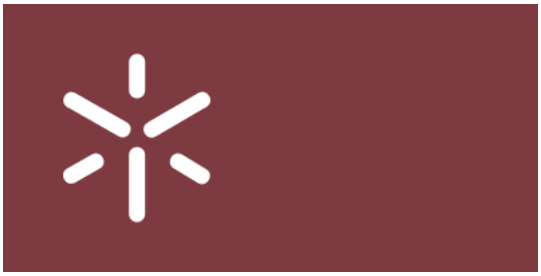

Benchmarking Java Streams

MIEI - 4º ano

Universidade do Minho

Processamento de Dados com Streams de Java

João Mourão A75064

José Silva A74576

Ricardo Certo A75315

11 de Janeiro de 2018

Conteúdo

1 Interface **5**

2 Carregamento dos Ficheiros **6**

2.1 Java 8 6

2.2 Java 9 7

2.3 Análise de Resultados 8

3 Testes **9**

3.1 Teste 1 10

3.1.1 Descrição 10

3.1.2 Observações 10

3.1.3 Código 10

3.1.4 Resultados 12

3.1.5 Análises e Conclusões 12

3.2 Teste 2 13

3.2.1 Descrição 13

3.2.2 Observações 13

3.2.3 Código 13

3.2.4 Resultados 16

3.2.5 Análises e Conclusões 16

3.3 Teste 3 17

3.3.1 Descrição 17

3.3.2 Observações 17

3.3.3 Código 17

3.3.4 Resultados 19

3.3.5 Análises e Conclusões 19

3.4 Teste 4 20

3.4.1 Descrição 20
3.4.2 Observações 20
3.4.3 Código 20
3.4.4 Resultados 22
3.4.5 Análises e Conclusões 22
3.5 Teste 5 23
3.5.1 Descrição 23
3.5.2 Observações 23
3.5.3 Código 23
3.5.4 Resultados 24
3.5.5 Análises e Conclusões 24
3.6 Teste 6 25
3.6.1 Descrição 25
3.6.2 Observações 25
3.6.3 Código 25
3.6.4 Resultados 28
3.6.5 Análises e Conclusões 28
3.7 Teste 7 29
3.7.1 Descrição 29
3.7.2 Observações 29
3.7.3 Código 29
3.7.4 Resultados 30
3.7.5 Análises e Conclusões 30
3.8 Teste 8 31
3.8.1 Descrição 31
3.8.2 Observações 31
3.8.3 Código 31
3.8.4 Resultados 32
3.8.5 Análises e Conclusões 32
3.9 Teste 9 33
3.9.1 Descrição 33
3.9.2 Observações 33
3.9.3 Código 33
3.9.4 Resultados 35
3.9.5 Análises e Conclusões 35
3.10 Teste 10 36

3.10.1 Descrição 36
3.10.2 Observações 36
3.10.3 Código 36
3.10.4 Resultados 38
3.10.5 Análises e Conclusões 38
3.11 Teste 11 39
3.11.1 Descrição 39
3.11.2 Observações 39
3.11.3 Código 39
3.11.4 Resultados 41
3.11.5 Análises e Conclusões 42
3.12 Teste 12 44
3.12.1 Descrição 44
3.12.2 Observações 44
3.12.3 Código 44
3.12.4 Resultados 46
3.12.5 Análises e Conclusões 47

Lista de Figuras

1.1 Menu 1 5
1.2 Menu 2 5

2.1 Load TransCaixa1M.txt Java 8 6
2.2 Load TransCaixa2M.txt Java 8 7
2.3 Load TransCaixa4M.txt Java 8 7
2.4 Load TransCaixa8M.txt Java 8 7
2.5 Load TransCaixa1M.txt Java 9 7
2.6 Load TransCaixa2M.txt Java 9 8
2.7 Load TransCaixa4M.txt Java 9 8
2.8 Load TransCaixa8M.txt Java 9 8

3.1 Gráfico Teste 1 12
3.2 Gráfico Teste 2 16
3.3 Gráfico Teste 3 19
3.4 Gráfico Teste 4 22
3.5 Gráfico Teste 5 24
3.6 Gráfico Teste 6 28
3.7 Gráfico Teste 7 30
3.8 Gráfico Teste 8 32
3.9 Gráfico Teste 9 35
3.10 Gráfico Teste 10 38
3.11 Gráfico Teste 11 - V1 41
3.12 Gráfico Teste 11 - V2 41
3.13 Gráfico Teste 11 - V3 42
3.14 Gráfico Teste 11 - V4 42
3.15 Gráfico Teste 12 - JAVA 8 46
3.16 Gráfico Teste 12 - JAVA 9 46

Capítulo 1

Interface

Como podemos visualizar nas figuras mais abaixo a nossa interface possui dois menus. O primeiro menu, é o menu onde o utilizador escolhe o ficheiro de dados com que quer efetuar os testes, o segundo depois de o ficheiro já ter sido carregado permite ao utlizador escolher qual o teste que quer realizar.

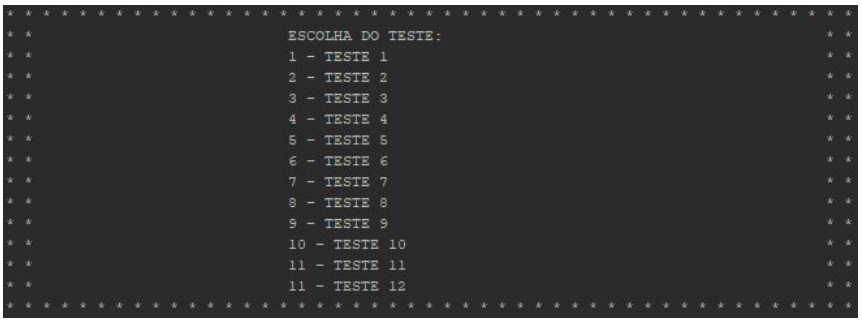

Figura 1.1: Menu 1

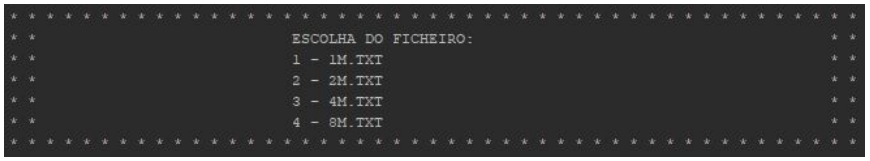

Figura 1.2: Menu 2

Capítulo 2

Carregamento dos Ficheiros

Para podermos realizar os testes temos que importar um ficheiro de Transações de Caixa. O utilizador pode escolher se esse ficheiro possui 1 milhão, 2 milhões, 4 milhões ou 8 milhões de transações. De seguida apresentamos o tempo que cada ficheiro demora a ser carregado, diferenciando os casos de quando usamos Java 8 ou Java 9 para efetuar esse carregamento.

