

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

DIM0124 - PROGRAMAÇÃO CONCORRENTE

Relatório de Desenvolvimento de Aplicação Concorrente

KNN – Nearest Neighboor

Matrícula: 20180063677

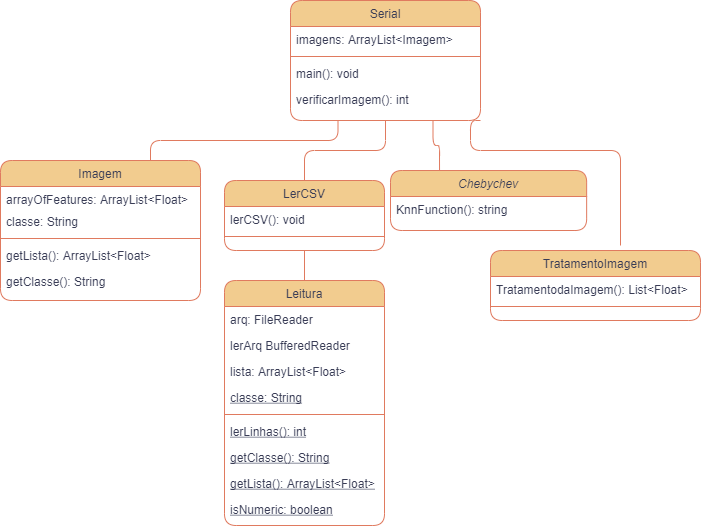
Nome: Gabriel Martins Spínola

1. **Introdução**

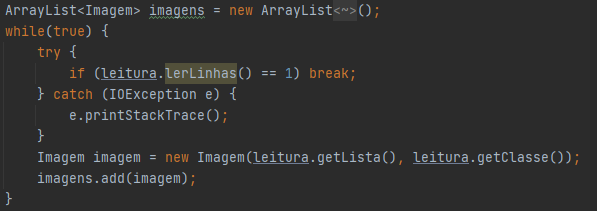
O algoritmo implementado foi o algoritmo do KNN, que tem o intuito de checar se existem ou não pessoas nas fotos que serão analisadas, para isso temos um arquivo em CSV contendo metadados de fotos com e sem pessoas para que possamos achar as fotos mais próximas quando processarmos.

Como resultado obtido, o programa imprime na tela se há ou não pessoas na foto a cada imagem que for processada. Com um dataset de 1gb o programa roda serialmente em 10 segundos em média, já com threads o cenário melhora, com semáforos e utilizando 5 threads a média cai para 7 segundos, com visibilidade e variáveis atômicas e utilizando também 5 threads a média continua em 7 segundos.

1. **Implementação Serial** 
   1. Descrição

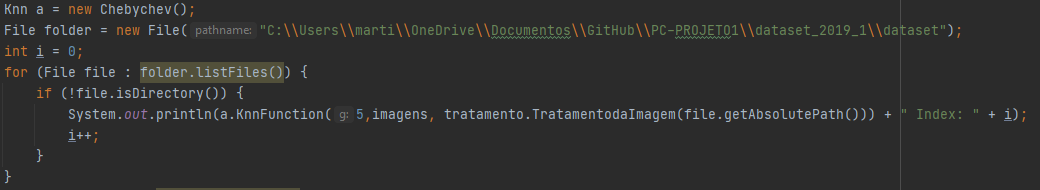


O algoritmo começa lendo um dataset em csv que possui vários metadados de algumas imagens que tenha ou não pessoas. Esse dataset é usado para calcular os vizinhos mais próximos da imagem que será processada posteriormente. Cada linha do csv é transformado em um objeto do tipo Imagem e armazenado em uma lista.

