Compressão de Imagens com Multithreading

Ana Carolina Ferreira de Figueiredo 121044191

Andrew da Silva Faria 121081559

Relatório Final

Programação Concorrente (ICP-361) — 2023/2

Descrição do problema escolhido

A compressão de imagens JPEG é um método popular para reduzir o tamanho de arquivos de imagem digital, mantendo uma qualidade visual aceitável, é uma técnica de compactação com perdas, o que significa que parte da informação da imagem é descartada para reduzir o tamanho do arquivo. A qualidade da imagem comprimida depende do valor de qualidade escolhido. Valores mais baixos resultam em maior compressão e menor qualidade, enquanto valores mais altos preservam mais detalhes, mas geram arquivos maiores.

```
File inputFile = new File("input_image.jpg");

BufferedImage inputImage = ImageIO.read(inputFile);

Iterator<ImageWriter> writers =
ImageIO.getImageWritersByFormatName("jpg");
ImageWriter writer = writers.next();

File outputFile = new File("output.jpg");
ImageOutputStream outputStream =
ImageIO.createImageOutputStream(outputFile);
writer.setOutput(outputStream);

ImageWriteParam params = writer.getDefaultWriteParam();
//Primeiro, o código obtém um objeto ImageWriteParam chamado params. O
ImageWriteParam é uma classe que contém configurações que afetam o
processo de escrita de imagens.

params.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
//"MODE_EXPLICIT" significa que você deseja ter controle explícito
sobre os parâmetros de compressão da imagem. Isso permitirá que você
defina manualmente a qualidade da compressão.

params.setCompressionQuality(0.5f);
//Neste caso, o valor é definido como 0.5f, o que equivale a uma
qualidade de compressão de 50%. Isso significa que, durante o processo
de compressão, a imagem será processada de tal forma que 50% dos
```

```
detalhes originais serão preservados, enquanto 50% da informação será
descartada para reduzir o tamanho do arquivo resultante.

writer.write(null, new IIOImage(inputImage, null, null), params);
//Aqui, o método write do objeto writer é chamado para escrever a
imagem de entrada (inputImage) no formato JPEG no arquivo de saída. Os
parâmetros de compressão definidos em params serão aplicados durante o
processo de escrita.

outputStream.close();
writer.dispose();
```

Dados de entrada: *Um arquivo de imagem.*

Funcionamento sequencial: *O código começa lendo a imagem de entrada em um objeto BufferedImage. Em seguida, obtém um escritor de imagem para o formato "jpg".Define os parâmetros de escrita da imagem, incluindo o modo de compressão explícita e a qualidade da compressão (neste caso, definida como 0.5f). Finalmente, escreve a imagem de entrada comprimida em um arquivo de saída.*

Saída: *Um arquivo de imagem contendo a imagem de entrada comprimida de acordo com os parâmetros especificados.*

O problema pode se beneficiar de uma solução concorrente em cenários onde é necessário processar várias imagens em paralelo. Se você tiver várias imagens para processar, cada uma delas pode ser tratada como uma tarefa separada. Uma solução concorrente pode distribuir o processamento de várias imagens em diferentes threads ou processos para acelerar o processo.

Além disso, em operações de leitura e gravação de arquivos, uma parte significativa do tempo é gasta esperando que os dados sejam lidos ou gravados. Enquanto uma thread está esperando, outras threads podem executar tarefas em segundo plano.

Projeto da solução concorrente

Para projetar uma solução concorrente para o problema de compressão de imagens, que envolve a leitura de uma imagem, aplicação de compressão e escrita da imagem comprimida, você pode adotar uma estratégia de divisão da tarefa principal entre fluxos de execução independentes, as threads, como por exemplo:

- *Divisão por imagem: Cada thread é responsável por processar uma imagem de entrada, quando você tiver várias imagens para processar em paralelo.*
- *Divisão por bloco: Cada thread é responsável por processar um bloco específico de uma única imagem, para acelerar o processamento de uma única imagem, dividindo-a em partes menores.*
- *Divisão por fase: Cada thread é atribuído a uma etapa do processo, como leitura, compressão ou escrita se uma das etapas for mais demorada que as outras.*

Optamos pela divisão por imagens porque nosso principal propósito era trabalhar com

um conjunto de imagens, otimizando o tempo de compressão do conjunto como um todo. Nossa escolha conta com um grande número de vantagens operacionais:

- *Ao dividir o processamento em imagens, cada thread ou unidade de processamento pode trabalhar em uma imagem separada de forma independente, o que permite um aproveitamento dos recursos de CPU, especialmente em sistemas com múltiplos núcleos, resultando em uma utilização mais eficiente do hardware.*
- *Essa abordagem é escalável, o que significa que você pode processar uma única imagem ou um grande conjunto de imagens com a mesma lógica de processamento.*
- *Cada imagem é tratada como uma tarefa independente, o que evita que erros em uma imagem afetem as outras. Se uma imagem apresentar problemas durante o processamento, isso não impedirá que as outras imagens sejam processadas.*

Por fim, nessa abordagem, cada imagem de entrada é atribuída a uma thread separada para processamento. Isso significa que cada thread lê uma imagem, aplica a compressão e escreve a imagem comprimida.