2.1 Java 8

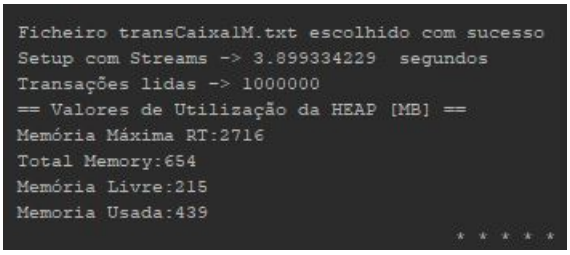

Figura 2.1: Load TransCaixa1M.txt Java 8

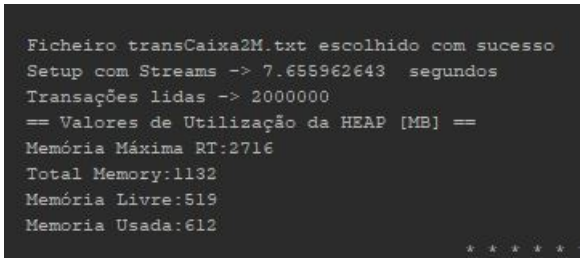

Figura 2.2: Load TransCaixa2M.txt Java 8

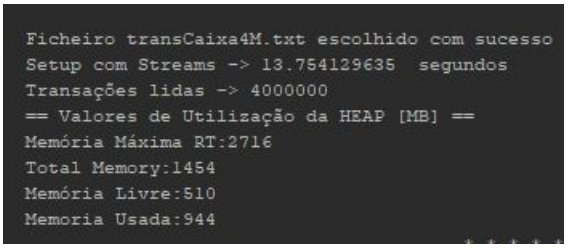

Figura 2.3: Load TransCaixa4M.txt Java 8

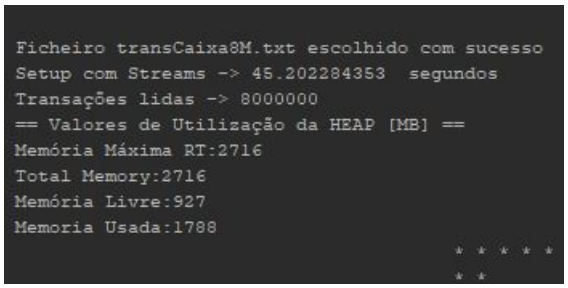

Figura 2.4: Load TransCaixa8M.txt Java 8

2.2 Java 9

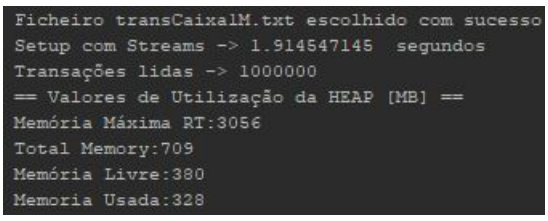

Figura 2.5: Load TransCaixa1M.txt Java 9

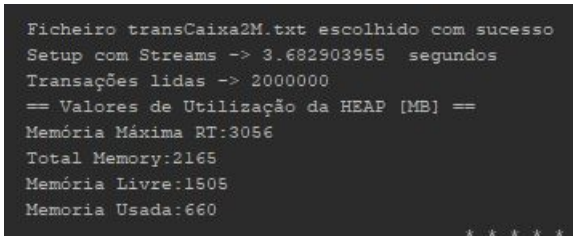

Figura 2.6: Load TransCaixa2M.txt Java 9

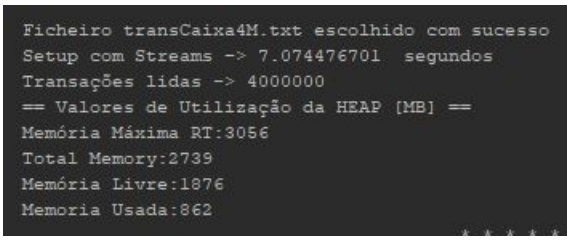

Figura 2.7: Load TransCaixa4M.txt Java 9

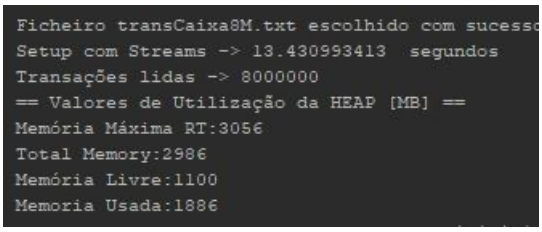

Figura 2.8: Load TransCaixa8M.txt Java 9

2.3 Análise de Resultados

Tendo em conta as imagens em cima podemos verificar que o Java 9 carrega os ficheiros mais rápido do que o Java 8.

Capítulo 3

Testes

Neste trabalho, vamos fazer o que se designa por *Java Benchmarking for Massive Data*, que tem como objetivo dar a compreender as vantagens e as desvantagens das várias alternativas disponíveis para uma igual solução. Todos os testes foram realizados tendo por base os ficheiros de texto transCaixa fornecidos, exceto os testes 3 e 4 que não necissitam do uso dos ficheiros. Por cada teste efetuado vamos apresentar as observações de cada teste, os diferentes códigos, os resultados obtidos e por fim tendo por base esses dados recolhidos vamos apresentar as nossas análises e conclusões.

3.1 Teste 1

3.1.1 Descrição

Criar um double[], uma DoubleStream e uma Stream <Double> contendo desde 1M até 8M dos valores das transacções registadas em List<TransCaixa>. Usando para o array um ciclo for() e um forEach() e para as streams as operações respectivas e processamento sequencial e paralelo, comparar para cada caso os tempos de cálculo da soma desses valores.

3.1.2 Observações

Nenhuma observação a fazer.

3.1.3 Código

For

```
 Supplier<Double> supplier1 = () -> forArray(array);

  private double forArray(double[] array){
    int i;
    double total = 0;

    for(i=0;i<array.length;i++)
            total += array[i];

    return total;
}
```

ForEach

```
Supplier<Double> supplier2 = () -> forEachArray(array);


private double forEachArray(double[] array){
    double total = 0;

    for(double d : array){
```

```
            total += d;
        }

        return total;
    }
```

Double Stream Sequencial

```
Supplier<Double> supplier3 = () -> ltc.stream()
.mapToDouble(l->l.getValor()).sum();
```

Stream < Double > Sequencial

```
Supplier<Double> supplier4 = () -> ltc.stream()
.map(l->l.getValor()).reduce(0.0,Double::sum);
```

Double Stream Paralela

```
Supplier<Double> supplier5 = () -> ltc.parallelStream()
.mapToDouble(l->l.getValor()).sum();
```

Stream < Double > Paralela

```
Supplier<Double> supplier6 = () -> ltc.parallelStream()
.map(l->l.getValor()).reduce(0.0,Double::sum);
```

3.1.4 Resultados

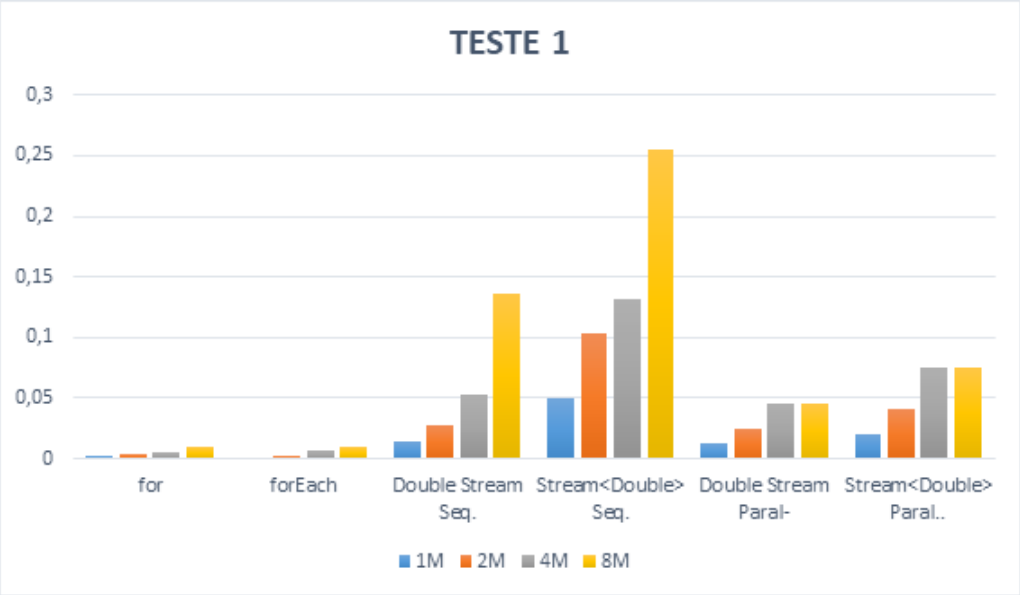

Figura 3.1: Gráfico Teste 1

3.1.5 Análises e Conclusões

Depois de analisarmos os resultados obtidos podemos afirmar que para este teste a opção mais rápida é usar o for ou o forEach enquanto que a forma mais lenta é quando usamos streams sequenciais.

3.2 Teste 2

3.2.1 Descrição

Considere o problema típico de a partir de um data set de dada dimensão se pretenderem criar dois outros data sets correspondentes aos 20% primeiros e aos 20% últimos do data set original segundo um dado critério. Defina sobre TransCaixa um critério de comparação que envol va datas ou tempos e use-o neste teste, em que se pretende comparar a solução com streams sequenciais e paralelas ás soluções usando List<> e TreeSet<>.

3.2.2 Observações

Nenhuma observação a fazer.

3.2.3 Código

List Streams Sequenciais

```
    List<TransCaixa> listaOrd = ltc.stream()
            .sorted(transPorData)
            .collect(toList());

    Supplier<List<List<TransCaixa>>> listSup = () ->
        selec2listas(listaOrd, vintep100);


  private List<List<TransCaixa>>
  selec2listas(List<TransCaixa> listaOrd, int vintep100) {
      List<TransCaixa> prim20 = listaOrd.stream()
              .limit(vintep100).collect(toCollection(() -> new ArrayList<>()));
      List<TransCaixa> ult20 = listaOrd.stream()
              .sorted(Caixa.transPorData2).limit(vintep100)
              .collect(toCollection(() -> new ArrayList<>()));


      List<List<TransCaixa>> result = new ArrayList<>();
      result.add(prim20);
      result.add(ult20);
```

```
    return result;
  }
```

TreeSet Streams Sequenciais

```
  TreeSet<TransCaixa> treeOrd = new TreeSet<>(transPorData);
  treeOrd.addAll(ltc);

  Supplier<List<TreeSet<TransCaixa>>> treeSup = () ->
  selec2tree(treeOrd,vintep100);

   private List<TreeSet<TransCaixa>> selec2tree(TreeSet<TransCaixa> treeOrd,
   int vintep100){

    TreeSet<TransCaixa> prim20 = treeOrd.stream()
            .limit(vintep100).collect(toCollection(() ->
            new TreeSet<>(Caixa.transPorData)));
    TreeSet<TransCaixa> ult20 = treeOrd.stream()
            .sorted(Caixa.transPorData2).limit(vintep100)
            .collect(toCollection(() -> new TreeSet<>(Caixa.transPorData)));

    List<TreeSet<TransCaixa>> result = new ArrayList<>();
    result.add(prim20);
    result.add(ult20);


    return result;
}
```