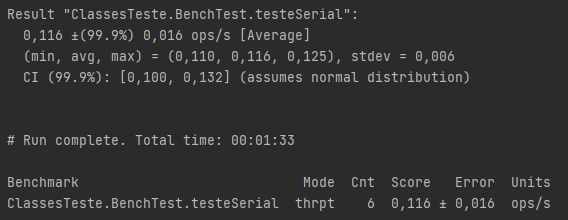


Após isso, criamos um objeto do tipo Chebchev (classe que tem o método que executa o cálculo de quais imagens são mais próximas) e percorremos a pasta do dataset executando a função de calculo para

cada imagem presente no dataset e imprimindo o resultado.

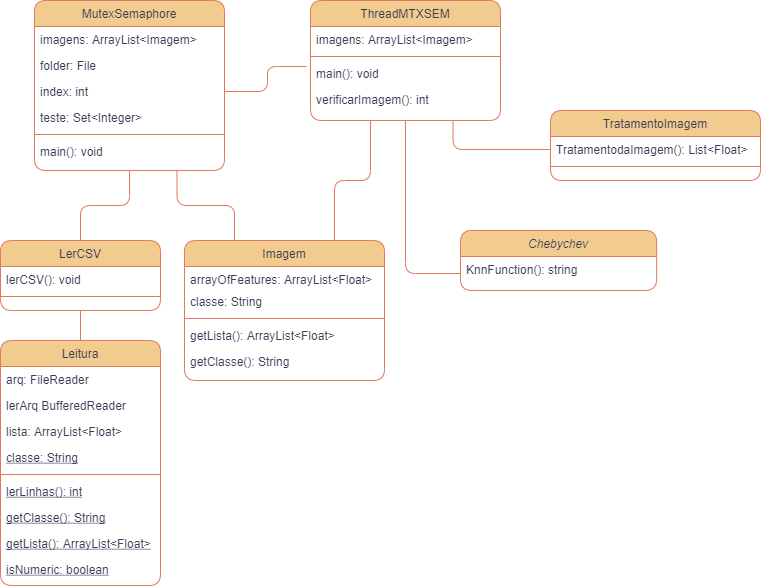


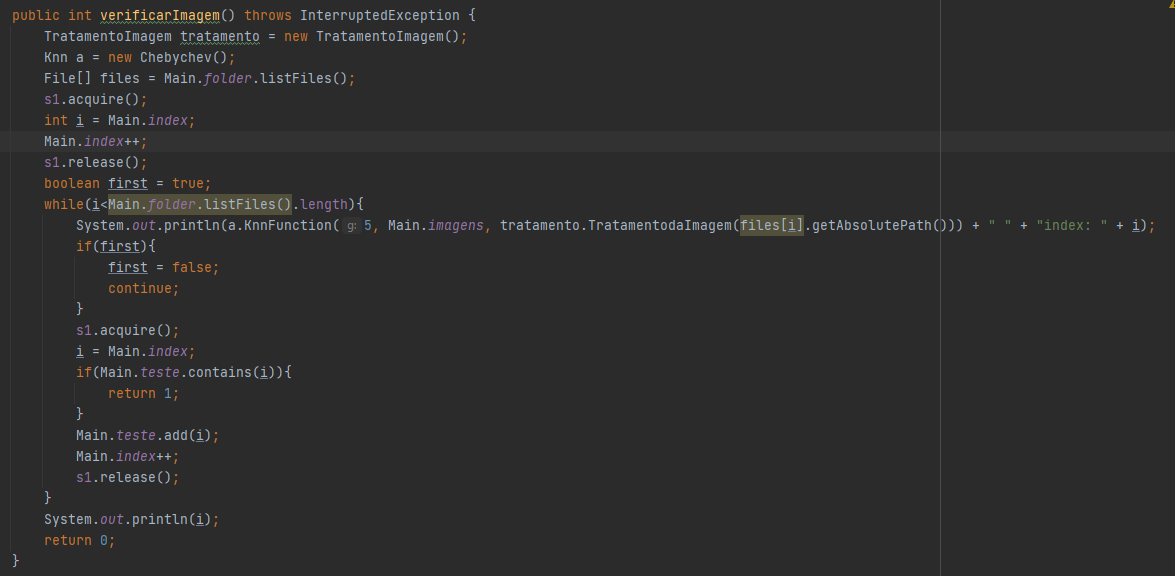
* 1. Avaliação com Microbenchmark

Foi realizado o teste de *benchmark* com o JMH com a implementação serial. A execução do teste retornou o seguinte resultado:

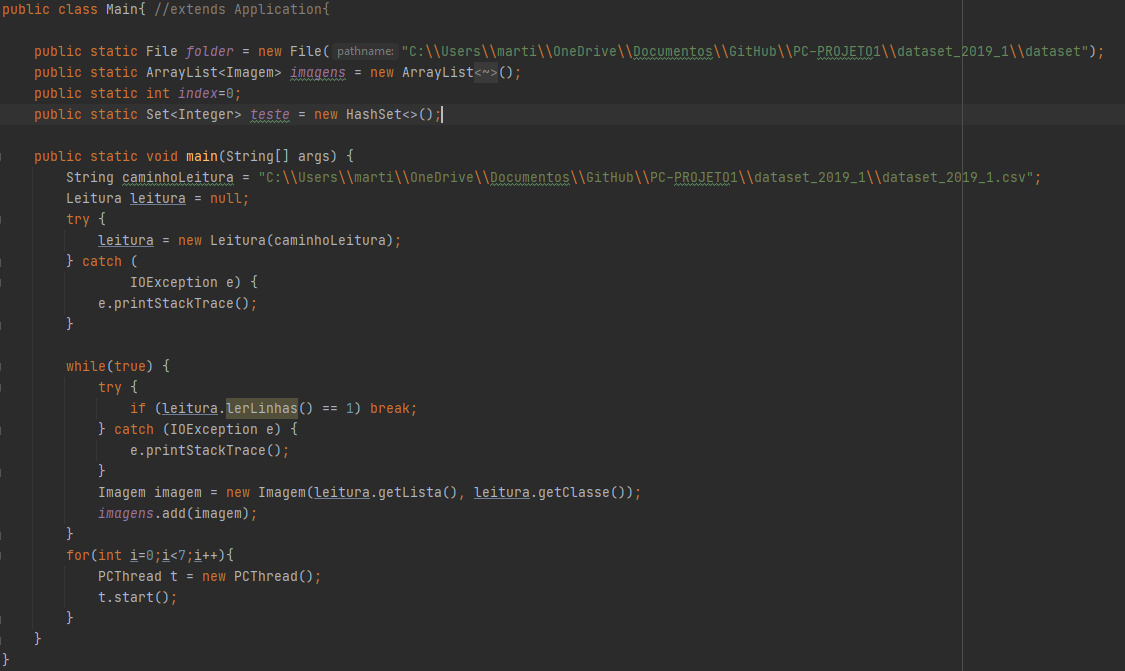
O método foi executado com o modo *throughput* com menos de 1 operação por segundo, com um intervalo de confiança de 99,9%, variação de 0,100 a 0,132 operação por segundo também foi possível notar que o método apresentou taxa de erro de 0,016 operações por segundo.

1. **Descrição da Implementação Concorrente - Abordagem Mutex/Semáforo**
   1. Descrição



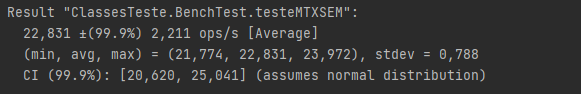
A primeira abordagem concorrente foi realizada utilizando semáforo, onde o semáforo controla o acesso a região critica que é o arquivo(imagem) que vai ser processado naquele momento pela *thread*, cada thread precisa percorrer a pasta do dataset processando as imagens, para isso, quando uma *thread* acessa um arquivo através de um indice, esse indice é bloqueado e então o algoritmo armazena o valor do indice e depois incrementa para poder liberar o acesso.

