Benchmarking Java Streams

Processamento de Dados com Streams de JAVA

Afonso Silva, a70387 Octávio Maia, a71369

Conteúdo

Parte I.

Contextualização e objetivos

Introdução

Pretende-se com este relatório dar a conhecer as diferenças de *performance* na análise de grandes sequências de dados recorrendo quer a técnicas tradicionais do JAVA7 ou a técnicas mais modernas do JAVA8 e JAVA9.

Para tal

Neste relatório, para um dos testes analisados, encontra-se o código-fonte que lhe diz respeito, bem como

Execução

Todos os testes deste projeto foram desenvolvidos no mesmo programa, que se encontra hospedado no GitHub:

https://github.com/OctavioMaia/PDSJ_TP2

O executável pode ser descarregado diretamente aqui e é corrido desta forma:

```
$ java -jar bjs.jar T
```

onde T é o número do teste que se pretende executar.

O resultado do teste será um tabela CSV que contém na primeira linha os nomes dos **indicadores** executados e na primeira coluna os nomes dos **inputs** indicados. Este CSV pode ser facilmente incluído numa folha de cálculo para melhor análise estatística do mesmo, ou numa table em formato de documento.

Implementação

De forma a uniformizar e facilitar a implementação de todos os testes requisitados, optou-se por implementar uma **interface** comun a todos os testes. Deste modo cada teste corresponderá a uma **classe** que implementa esta mesma interface.

Assume-se também que cada teste é composto por um conjunto de indicadores, que mais não são o conjunto dos métodos que o teste deve executar. Para além disso, cada teste pode receber ou não, um determinado **input** (por exemplo o conjunto de todas as transações realizadas, ou o número de números aleatórios a gerar).

```
public interface Test {
    /**
     * Returns a textual description of the input of this test.
     * If no input is given, then it should return an empty Optional.
     *
     * @return textual description of the input if it exists.
     */
```

```
    default Optional<String> input() {
        return Optional.empty();
    }

    /**
     * Returns all the indicators that this test should run.
     * Each indicator is an association between its textual description
     * and the function (Supplier) that it should run.
     *
     * @return Amap, where the keys are the textual description of the
     *         indicator, and the values are the supplier to run.
     */
    Map<String, Supplier<?>> indicators();

    /**
     * Calculates the times needed to run each one of the indicators.
     *
     * @return a map, where the keys are the textual description of the
     *         indicator, and the values are theirs results in seconds.
     */
    default Map<String, Double> results() {
        return indicators().entrySet().stream().collect(Collectors.toMap(
                Map.Entry::getKey,
                entry -> measure(entry.getValue()).getKey()
        ));
    }

    /**
     * Measures the required time to run the given supplier.
     * Before the supplier is ran, it does a 5 turns warmup and
     * cleans the memory.
     * @author FMM
     *
     * @param supplier supplier to run
     * @param <R> type of the result of the supplier
     * @return An entry composed by the time in seconds that the
     *         supplier took to run, and the result of that supplier.
     */
    static <R> SimpleEntry<Double, R> measure(Supplier<R> supplier) {
        for (int i = 0; i < 5; i++) supplier.get();
        System.gc();

        Crono.start();
        R result = supplier.get();
        Double time = Crono.stop();
        return new SimpleEntry<>(time, result);
    }
}
```

Com a utilização desta **interface** torna-se então bastante intuitiva a criação de novos testes. A classe principal do programa fica então responsável por ler o número do teste escolhido pelo utilizador e por corrê-lo com o **input** especificado.

Parte II.

Testes

Todos os testes aqui expostos foram executados num máquina com as seguintes especificações:

```
Windows 10 Pro v1607 (64 bits)
Intel Core i5 4690k (4c/4t)
16 GB RAM
JDK8_144, JDK9_01
```

1. Cálculo dos valores de transações registadas

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public double sumArray() {
    double[] values = new double[this.transactions.size()];
    int i = 0;

    for (TransCaixa transaction : this.transactions) {
        values[i++] = transaction.getValor();
    }

    double sum = 0.0;

    for (i = 0; i < values.length; i++) {
        sum += values[i];
    }

    return sum;
}
```

Listing 1.1: Cálculo da soma dos valores das transações atraveś de um array do tipo double

```
public double sumDoubleStream() {
    DoubleStream values = this.transactions.stream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}

public double sumDoubleStreamP() {
    DoubleStream values = this.transactions.parallelStream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}
```

Listing 1.2: Cálculo da soma dos valores das transações através de uma DoubleStream