List Streams Paralelas

```
  List<TransCaixa> listaOrd = ltc.stream()
          .sorted(transPorData)
          .collect(toList());

  Supplier<List<List<TransCaixa>>> listSuppar = () ->
  selec2listasparalelas(listaOrd, vintep100);

  private List<List<TransCaixa>> selec2listasparalelas
  (List<TransCaixa> listaOrd , int vintep100){
```

```
    List<TransCaixa> prim20 = listaOrd.parallelStream()
            .limit(vintep100).collect(toCollection(() -> new ArrayList<>()));
    List<TransCaixa> ult20 = listaOrd.parallelStream()
            .sorted(Caixa.transPorData2).limit(vintep100)
            .collect(toCollection(() -> new ArrayList<>()));

    List<List<TransCaixa>> result = new ArrayList<>();
    result.add(prim20);
    result.add(ult20);

    return result;


}
```

TreeSet Strems Paralelas

```
  TreeSet<TransCaixa> treeOrd = new TreeSet<>(transPorData);
  treeOrd.addAll(ltc);

  Supplier<List<TreeSet<TransCaixa>>> treeSuppar = () ->
  selec2treeparalelas(treeOrd,vintep100);

private List<TreeSet<TransCaixa>> selec2treeparalelas
   (TreeSet<TransCaixa> treeOrd, int vintep100){
    TreeSet<TransCaixa> prim20 = treeOrd.parallelStream()
            .limit(vintep100).collect(toCollection(() ->
            new TreeSet<>(Caixa.transPorData)));
    TreeSet<TransCaixa> ult20 = treeOrd.parallelStream()
            .sorted(Caixa.transPorData2).limit(vintep100)
            .collect(toCollection(() ->
            new TreeSet<>(Caixa.transPorData)));

    List<TreeSet<TransCaixa>> result = new ArrayList<>();
    result.add(prim20);
    result.add(ult20);


    return result;
}
```

3.2.4 Resultados

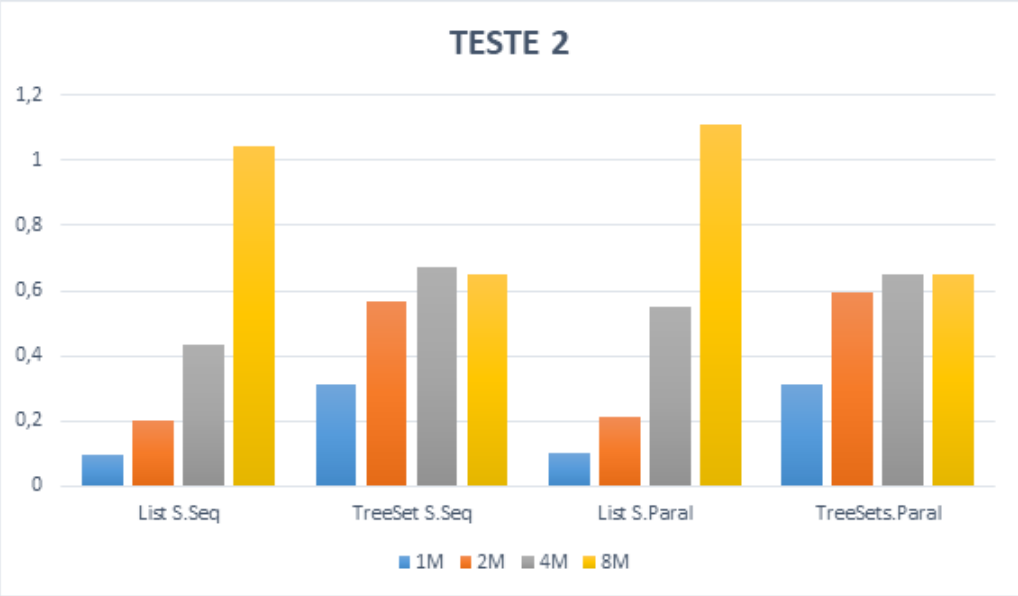

Figura 3.2: Gráfico Teste 2

3.2.5 Análises e Conclusões

Tendo em conta o gráfico dos resultados podemos concluir que é melhor nestes casos usar os Tree Sets independentemente de usarmos streams sequenciais ou paralelas pois ambas em todos os casos apresentam um tempo parecido.

3.3 Teste 3

3.3.1 Descrição

Crie uma IntStream, um int[] e uma List¡Integer¿ com de 1M a 8M de números aleatórios de valores entre 1 e 9_999. Determine o esforço de eliminar duplicados em cada situação.

3.3.2 Observações

O Teste 3 permite ao utilizador escolher se quer fazer o teste com 1 a 8 milhões de números aleatórios. Os ficheiros carregados não interferem com este teste.

3.3.3 Código

Criação Lista e Array Aleatórios

```
List<Integer> listaRand = randomList(i);
int[] arrayRand = listToArray(listaRand);
int tamanho = listaRand.size();

private int[] listToArray(List<Integer> lista){
    int i = 0;
    int[] arrayRand = new int[lista.size()];

    for(Integer t: lista){
        arrayRand[i] = t;
        i++;
    }

    return arrayRand;
}
```

IntStream

```
Supplier<IntStream> supplier1 = () ->
    listaRand.stream().mapToInt(l->l).distinct();
```

IntArray

```
    Supplier<int[]> supplier2 = () -> removeRepetidos(arrayRand);

     private int[] removeRepetidos(int[] array){
        Set<Integer> set = new HashSet<Integer>();
        int i;

        for(i=0;i<array.length;i++){
            set.add(array[i]);
        }
        i=0;
        int[] semRep = new int[set.size()];

        for(Integer s: set){
            semRep[i] = s;
            i++;
        }

        return semRep;
}
```

List<Integer>

```
    Supplier<List<Integer>> supplier3 = () ->
    listaRand.stream().distinct().collect(toList());
```

3.3.4 Resultados

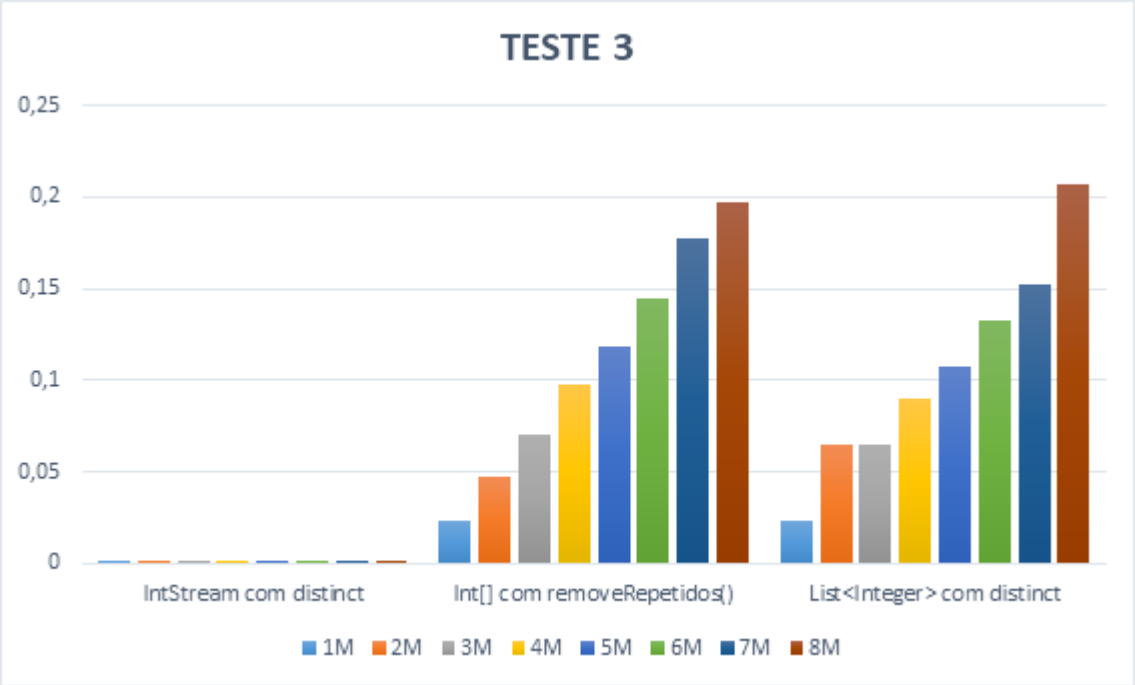

Figura 3.3: Gráfico Teste 3

3.3.5 Análises e Conclusões

Olhando para os resultados obtidos durante a realização dos testes, podemos concluir que usando uma IntStream com distinct, o seu esforço para remover repetidos a partir de uma lista gerada aleatoriamente é menor do que usar um array de inteiro com remoção de repetidos usando a função removeRep e também é menor do que usar um List<Integer> com distinct.

