UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE - UFRN
INSTITUTO METRÓPOLE DIGITAL - IMD
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO - BTI

MATHEUS RANGEL DE MELO - 20170070014

RELATÓRIO UNIDADE 1 - K-MEANS

NATAL, 2019

Algoritmo K-Means

Dado um conjunto de dados o algoritmo de k-means tem como objetivo agrupar os dados similares em diferentes clusters, para isso ele recebe como entrada o conjunto de dados a ser processado, a quantidade de clusters que o resultado deve ter e número máximo de iterações que deve ser feita.

A cada iteração o algoritmo gera aleatoriamente o valor inicial dos centróides que cada cluster deve ter, um valor do conjunto fará parte do cluster que tiver o centróide mais próximo dele, após agrupar todos os dados é calculada a média dos valores de cada cluster, essa média passa a ser o centróide de novos clusters repetindo o processo de agrupamento dos dados até que a média se estabilize. A cada iteração o algoritmo compara o resultado do agrupamento com o resultado do melhor agrupamento realizado até o momento, a qualidade de um agrupamento é definida pela soma das distâncias dos valores em relação ao centróide de cada cluster, quanto menor a distância melhor o agrupamento.

Quando o algoritmo finaliza ele retorna o melhor agrupado gerado pelas iterações.

Implementação

O algoritmo foi implementado de forma genérica, utilizando apenas as funções declaradas pela interface(ClusterData).

```java
package ufrn.br.kmeans;

import java.util.List;

public interface ClusterData<T> {
    Double calculateDistante(T v);
    T calculateMean(List<T> list);
    T getRandom();
}
```

Cada iteração do k-means pode ser executada de forma concorrente, pois o algoritmo apenas consome o conjunto de dados e não o modifica. Ao finalizar uma iteração o resultado do agrupamento precisa ser comparado com o melhor agrupamento realizado pelas iterações anteriores, substituindo-o caso o atual seja melhor. A única contenção necessária no algoritmo ocorre nesse momento pois o valor do melhor agrupamento é compartilhado entre as threads e é necessário garantir que apenas um agrupamento seja comparado por vez.

```
public class Kmeans<T extends ClusterData<T>> implements Runnable{
    private List<T> values;
    @Getter private List<Cluster<T>> bestClusters;
    @Getter private volatile double bestVariance;
    private List<List<T>> iterations;
    @Getter private final int numJobs;
    @Getter private final int maxIterations;
    @Getter private int curIteration;
    @Getter private int numClusters;
    public Kmeans(List<T> values, int numClusters, int numJobs, int maxIterations) {
```

Porém a única informação relevante sobre o cluster para fazer a comparação é a soma da distância dos pontos em relação ao centróide de cada cluster, o agrupamento que tiver o menor valor dessa soma é considerado o melhor(‘bestVariance’), então a contenção foi feita apenas na comparação dessa soma.

```
    public interface Callback<T extends ClusterData<T>>{
        void updateBestCluster(List<Cluster<T>> newClusters, Double variance);
    }
    private Callback<T> callback = (List<Cluster<T>> newClusters, Double variance) -> {
        synchronized (bestVariance){
            if(variance < bestVariance){
                bestVariance = variance;
                bestClusters = newClusters;
            }
        }
    };
```

A comparação é realizada quando a thread chama o callback apenas no final de sua execução, representando uma fração ínfima do tempo total de execução da thread.

O número de threads que irão realizar as iterações é flexível, cada thread recebe uma lista contendo diferente iterações que ela precisa realizar com o valores de centroids aleatórios.

```java
public void run() {
    List<List<Clusterize<T>>> jobs = new LinkedList<>();
    curIteration = 0;
    for (int i = 0; i < numJobs; i++) {
        LinkedList<Clusterize<T>>job = new LinkedList<>();
        for (int j = 0; j < maxIterations/numJobs; j++) {
            job.add(new Clusterize<>(values, callback, iterations.get(curIteration)));
            curIteration++;
        }
        jobs.add(job);
    }
    List<Thread> threads = new LinkedList<>();
    for(List<Clusterize<T>> job: jobs){
        Thread t = new Thread(() -> {
            for (Clusterize<T> clusterize: job
            ) {
                clusterize.run();
            }
        });
        threads.add(t);
        t.start();
    }
```

Conjunto de dados

O conjunto de dados escolhido para ser utilizado pelo k-means foi o Iris Plants Database, disponivel no link: https://archive.ics.uci.edu/ml/datasets/iris, ele tem 150 instancias e 3 classes, cada instância com 4 atributos a classe a qual ela pertence. A sumarização dos dados foi dada junto com o dataset:

Summary Statistics:

	Min	Max	Mean	SD	Class Correlation	
sepal length:	4.3	7.9	5.84	0.83	0.7826	
sepal width:	2.0	4.4	3.05	0.43	-0.4194	
petal length:	1.0	6.9	3.76	1.76	0.9490	(high!)
petal width:	0.1	2.5	1.20	0.76	0.9565	(high!)