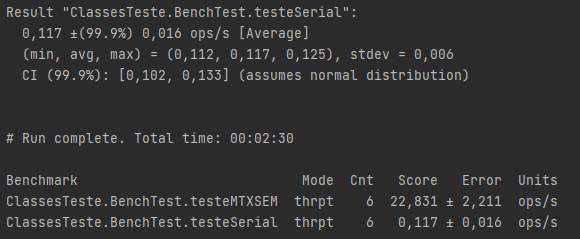
O indice é uma variável estática que está instanciada na classe Main



* 1. Avaliação com Microbenchmark

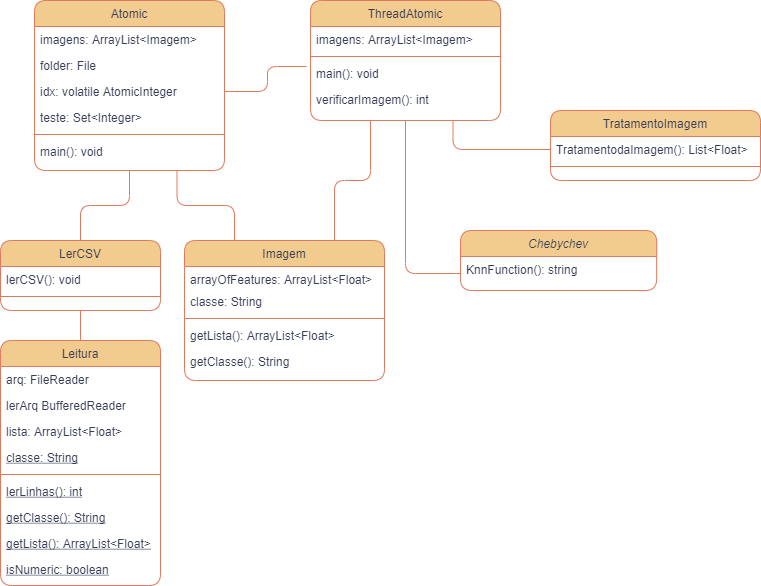
Ao realizar o teste com o JMH, o desempenho se mostrou muito mais eficiente que a implementação serial, onde conseguimos uma melhora significativa no *throuput* como vemos a seguir:

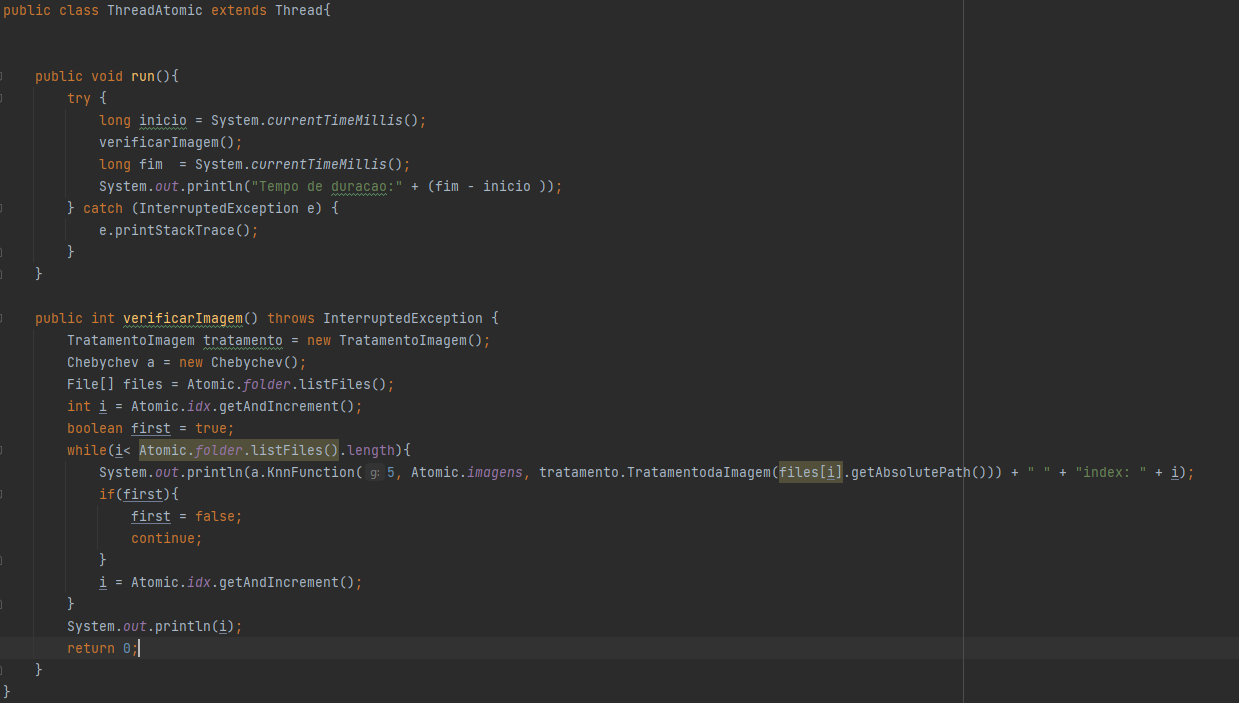




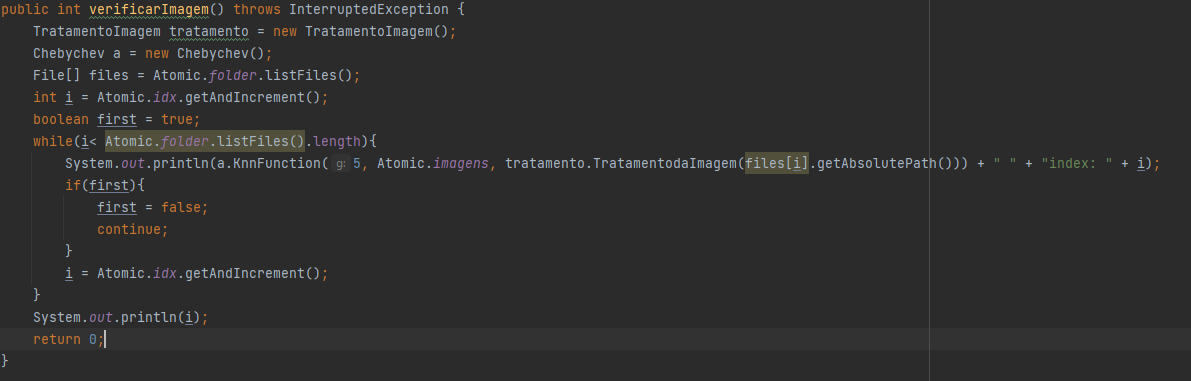
Vemos que a implementação concorrente, com mutex/semáforo, teve 22,831 operações por segundo com 99% de confiança com variação entre 20,060 e 25,041 vezes por segundos. Notou-se uma evolução de mais de 20 vezes comparado ao algoritmo implementado serialmente.

1. **Descrição da Implementação Concorrente - Abordagem Atomic/Volátil**
   1. Descrição



A implementação concorrente acima foi feita utilizando visibilidade e variáveis atômicas para garantir o acesso correto ao indice do arquivo que será executado o processamento. A classe que implementa a concorrência é a classe ThreadAtomic, essa que herda da classe *Thread* e contém os métodos *run* e o método verificarImagem

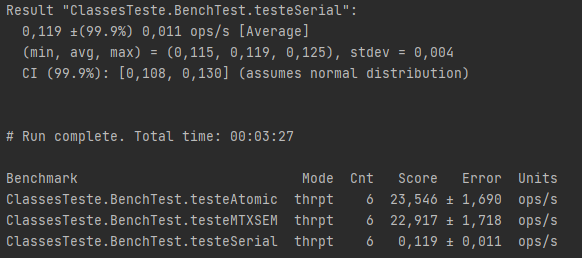
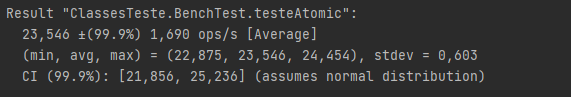
A variável idx é uma variável volátil criada na classe atomic, ela é uma variável atômica para podermos utilizar métodos como o getAndIncrement()



Dessa forma, podemos implementar a concorrência modificando poucas coisas da implementação vista anteriormente. A porcentagem do código concorrente é mais ou menos 25% do código.

