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

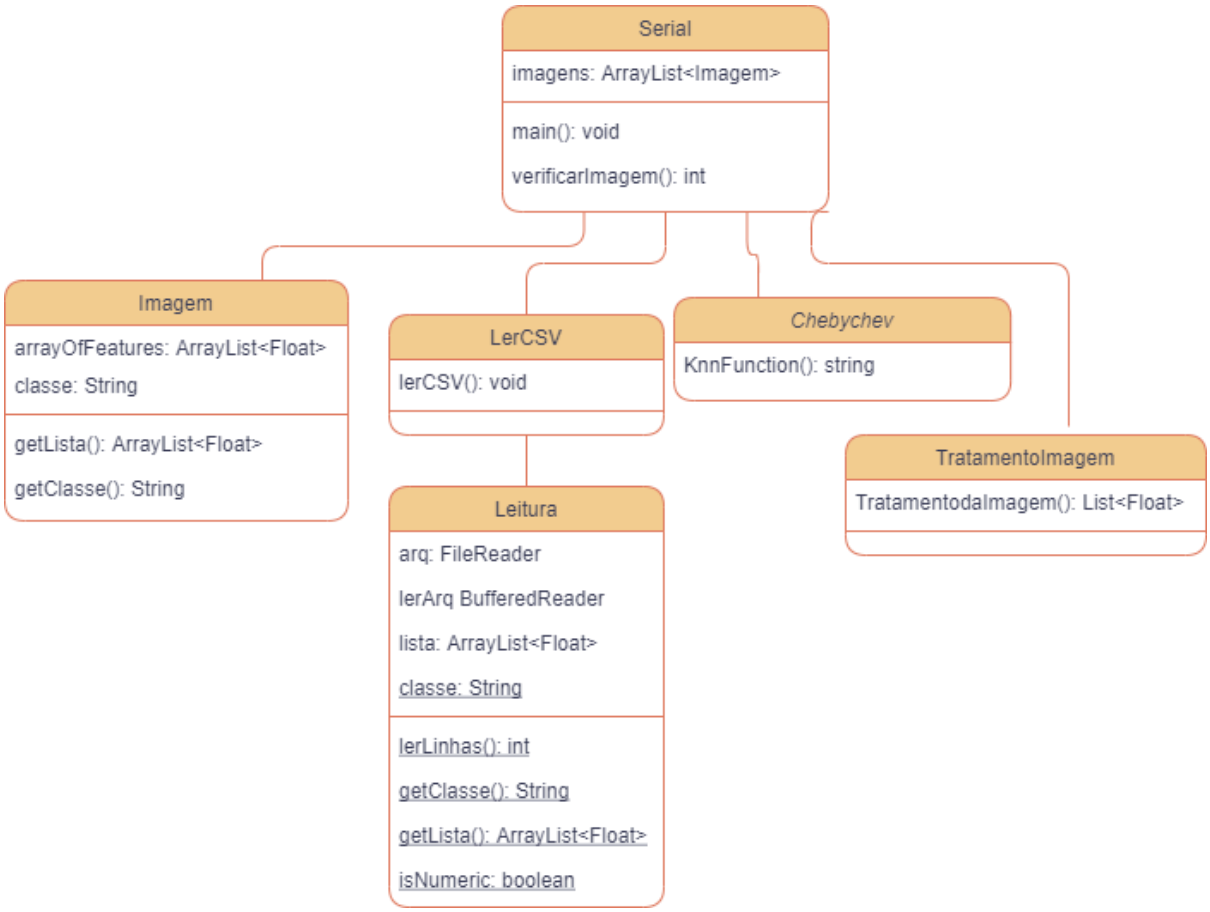
1. Introdução

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.

2. Implementação Serial

2.1. 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.

```
ArrayList<Imagem> imagens = new ArrayList<~>();
while(true) {
    try {
        if (leitura.lerLinhas() == 1) break;
    } catch (IOException e) {
        e.printStackTrace();
    }
    Imagem imagem = new Imagem(leitura.getLista(), leitura.getClasse());
    imagens.add(imagem);
}
```

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.

```
Knn a = new Chebychev();
File folder = new File( pathname: "C:\\Users\\marti\\OneDrive\\Documentos\\GitHub\\PC-PROJETO1\\dataset_2019_1\\dataset");
int i = 0;
for (File file : folder.listFiles()) {
    if (!file.isDirectory()) {
        System.out.println(a.KnnFunction( g: 5,imagens, tratamento.TratamentodaImagem(file.getAbsolutePath())) + " Index: " + i);
        i++;
    }
}
```