Com base neste sumário decidi utilizar apenas o petal length e o petal width normalizados entre 0 a 1 para calcular a distância entre duas iris.

```java
    private Tipo tipo;
    private double sepalLength;
    private double sepalWidth;
    private double petalLength;
    private double petalWidth;

    private Double getNormalizedPetalLength(){
        return (this.petalLength - minPetalLength)/(maxPetalLength - minPetalLength);
    }
    private Double getNormalizedPetalWidth(){
        return (this.petalWidth - minPetalWidth)/(maxPetalWidth - minPetalWidth);
    }
    @Override
    public Double calculateDistante(Iris v) {
        return Math.abs((this.getNormalizedPetalLength() - v.getNormalizedPetalLength())
                +(this.getNormalizedPetalWidth() - v.getNormalizedPetalWidth()));
    }
    @Override
    public Iris calculateMean(List<Iris> list) {
        double totalSepalLength = 0.0;
        double totalSepalWidth = 0.0;
        double totalPetalLength = 0.0;
        double totalPetalWidth = 0.0;
        for (Iris i: list) {
            totalPetalLength += i.getPetalLength();
            totalPetalWidth += i.getPetalWidth();
            totalSepalLength += i.getSepalLength();
            totalSepalWidth += i.getSepalWidth();
        }
        return new Iris(
            Tipo.UNCLASSIFIED,
             sepalLength: totalSepalLength/list.size(),
             sepalWidth: totalSepalWidth/list.size(),
             petalLength: totalPetalLength/list.size(),
             petalWidth: totalPetalWidth/list.size()
        );
    }
```

Especificação do Computador

Todos os teste foram realizados no meu computador com as seguintes especificações:

Processador: i5 4440m - 2,4ghz à 3.ghz - 2 núcleos e 4 threads.

Memoria RAM: 8GB 2400mhz

Disco Rigido: 1000TB 7200RPM.

Resultados

Os resultados obtidos foram excelentes, utilizando 100 iterações o k-means classificou corretamente 144 das 150 amostras contidas no conjunto de dados, o tempo de execução ficou 1,5 segundo em singlethread e o melhor tempo em multithread foi de 950ms com 4 threads. Testei também aumentando artificialmente o conjunto de dados para 3000 amostras e fazendo 5000 iterações, a porcentagem de acerto ficou a mesma e os tempos médios ficaram:

SingleThread: 14,8 segundos.

2 Threads: 11,6 segundos.
4 Threads: 9,9 segundos.
8 Threads: 8,8 segundos.
16 Threads: 8,6 segundos.

Testes e Benchmarks

O primeiro teste que realizei foi utilizando o Java Flight Recorder. Foi verificado o uso de memória, CPU, desempenho das threads e o impacto do garbage collector. Não estava sendo possível analisar completamente o comportamento do algoritmo com o conjunto de dados original por ser muito pequeno e terminar muito rápido. Com isso os benchmarks foram realizados com uma versão com os dados duplicados totalizando 3000 entradas e rodando 5000 iterações.

Uso de CPU

Mesmo executando com apenas uma thread a minha cpu ficou em média com 77% de utilização, diminuindo a margem para ganho de performance, o que justifica os resultados de tempo.

