o processo de encerramento do executor chamando “executor.shutdown()”, o que impede que novas tarefas sejam submetidas ao executor. Em seguida, a função tenta aguardar a conclusão de todas as tarefas em andamento por até 1 hora utilizando “executor.awaitTermination(1, TimeUnit.HOURS)”. Se todas as tarefas não forem concluídas dentro desse prazo, a função chama “executor.shutdownNow()” para forçar o encerramento imediato de todas as tarefas em execução. Caso a thread principal for interrompida enquanto aguarda a conclusão das tarefas, a função captura a exceção “InterruptedException” e chama “executor.shutdownNow()” para forçar o encerramento das tarefas.

```
89      //Funcao que aguarda a conclusao de todas as tarefas enviadas para as threads para nao ocorrer problemas de concorrencia
90      private static void encerramentoDaPoolDeThreads(ExecutorService executor) {
91          executor.shutdown();
92          try {
93              //Espera ate 1 hora para todas as tarefas terminarem
94              if (!executor.awaitTermination(1, TimeUnit.HOURS)) {
95                  executor.shutdownNow(); //Forca o encerramento das tarefas caso o tempo limite seja atingido
96              }
97          } catch (InterruptedException e) {
98              executor.shutdownNow();
99          }
100     }
```

2.4. TentarSenha:

Nessa parte da função “tentarSenha” tenta extrair o conteúdo de um arquivo ZIP usando uma senha fornecida. Se a senha estiver correta, ela adiciona essa senha a uma lista de senhas encontradas e marca o arquivo como desbloqueado.

A função “tentarSenha” é utilizada para tentar desbloquear arquivos ZIP usando uma senha fornecida. Se a senha estiver correta, a função atualiza uma lista de senhas encontradas e marca o arquivo como desbloqueado, garantindo que cada senha seja testada apenas uma vez por arquivo. A função também trata exceções de forma a continuar o processo de tentativa de outras senhas, mesmo em caso de falhas.

Mais abaixo na função “tentarSenha”, o programa tenta extrair o conteúdo de um arquivo ZIP usando uma senha fornecida. Inicialmente, ela verifica se a senha já foi encontrada para o arquivo específico; se sim, a função retorna imediatamente. Em seguida, obtém o

caminho do arquivo ZIP a ser testado e cria um objeto “ZipFile” para manipular o arquivo ZIP. Se o arquivo estiver criptografado, a senha fornecida é configurada para o objeto “ZipFile”.

A função então obtém a lista de cabeçalhos de arquivos contidos no arquivo ZIP e tenta extrair cada arquivo usando a senha fornecida. Se a extração for bem-sucedida, a função sincroniza o acesso à lista de senhas encontradas para evitar problemas de concorrência. Se a senha estiver correta e ainda não estiver na lista de senhas encontradas, a senha é adicionada à lista e o arquivo é marcado como desbloqueado. Uma mensagem é impressa no console indicando que a senha foi encontrada para o arquivo ZIP. Se ocorrer uma exceção “ZipException”, a função ignora a exceção e continua tentando outras senhas de forma sucessiva.

```
102     //Funcao que ira testar as senhas nos arquivos (Saulo) porem com algumas pequenas modificacoes comentadas
103     private static void tentarSenha(int arquivoIndex, String senha) {
104         //Verifica se a senha ja foi encontrada para o arquivo especifico
105         if (senha_encontrada.get(arquivoIndex)) {
106             return;
107         }
108 
109         //Pega o caminho do arquivo ZIP a ser testado
110         String arquivoZip = arquivos_zip[arquivoIndex];
111 
112         ZipFile zipFile = new ZipFile(new File(arquivoZip));
113         try {
114             if (zipFile.isEncrypted()) {
115                 zipFile.setPassword(senha.toCharArray());
116             }
117 
118             List<FileHeader> fileHeaderList = zipFile.getFileHeaders();
119             for (FileHeader fileHeader : fileHeaderList) {
120                 zipFile.extractFile(fileHeader, CAMINHO);
121                 synchronized (senhas_encontradas) {
122                     //Se a senha estiver certa adiciona na lista de senhas encontradas e marca como encontrada
123                     if (!senhas_encontradas.contains(senha)) {
124                         senhas_encontradas.add(senha);
125                         senha_encontrada.set(arquivoIndex, true);
126                         System.out.println("Senha encontrada para " + arquivoZip + ": " + senha);
127                     }
128                 }
129                 return; //Sai do metodo apos encontrar a senha
130             }
131         } catch (ZipException ex) {
132         }
133     }
```

2.5. extrairArquivoFinal:

Por fim, foi criado a função “extrairArquivoFinal” é responsável por extrair o arquivo final utilizando as senhas encontradas. Primeiro, ela cria um objeto “ZipFile” para manipular o arquivo ZIP final. Em seguida, verifica se todas as senhas dos arquivos ZIP individuais foram encontradas, comparando o tamanho da lista senhas_encontradas com o número de arquivos ZIP.