3.4 Teste 4

3.4.1 Descrição

Defina um método static, uma BiFunction e uma expressão lambda que dados dois inteiros calculam o resultado da sua divisão. Crie em seguida um int[] com sucessivamente 1M, 2M, 4M e 8M de inteiros. Finalmente processe o array de inteiros usando streams, sequenciais e paralelas, comparando os tempos de invocação e aplicação do método versus a bifunction e a expressão lambda explícita.

3.4.2 Observações

O Teste 4 permite ao utilizador escolher se quer fazer o teste com 1, 2, 4 ou 8 milhões de números aleatórios. Os ficheiros carregados não interferem com este teste.

3.4.3 Código

Sequencial Método Static Div

```
Supplier<Integer> supplier1 = () ->
Arrays.stream(array.clone()).reduce(1, (a,b)->div(a,b));
```

Sequencial BiFunction

```
Supplier<Integer> supplier2 = () ->
Arrays.stream(array.clone()).reduce(1,(a,b)->divisao.apply(a,b));
```

Sequencial Expressão Lambda

```
Supplier<Integer> supplier3 = () ->
Arrays.stream(array.clone()).reduce(1,(a,b)-> a!=0&&b!=0 ?
       (a>b ? a/b : b/a) : 1);
```

Paralelo Static Div

```
Supplier<Integer> supplier4 = () ->
Arrays.stream(array.clone()).parallel().reduce(1,(a,b)->div(a,b));
```

Paralelo BiFunction

```
Supplier<Integer> supplier5 = () ->
Arrays.stream(array.clone()).parallel().reduce(1,(a,b)->divisao.apply(a,b));
```

Paralelo Lambda

```
Supplier<Integer> supplier6 = () ->
Arrays.stream(array.clone()).parallel()
.reduce(1,(a,b)->a!=0&&b!=0 ? (a>b ? a/b : b/a) : 1);
```

3.4.4 Resultados

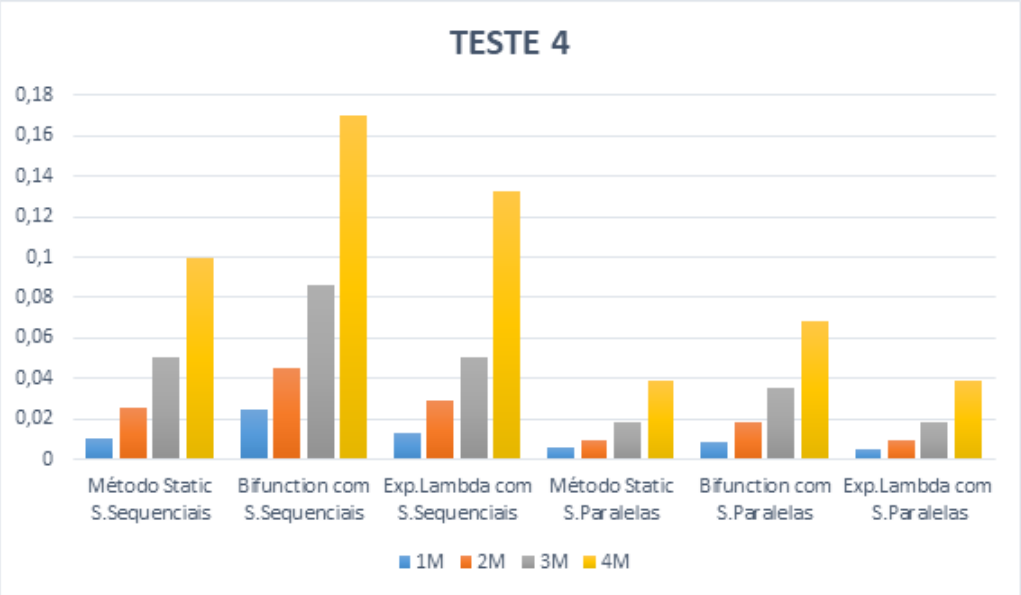

Figura 3.4: Gráfico Teste 4

3.4.5 Análises e Conclusões

Analisando os resultados obtidos no tópico anterior podemos concluir que o uso de Streams Parelelas é mais eficiente do que do que as Streams Sequencias para este caso de teste específico.

3.5 Teste 5

3.5.1 Descrição

Usando os dados disponíveis crie um teste que permita comparar se dada a List<TransCaixa> e um Comparator<TransCaixa>, que deverá ser definido, é mais eficiente, usando streams, fazer o collect para um TreeSet<TransCaixa> ou usar a operação sorted() e fazer o collect para uma nova List<TransCaixa>.

3.5.2 Observações

Nenhuma observação a fazer.

3.5.3 Código

List com Comparator

```
Supplier<List<TransCaixa>> listcomp = () -> listaComparador(ltc);

 List<TransCaixa> listaComparador(List<TransCaixa> ltc){
        return ltc.stream()
                .sorted(Caixa.transPorData).collect(toList());
     }
```

TreeSet com Sorted

```
Supplier<TreeSet<TransCaixa>> treesorte = () -> setsortetree(ltc);

TreeSet<TransCaixa> setsortetree(List<TransCaixa> ltc){
     return ltc.stream()
             .collect(toCollection(() ->
                new TreeSet<>(Caixa.transPorData)));
 }
```

3.5.4 Resultados

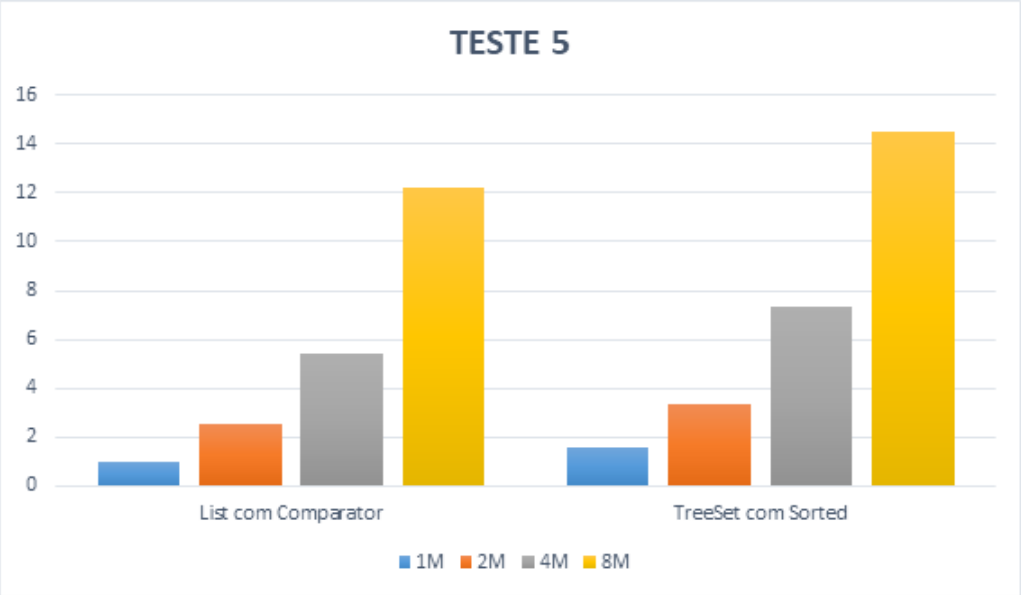

Figura 3.5: Gráfico Teste 5

3.5.5 Análises e Conclusões

Através da análise do gráfico que apresenta os resultados deste teste podemos concluir que o List com o Comparator é ligeiramente melhor do que o TreeSet com o Sorted

3.6 Teste 6

3.6.1 Descrição

Considere o exemplo prático das aulas de streams em que se criou uma tabela com as transacções catalogadas por Mês, Dia, Hora efectivos. Codifique em JAVA 7 o problema que foi resolvido com streams e compare tempos de execução.

3.6.2 Observações

Nenhuma observação a fazer.