Imagem: Uso de CPU 1 Thread

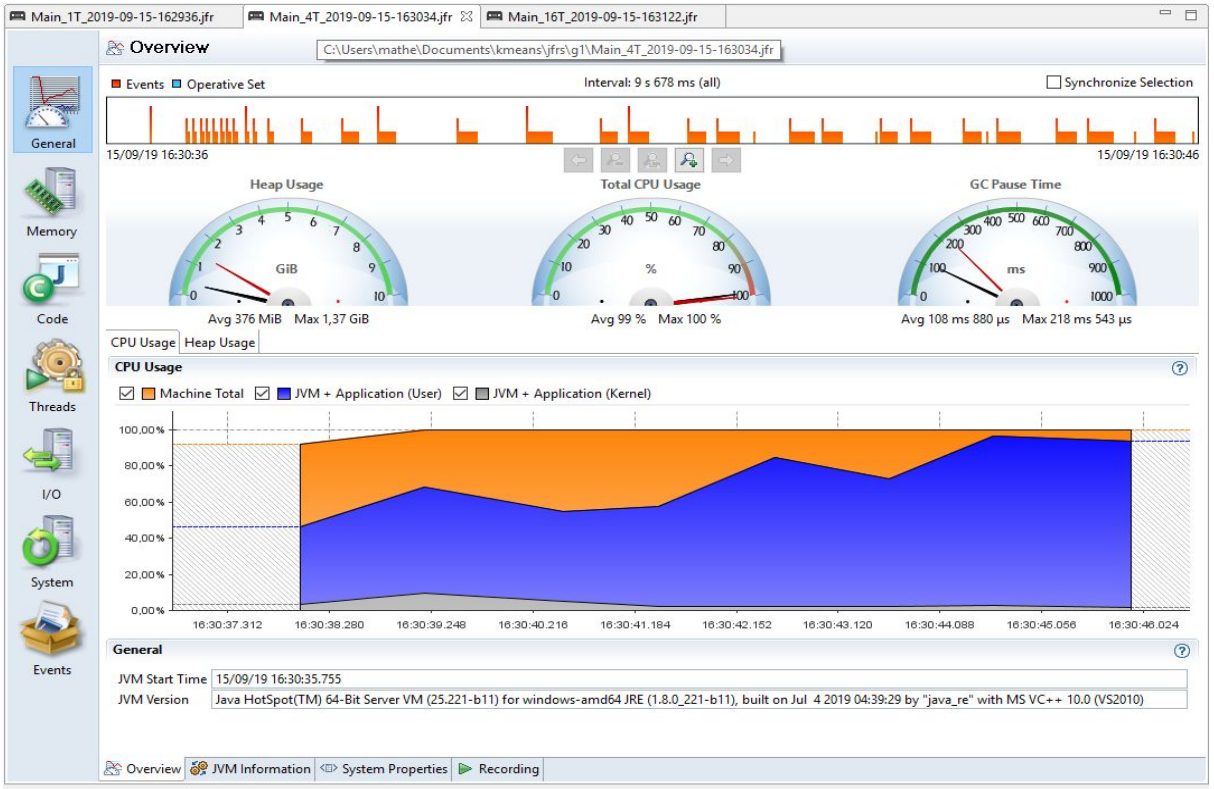


Imagem: Uso de CPU 4 Threads

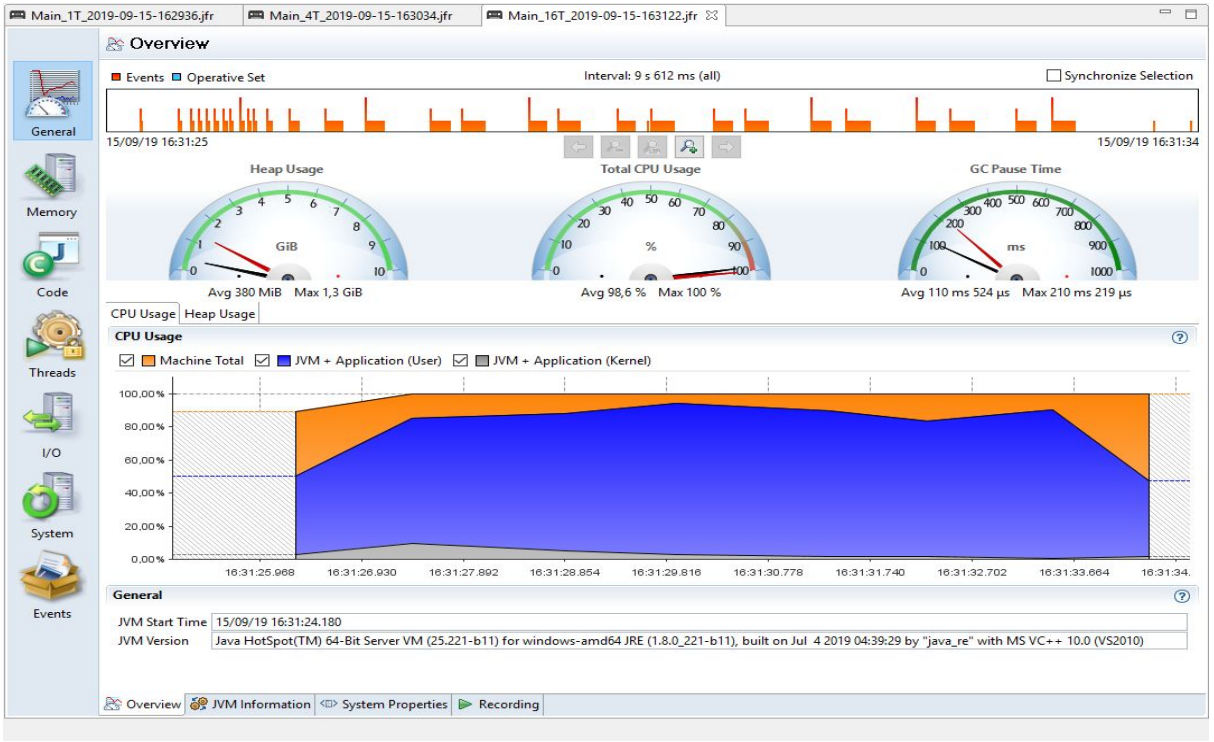


Imagem: Uso de CPU 16 Threads

Uso de Memória e Garbage Collector

O uso de memória não mudou muito independentemente da quantidade de threads instanciadas.