Para realizar a divisão de imagens por threads, pensamos em utilizar e comparar duas abordagens nessa separação, uma dinâmica e uma por blocos:

- *Como teremos um número variável de imagens para processar, e desejamos otimizar a distribuição de carga entre as threads de maneira flexível, uma divisão dinâmica de imagens entre threads, que envolve a alocação de imagens às threads à medida que as imagens estão disponíveis para processamento, pode ser bastante útil e eficiente.*
- *Já a divisão das threads por blocos, consiste em cada thread se tornar responsável por processar um subconjunto específico de imagens. Cada thread atua em um bloco de imagens consecutivas, distribuindo a carga de trabalho entre elas. aproveitando o paralelismo e proporcionando uma distribuição equitativa da carga de trabalho, o que é benéfico para a eficiência e o desempenho do programa.*

Implementação da solução concorrente

Em nosso projeto consta três códigos principais, todos implementam o problema escolhido com as diferentes abordagens que gostaríamos de comparar: um sequencial, um concorrente dinâmico e um concorrente e em blocos.

compresSeq.java

Como já apresentado, esse programa Java realiza a compressão sequencial de uma série de imagens JPEG, cada uma em um arquivo de saída correspondente, utilizando a API Java ImageIO.

Inicialmente, o programa espera receber três argumentos da linha de comando: o diretório de entrada das imagens (imgDir), o diretório de saída das imagens comprimidas (outDir), e o número total de imagens a serem processadas (qntImg).

Em seguida, um loop for é utilizado para iterar sobre o número total de imagens a serem processadas (qntImg). Dentro do loop, o programa constrói os caminhos dos arquivos

de entrada e saída para cada imagem. A classe ImageIO é utilizada para ler a imagem de entrada do arquivo especificado usando inputImage = ImageIO.read(inputFile);

Com isso, o programa obtém um iterador para os escritores de imagem disponíveis para o formato JPEG usando Iterator<ImageWriter> writers = ImageIO.getImageWritersByFormatName("jpg"). O primeiro escritor na lista é selecionado usando ImageWriter writer = writers.next();.

Assim, um novo arquivo de saída é criado usando File outputFile = new File("./"+ outDir +"/out (" + i + ").jpg"); e os parâmetros de compressão são obtidos usando ImageWriteParam params = writer.getDefaultWriteParam();.A qualidade de compressão é definida como 0,5 (50%).

Por fim, a imagem comprimida é escrita no arquivo de saída usando writer.write(null, new IIOImage(inputImage, null, null), params); e o escritor de imagem é liberado usando writer.dispose();.

compressDinamico.java

Como já apresentado, esse programa é uma aplicação multithreaded para comprimir dinamicamente uma sequência de imagens JPEG. Em resumo, essa implementação usa programação concorrente para comprimir várias imagens simultaneamente, utilizando diferentes threads para melhorar o desempenho e a eficiência do processo de compressão, utilizando um contador compartilhado para garantir que cada thread processe uma imagem única.

O programa conta com 3 classes principais:

A classe Contador é uma classe simples que encapsula uma variável de contagem (count), e possui métodos getCount() para obter o valor atual da contagem e getAndInc() para obter o valor atual e incrementar a contagem de forma atômica (usando synchronized para garantir que a operação seja thread-safe).

A classe Compressor é uma subclasse de Thread e representa cada thread que realiza a compressão.Possui variáveis de instância para armazenar o identificador da thread (id), o objeto Contador compartilhado (C), o número total de imagens (qntImg), diretório de entrada de imagens (imgDir), e o diretório de saída de imagens comprimidas (outDir).O método run() é o método principal executado pela thread e contém um loop que continua até que todas as imagens tenham sido processadas por todas as threads. Dentro do loop, cada thread obtém o próximo valor do contador usando o método getAndInc() e usa esse valor para construir os caminhos dos arquivos de entrada e saída. O restante do código dentro do loop é semelhante ao programa original, onde a imagem é lida, comprimida e salva no arquivo de saída.

A classe principal compressDinamico contém o método main que serve como ponto de entrada da aplicação. Os argumentos de linha de comando são usados para especificar o diretório de entrada das imagens (imgDir), o diretório de saída das imagens comprimidas (outDir), o número total de imagens a serem processadas (qntImg), e o número de threads (nthreads). Um array de objetos Compressor é criado, e cada objeto é inicializado com os parâmetros relevantes. As threads são iniciadas no loop usando compressores[i].start(). O programa principal aguarda a conclusão de todas as threads usando compressores[i].join().

compressBlocos.java

Como já apresentado, esse programa Java implementa uma aplicação de compressão de imagens JPEG distribuída em blocos utilizando múltiplas threads, dividindo o trabalho entre as threads para melhorar a eficiência do processo de compressão.