3.6.3 Código

Com Streams

```
Supplier<Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>>>
supcatalgst = () -> criacat(ltc);

Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>>
criacat(List<TransCaixa> ltc) {

    return ltc.stream().collect(groupingBy(util ->
    util.getData().getMonth(),
                                groupingBy(util ->
                                util.getData().getDayOfMonth(),
                                groupingBy(util ->
                                util.getData().getHour()))));

}
```

Java 7

```
Supplier<Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>>>
supcatalgj7 = () -> criacatalogo(ltc);

Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>>
criacatalogo(List<TransCaixa> ltc){
    Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>> catalogo =
```

```
            new HashMap<>();

        for(TransCaixa trans : ltc){

            Month mes = trans.getData().getMonth();
            int dia = trans.getData().getDayOfMonth();
            int hora = trans.getData().getHour();
            List<TransCaixa> colhora;
            Map<Integer,List<TransCaixa>> coldia;
            Map<Integer,Map<Integer,List<TransCaixa>>> colmes;

            colmes = getColmes(catalogo,mes);
            coldia = getColdia(colmes,dia);
            colhora = getColHora(coldia,hora);
            colhora.add(trans);

        }
        return catalogo;
    }

    private Map<Integer, Map<Integer, List<TransCaixa>>> getColmes
      (Map<Month, Map<Integer, Map<Integer, List<TransCaixa>>>> catalogo,
      Month mes) {
        Map<Integer, Map<Integer, List<TransCaixa>>> colunames;

        if(!catalogo.containsKey(mes)){
            colunames = new HashMap<>();
            catalogo.put(mes, colunames);
        }
        else{
            colunames = catalogo.get(mes);
        }

        return colunames;

    }

    private Map<Integer, List<TransCaixa>> getColdia
       (Map<Integer, Map<Integer, List<TransCaixa>>> colmes, int dia) {
        Map<Integer, List<TransCaixa>> colunadia;
```

```
    if(!colmes.containsKey(dia)){
        colunadia = new HashMap<>();
        colmes.put(dia, colunadia);
    }
    else{
        colunadia = colmes.get(dia);
    }

    return colunadia;

}

private List<TransCaixa> getColHora(Map<Integer, List<TransCaixa>> coldia,
int hora) {
    List<TransCaixa> colunahora;

    if(!coldia.containsKey(hora)){
        colunahora = new ArrayList<>();
        coldia.put(hora, colunahora);
    }
    else{
        colunahora = coldia.get(hora);
    }

    return colunahora;
}
```

3.6.4 Resultados

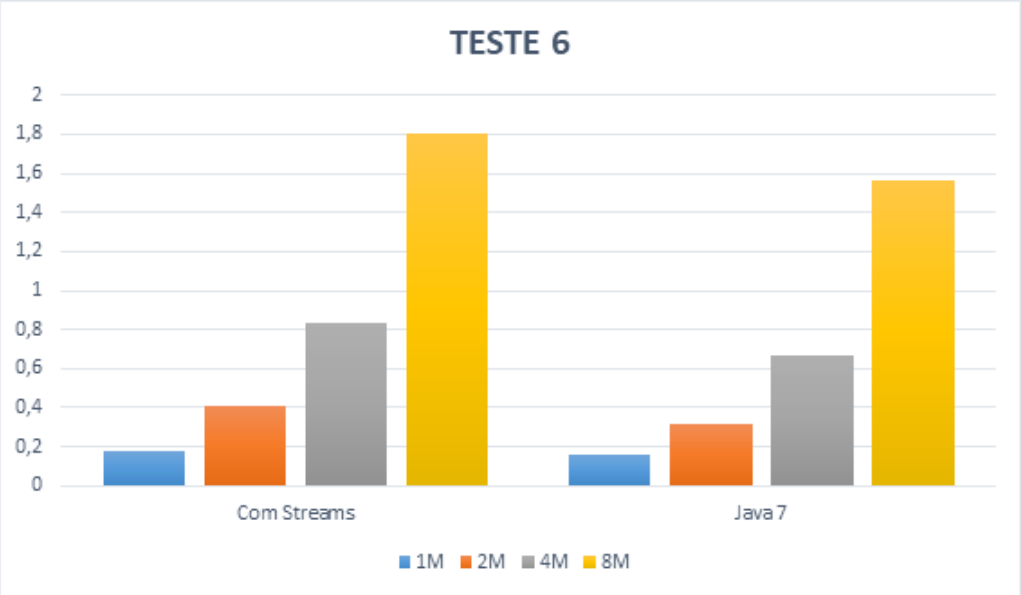

Figura 3.6: Gráfico Teste 6

3.6.5 Análises e Conclusões

Tendo em contas os dados do gráfico anterior usando o Java 7 tem uma ligeira vantagem sobre o uso de streams a nível do tempo, mas quando comparamos as linhas de código de escritas numa e noutra a alternativa que acabamos por escolher são as Streams. Tendo em consideração estes dois fatores de decisão mencionados atrás, as Streams apesar de não serem tão claras oferecem uma maior "limpeza"ao código e uma forma mais fácil de obter o pretendido.

3.7 Teste 7

3.7.1 Descrição

Usando List<TransCaixa> e Spliterator<TransCaixa> crie 4 partições cada uma com uma parte do data set. Compare os tempos de processamento de calcular a soma do valor das transacções com as quatro partições ou com o dataset inteiro, quer usando List< > e forEach() quer usando streams sequenciais e paralelas.

3.7.2 Observações

O esperado seria que o data set dividido em 4 partições demoraria menos tempo que o data set inteiro em relação as streams paralelas.

3.7.3 Código

Data Set Completo List

```
Supplier<Double> supplier1 = () -> forEach(ltc);
```

Data Set Completo Stream Sequencial

```
Supplier<Double> supplier2 = () ->
     ltc.stream().mapToDouble(l->l.getValor()).sum();
```

Data Set Completo Stream Paralela

```
Supplier<Double> supplier3 = () ->
ltc.parallelStream().mapToDouble(l->l.getValor()).sum();
```

Data Set Particionado Stream Sequencial

```
Supplier<Double> supplier4 = () -> spliteratorSeq();
```

Data Set Particionado Stream Paralela

```
Supplier<Double> supplier5 = () -> spliteratorParalelo();
```

3.7.4 Resultados

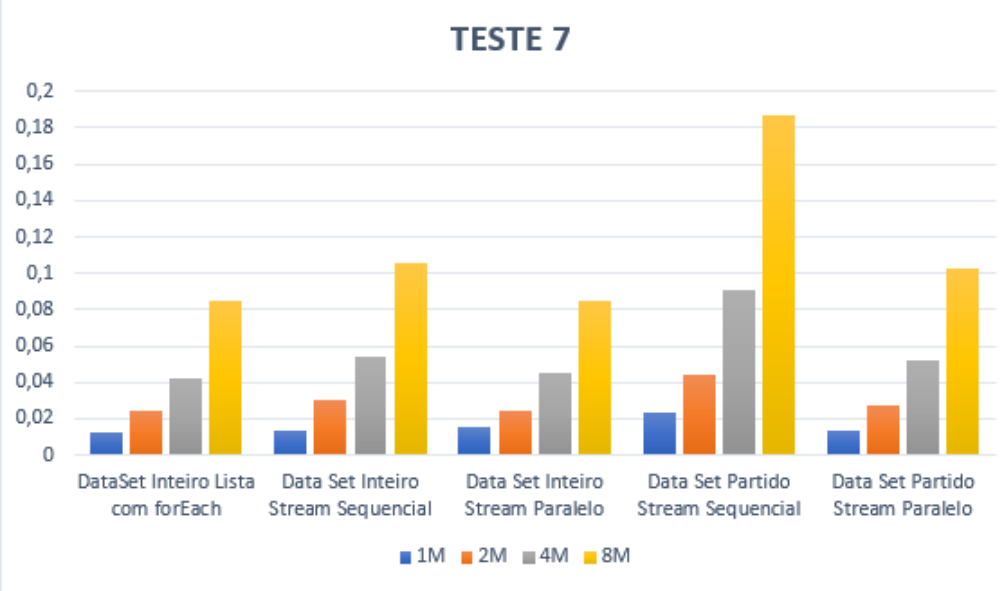

Figura 3.7: Gráfico Teste 7

3.7.5 Análises e Conclusões

Tendo em conta os resultados apresentados em cima podemos concluir que os resultados usando o data set completo apresentam tempos menores do que usando o data set com partições. Isso vai ficando cada vez mais evidente à medida que vamos aumentando o tamanho do input.

3.8 Teste 8

3.8.1 Descrição

Codifique em JAVA 7 e em Java 8 com streams, o problema de, dada a List<TransCaixa>, determinar o código da transacção de maior valor realizada num dado dia entre as 16 e as 20 horas.

3.8.2 Observações

Nenhuma observação a fazer.

