



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

DIM0124 - PROGRAMAÇÃO CONCORRENTE

Relatório de Desenvolvimento de Aplicação Concorrente

K-means

Matrícula: 20210057425

Nome: Talison Fabio Gomes De Medeiros

<b>1. Introdução.....</b>	<b>4</b>
1.1 Descrição do algoritmo.....	4
1.2 Base da dados.....	4
1.3 Descrição da Linguagem de Programação Escolhida.....	6
1.4 Setup do projeto.....	7
<b>2. Implementação Serial.....</b>	<b>11</b>
2.1 Descrição.....	11
2.2 Resultados.....	12
<b>3. Implementação Concorrente - Abordagem com Join.....</b>	<b>13</b>
3.1 Descrição.....	13
3.2 Resultados.....	16
<b>4. Implementação Concorrente - Abordagem Mutex/Semáforo.....</b>	<b>17</b>
4.1 Descrição.....	17
4.2 Mutex.....	17
4.3 Semáforos.....	21
4.4 Mutex + Semáforos.....	22
4.5 Rust.....	24
4.6 Resultados.....	26
<b>5. Implementação Concorrente - Abordagem Volátil.....</b>	<b>27</b>
5.1 Descrição.....	27
<b>6. Implementação Concorrente - ParallelStream.....</b>	<b>28</b>
6.1 Descrição.....	28
<b>7. Discussão.....</b>	<b>30</b>
<b>Segunda Unidade.....</b>	<b>34</b>
<b>8. Implementação Concorrente - Adder.....</b>	<b>34</b>
8.1 Descrição.....	34
8.2 Avaliação Microbenchmark.....	35
8.3 Avaliação de Profile.....	35
<b>9. Implementação Concorrente - Atomic.....</b>	<b>37</b>
9.1 Descrição.....	37
9.2 Avaliação Microbenchmark.....	37
9.3 Avaliação de Profile.....	38
<b>10. Implementação Concorrente - Executor.....</b>	<b>39</b>
10.1 Descrição.....	39
10.2 Avaliação Microbenchmark.....	40
10.3 Avaliação de Profile.....	40
<b>11. Implementação Concorrente - ForkJoin.....</b>	<b>42</b>
11.1 Descrição.....	42
11.2 Avaliação Microbenchmark.....	43
11.3 Avaliação de Profile.....	44
<b>12. Implementação Concorrente - Parallel Stream.....</b>	<b>45</b>
12.1 Descrição.....	45
12.2 Avaliação Microbenchmark.....	45

12.3 Avaliação de Profile.....	46
<b>13. Implementação Concorrente - Structured Conc.....</b>	<b>47</b>
13.1 Descrição.....	47
13.2 Avaliação Microbenchmark.....	48
13.3 Avaliação de Profile.....	48
<b>14. Implementação Concorrente - Concurrent Collections.....</b>	<b>50</b>
14.1 Descrição.....	50
14.2 Avaliação Microbenchmark.....	50
14.3 Avaliação de Profile.....	50
<b>Terceira Unidade.....</b>	<b>55</b>
<b>15. Spark - RDD.....</b>	<b>55</b>
15.1 Avaliação de Profile.....	57
<b>16. Spark - DataFrame.....</b>	<b>58</b>
16.1 Avaliação de Profile.....	60
<b>17. Spark - DataFrame + Parquet.....</b>	<b>61</b>
17.1 Avaliação de Profile.....	62
<b>18. Resultados e Discussão.....</b>	<b>62</b>

# 1. Introdução

## 1.1 Descrição do algoritmo

O algoritmo K-means é um método utilizado para agrupar pontos em K clusters, com base na distância entre eles. Cada ponto é atribuído ao cluster cujo centroíde está mais próximo, sendo o centroíde o ponto médio do cluster.

A cada iteração, o algoritmo calcula os centroides dos clusters como a média dos pontos atribuídos, e atribui a eles essa nova centroíde. Esse processo é repetido até que os centroides não mudem entre as iterações, ou até que seja atingido um número máximo de iterações.

Para todas as implementações foi feito o seguinte algoritmo:

- 1) **Inicialização:** São selecionados K pontos como centroides iniciais. Para conseguir os mesmos resultados em cada teste, esses valores eram selecionados ao invés de atribuição aleatória.
- 2) **Atribuição de clusters:** Eram percorridos todos os pontos no conjunto de dados, atribuía-o ao cluster com centroíde mais próximo, e o ponto era adicionado na lista de pontos daquele cluster.
- 3) **Cálculo de novos centróide:** Recalcula-se os centroides de cada cluster como a média de todos os pontos atribuídos a esse cluster.
- 4) **Convergência:** Eram comparados os novos centroides com os antigos
  - a) Se fossem iguais: algoritmo encerrado e clusters definidos.
  - b) Diferentes: é retornado ao passo 2

Para todas as implementações feitas a parte concorrente ficou no passo 2. Que se concentra a iteração que requer o processamento mais pesado.

## 1.2 Base da dados

Visando utilizar os algoritmo e o projeto para uma aplicação mais prática foi desenvolvido um programa para geração das bases de dados.

O Kmeans, além de muito usado em análise estatísticas, tem bastante aplicabilidade em análises de imagens e segmentação de elementos. Um caso básico que foi usado aqui é para segmentação de cores de uma imagem.

Foi desenvolvido um programa em *Rust* que converte arquivo um imagem recebido como parâmetro em um dataset CSV dos dados de RGB de cada pixel e suas coordenadas. E claro, consegue fazer o caminho inverso também: gerar uma imagem a partir de um dataset CSV com o mesmo formato de dados. Esse programa foi feito de forma que fosse facilmente integrado via redirect de *Stdout* e *StdIn* com os executáveis do algoritmo.

Com isso, era possível executar em uma mesma linha de comando um comando como esse...

```
$ pixels to-csv IMD-foto.png | java -jar Main.class | pixels to-image -o resultado.png
```

Esse *pixels* nesse exemplo é o binário do programa de conversão desenvolvido.

Nesse exemplo ele usa um arquivo de foto chamado `*imd-foto.png*` e transforma em dataset. Passa esse dataset para o programa em Java ler via o `System.in` e o programa (em Java) escreve a saída via `System.out`. Ao fim o programa recebe a saída de resultado e cria uma imagem com os dados recebidos.

Todos os programas de Kmeans ao fim do algoritmo tinham um processo de alterar os dados das imagens para refletir o resultado. Com isso, era possível obter a imagem que mostrava o resultado do algoritmo de forma instantânea.

Um exemplo abaixo há a imagem do IMD, usada no comando acima.



Embaixo seriam as imagens geradas ao fim, no primeiro caso com o K = 5 e na outra o K = 15.



Para todos os testes de benchmark e estresse foi usado uma base de dados de uma imagem com dimensões de 8500x6882 pixels, resultando num arquivo de dataset de 1.1GB.

### 1.3 Descrição da Linguagem de Programação Escolhida

Foi escolhida a linguagem *Rust*. Ela é uma linguagem compilada com foco em sistemas de baixo nível, buscando entregar um controle da memória manual e preciso, mas bem mais refinado que as linguagens dominantes no mercado. Rust se destaca por sua abordagem única de segurança. Ela foi projetada como uma alternativa mais segura ao C e ao C++, evitando problemas comuns de gerenciamento de memória, como erros de ponteiro e problemas de *memory leak*. Ao mesmo tempo mantendo um desempenho extremamente comparável. Com performance comparável e em alguns cenários melhor do que C.

Em Rust, não há um coletor de lixo (Garbage Collector). A memória é gerenciada manualmente, mas por meio de um sistema de propriedade. Cada valor

em Rust tem uma única propriedade, o que significa que apenas um proprietário pode existir para um determinado valor em um dado momento. Quando o proprietário sai de escopo, o valor é liberado automaticamente, semelhante à chamada de free() em C. Esse sistema garante que cada pedaço de memória esteja associado a apenas uma variável por vez e que todas as referências a esse pedaço de memória estejam válidas. Em Rust não há ponteiros nulos ou que apontem para espaços de memória já deslocados. Dessa forma ela exclui os maiores problemas de gerenciamento manual de memória, sem deixar de oferecer esse gerenciamento.

Uma das principais vantagens de Rust é que essa análise de propriedade é realizada apenas em tempo de compilação. Isso significa que o compilador garante a segurança de memória por meio de análise estática, sem custo adicional em tempo de execução.

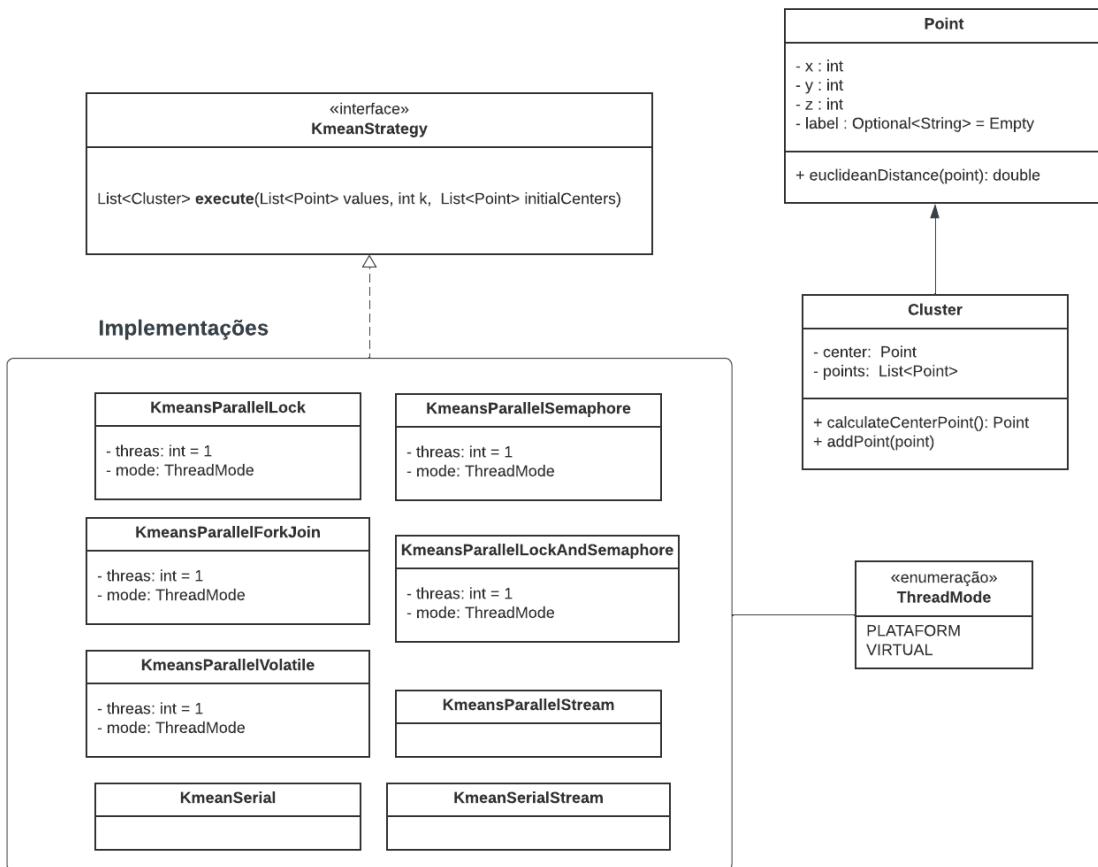
Sobre concorrência: devido ao seu foco em segurança e controle, Rust oferece implementações na sua biblioteca padrão para utilização de threads, Mutex, Semaphore, Barrier e Message Passing, mas também permite o uso de implementações externas. Existem diferentes runtimes em Rust para execução de código concorrente, que implementam recursos não suportados pela biblioteca padrão. Por exemplo, a biblioteca padrão não suporta algo similar às Virtual Threads, apenas threads nativas do SO. Para isso, existem bibliotecas externas que provêm runtimes para isso, como a Tokio ou a Async-STD. Ou Rayon para algo similar à parallelStreams de Java.

Além disso, não há JIT oficial de Rust, apenas projetos de terceiros ainda não estáveis. Reordering ocorre em programas em Rust.

## 1.4 Setup do projeto

Para todas as implementações foi utilizada a mesma estrutura e classes de entidades, interface e utilitárias. Alterando apenas as classes de implementação.

Foi usado o Strategy pattern para separar as implementações.



O código de interface do `KmeansStrategy` e as demais classes de entidade. Os exemplos acima possuem apenas os campos da classe para mais fácil explicação.

```

public interface KmeanStrategy {
    List<Cluster> execute(List<Point> values, int k, List<Point> initialCenters);
}

```

```

public class Cluster {
    Point center;
    List<Point> points;

    public Cluster(Point center, List<Point> points) {
        this.center = center;
        this.points = points;
    }
}

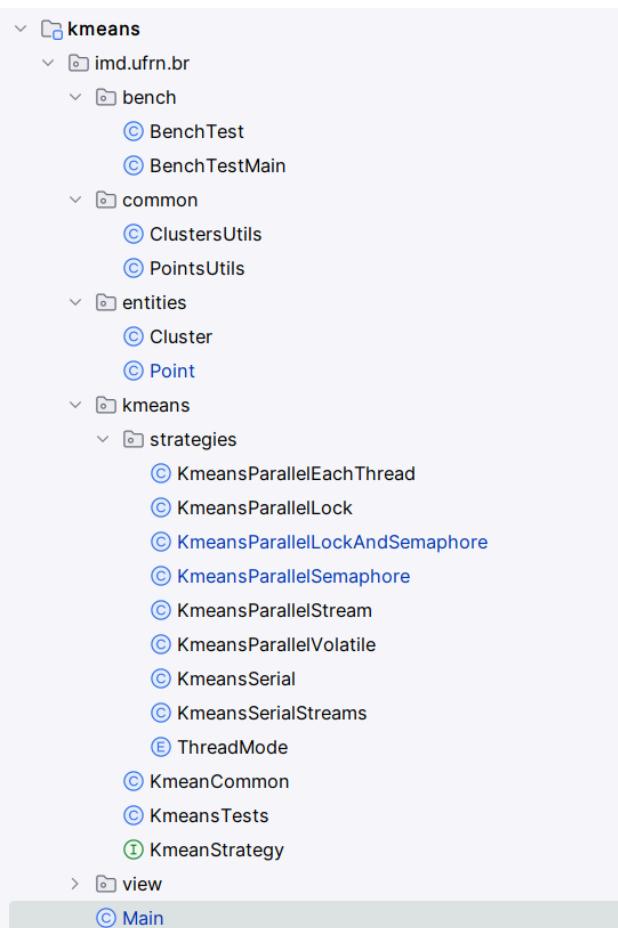
```

```

public class Point {
    private Optional<String> label = Optional.empty();
    private final int x;
    private final int y;
    private final int z;
}

```

Foi utilizado Maven e IntelliJ como IDEA de desenvolvimento. A estrutura no final do projeto ficou da seguinte maneira....