O programa conta com 2 classes principais:

A classe Compressor, por meio do método Compressor, recebe parâmetros relacionados à thread e inicializa as variáveis de instância. O método run implementa o método executado pela thread por meio de um loop, que percorre as imagens a serem processadas, começando a partir do id da thread e incrementando em nthreads. A lógica dentro do loop é semelhante à versão sequencial, onde a imagem é lida, comprimida e salva no arquivo de saída.

A classe compressBlocks contém o método main que serve como ponto de entrada da aplicação. Os argumentos de linha de comando são usados para especificar o diretório de entrada das imagens (imgDir), o diretório de saída das imagens comprimidas (outDir), o número total de imagens a serem processadas (qntImg), e o número de threads (nthreads). Um array de objetos Compressor é criado, e cada objeto é inicializado com os parâmetros relevantes. As threads são iniciadas no loop usando compressores[i].start(). O programa principal aguarda a conclusão de todas as threads usando compressores[i].join().

Para os testes de corretude e desempenho, foram utilizadas imagens de dois bancos de imagens públicos e gratuitos (Pexels e Pixabey).

Casos de teste de corretude

Para os testes de corretude precisamos fazer a demonstração de 2 aspectos importantes: precisamos verificar que os resultados de saída do nosso programa estão corretos (corretude), isto é, as imagens retornadas são aquelas dadas como entrada, porém comprimidas. Para tal, pensamos que não é necessário usarmos uma quantidade muito extensa de imagens (dados), já que queremos ver apenas se os resultados estão corretos. Podemos diminuir o número de dados para um valor em que podem ser utilizadas várias threads, mas sem exagerar. Teremos como dados de entrada cerca de 24 imagens, em que repetimos algumas delas, para que possamos ter threads diferentes comprimindo cópias da mesma imagem e possamos compará-las (optamos por 4 imagens diferentes e 6 cópias de cada imagem). Os testes foram realizados utilizando diferentes números de threads (1, 2, 4 e 8 threads), onde ao final de cada um deles, obtivemos os seguintes resultados:

- ***compressBlocos.java:***

Os resultados apontaram a corretude do programa, todas as imagens tiveram uma compressão de 50%, quando rodadas no programa, em cada um dos testes.

Exemplo do funcionamento do programa em uma das imagens do teste de corretude utilizando 8 threads:

imagem de entrada:

imagem de saída:

- ***compressDinamico.java:***

Os resultados apontaram a corretude do programa, todas as imagens tiveram uma compressão de 50%, quando rodadas no programa, em cada um dos testes.

Exemplo do funcionamento do programa em uma das imagens do teste de corretude utilizando 8 threads:

imagem de entrada:

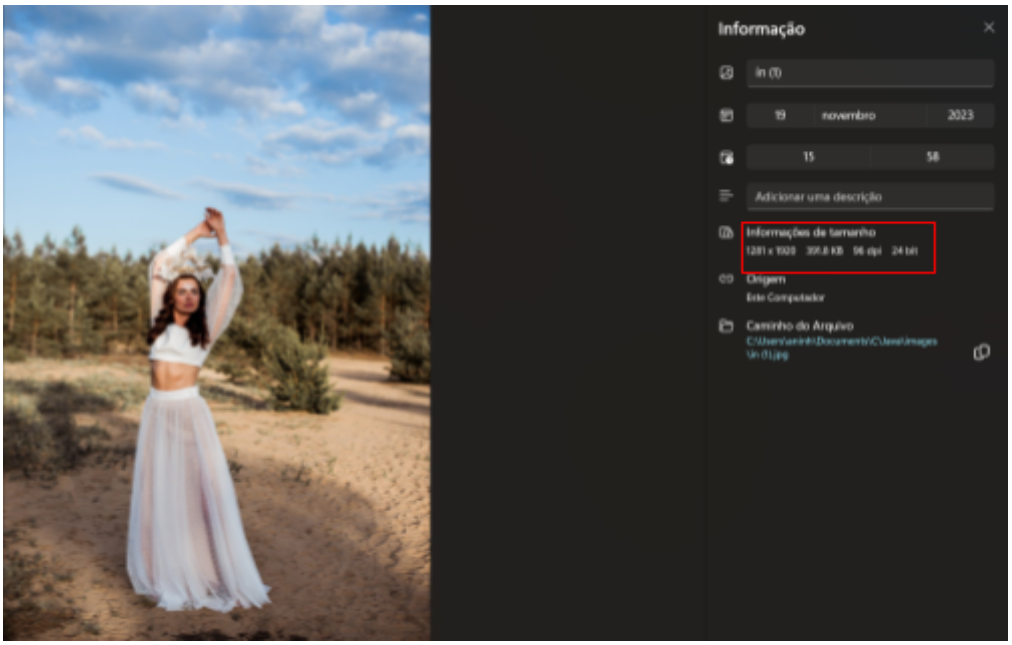
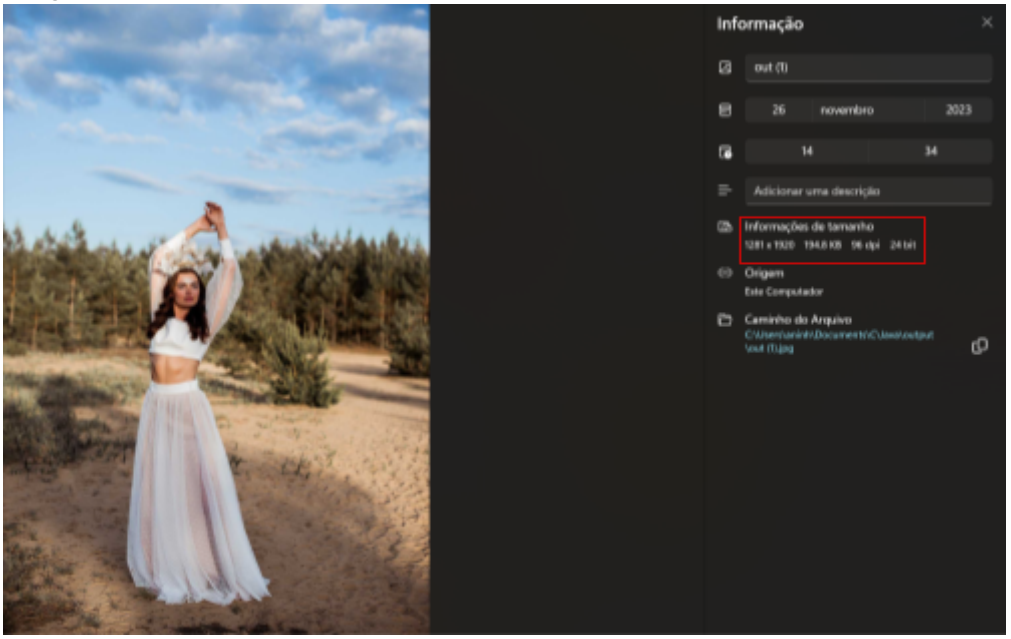