2.2. Avaliação com Microbenchmark

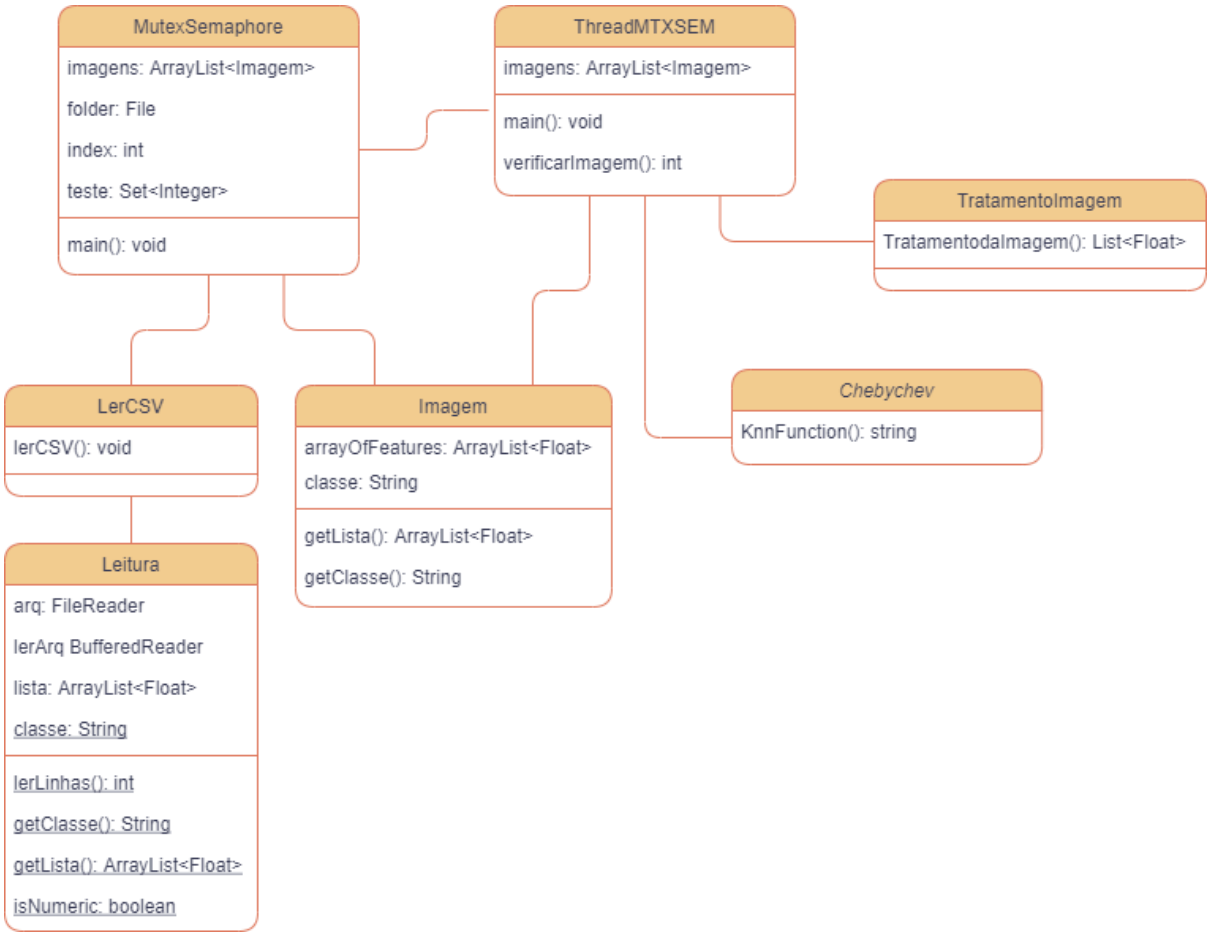
Foi realizado o teste de *benchmark* com o JMH com a implementação serial. A execução do teste retornou o seguinte resultado:

```
Result "ClassesTeste.BenchTest.testeSerial":
  0,116 ±(99.9%) 0,016 ops/s [Average]
  (min, avg, max) = (0,110, 0,116, 0,125), stdev = 0,006
  CI (99.9%): [0,100, 0,132] (assumes normal distribution)


# Run complete. Total time: 00:01:33

Benchmark                           Mode  Cnt  Score   Error  Units
ClassesTeste.BenchTest.testeSerial  thrpt    6  0,116 ± 0,016  ops/s
```

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.