A main por sua vez era a responsável por receber os dados de entrada (hard-coded ou lendo pela entrada padrão) e setar a implementação para a execução.

```
public class Main {
    public static void main(String[] args) {
        var K = 5;
        List<Point> values = Input.read(System.in);
        List<Point> initialCenters = PointsUtils.extractDistinctInitialValues(values, K);

        // AQUI muda a implementação.
        // Precisando apenas alterar para a classe da implementação desejada
        var kmeansRunner = new KmeansParallelSemaphore(ThreadMode.PLATFORM, 8);

        List<Cluster> output = kmeansRunner.execute(values, K, initialCenters);

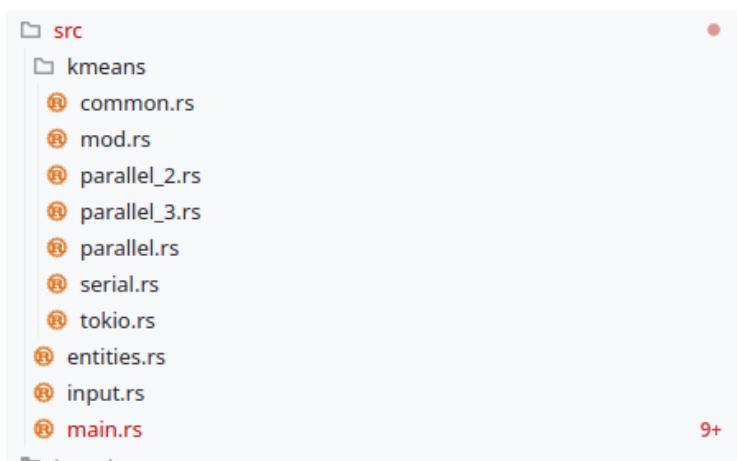
        for (var point : pointsFinal) {
            System.out.println(point.display());
        }
    }
}
```

Como Rust suporta parte do paradigma OO, foi usado a exata mesma organização de código para o projeto nele.

```
pub trait Kmeans {
    fn execute<'a>(
        &self,
        data: &'static Vec<Point>,
        k: u8,
        initial_centers: Vec<Point>,
    ) -> Vec<Cluster<'a>>;
}

#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
8 implementations
pub struct Point {
    x: u32,
    y: u32,
    z: u32,
    label: Option<Arc<str>>,
}

#[derive(Debug, PartialEq, Eq, Hash, Clone)]
6 implementations
pub struct Cluster<'a> {
    pub center: Point,
    pub points: Vec<&'a Point>,
}
```



Todas as implementações abaixo seguirão a partir dessa estrutura base. Sendo apenas classes novas que implementam a interface do KmeansStrategy.

## 2. Implementação Serial

### 2.1 Descrição

Em Java foram feitas duas implementações de serial. Uma totalmente imperativa, com laços de For e a outra utilizando essencialmente apenas Streams.

#### Versão imperativa

```
public class KmeansSerial implements KmeanStrategy {
    @Override
    public List<Cluster> execute(List<Point> values, int k, List<Point> initialCenters) {
        List<Cluster> clusters = new ArrayList<>();
        for (Point center : initialCenters) {
            clusters.add(new Cluster(center, new ArrayList<>()));
        }

        while (true) {
            for (var point : values) {
                int index = KmeanCommon.getIndexClosestCluster(point, clusters);
                clusters.get(index).addPoint(point);
            }

            var oldCenters = new ArrayList<Point>();
            var newCenters = new ArrayList<Point>();

            for (Cluster cluster : clusters) {
                oldCenters.add(cluster.getCenter());
                newCenters.add(cluster.calculateCenterPoint());
            }

            if (KmeanCommon.converged(newCenters, oldCenters)) {
                return clusters;
            }

            for (int i = 0; i < newCenters.size(); i++) {
                var cluster = clusters.get(i);
                var newCenter = newCenters.get(i);
                cluster.setCenter(newCenter);
                cluster.getPoints().clear();
            }
        }
    }
}
```

#### Versão com Streams

```

public class KmeansSerialStreams implements KmeanStrategy {
    @Override
    public List<Cluster> execute(List<Point> values, int k, List<Point> initialCenters) {
        List<Cluster> clusters = initialCenters.stream().map(Cluster::build_with_center).toList();

        while (true) {
            clusters = assignPoints(values, clusters);

            var newCenters = clusters.stream().map(Cluster::calculateCenterPoint).toList();
            var oldCenters = clusters.stream().map(Cluster::getCenter).toList();

            if (KmeanCommon.converged(newCenters, oldCenters)) {
                return clusters;
            }

            clusters = newCenters.stream().map(Cluster::build_with_center).toList();
        }
    }

    public static List<Cluster> assignPoints(List<Point> points, List<Cluster> clusters) {
        for (var point : points) {
            int index = KmeanCommon.getIndexClosestCluster(point, clusters);
            clusters.get(index).getPoints().add(point);
        }
        return clusters;
    }
}

```

A implementação em Rust foi extremamente parecida com a versão em Java. No seu caso foi utilizado essencialmente "Iterators", que devido à propriedade de *Zero-Cost Abstractions* do Rust, tem a mesma performance que formas imperativas com laço For.

```

#[derive(Default)]
pub struct KmeansSerialBuilder;

impl Kmeans for KmeansSerialBuilder {
    fn execute<'a>(&self, data: &'static Vec<Point>, _k: u8, initial_centers: Vec<Point>,) -> Vec<Cluster<'a>> {
        let mut clusters = initial_centers
            .into_iter()
            .map(|center| Cluster::from_center(center))
            .collect<'>(<Vec<Cluster>>());

        loop {
            clusters = common::assign_points(data, clusters);

            let new_centers: Vec<Point> = common::calculate_new_centers(&clusters);
            let old_centers: Vec<_> = clusters.iter().map(|cluster| &cluster.center).collect();

            if common::converged(new_centers.iter(), old_centers) {
                return clusters;
            }

            clusters = new_centers
                .into_iter()
                .map(|center| Cluster::from_center(center))
                .collect();
        }
    }
}

```

## 2.2 Resultados

Como dito na introdução, para os testes foi usado uma mesma base de dados de 1.1GB, com um K = 5 e ponto iniciais fixados iguais.

Java	Serial	214,424
Java	Serial com Stream	229,498
Rust	Serial	102,42

## 3. Implementação Concorrente - Abordagem com Join

### 3.1 Descrição

Para todas as classes de implementação concorrente foi definido campos da classes para serem usados como parâmetros de execução.

```
public class KmeansParallelEachThread implements KmeanStrategy {  
    public int threads = 1;  
  
    public ThreadMode mode = ThreadMode.PLATFORM;  
  
    public KmeansParallelEachThread(ThreadMode mode, int threads) {  
        this.mode = mode;  
        this.threads = threads;  
    }  
  
    @Override  
    public List<Cluster> execute(List<Point> values, int k, List<Point> ...
```

O ThreadMode é um Enum, em que é utilizado pattern Matching dentro da função execute para atribuir o ThreadBuilder correto na criação das threads.

```
public enum ThreadMode {  
    VIRTUAL,  
    PLATFORM,  
}
```

Na função de `execute()` é feito o comando seguinte, para criar o `threadBuilder` usado.

```
var threadBuilder = switch (this.mode) {  
    case VIRTUAL -> Thread.ofVirtual();  
    case PLATAFORM -> Thread.ofPlatform();  
};
```

Desta forma, para mudar a quantidade de threads máxima para abrir na execução, e se são threads do tipo Virtual ou de Platform, não é necessário mudar a implementação, apenas as entradas dos construtores.

Todas as implementações concorrentes seguintes vão se basear nessa mesma estrutura.

Como explicado na introdução 1.1, o código concorrente reside no passo 2. No cálculo de distância dos pontos de entrada com as centróides e atribuição dos pontos para seus clusters.

Além disso, há uma dependência de finalização de cada iteração. Pois é necessário, após a atribuição dos clusters de todos os pontos os cálculos de médias de cada cluster, para então realizar a nova iteração ou encerrar.

Nessa implementação de Fork/Join, a estratégia foi basicamente, a cada nova iteração, são criadas as threads que irão realizar os cálculos de distância, cada thread percorre todos os pontos de entrada concorrente, calculando as distância de cada ponto as centroides e adicionando os pontos à lista do clusters que estão mais próximos. Como essa adição de lista não é thread safe foi necessário usar *synchronized* no *.add()* das listas. Ao fim da iteração, é feito um join de todas as threads criadas para seguir para os próximos passos do algoritmo.

```

while (true) {
    List<Thread> threads = new ArrayList<>();
    for (int i = 0 ; i < this.threads; i++) {
        int indexOfThread = i;
        Thread thread = threadBuilder.start(() -> {
            int iteration = indexOfThread;
            while (iteration < values.size()) {
                var targetPoint = values.get(iteration);
                int indexClosestCluster = KmeanCommon.getIndexClosestCluster(targetPoint, clusters);
                Cluster targetCluster = clusters.get(indexClosestCluster);
                synchronized (targetCluster) {
                    targetCluster.addPoint(targetPoint);
                }
                iteration += this.threads;
            }
        });
        threads.add(thread);
    }

    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Resto do algoritmo.....

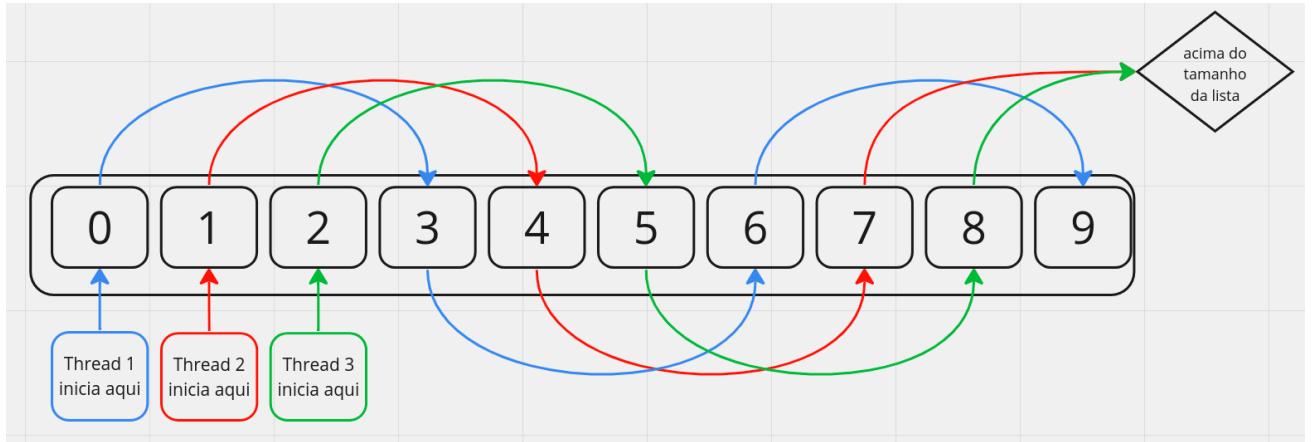
```

Essas iterações ocorrem várias vezes ao longo do programa. Logo, a função acaba criando e matando muitas threads antes de retornar efetivamente. Dado que há a garantia de encerramento de todas as threads antes dos próximos passos com o `join()`, nada mais foi necessário para a sincronização das threads. Foi utilizado apenas o `synchronized` para que a adição dos pontos nas listas de cada cluster não gere condições de corrida. O `Synchronized` nesse ponto foi inevitável em todas as implementações concorrentes seguintes.

Para percorrer a lista de pontos de entrada de forma concorrente foi adotada uma estratégia que foi repetida em todas as implementações seguintes. Baseada no index, cada threads recebe um valor de index inicial, que vai de 0 à N. Sendo N = (quantidade de threads - 1). E a cada iteração é somado à esse index o valor de N, para seguir como próximo index. Quando o index for maior que a quantidade de elementos da lista, é encerrado.

Ou seja, isso permite que todas as threads consigam percorrer a lista concorrentemente, independente.

Uma representação visual desse comportamento seria isso.



Cada Thread tem seu conjunto de números que irão acessar na iteração que não conflitam com os de outras threads. E como o acesso nesse caso será unicamente de leitura, não há condição de corrida nesses múltiplos acessos na mesma lista.

Em Rust foram feitas duas versões, uma usando a biblioteca padrão de Rust, que utiliza apenas Threads Nativas e outra versão usando a biblioteca Tokio, que oferece um Runtime com threads virtuais.

O algoritmo foi basicamente o mesmo nas duas versões, e bem parecido com o de Java. A diferença essencial foi que ao invés de alterar a lista de Clusters como foi feito com `synchronized`, em Rust foi usado passagem de mensagem com channel de `mpsc`.

Além das threads que processavam o passo 2 do algoritmo, havia uma thread para cada Cluster, que recebia por mensagem os pontos que nas outras threads definiram ele próprio como o mais próximo e outra threads para receber os cluster quando ele não tinham mais pontos para processar.

Infelizmente em Rust não há compatibilidade entre runtimes. Diferentes da abordagem feita nas implementações de Java, que pode-se alterar o tipo de threads apenas com parâmetro da classe, em Rust é necessário reescrever o código usando a outra biblioteca.

## 3.2 Resultados

Linguagem	Implementação	Modo Threads	MS
Java	Join	Plataform	397.316
Java	Join	Virtual	418.634
Rust	Join (STD)	OS Nativa	215.15
Rust	Join (Tokio)	Virtual Threads	333.45

## 4. Implementação Concorrente - Abordagem Mutex/Semáforo

### 4.1 Descrição

Para essa abordagem, na verdade, foram escritas 3 implementações. Essencialmente, todas seguem o mesmo algoritmo e tem boa parte do código idêntico, a diferença é só a estratégia de sincronização de threads.

1. Na primeira foi usado apenas Mutex
2. Na segunda apenas Semáforos
3. Foram utilizados um conjunto de Mutex e Semáforos estrategicamente onde cabiam melhor.

Diferente da abordagem usando o Join, foram criadas Threads apenas no começo da função. Essas threads criadas foram usadas em todas as iterações do algoritmo, sendo chamado o .join() apenas antes do retorno da função. Desse jeito se fez necessário estratégia para sincronizar as threads com os passos seguintes do algoritmo.

### 4.2 Mutex

No inicio da função são criadas as variáveis necessárias. Uma lista das Threads criadas para fazer .join() ao fim da função e um AtomicBoolean para sinalizar às threads que elas podem encerrar.

E para a sincronização de threads com os outros passos do algoritmo um lock para sinalizar às threads que elas podem iniciar o processamento, e um AtomicInteger que é decrementado todo momento que uma threads finaliza o processamento.

```
List<Thread> threads = new ArrayList<>();
var hasFinished = new AtomicBoolean(false);

var lockInitThreadComputation = new Object();
AtomicInteger threadsRunning = new AtomicInteger(this.threads);
```

```

for (int i = 0; i < this.threads; i++) {
    int indexOfThread = i;

    Thread thread = threadBuilder.start(() -> {
        int initialIndex = indexOfThread;

        boolean isFirstIteration = true;

        while (true) {
            int index = initialIndex;

            if (!isFirstIteration) {
                synchronized (lockInitThreadComputation) {
                    try {
                        synchronized (threadsRunning) {
                            threadsRunning.decrementAndGet();
                        }
                        lockInitThreadComputation.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            } else {
                isFirstIteration = false;
            }

            if (hasFinished.get())
                break;

            while (index < values.size()) {
                var targetPoint = values.get(index);
                int indexClosestCluster = KmeanCommon.getIndexClosestCluster(targetPoint, clusters);

                Cluster targetCluster = clusters.get(indexClosestCluster);
                targetCluster.addPointSync(targetPoint);

                index += this.threads;
            }
        }
    });
    threads.add(thread);
}

```

*From A Java concurrent Parallel with Lock*

No laço fora dessas threads, temos o resto da função.

```
while (true) {
    while (threadsRunnthreadsing.get() > 0) {
        // wait all finish
    }

    var oldCenters = new ArrayList<Point>();
    var newCenters = new ArrayList<Point>();

    for (Cluster cluster : clusters) {
        oldCenters.add(cluster.getCenter());
        newCenters.add(cluster.calculateCenterPoint());
    }

    if (KmeanCommon.converged(newCenters, oldCenters)) {
        hasFinished.set(true);

        synchronized (lockInitThreadComputation) {
            lockInitThreadComputation.notifyAll();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        return clusters;
    }

    for (int i = 0; i < newCenters.size(); i++) {
        var cluster = clusters.get(i);
        var newCenter = newCenters.get(i);
        cluster.setCenter(newCenter);
        cluster.getPoints().clear();
    }

    synchronized (lockInitThreadComputation) {
        synchronized (threadsRunning) {
            threadsRunning.set(threads.size());
        }
        lockInitThreadComputation.notifyAll();
    }
}
```

No primeiro while é verificado se (`threadsRunnthreadsing.get() > 0`), e a função é travada nesse ponto, esperando todas as threads decrementarem esse valor até chegar em zero. Quando o while é encerrado, o algoritmo segue normal como os outros passos até o final do while(true), que irá recomeçar o indicar as threads para reiniciar o processamento com:

```
synchronized (lockInitThreadComputation) {  
    synchronized (threadsRunning) {  
        threadsRunning.set(threads.size());  
    }  
    lockInitThreadComputation.notifyAll();  
}
```

Ele notifica o lock que todas as threads estão em `.wait()`. E seta o `AtomicInteger` para poder ser travado e liberado no `while(threadsRunnthreadsing.get() > 0)`.