* 1. Avaliação com Microbenchmark

Por fim temos o teste com o JMH da implementação concorrente através de visibilidade e variáveis atômicas. Pode se notar que não se obteve muitos ganhos em relação a implementação com Mutex e Semáforos. Porém, ainda assim apresentou resultado superior:

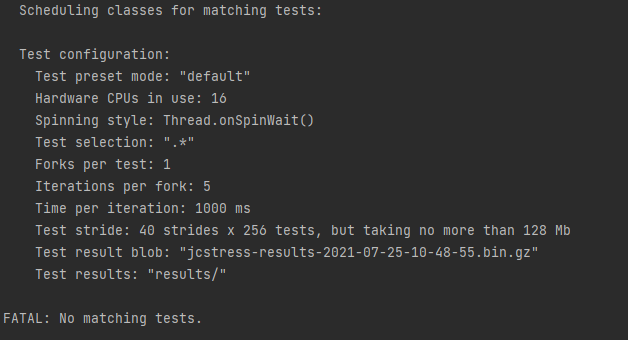


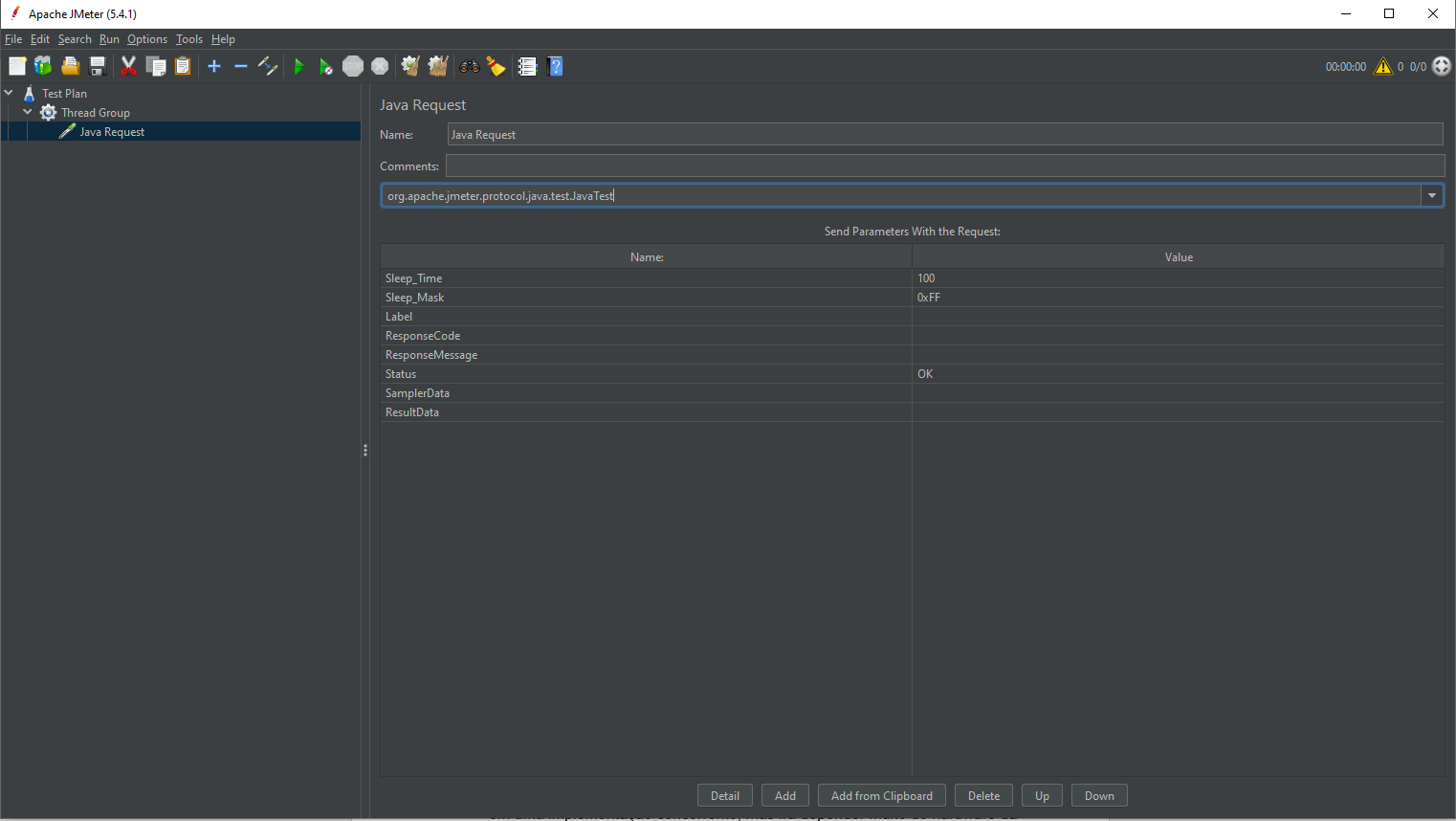
Podemos observar que foram realizadas 23,546 operações por segundo, superando as 22,917 da implementação anterior, a variação esteve ente 21,856 e 25,236 vezes por segundo e 1,69 operações de erro por segundo, o que mostra que com a variável atômica conseguiu mais eficiência e menos erro.