imagem de saída:



Casos de teste de desempenho

Para os teste de desempenho, precisamos realizar a avaliação de desempenho, procurando mostrar os ganhos de performance obtidos pela utilização de concorrência no programa em relação ao programa escrito de maneira totalmente sequencial. Para tal, precisaremos de um conjunto maior de dados. Isso porque dessa forma conseguiremos ver de maneira mais clara as vantagens da concorrência, que se beneficia justamente dessa maior carga de dados que podem ser processados por diferentes threads. Assim a ideia é utilizarmos como entrada diversas imagens, sendo essas imagens de tamanhos variados.

Para um teste mais completo, vamos testar alguns grupos de entrada:

- *25 imagens (tamanhos iguais ou muito parecidos)*
- *25 imagens (com uma maior variação entre os seus tamanhos)*
- *50 imagens (tamanhos iguais ou muito parecidos)*
- *50 imagens (com uma maior variação entre os seus tamanhos)*
- *100 imagens (tamanhos iguais ou muito parecidos)*
- *100 imagens (com uma maior variação entre os seus tamanhos)*

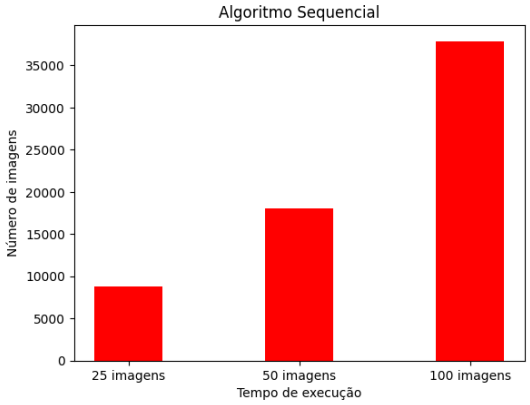
A ideia também é realizar os testes, para cada uma dessas entradas, usando 1, 2, 4 ou 8 threads, para observarmos o quanto de melhoria estaremos ganhando ao adicionar mais threads. Cada teste é realizado 10 vezes, e ao final utilizamos a média entre os valores obtidos.

Para o cálculo da aceleração, dividimos o tempo sequencial de execução pelo tempo médio de cada teste feito. Para o cálculo da eficiência, dividimos a aceleração do teste em questão pelo número de processadores utilizados nele.