O fim da função de fato ocorre aqui. Em que é situado o `AtomicBoolean` para que todas as threads possam encerrar, após serem liberadas com o `notifyAll()`. É feito o `.join()` para todas elas e a função é finalizada.

```
if (KmeanCommon.converged(newCenters, oldCenters)) {  
    hasFinished.set(true);  
  
    synchronized (lockInitThreadComputation) {  
        lockInitThreadComputation.notifyAll();  
    }  
  
    for (Thread thread : threads) {  
        try {  
            thread.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    return clusters;  
}
```

## 4.3 Semáforos

Como mencionado, a única diferença nesta implementação são as primitivas e a estratégia de sincronização.

Foram definidas essas variáveis ao invés dos Locks

```
List<Thread> threads = new ArrayList<>();
var hasFinished = new AtomicBoolean(false);

var semaphoreInitThreadComputation = new Semaphore(this.threads);
var semaphoreEndThreadComputation = new Semaphore(0);
```

Esse se tornou o código das threads, com a espera do início a partir do .acquire() do semáforo e o .release() em outro semáforo para sinalizar o seu término de iteração.

```
Thread thread = threadBuilder.start(() -> {
    int initialIndex = indexOfThread;

    while (true) {
        int index = initialIndex;

        try {
            semaphoreInitThreadComputation.acquire();

            if (hasFinished.get())
                break;

            while (index < values.size()) {
                var targetPoint = values.get(index);
                int indexClosestCluster = KmeanCommon.getIndexClosestCluster(targetPoint, clusters);

                Cluster targetCluster = clusters.get(indexClosestCluster);
                targetCluster.addPointSync(targetPoint);

                index += this.threads;
            }
            semaphoreEndThreadComputation.release();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
});
```

Esse é código de espera na main pela finalização de iteração das threads.

```
while (true) {
    try {
        semaphoreEndThreadComputation.acquire(this.threads);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

O código que inicia a iteração das threads

```
semaphoreInitThreadComputation.release(this.threads);
```

A fechamento de threads e retorno da função.

```
if (KmeanCommon.converged(newCenters, oldCenters)) {  
    hasFinished.set(true);  
    semaphoreInitThreadComputation.release(this.threads);  
  
    for (Thread thread : threads) {  
        try {  
            thread.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    return clusters;  
}
```

## 4.4 Mutex + Semáforos

Variáveis de sincronização

```
List<Thread> threads = new ArrayList<>();  
var hasFinished = new AtomicBoolean(false);  
  
var lockInitThread = new Object();  
var semaphoreEndThreadComputation = new Semaphore(0);
```

Código das threads

```

Thread thread = threadBuilder.start(() -> {
    int initialIndex = indexOfThread;
    boolean firstIteration = true;
    while (true) {
        int index = initialIndex;

        if (!firstIteration) {
            semaphoreEndThreadComputation.release();
            synchronized (lockInitThread) {
                try {
                    lockInitThread.wait();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        } else {
            firstIteration = false;
        }

        if (hasFinished.get())
            break;

        while (index < values.size()) {
            var targetPoint = values.get(index);
            int indexClosestCluster = KmeanCommon.getIndexClosestCluster(targetPoint, clusters);

            Cluster targetCluster = clusters.get(indexClosestCluster);
            targetCluster.addPointSync(targetPoint);

            index += this.threads;
        }
    }
});

```

Código de espera da main pela finalização da iteração das threads.

```

while (true) {
    try {
        semaphoreEndThreadComputation.acquire(this.threads);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}

```

Código que encerra as threads e retorna a função.

```

if (KmeanCommon.converged(newCenters, oldCenters)) {
    hasFinished.set(true);
    synchronized (lockInitThread) {
        lockInitThread.notifyAll();
    }

    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    return clusters;
}

```

Código que inicia uma nova iteração nas threads

```

synchronized (lockInitThread) {
    lockInitThread.notifyAll();
}

```

## 4.5 Rust

Em Rust, foi implementado uma versão, usando a biblioteca padrão (Threads nativas) e com Mutex. Como mencionado, Rust possui vários runtimes e bibliotecas que podem estender as funcionalidades padrões.

Na biblioteca padrão, por exemplo, ainda não existem semáforos, porém existem o CondVar, que é uma abstração similar ao lock do Java, com os métodos de .wait() e .notify().

Para guardar um mutex também existem dois tipos gerais, o Mutex e o RwLock. A diferença é que o Mutex requer uma chamada .lock() para fazer qualquer operação nele, seja escrita ou leitura, enquanto o RwLock tem os métodos de .read() e .write(). Múltiplas Threads podem ter o lock retornado de .read() ativo e utilizar o objeto simultaneamente, mas somente UMA thread pode estar com um lock ativo de .write().

## Higher-level synchronization objects

Most of the low-level synchronization primitives are quite error-prone and inconvenient to use, which is why the standard library also exposes some higher-level synchronization objects.

These abstractions can be built out of lower-level primitives. For efficiency, the sync objects in the standard library are usually implemented with help from the operating system's kernel, which is able to reschedule the threads while they are blocked on acquiring a lock.

The following is an overview of the available synchronization objects:

- **Arc**: Atomically Reference-Counted pointer, which can be used in multithreaded environments to prolong the lifetime of some data until all the threads have finished using it.
- **Barrier**: Ensures multiple threads will wait for each other to reach a point in the program, before continuing execution all together.
- **Condvar**: Condition Variable, providing the ability to block a thread while waiting for an event to occur.
- **mpsc**: Multi-producer, single-consumer queues, used for message-based communication. Can provide a lightweight inter-thread synchronisation mechanism, at the cost of some extra memory.
- **Mutex**: Mutual Exclusion mechanism, which ensures that at most one thread at a time is able to access some data.
- **Once**: Used for a thread-safe, one-time global initialization routine
- **OnceLock**: Used for thread-safe, one-time initialization of a global variable.
- **RwLock**: Provides a mutual exclusion mechanism which allows multiple readers at the same time, while allowing only one writer at a time. In some cases, this can be more efficient than a mutex.

Fonte:

<https://doc.rust-lang.org/std/sync/index.html#higher-level-synchronization-objects>

## State synchronization

The remaining synchronization primitives focus on synchronizing state. These are asynchronous equivalents to versions provided by `std`. They operate in a similar way as their `std` counterparts but will wait asynchronously instead of blocking the thread.

- **Barrier** Ensures multiple tasks will wait for each other to reach a point in the program, before continuing execution all together.
- **Mutex** Mutual Exclusion mechanism, which ensures that at most one thread at a time is able to access some data.
- **Notify** Basic task notification. `Notify` supports notifying a receiving task without sending data. In this case, the task wakes up and resumes processing.
- **RwLock** Provides a mutual exclusion mechanism which allows multiple readers at the same time, while allowing only one writer at a time. In some cases, this can be more efficient than a mutex.
- **Semaphore** Limits the amount of concurrency. A semaphore holds a number of permits, which tasks may request in order to enter a critical section. Semaphores are useful for implementing limiting or bounding of any kind.

Fonte: <https://docs.rs/tokio/latest/tokio/sync/#state-synchronization>

Na implementação em questão foi usado o `RwLock` para guardar os Clusters e usado apenas passagem de mensagem com os channel de `mpsc` de biblioteca padrão para sincronização das threads.

```
let has_finished = Arc::new(Mutex::new(false));

let (tx, rx) = mpsc::channel::<()>();
let mut tx_init_vec: Vec<mpsc::Sender<()>> = Vec::with_capacity(max_threads);
```

O código das threads

```

std::thread::spawn(move || {
    let initial_index = index_of_thread;
    loop {
        // Aguarda o mensagem da main para inicio
        rx_init.recv().unwrap();

        if *has_finished.lock().unwrap() {
            break;
        }

        let mut index = initial_index;
        while index < data.len() {
            let point = data.get(index).unwrap();

            let ind_closest_cluster = {
                let clusters_centers = clusters
                    .iter()
                    .map(|lock| lock.read().unwrap().center.clone())
                    .collect::<Vec<_>>();
                get_closest_cluster_index_based_in_centroids(point, clusters_centers.iter())
            };

            {
                clusters.get(ind_closest_cluster).unwrap().write().points.push(point);
            }

            index += max_threads;
        }
    }

    // Mensagem de encerramento do processamento
    tx.send(()).unwrap();
});
}

```

Por alguma razão essa foi a versão mais lenta de todos os testes, de todas as abordagens. Não foi possível encontrar com certeza a razão específica da lentidão, mas o melhor palpite tenha sido o uso do RwLock, que devido sua estratégia de gerência de lock, acabou nesse cenário gerando muito tempo de lock entre as threads. Pois quando as threads desejavam escrever num Lock, teriam que esperar todas as outras lendo esse mesmo valor terminar e fechar seus locks. Fazendo que todas as threads de escritas precisassem esperar todas as de leitura finalizar, e só então escreverem, e ainda de forma sequencial, visto que só uma poderia escrever por vez.

## 4.6 Resultados

Linguagem	Implementação	Modo Threads	MS
Java	Lock	Plataform	398.954
Java	Lock	Virtual	394.729
Java	Semaphore	Plataform	407.670

Java	Semaphore	Virtual	405.575
Java	Lock And Semaphore	Plataform	405.089
Java	Lock And Semaphore	Virtual	392.168
Rust	Mutex	OS Nativa	+900

## 5. Implementação Concorrente - Abordagem Volátil

### 5.1 Descrição

Dada a natureza do algoritmo, essa implementação basicamente é uma extensão da implementação utilizando apenas Lock. A diferença é a variável de contagem de threads que finalizaram as iterações. Ao invés de uma AtomicInteger na stack de função, é uma *volatile* na classe de execução. Dessa forma, não é necessário usar o *synchronized* no incremento dela. Apenas é chamado o método de incremento da variável e no while de espera é usado um get normal.

```
public class KmeansParallelVolatile implements KmeanStrategy {
    public int threads = 1;

    public ThreadMode mode = ThreadMode.PLATFORM;

    private volatile AtomicInteger threadsRunning = new AtomicInteger(0); You

    public KmeansParallelVolatile(ThreadMode mode, int threads) {
        this.mode = mode;
        this.threads = threads;
    }

    @Override
```

O resto do código é exatamente o mesmo da implementação usando apenas no lock no último tópico, com a exceção que é mais usado o synchronized para o threadsRunning para as operações de decrementAndGet() ou set().

Detalhe que foi necessário utilizar o AtomicInteger porque as operações de decrement dependem do valor atual da variável. Sendo assim, se threadsRunning fosse um `volatile int` as operações como `threadsRunning -=1` ou `threadsRunning--`, geraram condição de corrida.

Em Rust, a segurança de threads é tratada de maneira diferente. O compilador de Rust permite que várias threads tenham acesso a uma mesma variável, desde que ela seja marcada como imutável. No caso contrário, o compilador garante que apenas uma thread por vez tem acesso de escrita para uma variável mutável. Caso seja necessário um comportamento de leitura e escrita de um mesmo valor de memória entre Thread em Rust, só é possível alcançá-lo usando tipos de dados seguros específicos, como `Arc<Mutex<T>>` ou `Arc<RwLock<T>>`, que são basicamente mutexes, que garantem via lock a leitura e escrita segura dos dados.

No caso do `Mutex<T>` ele necessita de uma chamada de `.lock()` para travar a thread atual para usar o valor. Seja para ler ou escrever. O `RwLock` porém tem métodos para abrir um lock de leitura ou escrita, com `.read()` ou `.write()`. Locks de leitura podem ser abertos à vontade por quaisquer threads em conjunto, porém, no caso de lock de escrita, só uma terá acesso à memória enquanto possuir esse lock, bloqueando qualquer outra de ler enquanto ele não devolver o lock de volta.

Linguagem	Implementação	Modo Threads	MS
Java	Volatile	Plataform	396.226
Java	Volatile	Virtual	384.998

## 6. Implementação Concorrente - ParallelStream

### 6.1 Descrição

Essa é uma implementação adicional. Feita só como teste da funcionalidade de `ParallelStream`. Ela não emergiu de Java mas sim implementando em Rust. Utilizando um runtime chamado Rayon, foi possível criar um código baseado apenas em iteradores. Essa biblioteca de Rust abstraía a iteração paralela em threads alterando apenas a chamada de iteradores. Ao invés de chamar `.iter()` chamamos `par_iter()`. Similar aos Streams do Java. Por debaixo dos panos transformava em iteradores paralelos, que aproveitam o máximo da CPU em questão.

```

clusters = data
    .par_iter()
    .map(|point| {
        let index = common::get_closest_cluster_index(point, &clusters);
        let centroid = &clusters[index].center;
        (centroid, point)
    })
    .fold_with(
        BTreeMap::<&Point, Vec<&Point>>::default(),
        |mut acc, (centroid, point)| {
            acc.entry(centroid).or_insert_with(Vec::new).push(point);
            acc
        },
    )
    .reduce(
        || BTreeMap::<&Point, Vec<&Point>>::default(),
        |mut map1, map2| {
            for (key, values) in map2 {
                map1.entry(key).or_insert_with(Vec::new).extend(values);
            }
            map1
        },
    )
    .into_iter()
    .map(|(k, v)| Cluster {
        center: k.clone(),
        points: v,
    })
    .collect();

```

Nesse caso, o map() retorna o ponto e a centroide do cluster q ele está mais próximo. O fold\_with() e reduce() seguinte são para agrupar os pontos do cluster e em seguida exportar isso em uma struct de Cluster no último .map().

Em busca de algo similar em Java foi encontrado os ParallelStream, similar às streams normais, mas executando as iterações em paralelo. Com isso, acabei tentado utilizar apenas como extra e acabei conseguindo um código bem mais enxuto e com resultado consideravelmente melhores que os outros.