```
public double sumStream() {
    Stream<Double> values = this.transactions.stream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}

public double sumStreamP() {
    Stream<Double> values = this.transactions.parallelStream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}
```

Listing 1.3: Cálculo da soma dos valores das transações através de Stream<Double>

Resultados

Input	(1) sum Array	(2) sum DoubleStream	(3) sum DoubleStreamP	(4) sum Stream	(5) sum StreamP
1000000 transactions	0,010215	0,010794	0,009364	0,026481	0,015645
2000000 transactions	0,021934	0,021800	0,009912	0,053940	0,026226
4000000 transactions	0,044380	0,047799	0,026682	0,113063	0,051981
8000000 transactions	0,078939	0,089719	0,046436	0,214286	0,126348

Análise e conclusões

Após uma breve observação dos gráficos, podemos afirmar que a estrutura de dados mais adequada é de facto a *DoubleStreamP* que implementa streams paralelas. Em contraste, a pior estrutura em nível de performance é a *Stream*, sendo até 5 vezes mais lenta que a *DoubleStreamP*.

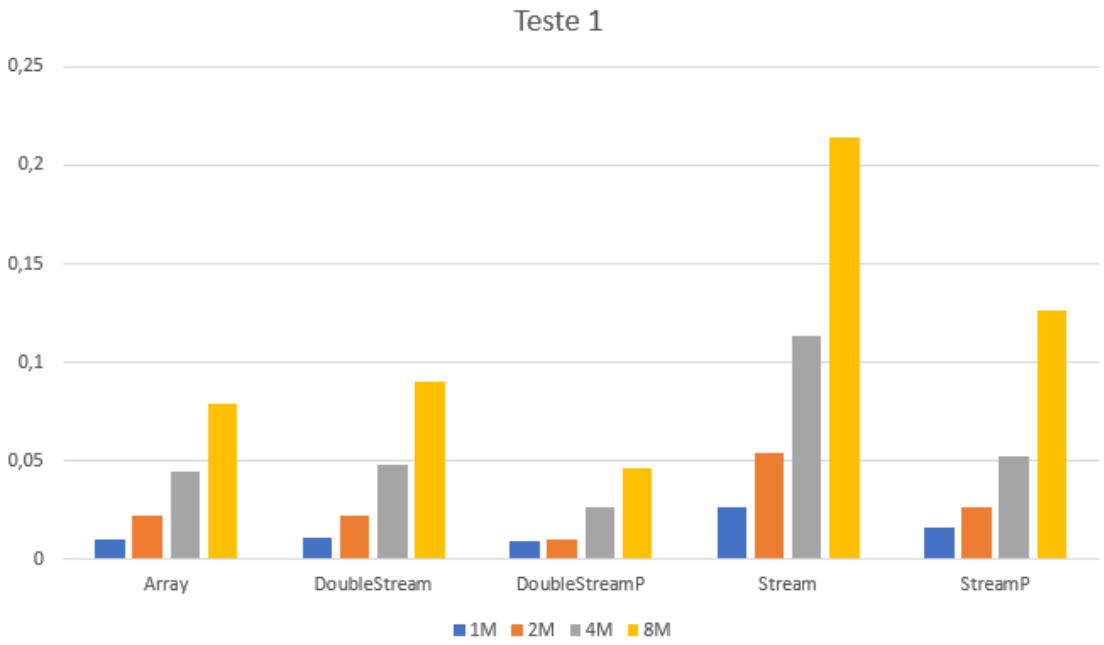

Figura 1.1.: Representação gráfica destes resultados

2. Extração dos primeiros e últimos 20% de transações realizadas

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateList() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> sorted = new ArrayList<>(this.transactions);
    sorted.sort(Comparator.comparing(TransCaixa::getData));

    List<TransCaixa> first = sorted.subList(0, nelems);
    List<TransCaixa> last = sorted.subList(
            sorted.size() - 1 - nelems, sorted.size() - 1);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateSet() {
    int nelems = 20 * this.transactions.size() / 100;

    TreeSet<TransCaixa> sorted = new TreeSet<>(
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1
    );
    sorted.addAll(this.transactions);

    List<TransCaixa> first = new ArrayList<>(sorted)
            .subList(0, nelems);
    List<TransCaixa> last = new ArrayList<>(sorted.descendingSet())
            .subList(0, nelems);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStream() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
            .sorted(Comparator.comparing(TransCaixa::getData))
```

```
                .limit(nelems)
                .collect(Collectors.toList());
        List<TransCaixa> last = this.transactions.stream()
                .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
                .limit(nelems)
                .collect(Collectors.toList());

        return new SimpleEntry<>(first, last);
    }
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStreamP() {
        int nelems = 20 * this.transactions.size() / 100;

        List<TransCaixa> first = this.transactions.stream()
                .sorted(Comparator.comparing(TransCaixa::getData))
                .limit(nelems)
                .collect(Collectors.toList());
        List<TransCaixa> last = this.transactions.stream()
                .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
                .limit(nelems)
                .collect(Collectors.toList());

        return new SimpleEntry<>(first, last);
    }
```