3.8.3 Código

Java 8

```
Supplier<String> j8sup = () -> ltc.stream().filter(t ->
t.getData().getHour() > 15 && t.getData().getHour() < 21)
                   .max((t1, t2) -> Double.compare(t1.getValor(),
                   t2.getValor())).get().getTrans();
```

Java 7

```
Supplier<String> j7sup = () -> getCodigoMaior(ltc);

private String getCodigoMaior(List<TransCaixa> ltc){
    double maior = 0;
    String codigo = null;
    double valor;

    for(TransCaixa trans : ltc){
        int hora = trans.getData().getHour();
        if( hora > 15 && hora < 21){
            valor = trans.getValor();
            if(valor > maior) {
                maior = valor;
                codigo = trans.getTrans();
            }
        }
    }
```

```
        return codigo;
    }
```

3.8.4 Resultados

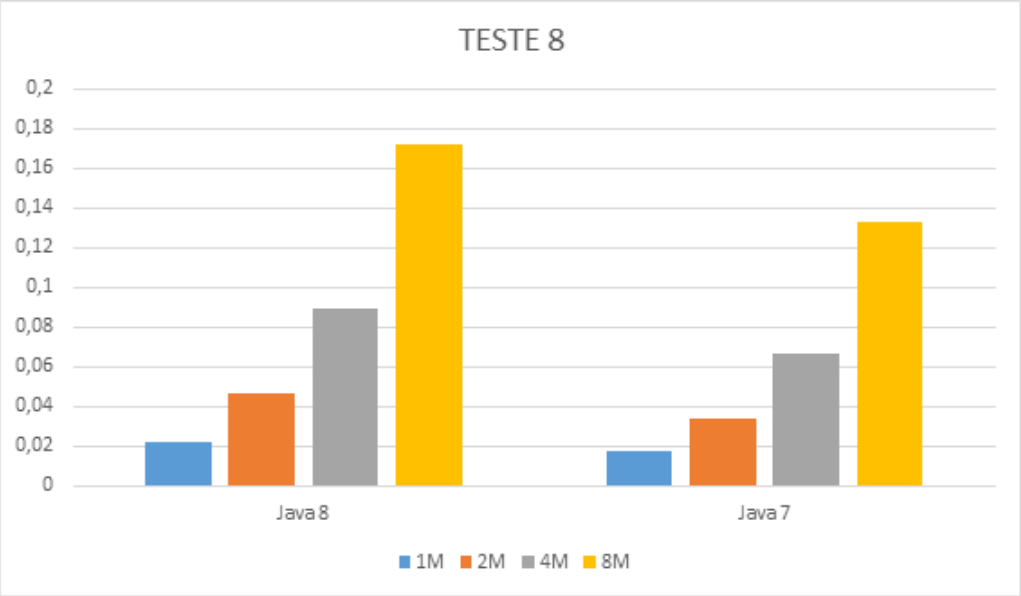

Figura 3.8: Gráfico Teste 8

3.8.5 Análises e Conclusões

Como podemos mais uma vez observar no gráfico de resultados verificamos que o java 7 á medida que vamos aumentando o input vai tornando-se mais rápido que o java 8. Estes tempos parecidos faz-nos levantar uma questão que já foi referida anteriormente noutro teste que é escolher a versão mais rápida e com elevado número de linhas ou a versão um bocado mais lenta mas engloba menos linhas de código, achamos que essa decisão vai depender um pouco do caso em estudo.

3.9 Teste 9

3.9.1 Descrição

Crie uma List<List<TransCaixa>> em que cada lista elemento da lista contém todas as transacções realizadas nos dias de 1 a 7 de uma dada semana do ano (1 a 52/53). Codifique em JAVA 7 e em Java 8 com streams, o problema de, dada tal lista, se apurar o total facturado nessa semana.

3.9.2 Observações

Este teste permite ao utilizador escolher o número da semana da qual pretende saber o total faturado.

3.9.3 Código

Java 7

```
Supplier<Double> j7sup = () -> faturadoSemana(ltc,sem);

private int semanaCalendario(LocalDateTime data) {
    return data.get(IsoFields.WEEK_OF_WEEK_BASED_YEAR);
   }

private List<List<TransCaixa>> listaSemanaTrans(List<TransCaixa> ltc){
     TreeMap<Integer,List<TransCaixa>> aux = new TreeMap<>();

     for(TransCaixa trans :ltc){
        LocalDateTime data = trans.getData();
        int nsem = semanaCalendario(data);
        List<TransCaixa> lsem = new ArrayList<>();

        if(aux.containsKey(nsem)){
            lsem = aux.get(nsem);
            lsem.add(trans);
            aux.put(nsem,lsem);
        }
        else{
            List<TransCaixa> ll = new ArrayList<>();
            ll.add(trans);
```

```
            aux.put(nsem,ll);
        }
    }

    List<List<TransCaixa>> lista = new ArrayList<>(aux.values());

    return lista;
}


private double faturadoSemana(List<TransCaixa> ltc , int semana){
    List<List<TransCaixa>> listasem = new ArrayList<>();
    listasem = listaSemanaTrans(ltc);

    List<TransCaixa> ll = new ArrayList<>();
    ll = listasem.get(semana);
    double valor = 0.0;

    for(TransCaixa tt : ll) {
        valor += tt.getValor();
    }

    return valor;
}
```

Java 8

```
Supplier<Double> j8sup = () ->
lista.get(sem).stream().map(t->t.getValor()).reduce(0.0, Double::sum);
```

3.9.4 Resultados

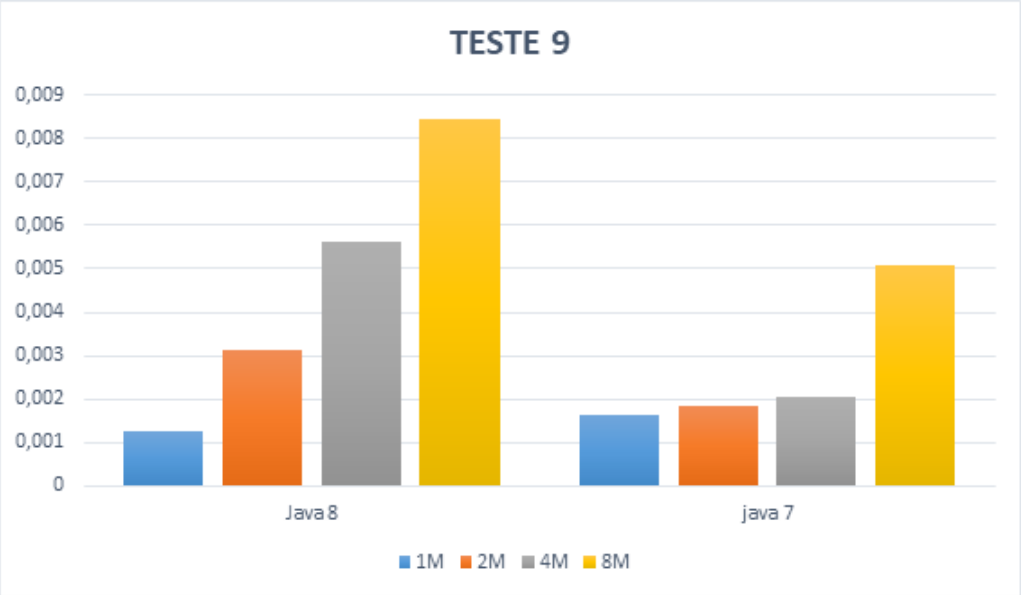

Figura 3.9: Gráfico Teste 9

3.9.5 Análises e Conclusões

Como podemos mais uma vez observar no gráfico de resultados verificamos que o java 7 á medida que vamos aumentando o input vai tornando-se mais rápido que o java 8.

3.10 Teste 10

3.10.1 Descrição

Admitindo que o IVA a entregar por transacção é de 15% para transacções menores que 20 Euros, 20% entre 20 e 29 e 23% para valores superiores, crie uma tabela com o valor de IVA total a pagar por mês. Compare as soluções em JAVA 7 e Java 8.

3.10.2 Observações

Nenhuma observação a fazer.

3.10.3 Código

Java 7

```
Supplier<Map<Month,Double>> j7sup = () -> ivapormes(ltc);


private Map<Month,Double> ivapormes(List<TransCaixa> ltc) {
    Map<Month,Double> ivam = new TreeMap<>();

    for(TransCaixa trans : ltc){

        Month mes = trans.getData().getMonth();
        double valor = trans.getValor();
        double valoriva = 0;

        if (valor < 20.0) {
            valoriva = valor * 0.15;
        }
        else if ( valor >= 20.0 && valor < 29.0 ){
            valoriva = valor * 0.20;
        }
        else if (valor >= 29.0){
            valoriva = valor * 0.23;
        }
```

```
            adicionaiva(mes,valoriva,ivam);

        }

        return ivam;
    }

    private void adicionaiva(Month mes, double valoriva,
      Map<Month, Double> ivam) {

        if(ivam.containsKey(mes)){
            ivam.put(mes, ivam.get(mes) + valoriva);
        }
        else {
            ivam.put(mes, valoriva);
        }
    }
```

Java 8

```
Supplier<Map<Month,Double>> j8sup = () -> ltc.stream()
                .collect(Collectors.groupingBy(t ->
     t.getData().getMonth(),TreeMap::new,
                 Collectors.summingDouble(t->t.getValor()<20.0 ?
                  t.getValor() *0.15 :
                     (t.getValor() >= 20.0 && t.getValor()<29.0
                                 ? t.getValor()*0.20 :
                                 t.getValor()*0.23))));
```