```

public class KmeansParallelStream implements KmeanStrategy {
    @Override
    public List<Cluster> execute(List<Point> values, int k, List<Point> initialCenters) {
        List<Cluster> clusters = initialCenters.stream().map(Cluster::build_with_center).toList();

        while (true) {
            List<Cluster> finalClusters = clusters;
            values.parallelStream().forEach(point -> {
                var index = KmeanCommon.getIndexClosestCluster(point, finalClusters);
                finalClusters.get(index).addPointSync(point);
            });

            var newCenters = clusters.parallelStream().map(Cluster::calculateCenterPoint).toList();
            var oldCenters = clusters.parallelStream().map(Cluster::getCenter).toList();

            if (KmeanCommon.converged(newCenters, oldCenters)) {
                return clusters;
            }

            clusters = newCenters.parallelStream().map(Cluster::build_with_center).toList();
        }
    }
}

```

Essa implementação realmente foi uma avaliação e trabalho adicional, apenas por curiosidade. Visto que eu havia implementado em Rust e busquei uma alternativa similar em Java.

Linguagem	Implementação	Modo Threads	MS
Java	ParallelStream	-	242.531
Rust	Rayon (parallel iterators)	Plataform	47.510

## 7. Discussão

Esse é o resultado de todos os testes realizados...

Como comentado antes foi usado uma base de dados de 1GB usando o valor de entrada de K = 5. Para ambos os casos foi usado o comando `time` do linux na execução dos executáveis.

No caso do Java...

\$ time java -jar Main.class

No caso do Rust...

\$ time ./binario-compilado

Foi executado o mesmo comando duas vezes para todos os testes e usado o menor valor entre os dois.

Linguagem	Implementação	Modo Threads	MS
*Java	Adder	Plataform	46
Rust	Rayon (parallel iterators)	Plataform	48
*Java	Adder Confinement	Plataform	51
Rust	Serial	-	102,42
Java	Serial	-	214,424
Rust	Join (STD)	OS Nativa	215,15
Java	Serial com Stream	-	229,498
Java	ParallelStream	-	243
Rust	Join (Tokio)	Green Threads	333,45
Java	Volatile	Virtual	384,998
Java	Lock And Semaphore	Virtual	392
Java	Lock	Virtual	395
Java	Volatile	Plataform	396
Java	Join	Plataform	397
Java	Lock	Plataform	399
Java	Lock And Semaphore	Plataform	405
Java	Semaphore	Virtual	406
Java	Semaphore	Plataform	408
Java	Join	Virtual	419
Rust	Mutex	OS Nativa	+ 900

Nesse teste ficou visível que as diferentes abordagens concorrentes em Java tiveram muito pouco impacto real. Visto que a diferença percentual se mostrou baixa. Entre a abordagem mais rápida (384 ms) e a mais lenta (419 ms) é de 8%. Sendo esse um percentual muito baixo de ganho, comparado, por exemplo, com as diferenças das implementações em Rust.

As diferenças de desempenho entre Virtual e Platform Threads em Java, nesses testes, não mostraram com exatidão a mais vantajosa ou uma diferença forte entre elas. Pela razão exposta logo antes. Testes em cenários e ambientes, ou usando implementações que usassem as diferentes técnicas em outros lugares do algoritmo poderiam destacar o diferencial de cada tipo de ferramentas de lock e tipos de Threads.

O mesmo não pode ser dito das implementações de Rust. É visível a diferença entre a utilização de thread nativas com a biblioteca padrão e as green threads com o runtime Tokio. Havendo uma diferença de 43%. A razão pra essa diferença pode se dar pelos códigos serem bem distintos. Como foi necessário usar outras bibliotecas e componentes diferentes para cada implementação, pouco se reutilizou de cada implementação. Exceto a própria lógica e algoritmo. No fim, a abordagem com Rayon e iteradores paralelos se mostrou superior em todos os aspectos comparada à todas as abordagens testadas.

Mas com certeza o problema mais destacado é o fato das implementações seriais, no fim, serem mais rápidas que boa parte das implementações paralelas. O que mostra que provavelmente há problemas de concorrência em boa parte dos códigos feitos que devem ser vistos e tratados com ferramentas mais adequadas para o problema do algoritmo em próximos testes.

Como hipótese principal, acredito que a principal razão desse gargalo é a contenção causada pelo *synchronized* usado na função de adicionar pontos aos clusters executado em todas as implementações...

```
while (_index < _values.size()) {
    var targetPoint = _values.get(_index);
    int indexClosestCluster = KmeanCommon.getIndexClosestCluster(targetPoint, _clusters);

    Cluster targetCluster = _clusters.get(indexClosestCluster);
    synchronized (targetCluster) {
        targetCluster.addPoint(targetPoint);
    }

    _index += this._threads;
}
```

No passo de separação de pontos para cada cluster foi feito via armazenamento, usando uma lista (não thread-safe) e fazendo um *.add()* de cada ponto para seu cluster respectivo. Devido isso, foi necessário esse *synchronized* nessa adição, em todas as implementações.

Apesar dos cálculos serem realizados de forma paralela, toda a adição de elementos se tornou serial. E como foi usado um valor baixo de K para os testes, a contenção e aglomeração ficou ainda maior do que se fosse usado valores maiores

de K, que iriam dividir a contenção do *synchronized* em mais listas, dado que teriam mais clusters para os pontos se associarem.

Dado isso, fica claro que próximas implementações devem achar outras abordagem que descarte esse *synchronized* e evite a contenção de pontos para cada clusters. Além disso buscar outros problemas que podem haver nas implementações que causem esse gargalo.

# Segunda Unidade

## 8. Implementação Concorrente - Adder

### 8.1 Descrição

Dada a análise feita na seção passada (Discursão), para os códigos da segunda unidade foi realizado uma outra abordagem para o algoritmo de cálculo do Kmeans: Ao invés de uma algoritmo baseado em separar pontos entre listas (o que resulta inevitavelmente no uso de um synchronized para adicionar nas listas) foi feito uma abordagem de acumulo de inteiros. Somando os valores dos pontos para no final obter suas médias. Nesse caso, ao invés da adição na lista, é feita uma operação de soma em algum valor, para no final calcular a média.

O uso do Adder foi perfeito para a adoção dessa abordagem, dado seu propósito, já que é possível acumular os valores, realizando a soma no final e dividir pela quantidade (fixa) dos elementos. Para isso foi criado uma classe auxiliar para compartilhar entre as threads. Ela serve para acumular a soma dos valores dos pontos e quantidade de valores acumulados.

```
Codeium: Refactor | Explain
private class ClusterAccumulator {
    public LongAdder accX;
    public LongAdder accY;
    public LongAdder accZ;
    public LongAdder couting;
}
```

E para as threads, foi criado um runnable que possui a lista de ClusterAccumulator para realizar os .add() do LongAdder. Como todas as lista só possuem operações de ReadOnly e ClusterAccumulator só possui campos thread-safe, então a classe auxiliar é Thread-Safe.

```

Codeium: Refactor | Explain
private class ThreadRunner implements Runnable {
    private int initialIndex;
    private int intervalIndex;
    private List<Point> values;
    private List<Point> centroids;
    private List<ClusterAccumulator> clusterAccumulators;

    Codeium: Refactor | Explain | Generate Javadoc | ×
    @Override
    public void run() {
        for (int i = this.initialIndex; i < this.values.size(); i += this.intervalIndex) {
            Point targetPoint = values.get(i);
            int clusterIndex = KmeanCommon.getIndexClosestCentroid(targetPoint, this.centroids);
            var clusterAccumulator = this.clusterAccumulators.get(clusterIndex);
            clusterAccumulator.accX.add(targetPoint.getX());
            clusterAccumulator.accY.add(targetPoint.getY());
            clusterAccumulator.accZ.add(targetPoint.getZ());
            clusterAccumulator.couting.increment();
        }
    }
}

```

A abordagem para criação de threads foi a mesma das implementações passadas, mudando apenas o Runnable usado.

## 8.2 Avaliação Microbenchmark

Para essa e todas as abordagens seguinte os testes com o JMH seguiu a exata mesma estrutura, com um teste verificando o tempo médio de processamento:

```

@Benchmark
@Fork(value = 1)
@Warmup(iterations = 1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.SECONDS)
public void adder(Blackhole bh) throws IOException {
    var K = this.K;
    var values = BenchTest.getInputValues();
    var initialCenters = this.extractInitialCenters(K);
    var threadMode = ThreadMode.PLATFORM;
    var output = new KmeansAdder(threadMode, thiscores).execute(values, K, initialCenters);
    bh.consume(output);
}

```

Outra coisa importante é que o valor de K foi de 25, ao invés de 5, como os testes da primeira unidade, que forçam bem mais a concorrência do algoritmo, visto que há mais valores para dividir os pontos e calcular a média.

Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.adder	25	8	avgt	5	87,088 ± 192,871	s/op	

## 8.3 Avaliação de Profile

Para todos as outras abordagens abaixo, foi feito o profile com o uso do JMC, ou seja, o testes de cima. Extraiendo dez minutos.

Para a alocação mostrou-se a alocação intensa dos valores de inteiros, dos pontos calculados e Labels (as listas de String). Nada anormal.

Class	Alloc Total	Total Allocation (%)
int[]	8,17 GiB	38,8 %
java.lang.Object[]	3,84 GiB	18,2 %
byte[]	3,36 GiB	16 %
java.lang.String	2,6 GiB	12,3 %
java.lang.String[]	1,06 GiB	5,04 %
java.util.regex.IntHashSet[]	1,01 GiB	4,79 %
imd.ufn.br.entities.Point	832 MiB	3,86 %
java.util.Optional	91,2 MiB	0,423 %
char[]	37,7 MiB	0,175 %
java.util.concurrent.Concurrent	32,6 MiB	0,151 %

Para o GC, a porcentagem se mostrou bem baixa, e com alguma pausas durante a execução do algoritmo, mas com um padrão, da grande maioria das pausa de GC no começo do programa, ou seja, na leitura do arquivo e não no processamento do algoritmo em si.



GC Summary	
<b>Young Collection Total Time</b>	<b>Old Collection Total Time</b>
GC Count 46	GC Count 9
Average GC Time 68,604 ms	Average GC Time 1,517 s
Maximum GC Time 317,583 ms	Maximum GC Time 4,269 s
Total GC Time 3,156 s	Total GC Time 13,654 s
<b>All Collections Total Time</b>	<b>All Collections Pause Time</b>
GC Count 55	Average Pause 51,544 ms
Average GC Time 305,638 ms	Longest Pause 317,583 ms
Maximum GC Time 4,269 s	Sum of Pauses 3,299 s
Total GC Time 16,810 s	

Na média, teve-se em resumo o seguinte resultado (considerando a porcentagem de GC como o tempo total sobre o tempo de 10 minutos da captura do JFR):

Tipo	Tempo Execução (segundos)	GC time
Adder	87,088	2,80 %

# 9. Implementação Concorrente - Atomic

## 9.1 Descrição

Nessa abordagem foi feita basicamente a mesma coisa da abordagem do LongAdder, alterando apenas a classe auxiliar que contém os LongAdder, além do método de .add() do LongAdder para o método equivalente no Atomic.

```
Codeium: Refactor | Explain
private class ClusterAccumulator {
    public AtomicInteger accX;
    public AtomicInteger accY;
    public AtomicInteger accZ;
    public AtomicInteger couting;
}
```

```
@Override
public void run() {
    for (int i = this.initialIndex; i < this.values.size(); i += this.intervalIndex) {
        Point targetPoint = values.get(i);
        int clusterIndex = KmeanCommon.getIndexClosestCentroid(targetPoint, this.centroids);
        var clusterAccumulator = this.clusterAccumulators.get(clusterIndex);
        clusterAccumulator.accX.addAndGet(targetPoint.getX());      You, last month + add TCP
        clusterAccumulator.accY.addAndGet(targetPoint.getY());
        clusterAccumulator.accZ.addAndGet(targetPoint.getZ());
        clusterAccumulator.couting.incrementAndGet();
    }
}
```

Dessa forma a comparação fica bem justa, visto que realmente a diferença é apenas a primitiva usada.

## 9.2 Avaliação Microbenchmark

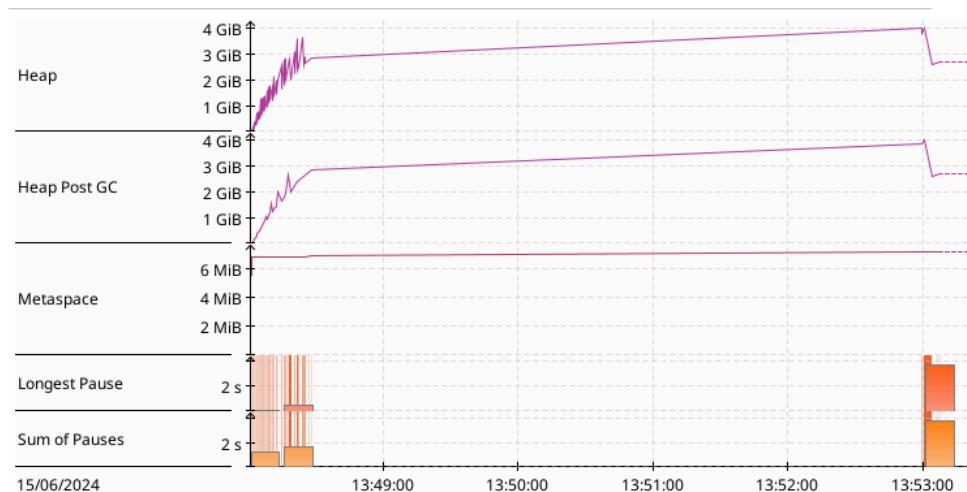
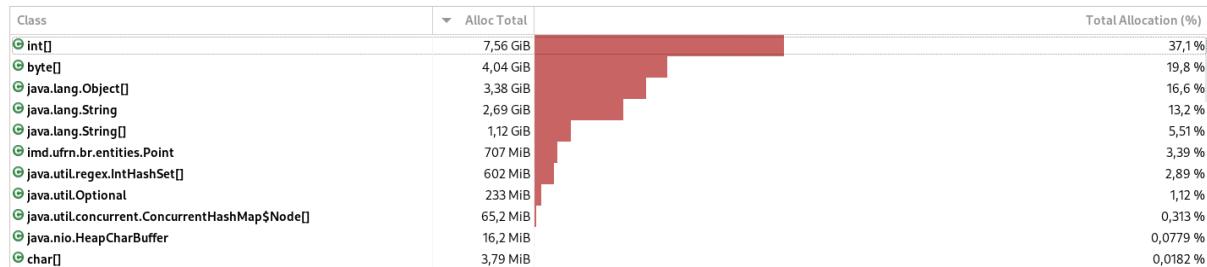
Foi feita o mesmo padrão de teste da última implementação mostrada:

```
@Benchmark
@Fork(value = 1)
@Warmup(iterations = 1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.SECONDS)
public void atomic(Blackhole bh) throws IOException {
    var K = this.K;
    var values = BenchTest.getInputValues();
    var initialCenters = this.extractInitialCenters(K);
    var threadMode = ThreadMode.PLATAFORM;
    var output = new KmeansAtomic(threadMode, thiscores).execute(values, K, initialCenters);
    bh.consume(output);
}
```

Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.atomic	25	8	avgt	5	179,025 ± 425,706		s/op

## 9.3 Avaliação de Profile

Os resultados de profiles teve pontos iguais ao LongAdder, mas um diferença nítida foi o tempo de GC.



GC Summary	
<b>Young Collection Total Time</b>	
GC Count	39
Average GC Time	80,432 ms
Maximum GC Time	522,236 ms
Total GC Time	3,137 s
<b>Old Collection Total Time</b>	
GC Count	7
Average GC Time	2,194 s
Maximum GC Time	5,669 s
Total GC Time	15,355 s
<b>All Collections Total Time</b>	
GC Count	46
Average GC Time	402,000 ms
Maximum GC Time	5,669 s
Total GC Time	18,492 s
<b>All Collections Pause Time</b>	
Average Pause	135,918 ms
Longest Pause	3,849 s
Sum of Pauses	7,068 s

Tipo	Tempo Execução (segundos)	GC time
------	---------------------------	---------

Atomic	179,025	3,08 %
--------	---------	--------

Apesar de uma porcentagem maior de GC, pelo gráfico de pausa é nítido que as únicas pausas foram da leitura do arquivo inicialmente e no final do processamento. Durante a função de cálculo concorrente de fato, teve-se nenhuma pausa, apenas no final, que de fato resultou uma pausa muito longa. Mas durante o algoritmo concorrente teve-se um GC pause zerado.

Apesar disso, se mostrou que o Atomic perde consideravelmente em comparação com o LongAdder, com quase o dobro de tempo de execução.

## 10. Implementação Concorrente - Executor

### 10.1 Descrição

Com os resultados passados do LongAdder e Atomic, para todas as abordagens seguintes foi usado o mesmo algoritmo de Adder como base, alterando apenas a estratégia de criação e gerenciamento das threads que realizam a acumulação dos valores dos pontos.

Inicialmente foi criada um executor do FixedThreadPool

```
ExecutorService executorService = Executors.newFixedThreadPool(this.threads);
```

Foi usado também Futures, uma lista de Futures que recebem cada uma das Futures retornadas na criação de novas tasks.