3. Descrição da Implementação Concorrente - Abordagem Mutex/Semáforo

3.1. 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.

```java
public int verificarImagem() throws InterruptedException {
    TratamentoImagem tratamento = new TratamentoImagem();
    Knn a = new Chebychev();
    File[] files = Main.folder.listFiles();
    s1.acquire();
    int i = Main.index;
    Main.index++;
    s1.release();
    boolean first = true;
    while(i<Main.folder.listFiles().length){
        System.out.println(a.KnnFunction( g: 5, Main.imagens, tratamento.TratamentodaImagem(files[i].getAbsolutePath())) + " " + "index: " + i);
        if(first){
            first = false;
            continue;
        }
        s1.acquire();
        i = Main.index;
        if(Main.teste.contains(i)){
            return 1;
        }
        Main.teste.add(i);
        Main.index++;
        s1.release();
    }
    System.out.println(i);
    return 0;
}
```

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

```java
public class Main{ //extends Application{

    public static File folder = new File( pathname: "C:\\Users\\marti\\OneDrive\\Documentos\\GitHub\\PC-PROJETO1\\dataset_2019_1\\dataset");
    public static ArrayList<Imagem> imagens = new ArrayList<~>();
    public static int index=0;
    public static Set<Integer> teste = new HashSet<>();

    public static void main(String[] args) {
        String caminhoLeitura = "C:\\Users\\marti\\OneDrive\\Documentos\\GitHub\\PC-PROJETO1\\dataset_2019_1\\dataset_2019_1.csv";
        Leitura leitura = null;
        try {
            leitura = new Leitura(caminhoLeitura);
        } catch (
                IOException e) {
            e.printStackTrace();
        }

        while(true) {
            try {
                if (leitura.lerLinhas() == 1) break;
            } catch (IOException e) {
                e.printStackTrace();
            }
            Imagem imagem = new Imagem(leitura.getLista(), leitura.getClasse());
            imagens.add(imagem);
        }
        for(int i=0;i<7;i++){
            PCThread t = new PCThread();
            t.start();
        }
    }
}
```

3.2. 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:

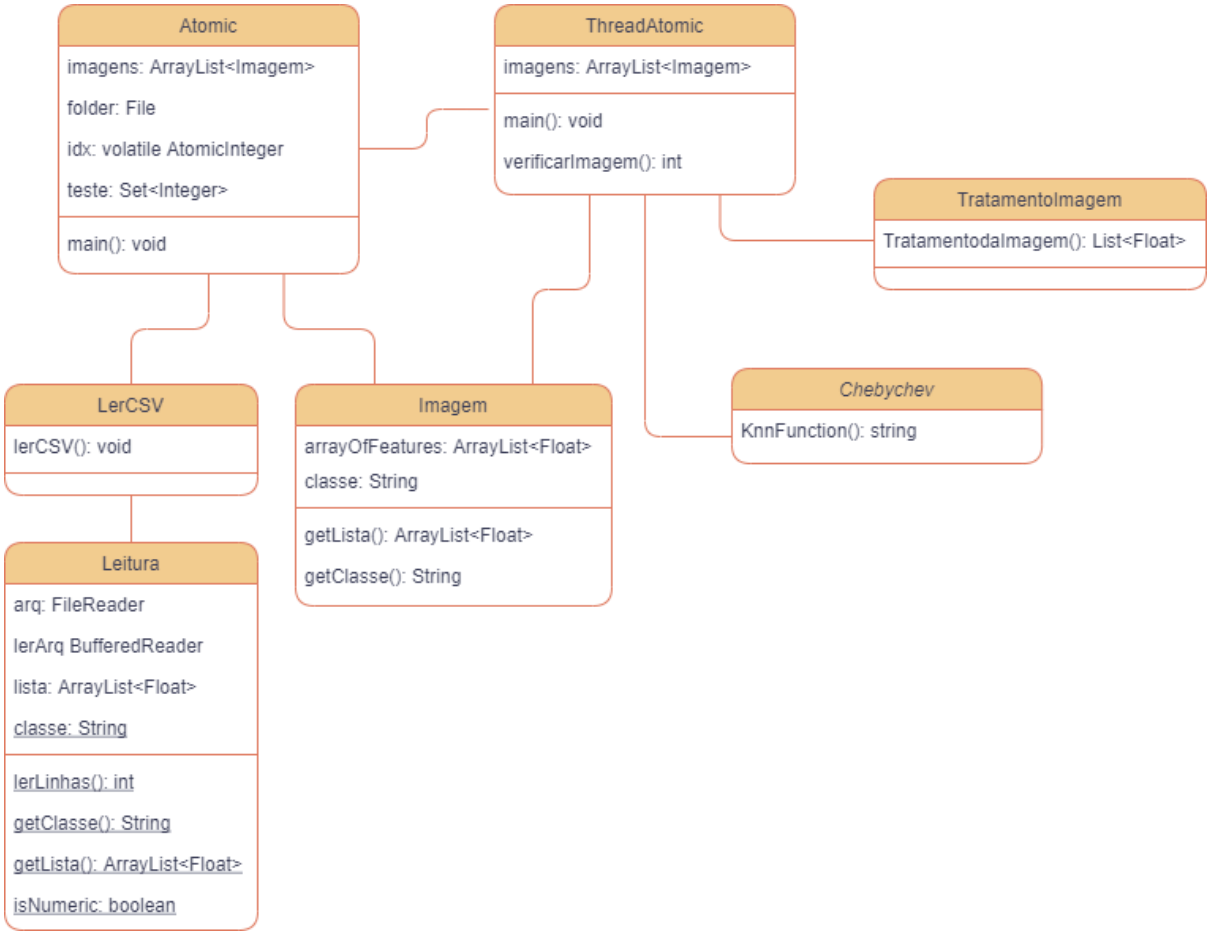
```
Result "ClassesTeste.BenchTest.testeMTXSEM":
  22,831 ±(99.9%) 2,211 ops/s [Average]
  (min, avg, max) = (21,774, 22,831, 23,972), stdev = 0,788
  CI (99.9%): [20,620, 25,041] (assumes normal distribution)
```

```
Result "ClassesTeste.BenchTest.testeSerial":
  0,117 ±(99.9%) 0,016 ops/s [Average]
  (min, avg, max) = (0,112, 0,117, 0,125), stdev = 0,006
  CI (99.9%): [0,102, 0,133] (assumes normal distribution)


# Run complete. Total time: 00:02:30

Benchmark                              Mode  Cnt   Score   Error  Units
ClassesTeste.BenchTest.testeMTXSEM    thrpt    6  22,831 ± 2,211  ops/s
ClassesTeste.BenchTest.testeSerial    thrpt    6   0,117 ± 0,016  ops/s
```

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.

4. Descrição da Implementação Concorrente - Abordagem Atomic/Volátil

4.1. 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

```
public class ThreadAtomic extends Thread{


    public void run(){
        try {
            long inicio = System.currentTimeMillis();
            verificarImagem();
            long fim  = System.currentTimeMillis();
            System.out.println("Tempo de duracao:" + (fim - inicio ));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public int verificarImagem() throws InterruptedException {
        TratamentoImagem tratamento = new TratamentoImagem();
        Chebychev a = new Chebychev();
        File[] files = Atomic.folder.listFiles();
        int i = Atomic.idx.getAndIncrement();
        boolean first = true;
        while(i< Atomic.folder.listFiles().length){
            System.out.println(a.KnnFunction( g: 5, Atomic.imagens, tratamento.TratamentodaImagem(files[i].getAbsolutePath())) + " " + "index: " + i);
            if(first){
                first = false;
                continue;
            }
            i = Atomic.idx.getAndIncrement();
        }
        System.out.println(i);
        return 0;
    }
}
```

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

```
public int verificarImagem() throws InterruptedException {
    TratamentoImagem tratamento = new TratamentoImagem();
    Chebychev a = new Chebychev();
    File[] files = Atomic.folder.listFiles();
    int i = Atomic.idx.getAndIncrement();
    boolean first = true;
    while(i< Atomic.folder.listFiles().length){
        System.out.println(a.KnnFunction( g: 5, Atomic.imagens, tratamento.TratamentodaImagem(files[i].getAbsolutePath())) + " " + "index: " + i);
        if(first){
            first = false;
            continue;
        }
        i = Atomic.idx.getAndIncrement();
    }
    System.out.println(i);
    return 0;
}
```

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.