Resultados

Input	(1) byDateList	(2) byDateSet	(3) byDateStream	(4) byDateStreamP
1000000 transactions	1,029137	1,939311	1,861916	2,239986
2000000 transactions	2,331822	3,706627	3,858658	4,378531
4000000 transactions	5,745454	9,411794	9,281572	10,556521
8000000 transactions	12,635500	21,085741	20,579760	22,002980

Análise e conclusões

Após uma breve análise do gráfico podemos concluir que a implementação usando *List* é a mais rápida por uma margem substancial (quase 2x mais rápida que as outras alternativas implementadas). Curiosamente a implementação paralela utilizando Streams, *byDateStreamP* é mais lenta que a sua implementação sequencial, *byDateStream*. Podemos concluir que para os dataset fornecido a paralelização não é benéfica, antes pelo contrário. Por fim, a implementação usando Sets e Streams sequenciais têm performance quase

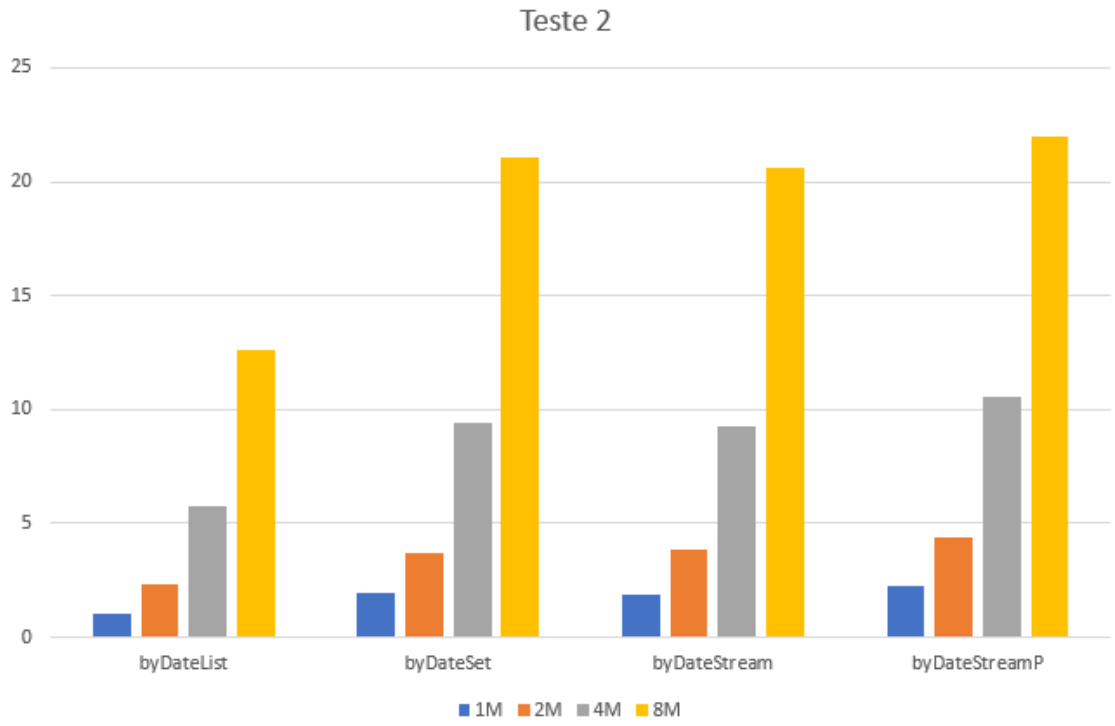

Figura 2.1.: Representação gráfica destes resultados

idênticas, sendo a sua diferença em tempos de execução na casa dos 5%.