```
ExecutorService executorService = Executors.newFixedThreadPool(this.threads);
List<Future<?>> futures = new ArrayList<>();
```

O código para criação de threads e gerenciamento foi esse:

```
clusterAccumulators.forEach(ClusterAccumulator::reset);
for (int futureIndex = 0 ; futureIndex < this.threads; futureIndex++) {
    int finalFutureIndex = futureIndex;
    List<Point> finalCentroids = centroids;
    Future<?> future = executorService.submit(() -> { You, 3 weeks ago • refactor ThreadRunner
        for (int i = finalFutureIndex; i < values.size(); i += this.threads) {
            Point targetPoint = values.get(i);
            int clusterIndex = KmeanCommon.getIndexClosestCentroid(targetPoint, finalCentroids);
            var clusterAccumulator = clusterAccumulators.get(clusterIndex);
            clusterAccumulator.addPoint(targetPoint);
        }
    });
    futures.add(future);
}

for (var future: futures) {
    try {
        future.get();
    } catch (InterruptedException | ExecutionException e) {
        throw new RuntimeException(e);
    }
}
```

Do you want to investigate?

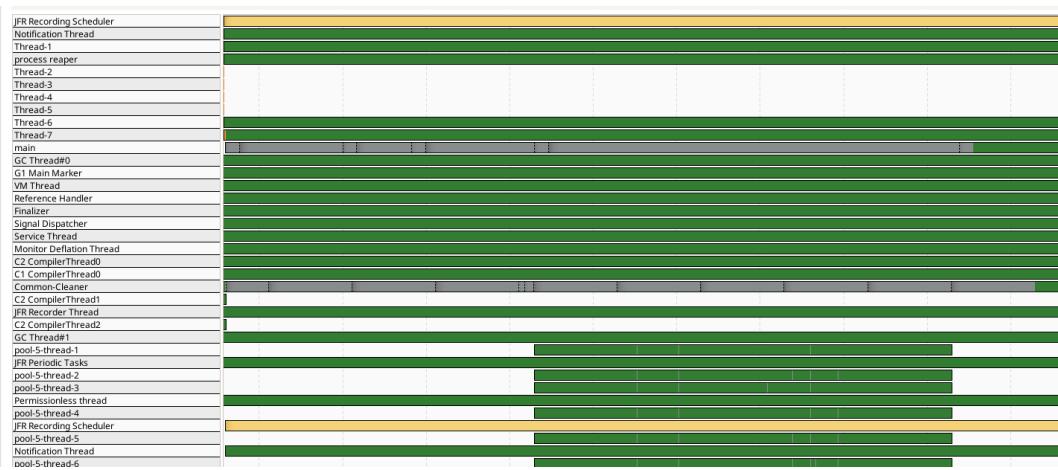
O resto seguiu o mesmo do código da abordagem do Adder.

## 10.2 Avaliação Microbenchmark

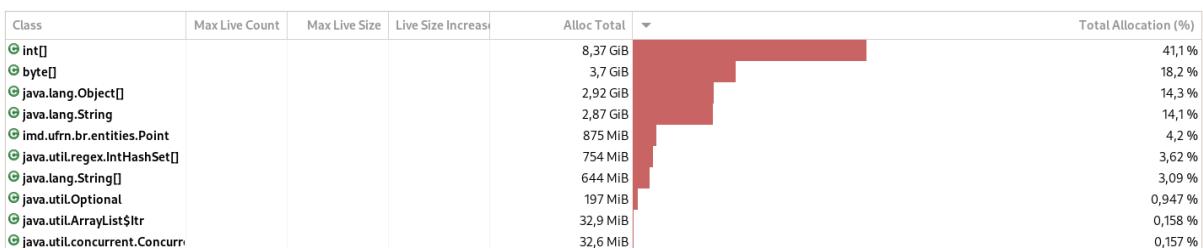
Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.adder_executor	25	8	avgt	5	219,639	± 534,180	s/op

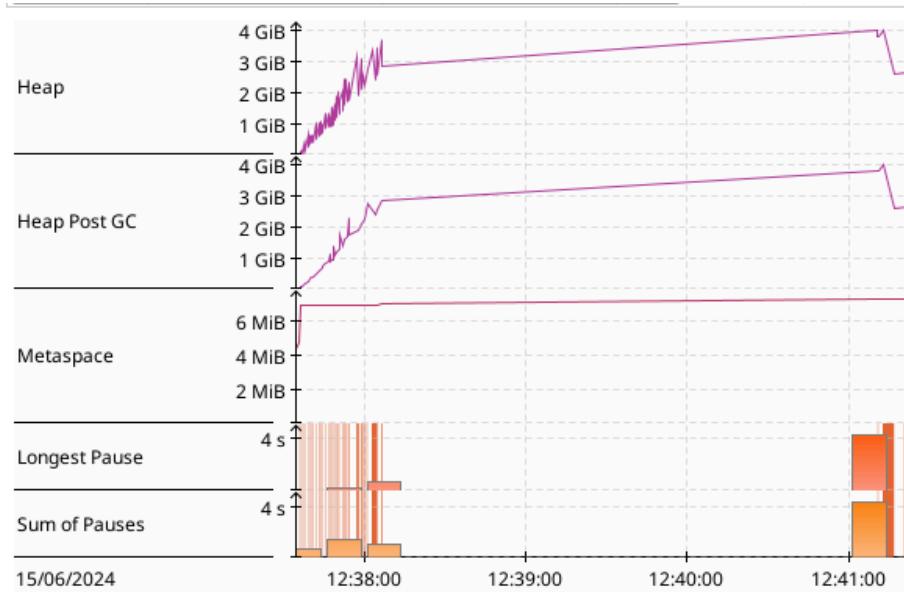
## 10.3 Avaliação de Profile

Dado o uso do Executor foi possível ter uma visualização melhor da contenção das threads no Profile. O que antes era impossível, devido a criação de varias threads em cada iteração, que acarretava em centenas de threads durante a vida do programas.



Foi possível observar a baixissima contenção do algoritmo.





GC Summary	
<b>Young Collection Total Time</b>	
GC Count	44
Average GC Time	82,353 ms
Maximum GC Time	762,760 ms
Total GC Time	3,624 s
<b>Old Collection Total Time</b>	
GC Count	9
Average GC Time	2,174 s
Maximum GC Time	6,228 s
Total GC Time	19,565 s
<b>All Collections Total Time</b>	
GC Count	53
Average GC Time	437,519 ms
Maximum GC Time	6,228 s
Total GC Time	23,188 s
<b>All Collections Pause Time</b>	
Average Pause	141,715 ms
Longest Pause	4,357 s
Sum of Pauses	8,078 s

Tipo	Tempo Execução (segundos)	GC time
Adder Executor	60,056	2,86 %

O comportamento de GC e os valores alocados se mantiveram iguais, com um GC pause durante o processamento zerado.

As vantagens foram nítidas, o consumo de memoria ficou quase idêntico, a desalocação pelo GC foi mínima e o tempo total foi extremamente reduzido, provavelmente graças ao tempo de criação e desalocação de threads que no caso do teste do Adder eram feito a cada iteração. E no executor, se mantiveram as mesmas em toda a execução.

# 11. Implementação Concorrente - ForkJoin

## 11.1 Descrição

Dada a natureza do ForkJoin foi necessário um algoritmo diferente para o cálculo e uso do ClusterAccumulator.

Dado que o retorno do processamento é dado pelos ClusterAccumulator compartilhado pelas threads, e que ao fim de todos os processamentos é usado o .sum() do LongAdder para obter esse valor, foi usado o RecursiveAction para a task passada para a instância do ForkJoin.

```
public class MeanTask extends RecursiveAction {    You, last month • Add ForkJoin Impl
    private static final int THRESHOLD = 20;

    private final List<Point> centroids;
    private final List<ClusterAccumulator> accumulators;
    private final List<Point> values;

    public MeanTask(List<Point> centroids, List<ClusterAccumulator> accumulators, List<Point> values) {
        this.centroids = centroids;
        this.accumulators = accumulators;
        this.values = values;
    }

    @Override
    protected void compute() {
        if (this.values.size() <= THRESHOLD) {
            for (Point value : values) {
                int clusterIndex = KmeanCommon.getIndexClosestCentroid(value, this.centroids);
                var clusterAccumulator = this.accumulators.get(clusterIndex);
                clusterAccumulator.addPoint(value);
            }
        } else {
            int half = this.values.size() / 2;
            List<Point> left = this.values.sublist(0, half);
            List<Point> right = this.values.subList(half, this.values.size());

            // Create subtasks for each half
            MeanTask leftTask = new MeanTask(this.centroids, this.accumulators, left);
            MeanTask rightTask = new MeanTask(this.centroids, this.accumulators, right);

            // Fork the subtasks
            leftTask.fork();
            rightTask.fork();

            // Join the subtasks and combine results
            leftTask.join();
            rightTask.join();
        }
    }
}
```

O .addPoint(), que consiste no cálculo principal da concorrência é um método da classe auxiliar que foi refeita de forma mais segura.

```

private class ClusterAccumulator {
    public final LongAdder accX;
    public final LongAdder accY;
    public final LongAdder accZ;
    public final LongAdder couting;

    public ClusterAccumulator() {
        this.accX = new LongAdder();
        this.accY = new LongAdder();
        this.accZ = new LongAdder();
        this.couting = new LongAdder();
    }
    public void reset() {
        this.accX.reset();
        this.accY.reset();
        this.accZ.reset();
        this.couting.reset();
    }
    public void addPoint(Point point) {
        this.accX.add(point.getX());
        this.accY.add(point.getY());
        this.accZ.add(point.getZ());
        this.couting.increment();
    }
    public Point getMean() {
        long size = this.couting.sum();
        if (size == 0)
            return new Point(0,0,0);

        long meanX = this.accX.sum() / size;
        long meanY = this.accY.sum() / size;
        long meanZ = this.accZ.sum() / size;
        return new Point((int)meanX, (int)meanY, (int)meanZ);
    }
}

```

## 11.2 Avaliação Microbenchmark

Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.adder_forkjoin	25	8	avgt	5	95,887 ± 11,026		s/op

## 11.3 Avaliação de Profile



GC Summary	
<b>Young Collection Total Time</b>	<b>Old Collection Total Time</b>
GC Count 170	GC Count 56
Average GC Time 42,532 ms	Average GC Time 3,544 s
Maximum GC Time 402,923 ms	Maximum GC Time 7,036 s
Total GC Time 7,230 s	Total GC Time 3 min 18 s
<b>All Collections Total Time</b>	<b>All Collections Pause Time</b>
GC Count 226	Average Pause 31,867 ms
Average GC Time 910,115 ms	Longest Pause 402,923 ms
Maximum GC Time 7,036 s	Sum of Pauses 8,986 s
Total GC Time 3 min 26 s	

Tipo	Tempo Execução (segundos)	GC time
Adder ForkJoin	95,887	34,33 %

A principal perda visivel foi o GC time e as pausas durante a execução, que pelo gráfico foram numerosas durante a execução. Devido ao objetos criados pela abordagem recursiva do ForkJoin, visíveis nos objetos alocados. Com um total de 80GB apenas das Tasks recursivas necessárias se alocar. Além de que o tempo médio teve o pior resultado das abordagens até agora usando o Adder.

## 12. Implementação Concorrente - Parallel Stream

### 12.1 Descrição

Essa foi a implementação mais simples de todas. Todo o setup, criação, sincronização e gerência de threads foi realizado nessas poucas linhas. A classe auxiliar se manteve mas não foi necessário nenhum Runnable.

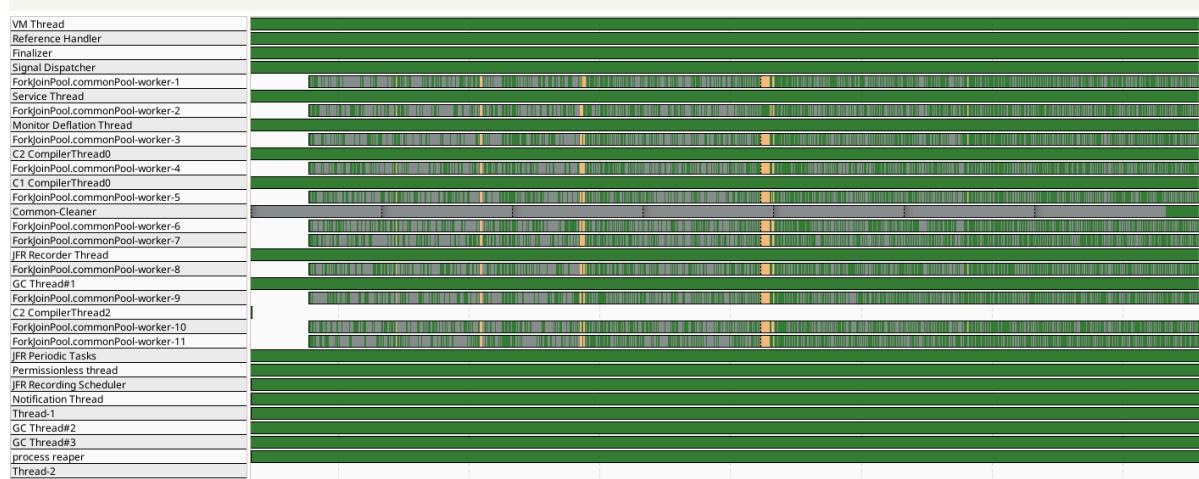
```
List<Point> finalCentroids = centroids;
values.parallelStream()
    .forEach(point -> {
        int clusterIndex = KmeanCommon.getIndexClosestCentroid(point, finalCentroids);
        var clusterAccumulator = clusterAccumulators.get(clusterIndex);
        clusterAccumulator.addPoint(point);
    });
}
```

### 12.2 Avaliação Microbenchmark

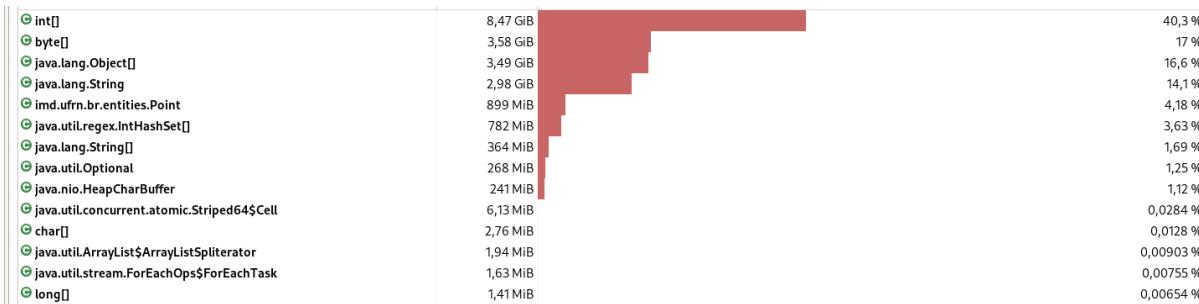
Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.adder_parallel_stream	25	8	avgt	5	73,841 ± 112,057		s/op

## 12.3 Avaliação de Profile

Assim como o ForkJoin percebeu-se uma contenção inerente aos métodos de ForkJoin.



Mas um diferença do ForkJoin foi que não foi alocado de forma intensa as task recursivas como foi lá. Se mantendo a mesma alocação de objetos das outras abordagens. Os que são inerentes ao Parallel Stream tiveram apenas alguns MB de alocação.



Isso acabou resultando em pausas mínimas do GC durante o algoritmo.



GC Summary	
<b>Young Collection Total Time</b>	<b>Old Collection Total Time</b>
GC Count 41	GC Count 7
Average GC Time 80,856 ms	Average GC Time 2,501 s
Maximum GC Time 515,481 ms	Maximum GC Time 5,497 s
Total GC Time 3,315 s	Total GC Time 17,509 s
<b>All Collections Total Time</b>	<b>All Collections Pause Time</b>
GC Count 48	Average Pause 130,841 ms
Average GC Time 433,827 ms	Longest Pause 3,670 s
Maximum GC Time 5,497 s	Sum of Pauses 7,065 s
Total GC Time 20,824 s	

Tipo	Tempo Execução (segundos)	GC time
Adder Parallel Stream	73,841	4,75 %

Essa se mostrou uma abordagem bem interessante, apesar de não ser a melhor em tempo e nem em GC Pause, ela teve uma complexidade de mínima de implementação. Essa é a implementação com menos linhas e menos complexidades. À um custo baixo de perda de performance, com o segundo melhor tempo e a porcentagem de GC próximo das melhores.

## 13. Implementação Concorrente - Structured Conc

### 13.1 Descrição

Essa abordagem se baseou nos métodos de `.fork()` e `.join()`. Baseando-se na implementação usando de Futures do Executor, a diferença foi que ao invés de esperar os futures, apenas foi feito o `.join` para o `StructuredTaskScope`.