3.10.4 Resultados

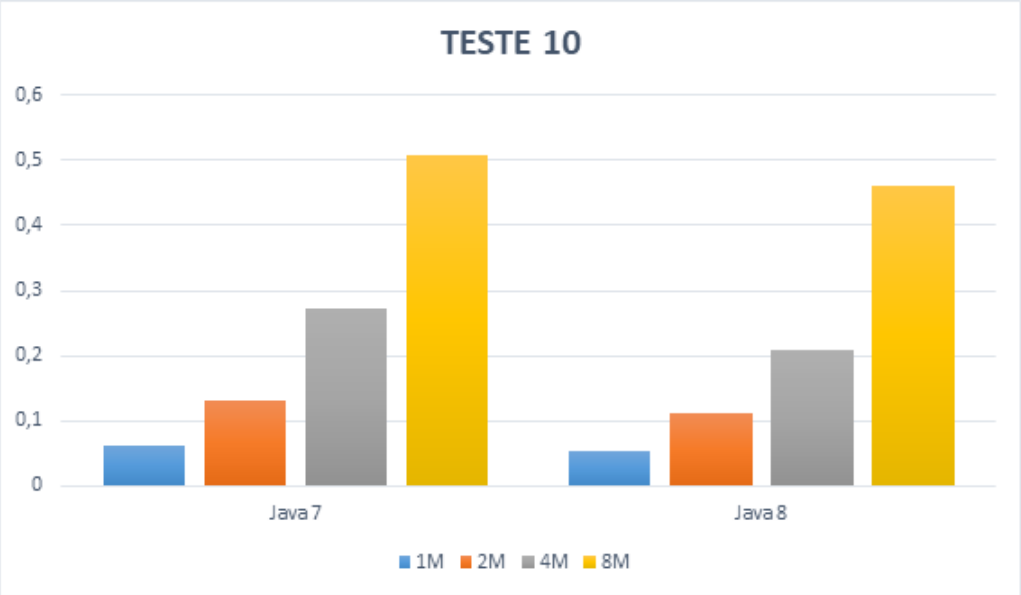

Figura 3.10: Gráfico Teste 10

3.10.5 Análises e Conclusões

Pelos resultados presentes em cima podemos observar que o Java 8 é ligeiramente mais eficiente quando temos de fazer operações em todos os elementos e isso vai notando-se à medida que vamos aumento o tamanho do ficheiro de input.

3.11 Teste 11

3.11.1 Descrição

Seleccione 4 exemplos de processamento com streams que programou nestes testes. Compare os tempos encontrados em Java 8 com os tempos obtidos usando JDK 9, quer em processamento sequencial quer em paralelo.

3.11.2 Observações

Nenhuma observação a fazer.

3.11.3 Código

Teste 1

```
//STREAM SEQUENCIAL
Supplier<Double> supplier1 = () ->
ltc.stream().mapToDouble(l->l.getValor()).sum();

//STREAM PARALELO
Supplier<Double> supplier2 = () ->
ltc.parallelStream().mapToDouble(l->l.getValor()).sum();
```

Teste 2

```
// STREAM SEQUENCIAL
Supplier<List<List<TransCaixa>>> listSup = () -> selec2listas(listaOrd,
vintep100);

// STREAM PARELELO
Supplier<List<List<TransCaixa>>> listSuppar = () ->
selec2listasparalelas(listaOrd, vintep100);
```

Teste 5

```
Supplier<List<TransCaixa>> listcomp = () -> listaComparador(ltc);
```

Teste 8

```
Supplier<String> sup = () -> ltc.stream().filter(t ->
t.getData().getHour() > 15 && t.getData().getHour() < 21)
                                        .max((t1, t2) ->
                                        Double.compare(t1.getValor(),
                                        t2.getValor())).get().getTrans();
```