3. Esforço de eliminação de duplicados

Observações

Este teste recebe como input a quantidade de números a gerar.

Métodos a testar

```
public Integer[] uniqueArray() {
    Set<Integer> nodups = new HashSet<>();
    for (int value : this.values) {
        nodups.add(value);
    }

    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.1: Eliminação dos duplicados através de um array de inteiros

```
public Integer[] uniqueList() {
    List<Integer> aux = new ArrayList<>();
    for (int value : this.values) {
        aux.add(value);
    }

    List<Integer> nodups = new ArrayList<>(new HashSet<>(aux));
    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.2: Eliminação dos duplicados através de uma lista de inteiros

```
public int[] uniqueIntStream() {
    IntStream values = new Random().ints(this.values.length, 0, 9999);
    return values.distinct().toArray();
}
```

Listing 3.3: Eliminação dos duplicados através de uma stream de inteiros

Resultados

Input	(1) uniqueArray	(2) uniqueList	(3) uniqueIntStream
1000000 random numbers	0,014765	0,027333	0,022026
2000000 random numbers	0,028226	0,055699	0,045615
4000000 random numbers	0,055047	0,112336	0,087037
8000000 random numbers	0,113187	0,296811	0,177688

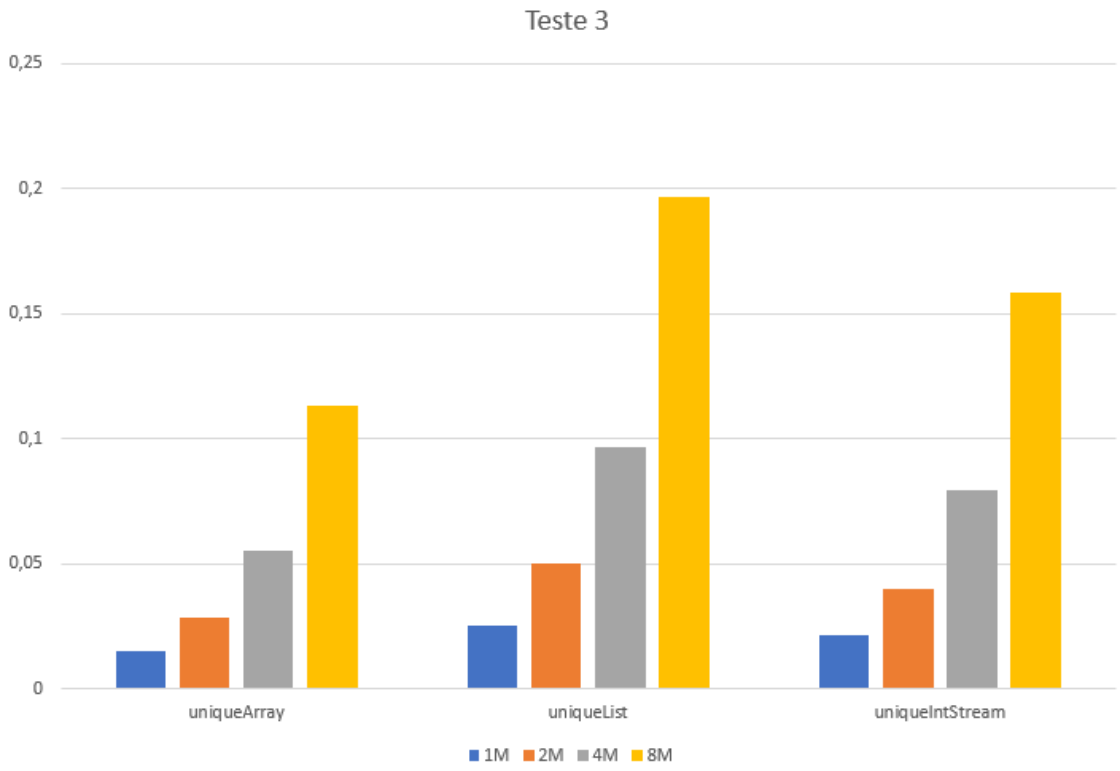

Figura 3.1.: Representação gráfica destes resultados

Análise e conclusões

Após a respetiva geração de 1, 2, 4 ou 8 milhões de números aleatórios compreendidos entre 1 e 9999, procedemos à remoção dos elementos duplicados. A alternativa mais viável para tal feito demonstrou tratar-se do *uniqueArray* que utiliza um *HashSet* como estrutura

auxiliar. A segunda melhor alternativa seria a *uniqueStream* que utiliza o operando *distinct()* seguindo de uma coleção para array. Em último lugar vem a *uniqueList*, que embora também use um *HashSet* como estrutura auxiliar como o *uniqueArray* tem uma performance muito pior, tal que o tempo de execução do *uniqueArray* para 8 milhões de números aleatórios é igual ao tempo de execução do *uniqueList* para 4 milhões.