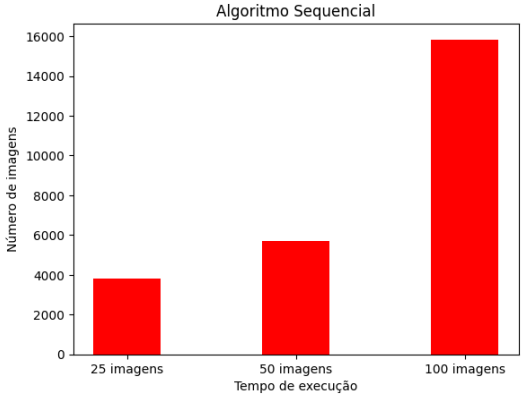
Resultados brutos obtidos:

Algoritmo sequencial

- *imagens-distintas 25 img -> 8754 ms 50 img -> 18084 ms 100 img -> 37865 ms*

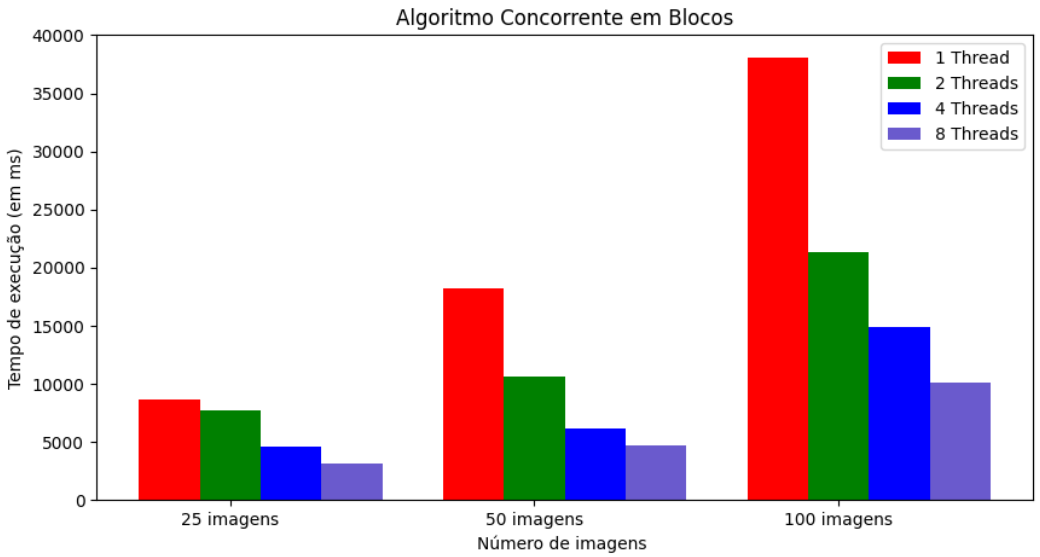


- *imagens-parecidas 25 img -> 3804 ms 50 img -> 5723 ms 100 img -> 15837 ms*



Algoritmo concorrente em blocos

- *imagens-distintas*



1 thread: 25 img -> 8645 ms 50 img -> 18225 ms 100 img -> 38122 ms

aceleração: 25 img -> 1,01 50 img -> 0,99 100 img -> 0,99

eficiência: 25 img -> 1,01 50 img -> 0,99 100 img -> 0,99

2 threads: 25 img -> 7670 ms 50 img -> 10675 ms 100 img -> 21300 ms

aceleração: 25 img -> 1,14 50 img -> 1,69 100 img -> 1,77

eficiência: 25 img -> 0,57 50 img -> 0,84 100 img -> 0,88

4 threads: 25 img -> 4629 ms 50 img -> 6196 ms 100 img -> 14908 ms

aceleração: 25 img -> 1,89 50 img -> 2,91 100 img -> 2,54

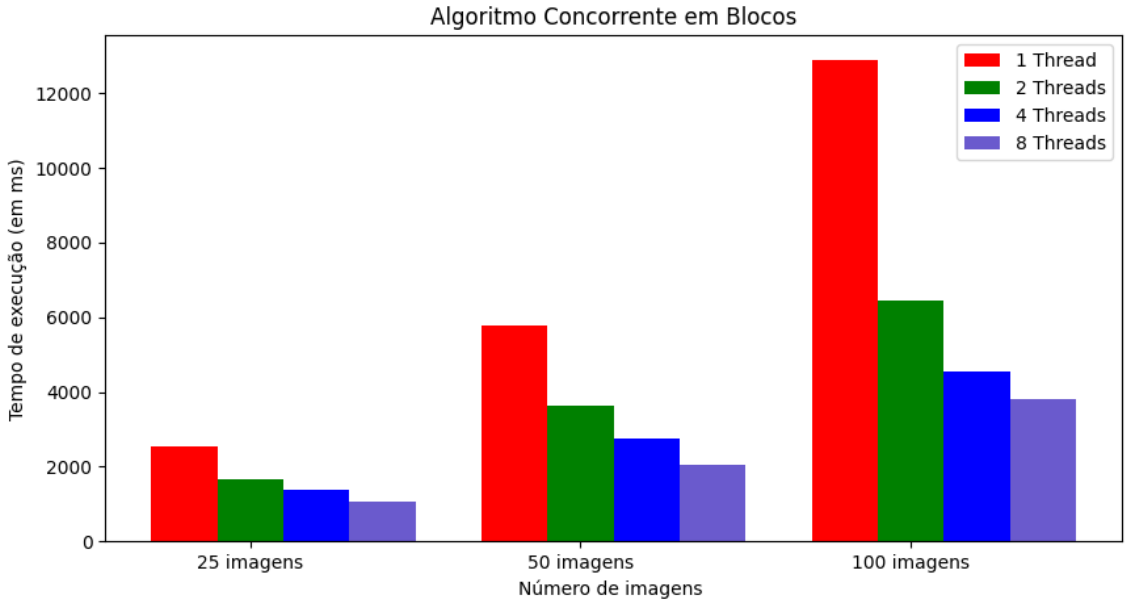
eficiência: 25 img -> 0,47 50 img -> 0,72 100 img -> 0,63