```

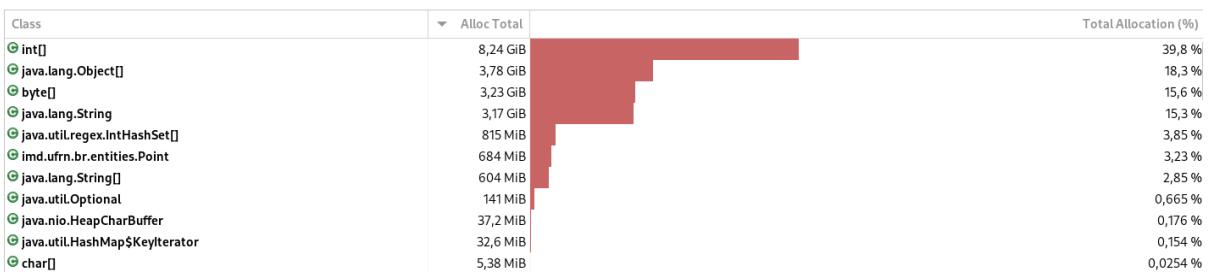
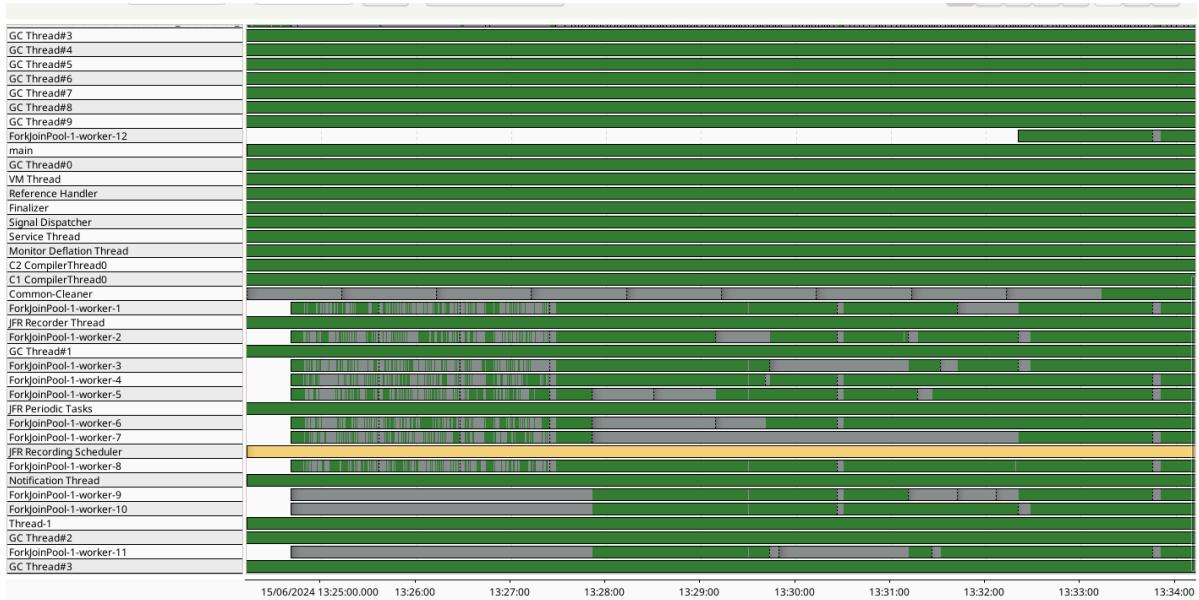
try (var taskScope = new StructuredTaskScope<>()) {
    List<ClusterAccumulator> clusterAccumulators = initialCenters.stream().map(_el -> new ClusterAccumulator()).toList();
    System.out.println("initial: " + initialCenters);
    List<Point> centroids = initialCenters;
    while (true) {
        clusterAccumulators.forEach(ClusterAccumulator::reset);
        for (int futureIndex = 0 ; futureIndex < this.threads; futureIndex++) {
            int finalFutureIndex = futureIndex;
            List<Point> finalCentroids = centroids;
            taskScope.fork(() -> {
                for (int i = finalFutureIndex; i < values.size(); i += this.threads) {
                    Point targetPoint = values.get(i);
                    int clusterIndex = KmeanCommon.getIndexClosestCentroid(targetPoint, finalCentroids);
                    var clusterAccumulator = clusterAccumulators.get(clusterIndex);
                    clusterAccumulator.addPoint(targetPoint);
                }
            });
        }
        return null;
    }]; You, 3 weeks ago • structured classes and parallel stream
}
taskScope.join();

```

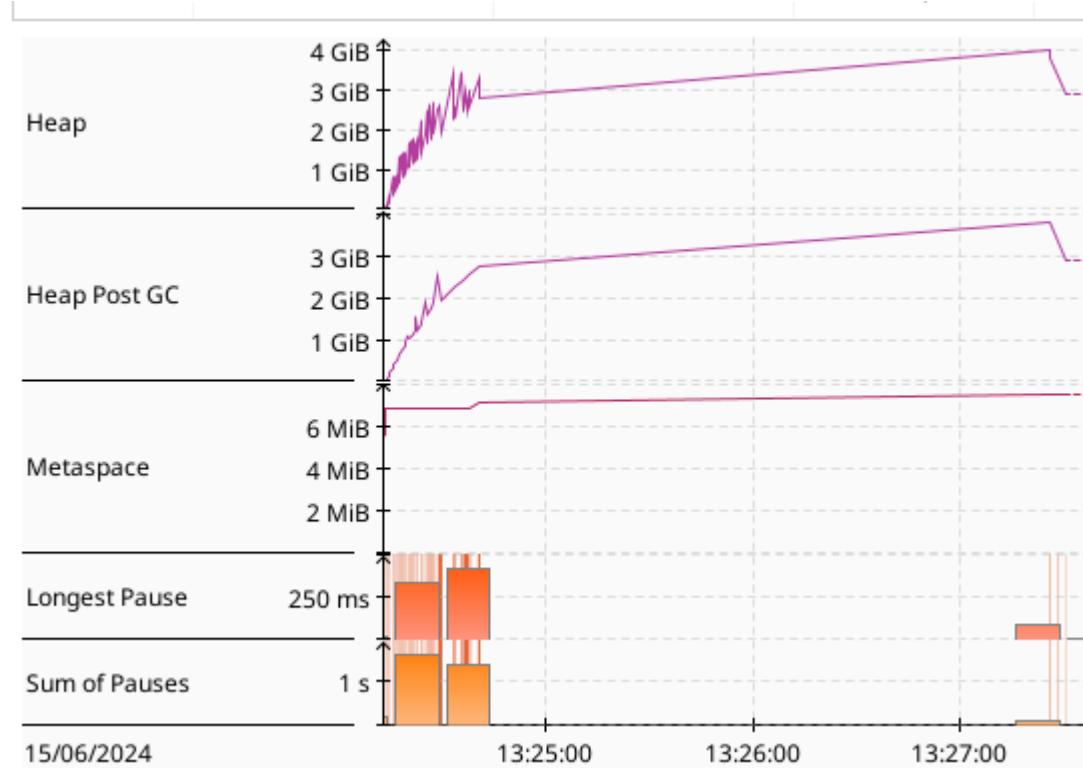
## 13.2 Avaliação Microbenchmark

Benchmark	(K)	(cores)	Mode	Cnt	Score	Error	Units
BenchTest.adder_structured_conc	25	8	avgt	5	163,541 ± 438,267	s/op	

## 13.3 Avaliação de Profile



GC Summary	
<b>Young Collection Total Time</b>	<b>Old Collection Total Time</b>
GC Count 39	GC Count 6
Average GC Time 85,998 ms	Average GC Time 2,371 s
Maximum GC Time 436,140 ms	Maximum GC Time 7,066 s
Total GC Time 3,354 s	Total GC Time 14,226 s
<b>All Collections Total Time</b>	<b>All Collections Pause Time</b>
GC Count 45	Average Pause 68,410 ms
Average GC Time 390,655 ms	Longest Pause 436,140 ms
Maximum GC Time 7,066 s	Sum of Pauses 3,489 s
Total GC Time 17,579 s	



Tipo	Tempo Execução (segundos)	GC time
Adder Strucutred Conc	163,541	2,92 %

Esse método não teve grandes diferenças quanto ao GC e alocação, o problema visível foi a contenção gerada. Que provavelmente gerou um tempo bem pior que todas as outras abordagens.

# 14. Implementação Concorrente - Concurrent Collections

## 14.1 Descrição

Essa implementação usou outra abordagem. Ao invés do Adder, retomou os algoritmos feitos na primeira unidade, onde era feito a adição de classes em listas usando o synchronized. Aqui, a diferença foi o uso da Collections.synchronizedList para essas adições

```
public List<Cluster> execute(List<Point> values, int k, List<Point> initialCenters) {
    List<Cluster> clusters = initialCenters.stream().map(p -> new Cluster(p, Collections.synchronizedList(new ArrayList<>()))).toList();
    System.out.println("initial: " + initialCenters);

    while (true) {
        final var finalClusters = clusters;
        values.parallelStream().forEach(point -> {
            var index = KmeanCommon.getIndexClosestCluster(point, finalClusters);
            finalClusters.get(index).getPoints().add(point);
        });

        List<Point> newCenter = clusters.stream().map(Cluster::calculateCenterPoint).toList();
        List<Point> center = clusters.stream().map(Cluster::getCenter).toList();

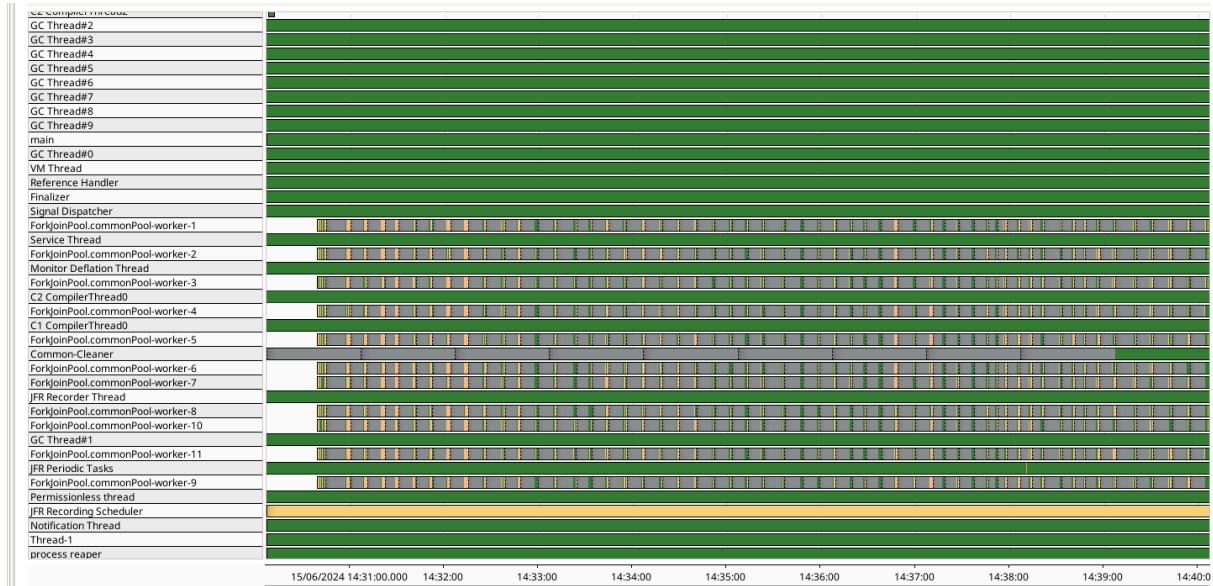
        if (KmeanCommon.converged(newCenter, center)) {
            return clusters;
        }