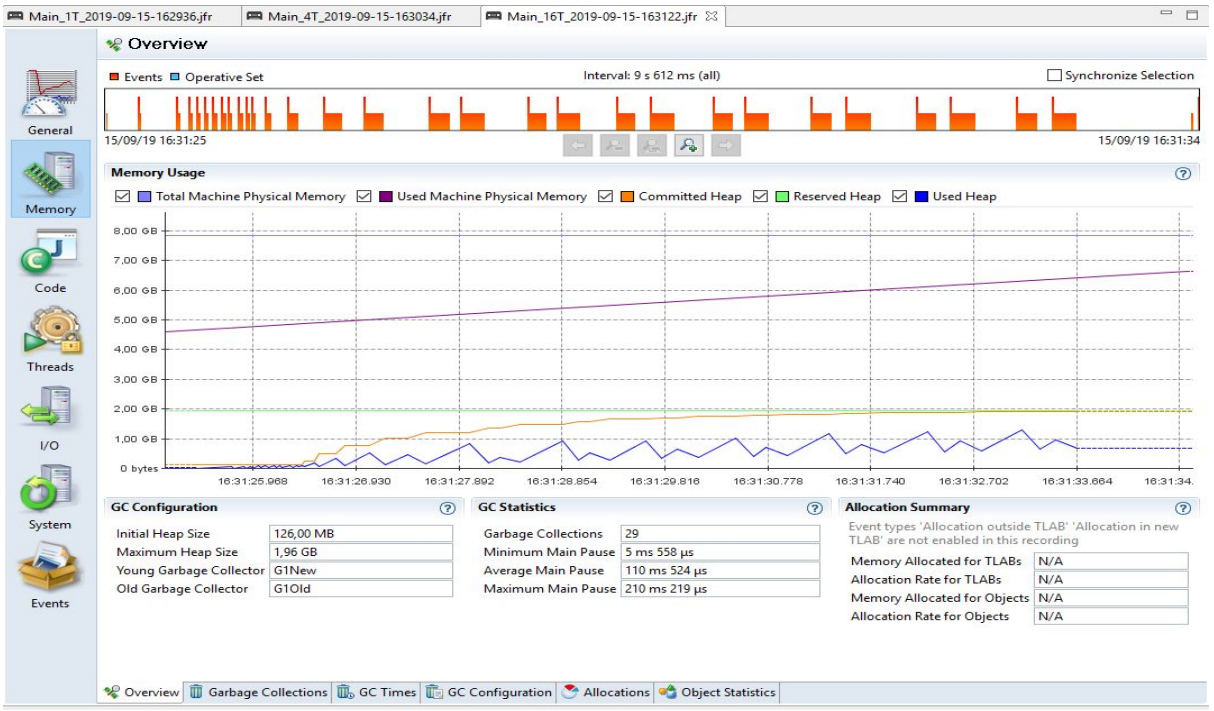
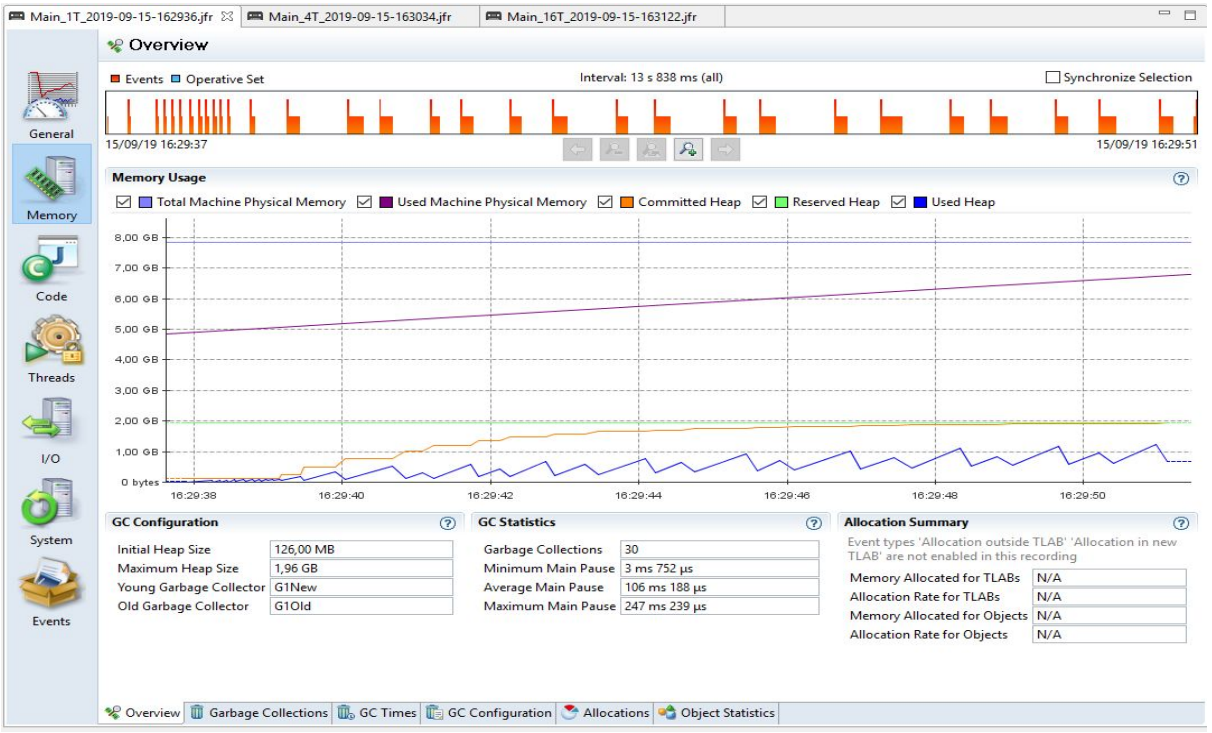