Caso todas as senhas tenham sido encontradas, a função concatena todas essas senhas em uma única “String” chamada de “senhaFinal”. Utilizando essa senha concatenada, a função verifica se o arquivo ZIP final está criptografado e, se sim, configura a senha no objeto “ZipFile”. Depois, obtém a lista de cabeçalhos de arquivos contidos no arquivo ZIP final e tenta extrair cada arquivo usando a senha concatenada.

Se a extração for bem-sucedida, a função imprime uma mensagem no console indicando que o arquivo final foi extraído com a senha combinada e retorna, encerrando o método. Caso ocorrer uma exceção, foi utilizado “ZipException” durante a extração, a função imprime uma mensagem de erro no console indicando que houve um problema ao extrair o arquivo final com a senha combinada. Caso nem todas as senhas tenham sido encontradas, a função imprime uma mensagem no console informando que nem todas as senhas foram encontradas.

```
135         //Extrai o arquivo final usando as senhas encontradas
136         private static void extrairArquivoFinal() {
137             ZipFile zipFile = new ZipFile(new File(arquivo_final));
138 
139             //Verifica se todas as senhas foram encontradas
140             if (senhas_encontradas.size() == arquivos_zip.length) {
141                 //Concatena todas as senhas em uma unica string
142                 StringBuilder senhaFinalBuilder = new StringBuilder();
143                 for (String senha : senhas_encontradas) {
144                     senhaFinalBuilder.append(senha);
145                 }
146                 String senhaFinal = senhaFinalBuilder.toString();
147 
148                 try {
149                     if (zipFile.isEncrypted()) {
150                         zipFile.setPassword(senhaFinal.toCharArray());
151                     }
152 
153                     List<FileHeader> fileHeaderList = zipFile.getFileHeaders();
154                     for (FileHeader fileHeader : fileHeaderList) {
155                         zipFile.extractFile(fileHeader, CAMINHO);
156                         System.out.println("Arquivo final.zip extraido com a senha: " + senhaFinal);
157                         return; // Sai do metodo apos extrair o arquivo final
158                     }
159                 } catch (ZipException ex) {
160                     System.out.println("Erro ao extrair o arquivo final com a senha combinada: " + senhaFinal);
161                 }
162             } else {
163                 System.out.println("Nem todas as senhas foram encontradas.");
164             }
165         }
166     }
167     //Codigo desenvolvido por Joao Victor Dutra e Felipe Leal
168 
```

3.Conclusão:

Neste trabalho prático, fomos desafiados a quebrar a senha de 5 arquivos ".zip" utilizando um algoritmo de força bruta, explorando a tabela ASCII para testar diversas combinações de caracteres possíveis. Este exercício técnico nos proporcionou uma valiosa oportunidade para aplicar e aprimorar nossas habilidades em segurança digital e criptografia, além de nos familiarizarmos com técnicas de quebra de senha e desencriptação de arquivos protegidos.

Ao longo do processo, enfrentamos a complexidade de testar combinações de letras maiúsculas e minúsculas, números e caracteres especiais, todos mapeados a valores inteiros através da tabela ASCII. Para tornar o processo mais eficiente, utilizamos programação paralela, dividindo as tarefas entre múltiplas threads para reduzir o tempo necessário para a desencriptação.

Um ponto a se destacar, e que os threads continuam executando por um tempo, mesmo após encontrar a senha correta, no nosso programa conseguimos contornar esse problema, porém ainda ocorria esse problema por um curto período de tempo, porque os threads já estavam executados no momento que a senha foi encontrada.