        clusters = newCenter.stream().map(p -> new Cluster(p, Collections.synchronizedList(new ArrayList<>()))).toList();
    }
}
```

## 14.2 Avaliação Microbenchmark

Para esse algoritmo, o JMH não foi capturado por uma simples razão, a implementação foi tão mal performática que cada iteração demorava mais que 10 minutos. Ou seja, simplesmente ordem de grandeza mais lenta e pior que as piores implementações anteriores. Não foi aguardado até o fim da execução dele, já que era visível que passaria os 600 segundos.

## 14.3 Avaliação de Profile



GC Summary	
<b>Young Collection Total Time</b>	<b>Old Collection Total Time</b>
GC Count	239
Average GC Time	36,468 ms
Maximum GC Time	840,649 ms
Total GC Time	8,716 s
<b>All Collections Total Time</b>	<b>All Collections Pause Time</b>
GC Count	306
Average GC Time	652,051 ms
Maximum GC Time	5,810 s
Total GC Time	3 min 20 s
Average Pause	28,425 ms
Longest Pause	840,649 ms
Sum of Pauses	10,546 s

Tipo	Tempo Execução (segundos)	GC time
Concurrent Collections	+600	33,3 %

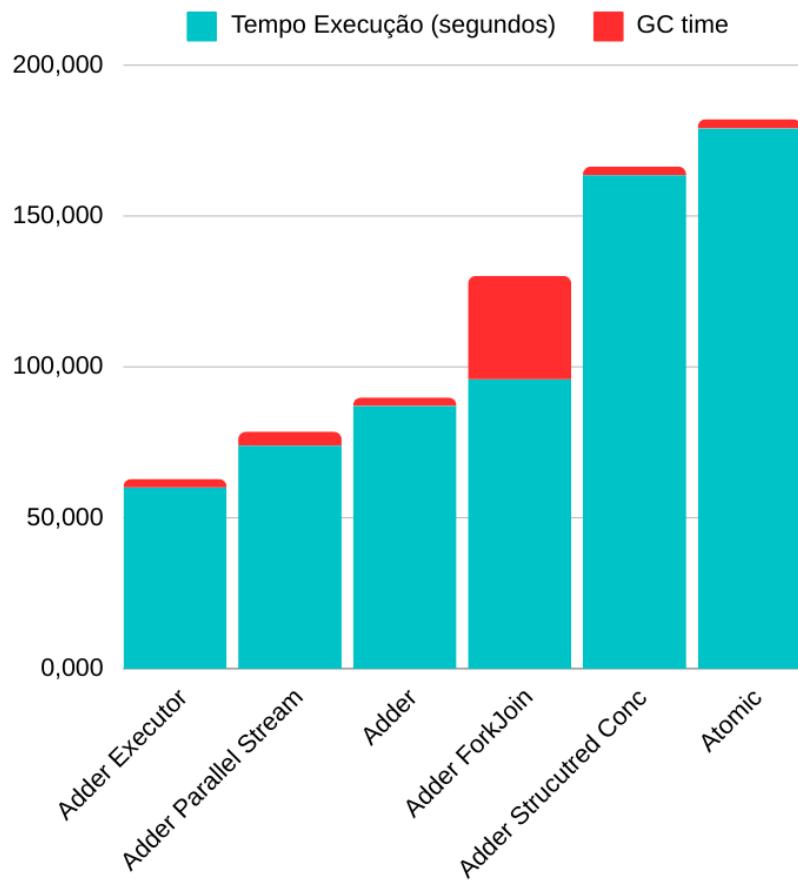
Pra essa implementação foi nítido que não cabia essa abordagem. Com um tempo de resposta e de pausa de GC extremamente alta. Mostrando que a abordagem usando GC eram muito mais ideais.

## Discussão

Esse foi o apanhado geral do sumário dos resultados do JMH e porcentagem de GC:

Tipo	Tempo Execução (segundos)	GC time
Adder Executor	60,056	2,86 %
Adder Parallel Stream	73,841	4,75 %
Adder	87,088	2,80 %
Adder ForkJoin	95,887	34,33 %
Adder Structured Conc	163,541	2,92 %
Atomic	179,025	3,08 %
Concurrent Collections	+600	33,3%
Implementação Serial	+600	**
Qualquer implementação da unidade 1	+600	**

Para melhor visualização há esse gráfico com as implementações e quantos porcentos do tempo total é de GC.



Como comentado antes, esses testes foram diferentes da unidade passada, foi usado um K de 25 (separar os dados em 25 clusters) ao inves de 5, visto que após alguns testes, se usado um K pequeno a concorrência não se justificava e abordagens seriais acabavam sendo até mais perfomaticas.

Além disso, devido a limitação do computador utilizado, o arquivo utilizado era de 700MB. Diferente da primeira unidade que foi usado sempre um arquivo de 1.1GB como comentado antes. Esse tamanho de arquivo foi o mais próximo do limite computacional testado. O tamanho máximo possível em que todos os testes conseguiram rodar sem estouro de memória da máquina.

Como mostrado na tabela, foi testado todas as implementações da primeira unidade, e devido a abordagem do algoritmo, que se baseia em adição em massa de pontos em listas na memória (usando synchronized) todas acabaram com uma perfomance péssima nesse cenário. Dessa forma, descartadas. O teste do Concurrent Collections deixou nítido ainda seguiu essa abordagem e deixou nítido como ela funciona.

Pode-se concluir a superioridade do Adder ao inves do Atomic para o algoritimo, devido a necessidade do algoritmo incrementar valores intensamente, que faz o Atomic pecar.

O ForkJoin não se mostrou eficiente. Graça ao fato do algoritmo ter uma natureza iterativa ao inves de recursivo, o que não justifica a recursão advinda dele. Que não gera grande ganhos e ainda necessita de trabalho do GC desnecessário.

O uso de Adder se encaixou perfeitamente para criação de novas threads em cada iteração, já que ele não é necessário realmente criar threads mas ele reaproveita as iniciais, e desta forma o gargalo entre a criação e desalocação de threads é minimizado.

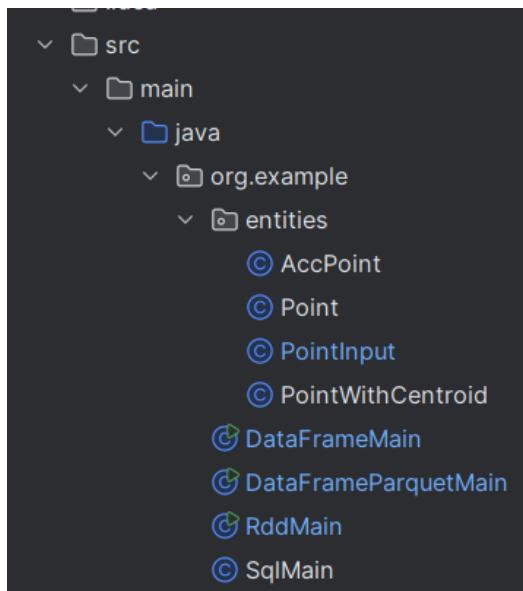
Como observado, a questão de GC foi bem realizada, ao ponto de que muitas implementações tem GC pause basicamente mínimo, zerado em alguns caso durante a execução, tendo pausas apenas na leitura inicial dos arquivos. Ou seja, pode-se dizer que foi alcançado o ponto ideal acerca do controle do GC em muitas implementações.

Mas em relação custo beneficio entre complexidade de código e performance a versão com ParallelStream ganhou disparado. O código da implementação foi mínimo, com poucas linhas, sem classes auxiliares além da classe do LongAdder. O código ficou legível, simples e com uma performance impressionante, perdendo apenas para o Adder usando Executors. Apesar de um tempo maior de GC, que não passa de 5% ainda advindo do fato de ParallelStream ainda usar ForkJoin por debaixo dos panos.

# Terceira Unidade

## 15. Spark - RDD

Para as implementações com o Spark foi feito um novo projeto Java com Maven para as três implementações. Tendo cada implementação uma classe main própria.



Nessa implementação, o RDD do spark foi usado tanto para leitura de arquivos quanto para o algoritmo em si. A leitura foi realizada com o .textFile, que retornava um RDD de cada linha, foi feito então o map para passar cada linha para uma classe de Point específica.

```
SparkConf conf = new SparkConf().setMaster("local[*]").setAppName("Spark Application");
JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> lines = sc.textFile("input.csv");

JavaRDD<Point> points = lines
    .map(line -> line.split(" "))
    .map(parts -> {
        String label = parts[0];
        int red = Integer.parseInt(parts[1]);
        int green = Integer.parseInt(parts[2]);
        int blue = Integer.parseInt(parts[3]);
        return new Point(label, red, green, blue);
    });
}
```

Com o RDD de todos os feitos foi criado a função que de fato executa o algoritmo. Se baseando principalmente na função de .aggregateByKey para agrupar todos os Point que possuem os mesmos centroids próximos. Foi criado uma classe auxiliar AccPoint como um acumulador de todos os pontos de cada agrupamento.

```
public static List<Point> kmeansCentroidsRDD(JavaRDD<Point> points, int K) { 3 usages • Talison Fabio
    List<Point> centroids = points.distinct().take(K);

    boolean isFinished = false;
    while(!isFinished) {
        final List<Point> finalCentroids = centroids;

        var accumulators = points
            .mapToPair(point -> {
                int i = point.closest(finalCentroids);
                return new Tuple2<Integer, Point>(i, point);
            })
            .aggregateByKey(new AccPoint(), AccPoint::add, AccPoint::merge);

        var newCentroids = accumulators.mapValues(AccPoint::meanResult).sortByKey().values().collect();

        isFinished = Point.converged(centroids, newCentroids);
        centroids = newCentroids;
    }

    return centroids;
}
```

O AccPoint basicamente é um acumulador, que guarda a soma dos pontos e a quantidade e um método de .meanResult() que realizar o cálculo do ponto médio usando esses valores.

```

public class AccPoint implements Serializable {
    final int counting; 8 usages
    final int sumX; 6 usages
    final int sumY; 6 usages
    final int sumZ; 6 usages

    > public AccPoint(){...}
    > private AccPoint(int counting, int sumX, int sumY, int sumZ){...}

    public AccPoint add(Point point) { ± Talison Fabio
        return new AccPoint(
            counting: counting + 1,
            sumX: sumX + point.x,
            sumY: sumY + point.y,
            sumZ: sumZ + point.z
        );
    }
    public AccPoint merge(AccPoint acc) { 1 usage ± Talison Fabio
        return new AccPoint(
            counting: counting + acc.counting,
            sumX: sumX + acc.sumX,
            sumY: sumY + acc.sumY,
            sumZ: sumZ + acc.sumZ
        );
    }
    public Point meanResult() { 1 usage ± Talison Fabio
        return new Point(
            x: sumX / counting,
            y: sumY / counting,
            z: sumZ / counting
        );
    }
}

```

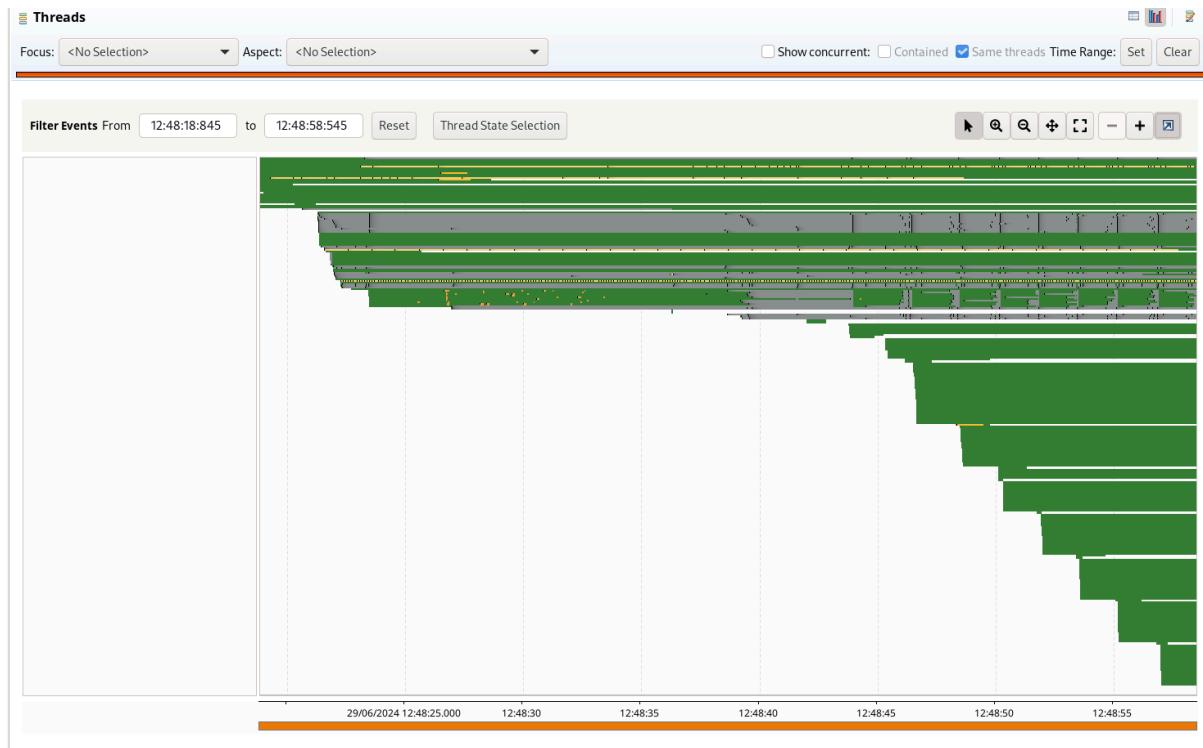
A escrita ao fim foi feito com o `saveAsTextFile` do RDD.

```

points.map(p -> {
    int i = p.closest(finalCentroids1);
    Point centroid = finalCentroids1.get(i);
    return new Point(p.label, centroid.x, centroid.y, centroid.z);
}) JavaRDD<Point>
.map(p -> String.join(" ", new String[]{p.label, String.valueOf(p.x), String.valueOf(p.y), String.valueOf(p.z)}))
.saveAsTextFile( path: "output.csv");

```

## 15.1 Avaliação de Profile



**Tempo de GC: 17%**

Pode-se perceber que ele teve baixíssima contenção. Apenas em threads internas do próprio Spark. E a leitura do arquivo tomou boa parte do tempo total da execução.

## 16. Spark - DataFrame

Para essa implementação foi buscado uma forma de, assim como o RDD, criar uma implementação do algoritmo usando as funções e abstrações de transformação de dados do DataFrame, porém, realmente não foi possível chegar num resultado satisfatório. Problemas com a documentação para achar funções que executassem operações como o

RDD e que pudessem lidar com leitura de listas externas. As operações com os DataSet não são só entre as próprias colunas em si, mas precisam ser calculadas com uma lista de centroides, externas, que são recalculadas em cada rodada. Com as funções de UDF, agregação do DataSet ou ademais não foi encontrado uma solução para esse problema.

O Dataframe foi usado então para a leitura de arquivos, usando uma classe PointInput como schema.