Imagem: Consumo de memória com 16 Threads



Consumo de memória com 1 Thread

O que mais preocupa em relação a implementação foi quantidade de lixo que estava sendo gerado. A cada iterações vários agrupamentos são gerados e descartados, com isso o impacto do GC foi tremendo correspondendo a quase 70% do tempo de execução utilizando o Parallel GC e 30% utilizando o G1. Inicializando mais threads o GC foi executado com um pouco menos de frequência.

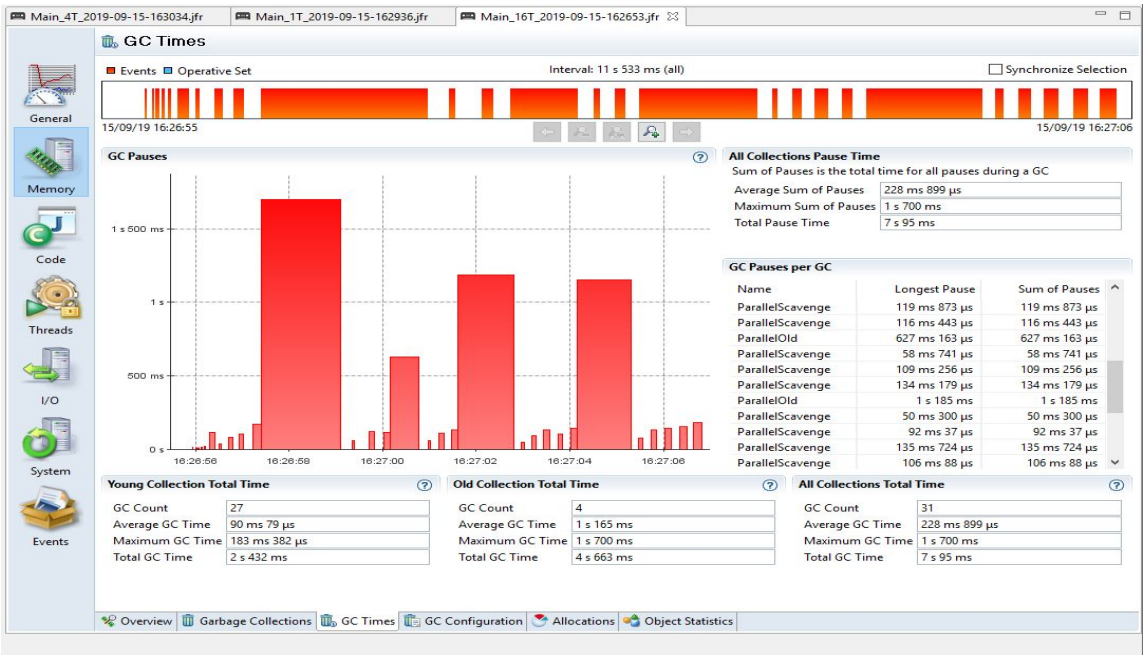


Imagem: Parallel GC Times

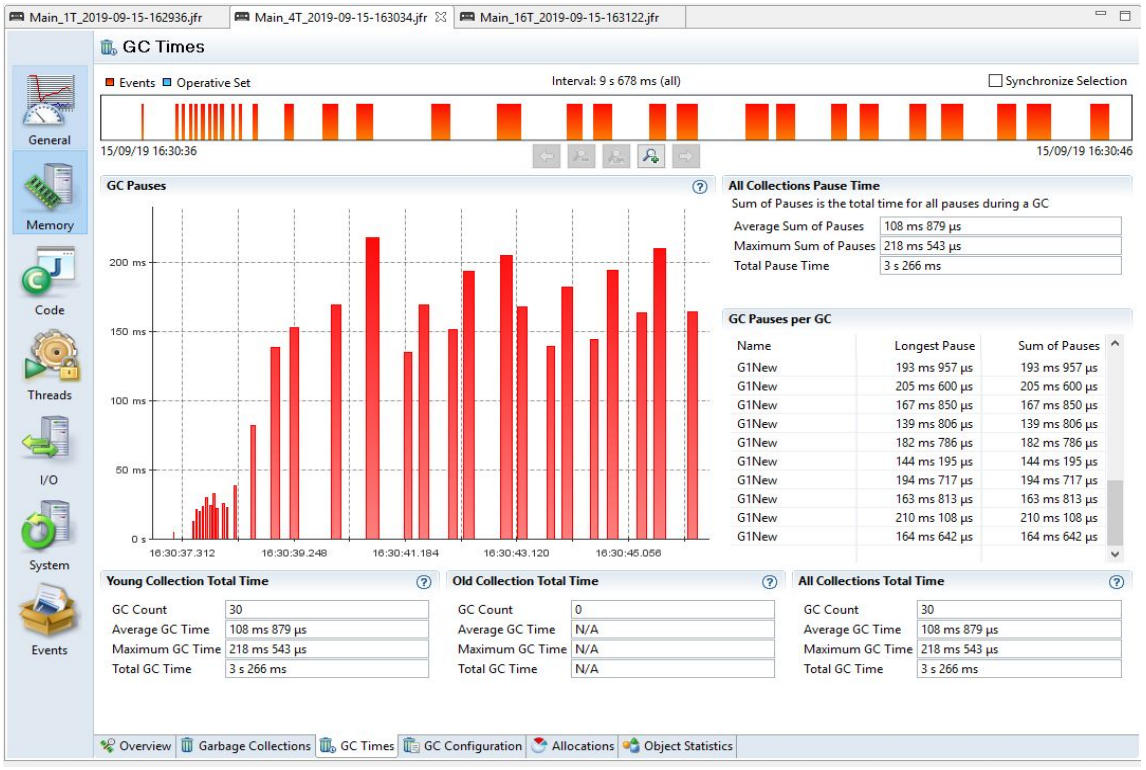


Imagem: G1 GC times

Threads

O algoritmo permitiu que as threads pudessem executar praticamente sem contenção e com a carga bastante balanceada entre as threads.

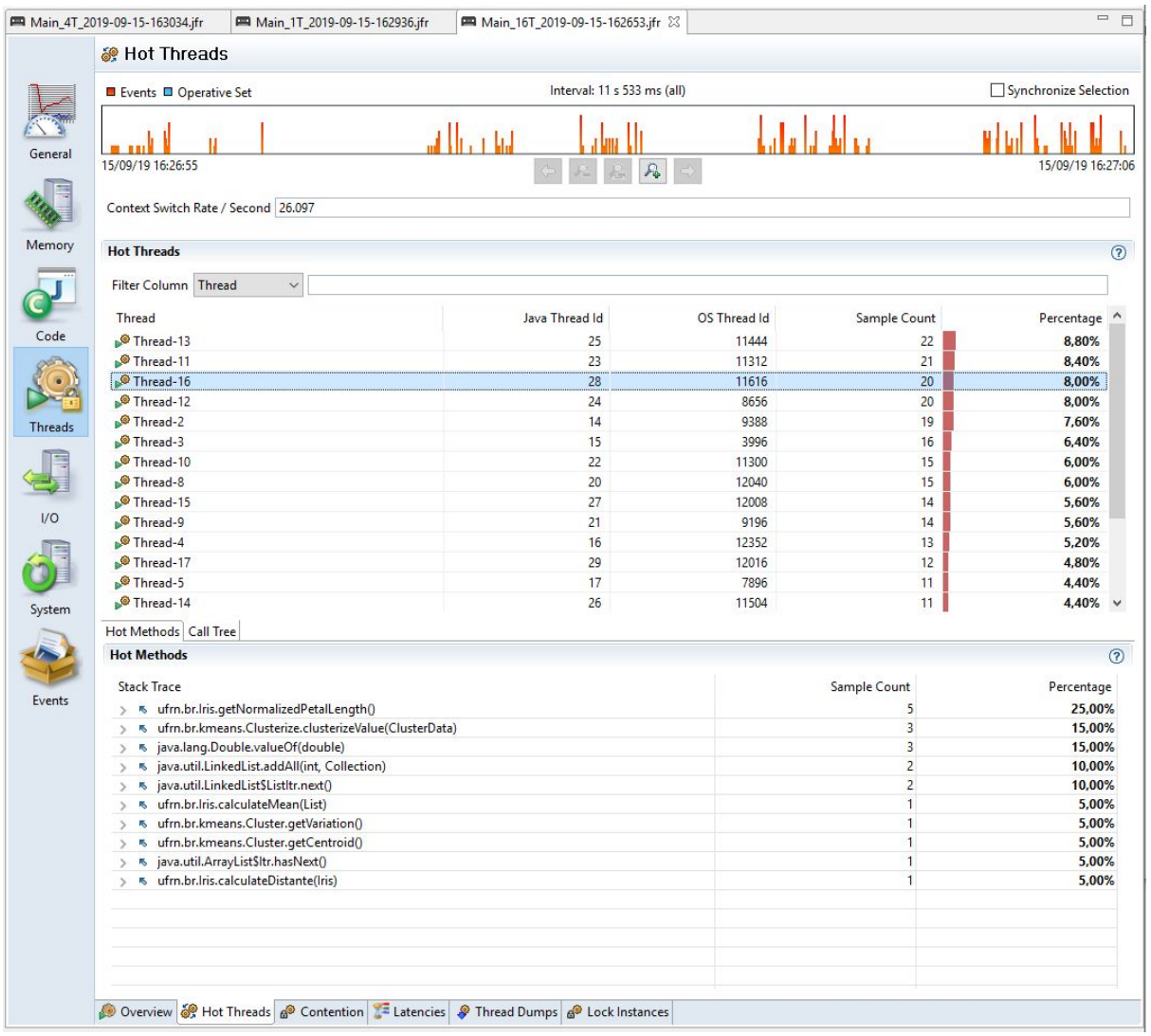


Imagem: Uso das threads.