4.2. 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:

```
Result "ClassesTeste.BenchTest.testeAtomic":
  23,546 ±(99.9%) 1,690 ops/s [Average]
  (min, avg, max) = (22,875, 23,546, 24,454), stdev = 0,603
  CI (99.9%): [21,856, 25,236] (assumes normal distribution)
```

```
Result "ClassesTeste.BenchTest.testeSerial":
  0,119 ±(99.9%) 0,011 ops/s [Average]
  (min, avg, max) = (0,115, 0,119, 0,125), stdev = 0,004
  CI (99.9%): [0,108, 0,130] (assumes normal distribution)


# Run complete. Total time: 00:03:27

Benchmark                              Mode  Cnt   Score   Error  Units
ClassesTeste.BenchTest.testeAtomic    thrpt    6  23,546 ± 1,690  ops/s
ClassesTeste.BenchTest.testeMTXSEM    thrpt    6  22,917 ± 1,718  ops/s
ClassesTeste.BenchTest.testeSerial    thrpt    6   0,119 ± 0,011  ops/s
```

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.

5. 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.

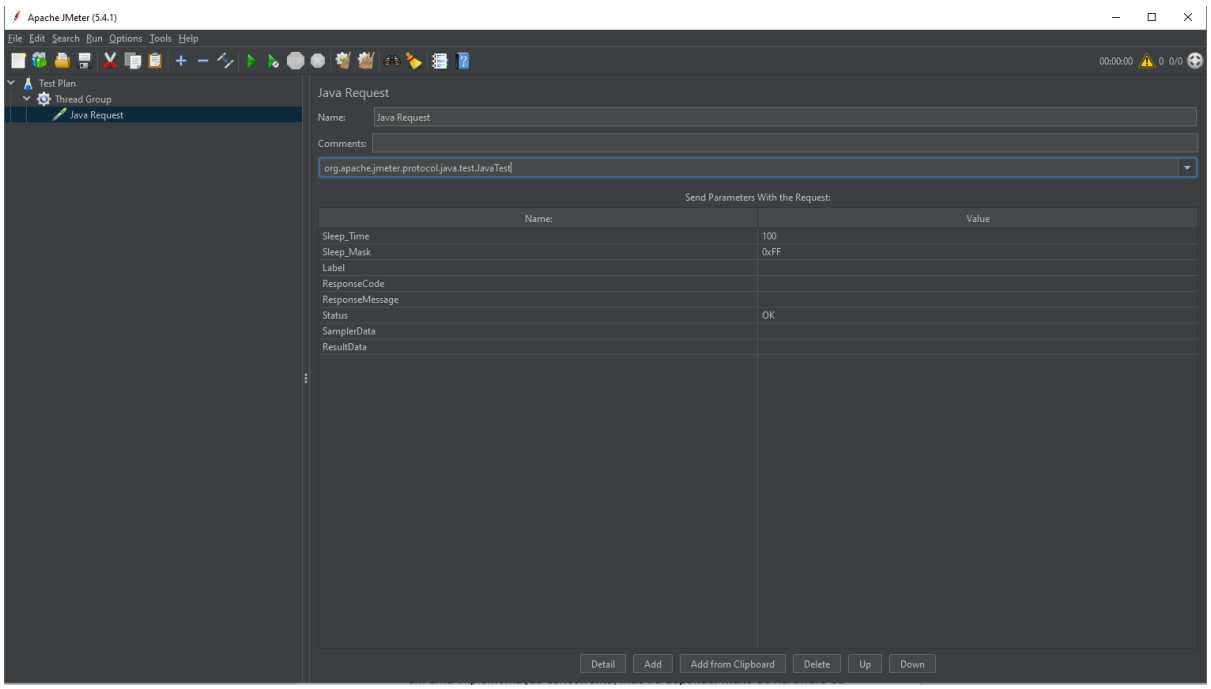
Analisando os resultados oferecidos pelo JMH pude perceber que a implementação concorrente, tanto com mutex e semáforos quanto com visibilidade e variáveis atômicas ofereceram uma melhora significativa no tempo de execução do algoritmo, onde algumas vezes no meu computador, que possui 8 núcleos, a implementação 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:

```
  Scheduling classes for matching tests:

  Test configuration:
    Test preset mode: "default"
    Hardware CPUs in use: 16
    Spinning style: Thread.onSpinWait()
    Test selection: ".*"
    Forks per test: 1
    Iterations per fork: 5
    Time per iteration: 1000 ms
    Test stride: 40 strides x 256 tests, but taking no more than 128 Mb
    Test result blob: "jcstress-results-2021-07-25-10-48-55.bin.gz"
    Test results: "results/"

FATAL: No matching tests.
```

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.