3.1. Teste:

Foi feito um teste, para uma melhor compreensão, e para ver se o tempo para quebrar a senha fosse menor. Após todos os arquivos serem desencriptados, a CPU foi 100% utilizado novamente, em uma máquina, com as configurações abaixo:

Processador: I5 – 9400

Núcleos: 6

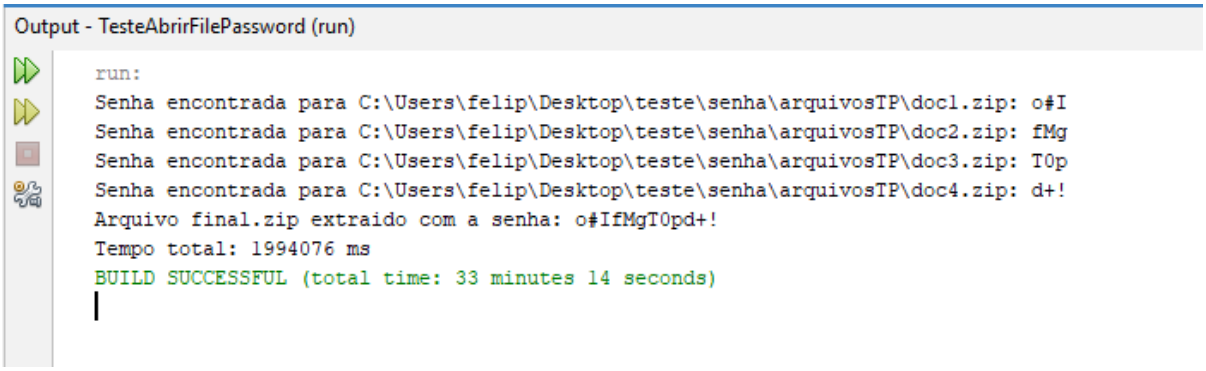
Processadores lógicos: 6

Memória Ram: 16 GB

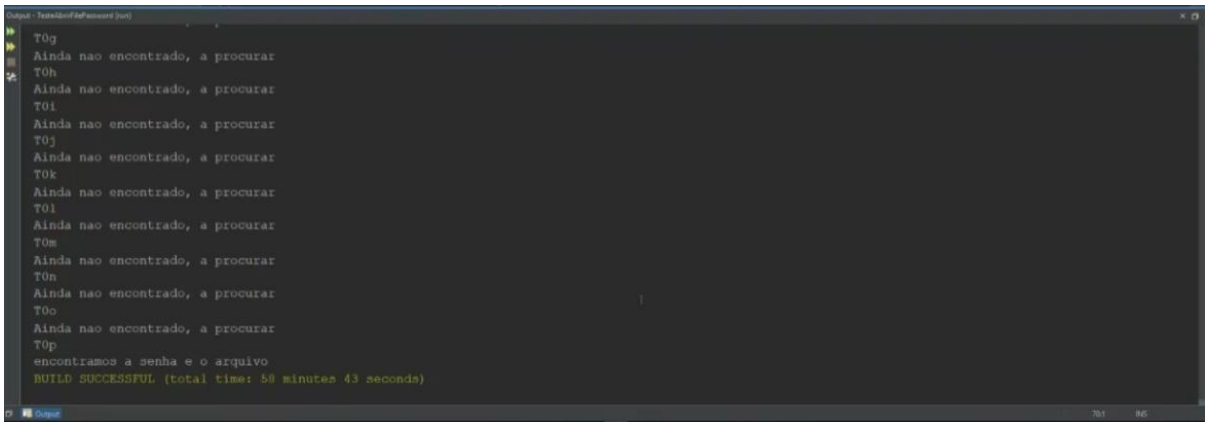
Placa de Vídeo: RTX 2060 SUPER

SSD: Sim, SSD NVME m.2

Com isso conseguimos que todos os arquivos fossem desencriptados em 33 minutos e 14 segundos.



Por fim, deixamos também o tempo da execução de um único arquivo, sem a utilização da programação paralela, tendo um total de 58 minutos e 43 segundos.



Como podemos a programação paralela nesse programa e extremamente importante para o ganho de tempo de execução. Apesar dos desafios, como a necessidade de configurar corretamente a programação paralela para evitar uma maior demora ou ganho pouco significativo de tempo, conseguimos entender e aplicar conceitos importantes. Aprendemos que a distribuição adequada das tarefas e a sincronização correta entre os threads são cruciais para o sucesso da programação paralela.