4. Comparação entre a aplicação de método estático, BiFunction e Lambda

Observações

Este teste recebe como input a quantidade de números a gerar.

Métodos a testar

```
public static int div(int x, int y) {
    return x / y;
}

public int[] divSMethodStream() {
    return Arrays.stream(this.values)
        .map(x -> div(x, 2)).toArray();
}

public int[] divSMethodStreamP() {
    return Arrays.stream(this.values).parallel()
        .map(x -> div(x, 2)).toArray();
}
```

Listing 4.1: Divisão de todos os números por 2 através de um método estático

```
public int[] divBiFunStream() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values)
        .map(x -> f.apply(x, 2)).toArray();
}

public int[] divBiFunStreamP() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values).parallel()
        .map(x -> f.apply(x, 2)).toArray();
}
```

Listing 4.2: Divisão de todos os números por 2 através de uma BiFunction

```
public int[] divLambdaStream() {
    return Arrays.stream(this.values).map(x -> x / 2).toArray();
}

public int[] divLambdaStreamP() {
    return Arrays.stream(this.values).parallel().map(x -> x / 2).toArray();
```

```
}
```

Listing 4.3: Divisão de todos os números por 2 através de um Lambda

Resultados

Input	(1) divS Method Stream	(2) divS Method StreamP	(3) div BiFun Stream	(4) div BiFun StreamP	(5) div Lambda Stream	(6) div Lambda StreamP
1000000 random numbers	0,004767	0,001324	0,007617	0,004647	0,004802	0,002074
2000000 random numbers	0,008992	0,003859	0,015164	0,007746	0,008965	0,002696
4000000 random numbers	0,018092	0,008287	0,030338	0,017974	0,023628	0,007737
8000000 random numbers	0,036636	0,010449	0,060451	0,032753	0,036349	0,012460

Análise e conclusões

Embora neste dataset todos os métodos tenham uma performance excelente, na casa das centésimas de segundo, embora a implementação do *divSMethodStreamP* e *divLambdaStreamP* sejam as melhores, sendo a sua performance idêntica (diferenças na casa das milésimas de segundo). O pior método de implementação trata-se da *divBiFunStream* que utiliza uma *BiFunction* para o cálculo da metade do número em causa.