8 threads: 25 img -> 3177 ms 50 img -> 4730 ms 100 img -> 10078 ms

aceleração: 25 img -> 2,75 50 img -> 3,82 100 img -> 3,75

eficiência: 25 img -> 0,34 50 img -> 0,47 100 img -> 0,46

- *imagens-parecidas*



1 thread: 25 img -> 2533 ms 50 img -> 5767 ms 100 img -> 12899 ms

aceleração: 25 img -> 1,5 50 img -> 0,99 100 img -> 1,22

eficiência: 25 img -> 1,5 50 img -> 0,99 100 img -> 1,22

2 threads: 25 img -> 1662 ms 50 img -> 3623 ms 100 img -> 6447 ms

aceleração: 25 img -> 2,28 50 img -> 1,57 100 img -> 2,45

eficiência: 25 img -> 1,24 50 img -> 0,78 100 img -> 1,22

4 threads: 25 img -> 1382 ms 50 img -> 2762 ms 100 img -> 4544 ms

aceleração: 25 img -> 2,75 50 img -> 2,07 100 img -> 3,48

eficiência: 25 img -> 0,68 50 img -> 0,51 100 img -> 0,87

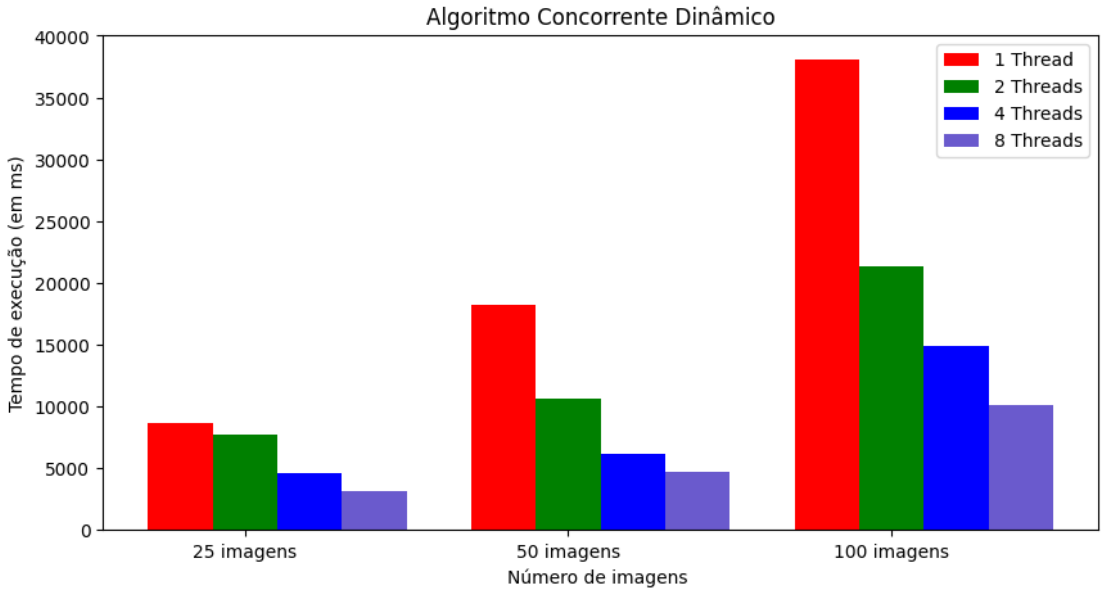
8 threads: 25 img -> 1062 ms 50 img -> 2062 ms 100 img -> 3791 ms