```
SparkSession spark = SparkSession.builder()
    .appName("Kmeans")
    .master("local[*]")
    .getOrCreate();

Encoder<PointInput> pointEncoder = Encoders.bean(PointInput.class);

Dataset<PointInput> inputPoints = spark.read().format("csv")
    .option("header", "true")
    .option("sep", " ")
    .option("inferSchema", "true")
    .load(path: "input.csv").as(pointEncoder);
```

```
public class PointInput implements Serializable { 15 usages  ↗ Talison Fabi
    public String label;  4 usages
    public int x;  8 usages
    public int y;  8 usages
    public int z;  8 usages

    public Point toAppPoint() { return new Point(label, x, y, z); }
    public int closest(List<PointInput> others) {...}

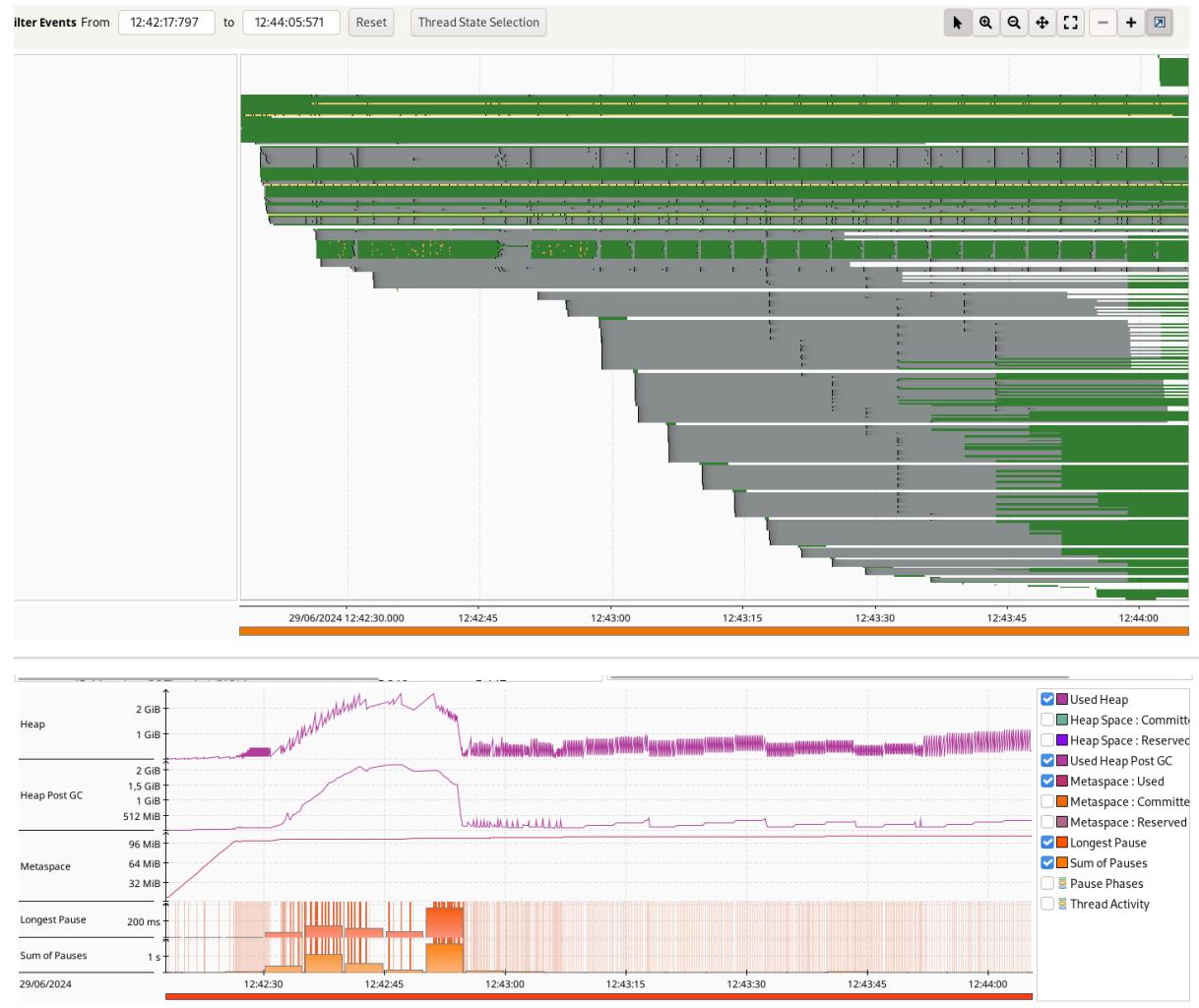
    public String getLabel() { return label; }
    public void setLabel(String label) { this.label = label; }
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
    public int getY() { return y; }
    public void setY(int y) { this.y = y; }
    public int getZ() { return z; }
    public void setZ(int z) { this.z = z; }
```

Para conseguir executá-lo o DataFrame foi convertido para RDD e foi usado a mesma função criada para a implementação do RDD

```
JavaRDD<Point> pointsRDD = inputPoints.javaRDD().map(p -> new Point(p.label, p.x, p.y, p.z));  
  
final List<Point> finalCentroids1 = RddMain.kmeansCentroidsRDD(pointsRDD, K);
```

A partir desse ponto o código é exatamente o mesmo que a implementação anterior.

## 16.1 Avaliação de Profile



**Tempo de GC: 20%**

Pode-se perceber um problema com a contenção das threads, em que apesar de exatamente a mesma função usada para o processamento dos dados, a implementação usando DataFrame teve uma contenção muito elevada.

## 17. Spark - DataFrame + Parquet

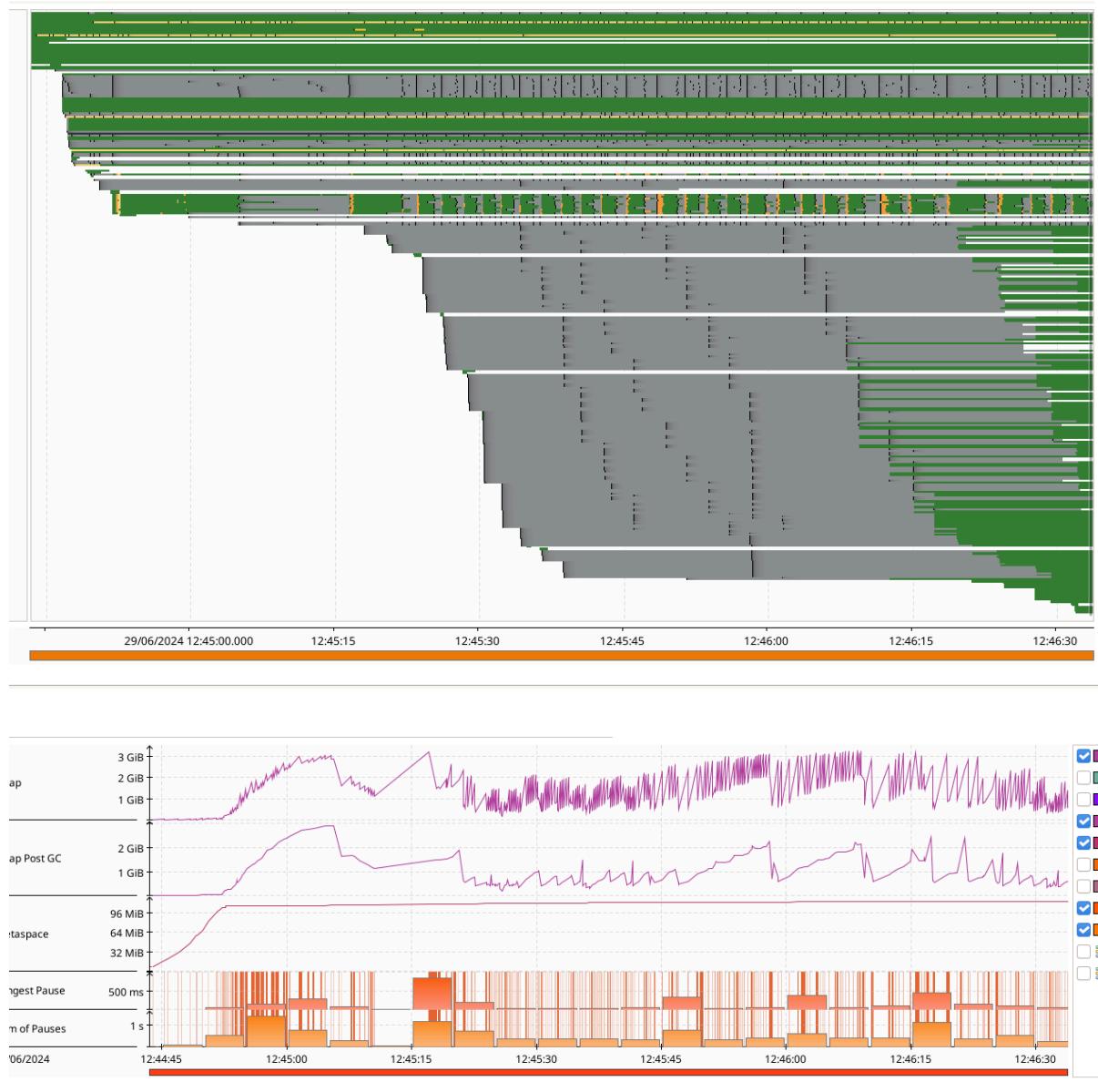
```
SparkSession spark = SparkSession.builder()
    .appName("Kmeans")
    .master("local[*]")
    .getOrCreate();

JavaRDD<Point> points = spark.read().parquet( path: "input.parquet") D
    .javaRDD() JavaRDD<Row>
    .map(parts -> {
        String label = parts.get(0).toString();
        int red = Integer.parseInt(parts.get(1).toString());
        int green = Integer.parseInt(parts.get(2).toString());
        int blue = Integer.parseInt(parts.get(3).toString());
        return new Point(label, red, green, blue);
    });
}
```

A única diferença real no código foi o código para leitura do arquivo. O resto para processamento dos dados se manteve o mesmo.

O arquivo parquet usado nos testes foi o mesmo csv usado nas ultimas implementações convertido para parquet. Foi usado um script python para fazer a conversão.

## 17.1 Avaliação de Profile



**Tempo de GC: 20%**

As observações do profile são bem similares com a implementação de DataFrame com csv. Com exceção das pausas e coletas do GC, que foram bem mais inconstantes que a implementação anterior, mas que não alterou a porcentagem do GC.

## 18. Resultados e Discussão

Um detalhe importante desses testes é que o dataset precisou ser ainda mais reduzido em comparação aos testes da segunda unidade, pois foram realizados no mesmo computador utilizado anteriormente. O consumo de RAM do Spark, com o dataset já utilizado, rapidamente esgotava a memória, travando completamente o computador. Assim, foi testado várias bases de dados de diferentes tamanhos para encontrar o limite máximo

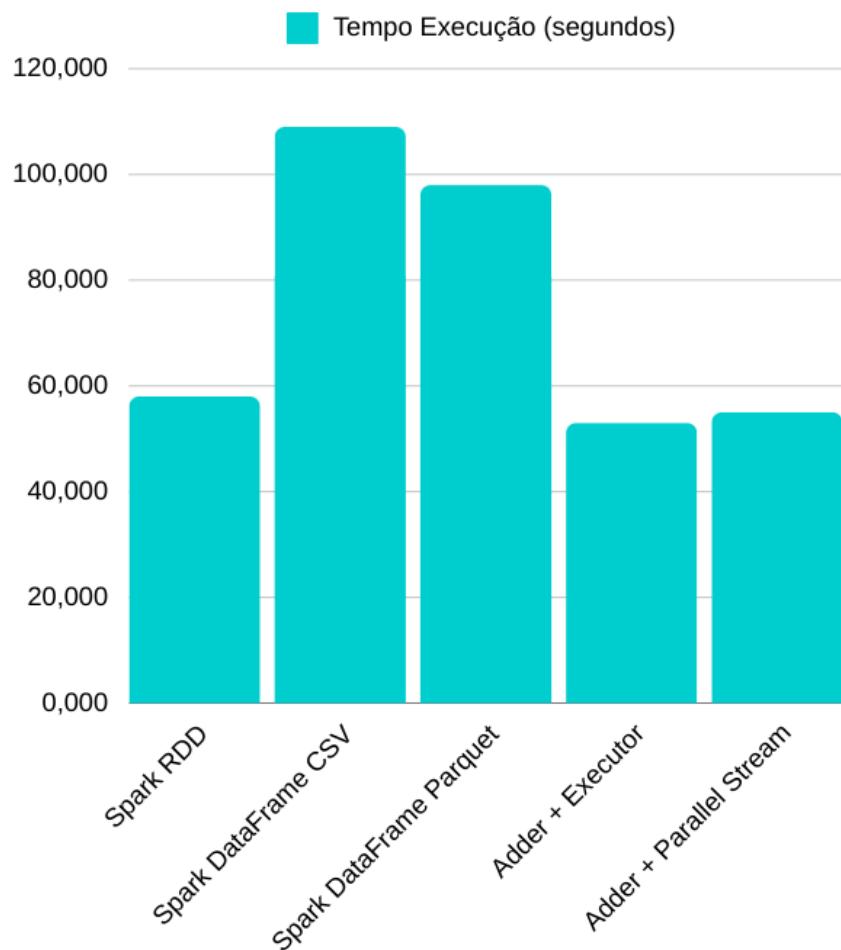
que pudesse ser executado sem travar todas as implementações. O tamanho encontrado foi um arquivo CSV de 430MB

Esses foram os tempos médios de execução de cada implementação.

Implementação	Tempo (ms)
RDD	58 356
DataFrame	109 752
DataFrame + Parquet	98 444

Para comparar com as implementações Java feitas anteriormente foi executado as implementações com os melhores resultados anteriores usando o mesmo dataset.

Implementação	Tempo (ms)
Adder + Executor	53 509
Adder + ParallelStream	55 103
Adder + ForkJoin	58 893



Os tempos de GC entre todas as implementações com Spark foram bastante semelhantes, então não há muito o que comentar a respeito.

Foi possível tirar algumas conclusões com os testes, mas infelizmente, devido às limitações técnicas do computador utilizado, não foi possível realizar análises mais profundas que só poderiam ser feitas com datasets maiores, acima de 1GB. Com datasets menores, como o utilizado, que foi o único viável, podemos teorizar que a performance inferior do Spark se deve ao overhead básico do próprio framework. Também é possível que a implementação tenha espaço para melhorias. Sem comparar com bases de dados maiores, não é possível confirmar essas teorias com precisão.

No entanto, uma conclusão clara é a comparação entre a conversão de DataSet usando DataFrame e JavaRDD. Observou-se que, nos algoritmos baseados em DataFrame, o tempo de leitura inicial dos arquivos é menor do que nos baseados em RDD. Isso foi verificado tanto no Profile quanto na medição de tempo durante o processamento, calculada usando a estratégia simples do System.currentTimeMillis().

```
long timeMsStart = System.currentTimeMillis();

// Implementação da leitura

long elapsedTimeBeforeProcess = System.currentTimeMillis() - timeMsStart;
System.out.println("Parse OK");
System.out.println(elapsedTimeBeforeProcess);
```

Foi executado cada implementação cinco vezes e calculado a média.

Implementação	Tempo médio inicial apenas da leitura do arquivo (segundos)
RDD	20
DataFrame	13
DataFrame + Parquet	11

Claro que, dada a natureza "Lazy" do Spark, é provável que parte dessa leitura seja processada após a medição inicial, durante o processamento dos dados. No entanto, dado o uso do DataFrame com Schema para leitura e a implementação de leitura de arquivo do RDD ser mais manual (com alocação e split de String), é um bom indício de que, de fato, a leitura no DataFrame seja mais rápida, dada a própria natureza do DataFrame e RDD para essa tarefa.

Ao final, considerando que o algoritmo de processamento de todas as implementações é literalmente idêntico, e a única diferença está na leitura do arquivo, se a leitura do DataFrame é mais rápida, o tempo total também deveria ser. Ou seja, nesse cenário, teoricamente, o DataFrame deveria ser mais performático. O que não é o caso. A teoria mais provável é que a conversão de DataSet para JavaRDD não é gratuita, e há um custo inerente nessa conversão de DataSet para RDD, que, se fosse implementado com as funções do próprio DataSet, não teria essa perda.

No Profile das implementações com DataFrame, é possível notar também uma contenção muito alta das threads, o que certamente influenciou nessa perda de performance. Dado o fato de que, novamente, temos a mesma função rodando em ambos, reforça-se mais o fato da perda nessa conversão de DataSet para JavaRDD. Pois, caso a

causa fosse a implementação, a versão com RDD também expressaria o mesmo problema no Profile.

Portanto, podemos concluir que o ideal de verdade seria realizar novos testes em um computador potente com uma base de dados de fato grande para extrair os dados.

E além disso, para melhorar essas implementações o ideal seria usar as funções de agregação e UDF do próprio DataSet e não realizar a conversão para JavaRDD, visto que essa operação se mostrou custosa.