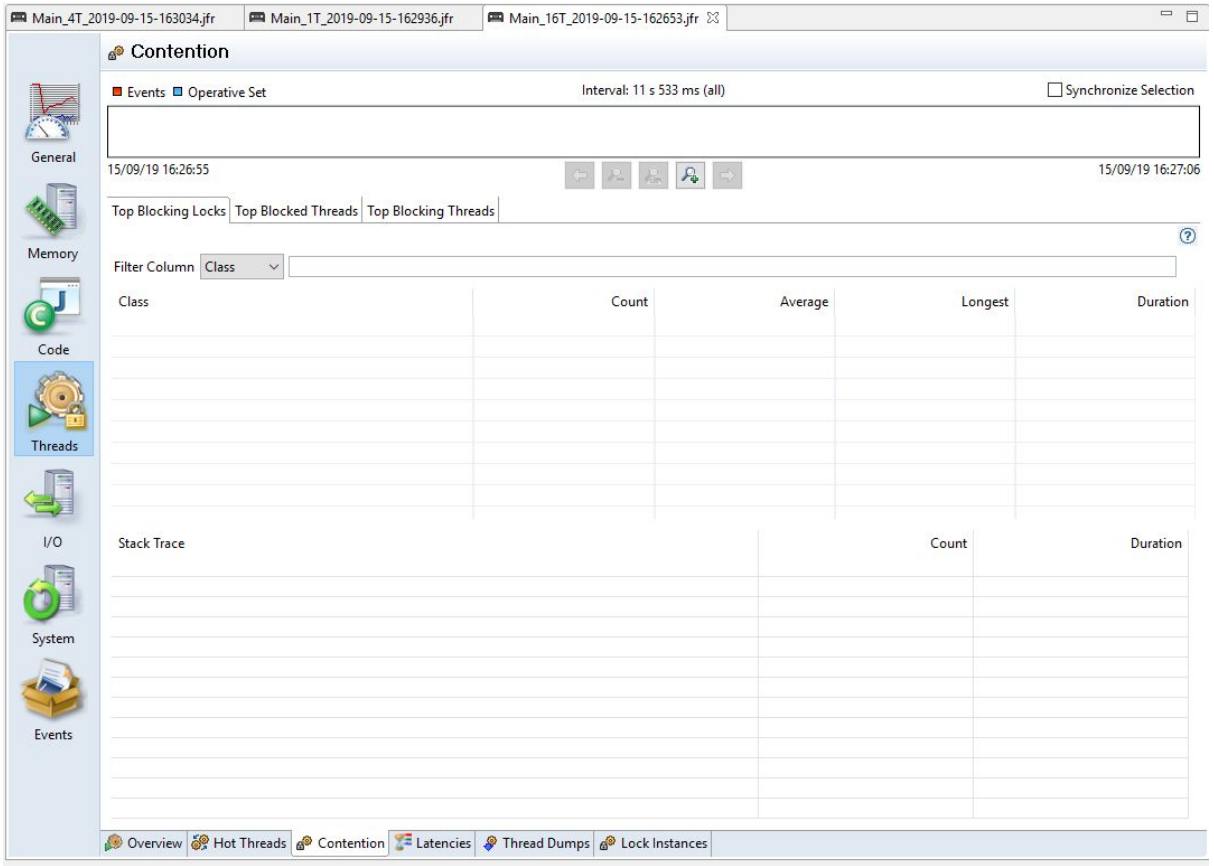


Imagem: Algoritmo executando com 16 threads e sem contenção.