1. **Discussão (Vale 1.0)**

O trabalho desenvolvido teve o intuito de praticar e aprender a programar de forma concorrente, permitindo que os algoritmos desenvolvidos fossem feitos e testados pelas ferramentas vistas em aula. Para realizar o tratamento das imagens do dataset utilizei a biblioteca openCV.

Analisando os resultados oferecidos pelo JMH pude perceber que a implementação concorrente, tanto com mutex e semáforos quanto com visibilidade e variáveis atômicas ofereceram uma melhora significativa no tempo de execução do algoritmo, onde algumas vezes no meu computador, que possui 8 núcleos, a implementação com semáforo foi mais eficaz do que a com variável atômica e outras vezes foi o oposto, porém sempre mostrando ser bem mais rápido que a implementação serial.

Infelizmente, tive muitos problemas com os outros testes a ser realizado, só consegui rodar o JMH. Nas classes do projeto pode ser visto que está implementado as classes de testes de praticamente todos os testes, porém sempre algo dava errado e não conseguia roda-los, no teste do JCStress por exemplo, tive essa resposta ao rodar o código: 

Porém a classe de teste está implementada e está no projeto. Já no JMeter, fiz a classe de teste, rodei o mvn clean e install (assim como rodei antes de executar o JCStress) copiei a .jar para a pasta lib/ext porém, quando abro o programa e faço os procedimentos como está no slide de aula, não aparece a classe a ser testada 

Além disso, também não obtive sucesso com os outros testes, como o JFR que tentei várias formas de executar, vi diversos vídeos ensinando, fiz os procedimentos, mas nunca aparecia a opção de rodar o teste.

Como lição, aprendi que o ganho de eficiência pode ser significativo em uma implementação concorrente, mas irá depender muito do hardware da sua máquina. Também aprendi que pode ser mais interessante utilizar visibilidade e variáveis atômicas do que mutex/semáforos pois elas evitam locks desnecessários. Outros aprendizados sobre condições de corrida, garbage collection e outros assuntos estudados não puderam ser testados por diversos problemas portanto, como não consegui analisar esses dados na minha aplicação, acho que não faz sentido comentar a respeito, logo irei comentar apenas o que consegui realizar com sucesso no projeto.