3.11.4 Resultados

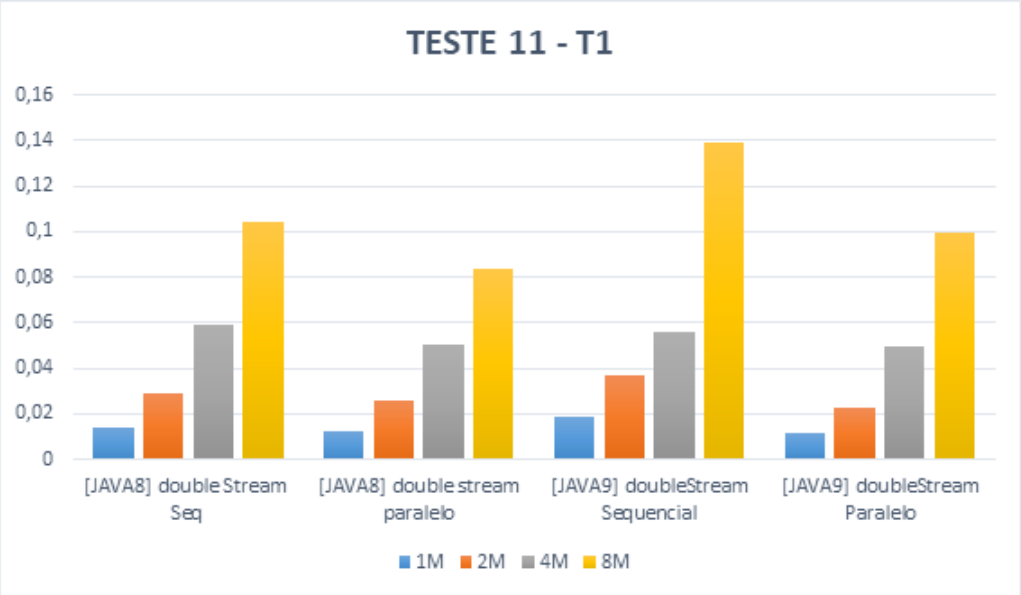

Figura 3.11: Gráfico Teste 11 - V1

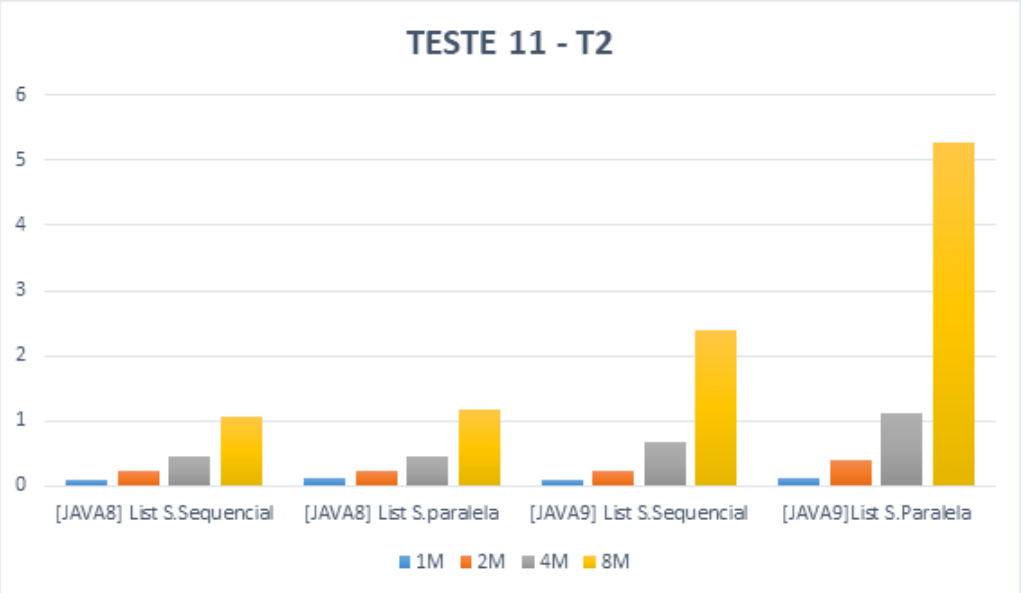

Figura 3.12: Gráfico Teste 11 - V2

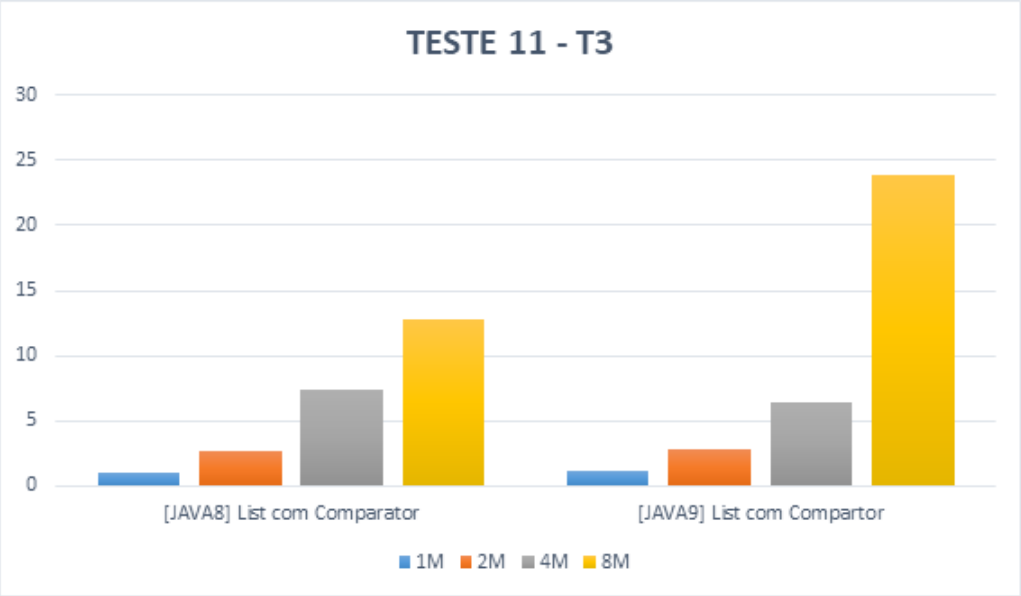

Figura 3.13: Gráfico Teste 11 - V3

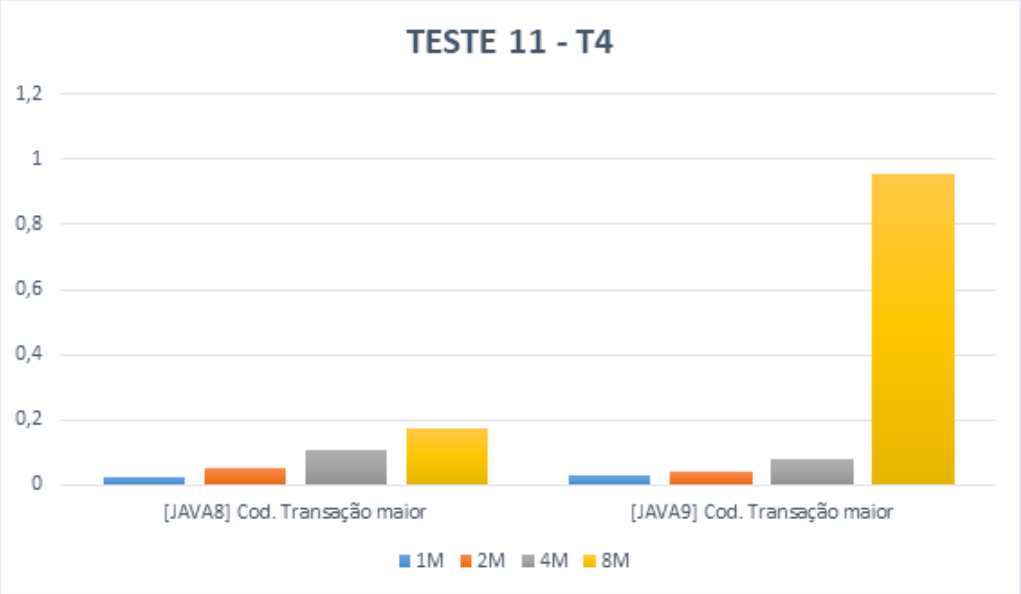

Figura 3.14: Gráfico Teste 11 - V4

3.11.5 Análises e Conclusões

Verificamos que para os testes efetuados no teste 11 existe uma perda de performance do java 8 para o java 9, perda essa não esperada. Apesar

disso, foi verificado que o uso de streams paralelas obteve um menor tempo de execução para o teste 1 ao contrário do teste 2 onde o tempo de execução em paralelo piorou.

3.12 Teste 12

3.12.1 Descrição

Considerando List < TransCaixa > criar uma tabela que associa a cada nº de caixa uma tabela contendo para cada mês as transacções dessa caixa. Desenvolva duas soluções, uma usando um Map<> como resultado e a outra usando um ConcurrentMap(). Em ambos os casos calcule depois o total facturado por caixa em Java 8 e em Java 9.

3.12.2 Observações

Usamos uma ConcurrentSkipListMap.

3.12.3 Código

Resultado com o Map

```
Map<String,Map<Month,List<TransCaixa>>>
transMap = getMap(ltc);

    Supplier<Map<String,Double>> supplierMap = ()->
    transMap.entrySet().stream()

        .collect(Collectors.toMap(t->t.getKey(),
            t->t.getValue().values().stream().mapToDouble(l->l.stream()
            .collect(Collectors.summingDouble(s->s.getValor())))).sum()));

SimpleEntry<Double,Map<String,Double>> resultMap =
  Utilidades.testeBoxGenW(supplierMap);
```

Resultado com o Concurrent Map

```
Map<String,Map<Month,List<TransCaixa>>> transConcurrentMap =
 getConcurrentMap(ltc);

Supplier<Map<String,Double>> supplierConcurrentMap = () ->
   transConcurrentMap.entrySet().stream().
```

```
.mapToDouble(l->l.stream()
.collect(Collectors.summingDouble(s->s.getValor()))).sum()));
```

3.12.4 Resultados

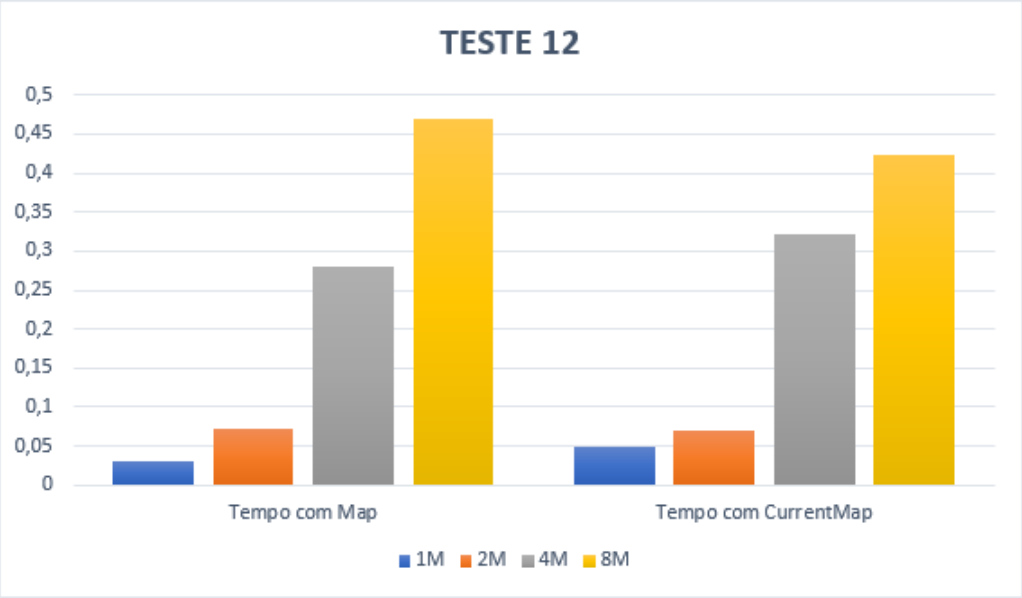

Figura 3.15: Gráfico Teste 12 - JAVA 8

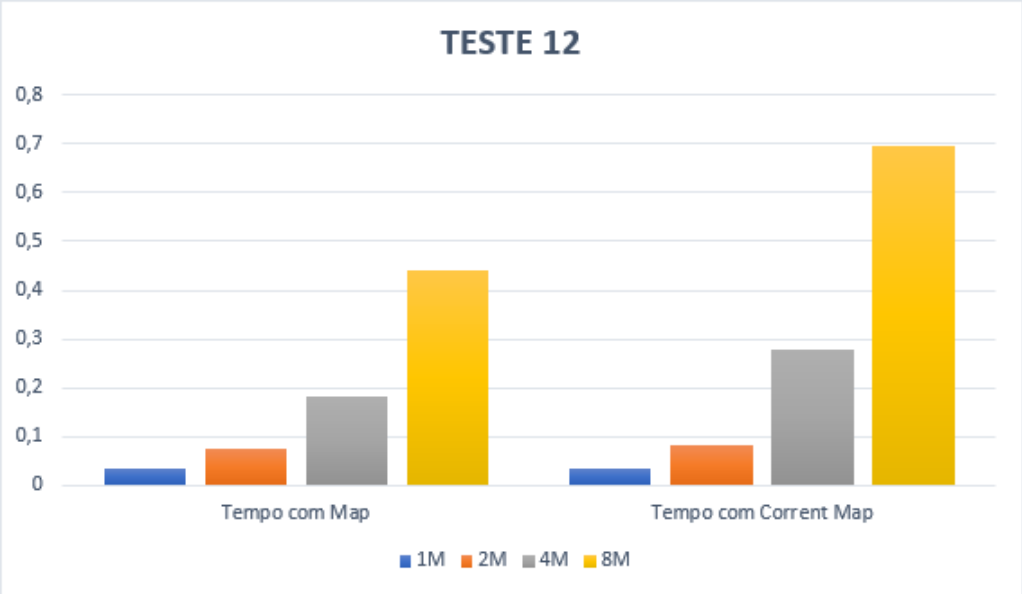

Figura 3.16: Gráfico Teste 12 - JAVA 9

3.12.5 Análises e Conclusões

Analisando os resultados obtidos com Java 8 e Java 9 podemos concluir que usando o Java 9 vamos obter tempos mais rápidos ao longo da sua execução. E obtivemos um melhor tempo usando CorrentMap no Java 8, enquanto no Java 9 verificou-se o oposto.