JCStress

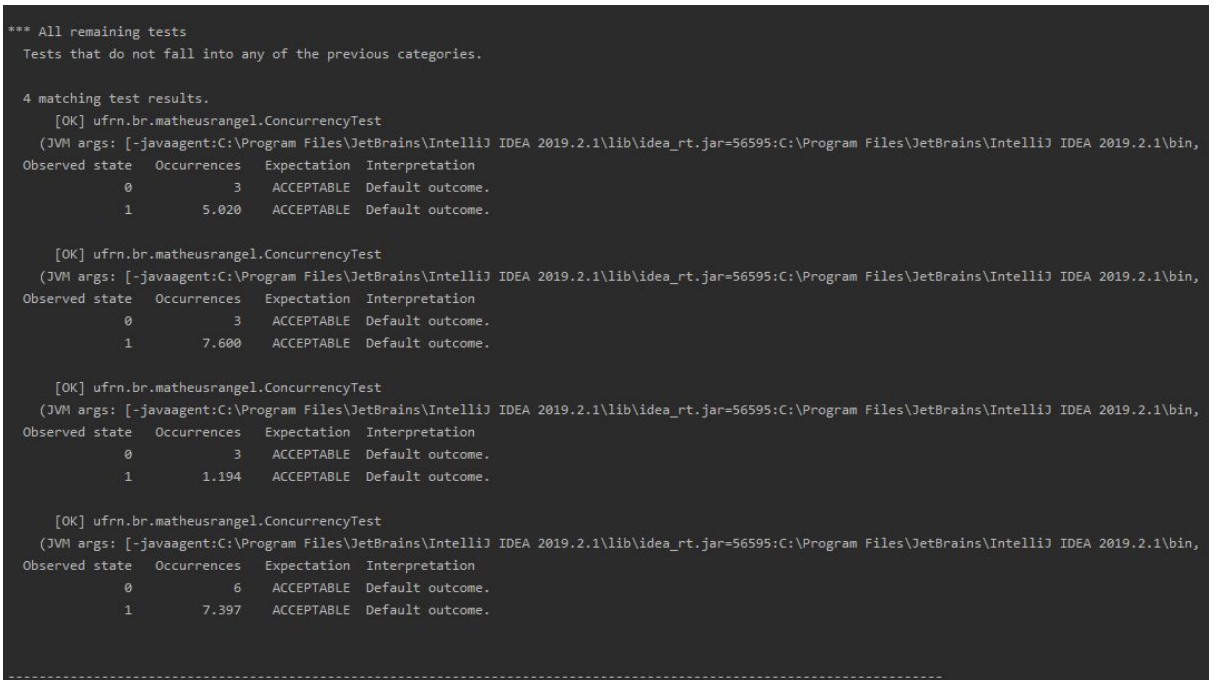
Utilizei o JCStress para verificar se a melhor variância calculada realmente corresponde a variância do melhor agrupamento.

```java
private List<Iris> randomValues = randomList( size: 50);
private volatile double variance = 0;
private Kmeans<Iris> kmeans = new Kmeans<>(randomValues, numClusters: 3, numJobs: 4, maxIterations: 10);
private static List<Iris> randomList(int size){
    Iris i = new Iris(Iris.Tipo.UNCLASSIFIED, sepalLength: 0, sepalWidth: 0, petalLength: 0, petalWidth: 0);
    List<Iris> list = new ArrayList<>(size);
    for (int j = 0; j < size; j++) {
        list.add(i.getRandom());
    }
    return list;
}

@Actor
public void actor1(I_Result r) {
    kmeans.run();
    while(kmeans.getBestVariance() == null){};
    if(variance == 0){
        variance = kmeans.getBestVariance();
    }else{
        boolean flag = variance == kmeans.getBestVariance();
        r.r1 = (byte) (flag ? 1:0);
    }
}

@Actor
public void actor2(I_Result r) {
    kmeans.run();
    while(kmeans.getBestClusters() == null){};
    double v = 0;
    for (Cluster<Iris> cluster: kmeans.getBestClusters()) {
        v += cluster.getVariation();
    }
    if(variance == 0){
        variance = v;
    }else{
        boolean flag = variance == v;
        r.r1 = (byte) (flag ? 1:0);
    }
}
```

O algoritmo passou em todos os testes.



```
*** All remaining tests
  Tests that do not fall into any of the previous categories.

  4 matching test results.
      [OK] ufrn.br.matheusrangel.ConcurrencyTest
    (JVM args: [-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\lib\idea_rt.jar=56595:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\bin,
  Observed state   Occurrences   Expectation  Interpretation
              0             3    ACCEPTABLE  Default outcome.
              1         5.020    ACCEPTABLE  Default outcome.

      [OK] ufrn.br.matheusrangel.ConcurrencyTest
    (JVM args: [-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\lib\idea_rt.jar=56595:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\bin,
  Observed state   Occurrences   Expectation  Interpretation
              0             3    ACCEPTABLE  Default outcome.
              1         7.600    ACCEPTABLE  Default outcome.

      [OK] ufrn.br.matheusrangel.ConcurrencyTest
    (JVM args: [-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\lib\idea_rt.jar=56595:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\bin,
  Observed state   Occurrences   Expectation  Interpretation
              0             3    ACCEPTABLE  Default outcome.
              1         1.194    ACCEPTABLE  Default outcome.

      [OK] ufrn.br.matheusrangel.ConcurrencyTest
    (JVM args: [-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\lib\idea_rt.jar=56595:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.1\bin,
  Observed state   Occurrences   Expectation  Interpretation
              0             6    ACCEPTABLE  Default outcome.
              1         7.397    ACCEPTABLE  Default outcome.


------------------------------------------------------------------------------------------------------------------------
```