aceleração: 25 img -> 3,58 50 img -> 2,77 100 img -> 4,17

eficiência: 25 img -> 0,44 50 img -> 0,34 100 img -> 0,52

Algoritmo concorrente em dinâmico

- *imagens-distintas*



1 thread: 25 img -> 8567 ms 50 img -> 17928 ms 100 img -> 37621 ms

aceleração: 25 img -> 1,02 50 img -> 1,008 100 img -> 1,006

eficiência: 25 img -> 1,02 50 img -> 1,008 100 img -> 1,006

2 threads: 25 img -> 4765 ms 50 img -> 9215 ms 100 img -> 19935 ms

aceleração: 25 img -> 1,83 50 img -> 1,96 100 img -> 1,89

eficiência: 25 img -> 0,91 50 img -> 0,98 100 img -> 0,94

4 threads: 25 img -> 2926 ms 50 img -> 5289 ms 100 img -> 11801 ms

aceleração: 25 img -> 2,99 50 img -> 3,43 100 img -> 3,2

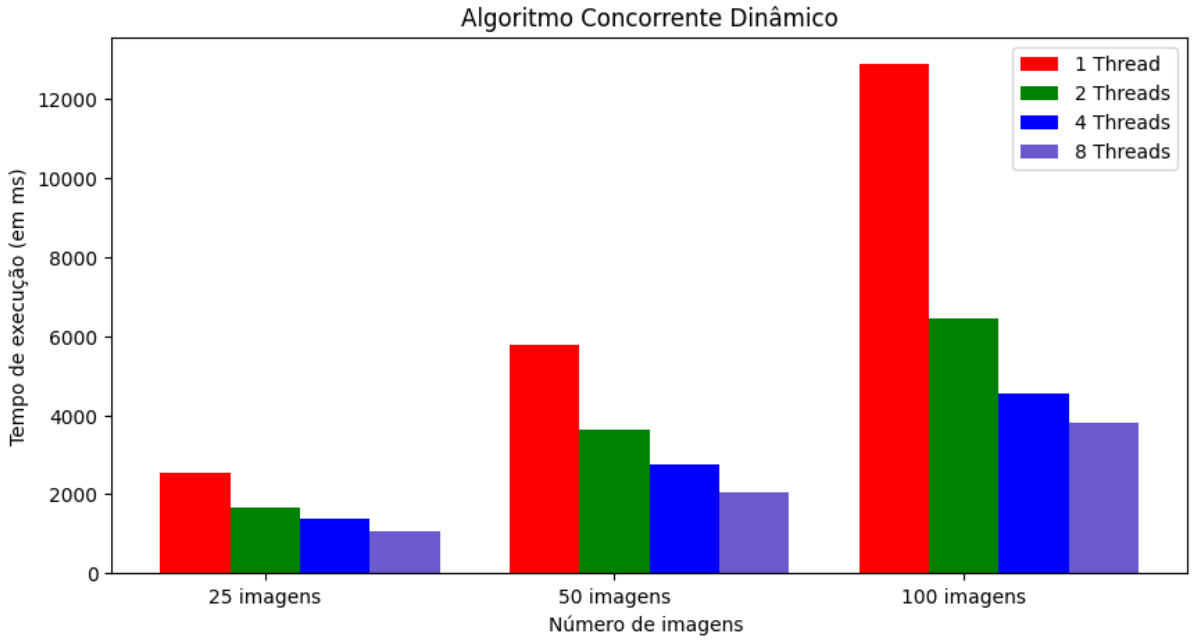
eficiência: 25 img -> 0,74 50 img -> 0,85 100 img -> 0,8

8 threads: 25 img -> 2114 ms 50 img -> 3459 ms 100 img -> 7802 ms

aceleração: 25 img -> 4,14 50 img -> 5,25 100 img -> 4,84

eficiência: 25 img -> 0,51 50 img -> 0,65 100 img -> 0,6

- *imagens-parecidas*



1 thread: 25 img -> 2613 ms 50 img -> 5622 ms 100 img -> 11287 ms

aceleração: 25 img -> 1,45 50 img -> 1,01 100 img -> 1,4

eficiência: 25 img -> 1,45 50 img -> 1,01 100 img -> 1,4

2 threads: 25 img -> 1493 ms 50 img -> 3086 ms 100 img -> 6867 ms

aceleração: 25 img -> 2,54 50 img -> 1,85 100 img -> 2,3

eficiência: 25 img -> 1,27 50 img -> 0,92 100 img -> 1,15

4 threads: 25 img -> 1063 ms 50 img -> 2399 ms 100 img -> 5401 ms

aceleração: 25 img -> 3,57 50 img -> 2,38 100 img -> 2,93

eficiência: 25 img -> 0,89 50 img -> 0,59 100 img -> 0,73

8 threads: 25 img -> 964 ms 50 img -> 1866 ms 100 img -> 3675 ms

aceleração: 25 img -> 3,94 50 img -> 3,06 100 img -> 4,3

eficiência: 25 img -> 0,49 50 img -> 0,38 100 img -> 0,53

Avaliação dos resultados obtidos

Para a avaliação e comparação dos tempos de execução obtidos por cada programa em cada teste feito, separamos nossa comparação entre três aspectos distintos:

- *Imagens de Tamanhos Semelhantes vs. Imagens de Tamanhos Variados*

A compressão de imagens com tamanhos semelhantes pode ser mais eficiente, uma vez que os algoritmos podem explorar padrões de repetição de forma mais consistente em imagens que compartilham dimensões similares. Nos resultados apresentados, os tempos de execução para imagens de tamanhos semelhantes geralmente são menores do que para imagens de tamanhos variados em todos os algoritmos avaliados.

- *Algoritmo Sequencial vs Algoritmo em Blocos vs Algoritmo Dinámico*

Algoritmo Sequencial

Como esperado, o algoritmo sequencial, apesar de apresentar uma fácil implementação e poder ser mais eficiente para conjuntos de dados pequenos, possuí um desempenho limitado, especialmente ao lidar com grandes conjuntos de dados, uma vez que não aproveita totalmente o poder de processamento de sistemas multicore.

Algoritmo Concorrente em Blocos

Como esperado, apresentou um melhor desempenho em comparação com o sequencial, especialmente com conjuntos de dados grandes, utilizando mais de uma thread de execução. Para imagens distintas, observamos um crescimento na aceleração bastante considerável e taxas de eficiência dentro do esperado, com destaque a utilização de 2 threads com grandes conjuntos de dados. Para imagens parecidas, observamos um cenário semelhante, mas com uma taxa de crescimento da aceleração e da eficiência levemente mais baixos que para imagens distintas, isso se da muito provavelmente, porque cada thread terá uma menor carga de padrões de repetição para explorar, devido a divisão de trabalho entre elas. Ou seja, em geral obtivemos um declínio no tempo de execução considerável e favorável quando comparado ao algoritmo sequencial.

Entretanto, podendo haver overhead na divisão e coordenação do trabalho entre os threads, a eficiência pode depender do tamanho dos blocos e da natureza dos dados.

Algoritmo Concorrente Dinâmico

Como esperado, o algoritmo dinâmico apresentou um melhor desempenho em comparação com o sequencial, especialmente com conjuntos de dados grandes, utilizando mais de uma thread de execução. Em cenários em que a carga de trabalho não é uniformemente distribuída, pode ser mais eficiente do que o concorrente em blocos. Apesar da implementação mais complexa em comparação com o sequencial e o concorrente em blocos, em geral, observamos taxas de aceleração e eficiência consideravelmente altas, e maiores muitas vezes que quando comparado com o algoritmo em blocos. Ou seja, em geral obtivemos um declínio no tempo de execução considerável e favorável quando comparado ao algoritmo sequencial.

Em suma, ambos os algoritmos concorrentes superam o sequencial, especialmente quando há recursos de hardware multicore disponíveis. O desempenho dos algoritmos concorrentes melhora significativamente com o aumento do número de threads, até atingir um ponto de saturação. O ponto ideal de threads pode variar dependendo das características específicas do hardware e do problema.

- *mundo ideal vs resultados obtidos*

Primeiramente precisamos entender mais a fundo as métricas de avaliação consideradas para seguirmos com esse estudo. Sabemos que a aceleração (A) (do termo em inglês speedup) é a razão entre o tempo de execução da melhor versão sequencial do algoritmo Ts e o tempo de execução da versão paralela Tp(p) usando p processadores, ou seja A(p) = Ts / Tp(p).

Quando paralelizamos um problema, o ideal seria conseguir dividir a tarefa igualmente entre os fluxos de execução paralela sem adicionar carga de trabalho extra. Nesse caso, teríamos Tp(p) = Ts / p e a aceleração seria perfeitamente linear, ou seja, com p processadores o algoritmo concorrente executaria p vezes mais rápido que o algoritmo sequencial. Teoricamente esse é o valor máximo de aceleração de um algoritmo concorrente.

Além disso, também sabemos que a medida de eficiência (E) é usada para mostrar como a aceleração varia com o aumento do número de processadores, e é dada pela seguinte equação E(p) = A / p. Essa métrica permite avaliar, em particular, como a carga de trabalho extra do algoritmo concorrente cresce quando o número de processadores aumenta. Quando a aceleração é linear, temos eficiência igual a 1 (valor ideal). Como normalmente a aceleração é menor que o número de unidades de processamento, o valor da eficiência fica abaixo de 1, sendo desejável se buscar o valor mais próximo de 1 possível.

Nos resultados obtidos, pudemos observar acelerações lineares nos testes com imagens parecidas com 2 threads, para ambos os algoritmos concorrentes e nos testes de imagens distintas com 2 threads para o algoritmo dinâmico.

Entretanto, podemos notar também como houveram casos em que não foram apresentadas acelerações lineares, nos testes com 4 e 8 threads, principalmente devido a complexidade de gerência dos fluxos de execução e a competição pelo acesso à memória, que cresceram com o aumento do número de processadores.

Possíveis melhorias do programa

Notamos que para um mesmo número de threads, a proporção das métricas de avaliação não se mantém conforme aumentamos o número de imagens. Isso pode ser um indício de que as imagens utilizadas influenciam para esse resultado, ou seja é possível que hajam características da imagem, além de seus tamanhos e quantidade, que possam impactar na execução do programa. Nossa ideia seria realizar uma nova categoria de testes, considerando as demais características de uma imagem, para identificar exatamente quais são esses aspectos, para descobrir as imagens que obtêm um melhor desempenho do programa.

Referências bibliográficas

https://www.baeldung.com/java-image-compression-lossy-lossless