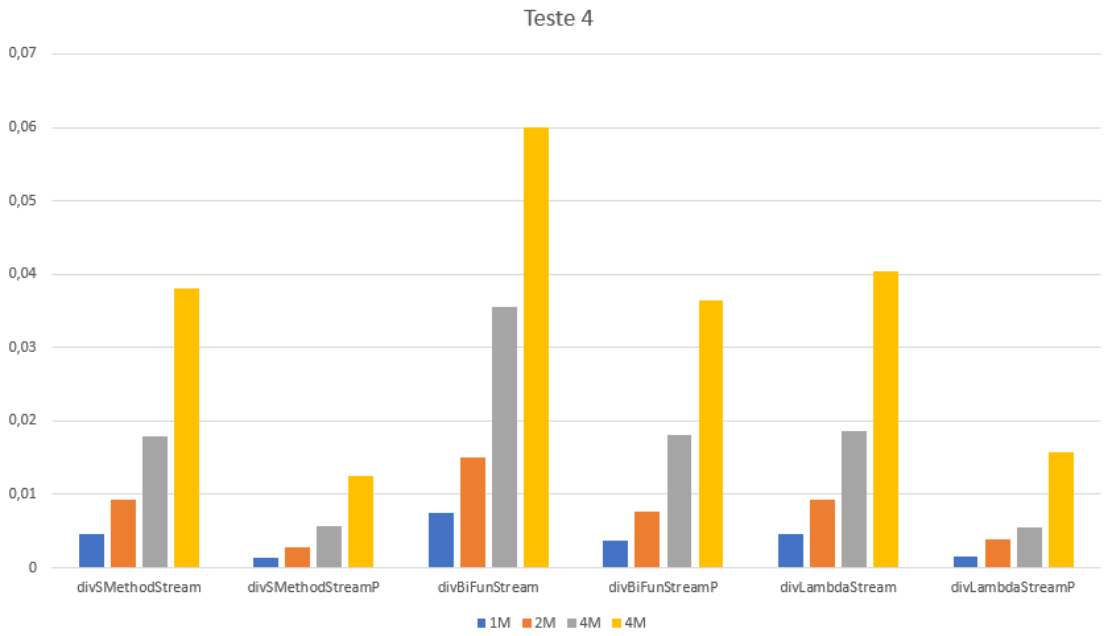

Figura 4.1.: Representação gráfica destes resultados

5. Comparação da ordenação através de um TreeSet e do método `sorted`

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public TreeSet<TransCaixa> sortTreeSet() {
    Comparator<TransCaixa> byDate =
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1;

    return this.transactions.stream()
        .collect(Collectors.toCollection(
                () -> new TreeSet<>(byDate)));
}
```

Listing 5.1: Ordenação através de um TreeSet

```
public List<TransCaixa> sortList() {
    Comparator<TransCaixa> byDate =
        Comparator.comparing(TransCaixa::getData);

    return this.transactions.stream()
        .sorted(byDate).collect(Collectors.toList());
}
```

Listing 5.2: Ordenação através do método sorted

Resultados

Input	(1) sortTreeSet	(2) sortList
1000000 transactions	1,510102	0,849679
2000000 transactions	3,449318	1,840600
4000000 transactions	8,242589	4,047586
8000000 transactions	19,584347	8,851000

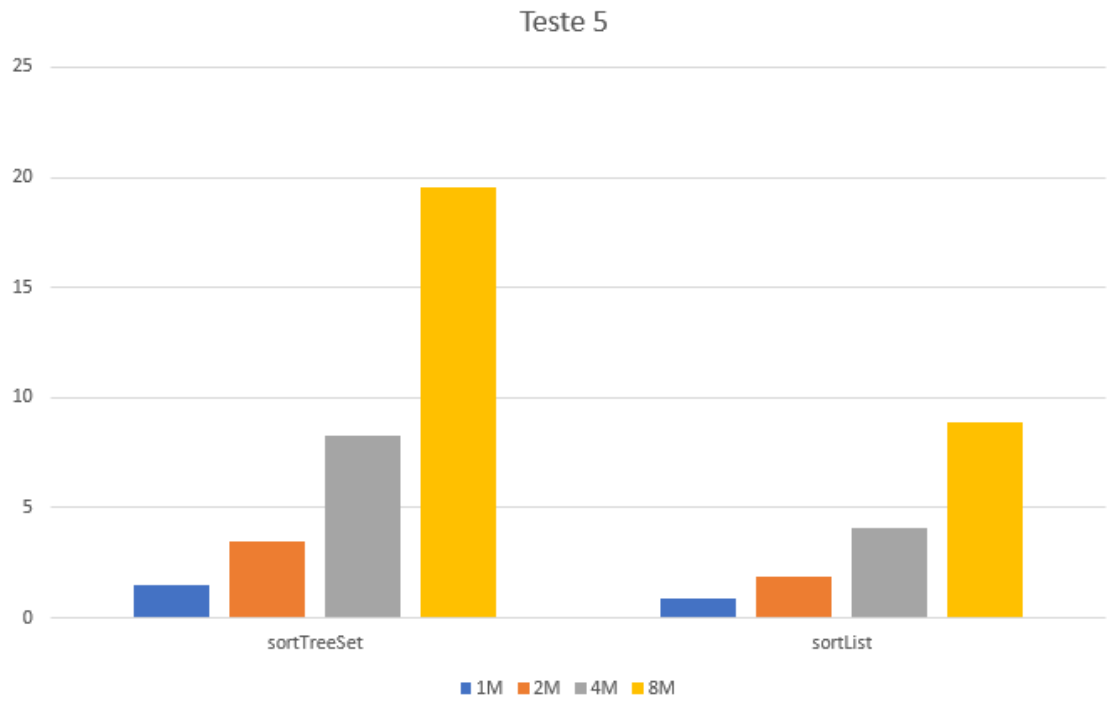

Figura 5.1.: Representação gráfica destes resultados

Análise e conclusões

Como podemos verificar pela breve análise do gráfico e tabela, a função *sortList* aplica o método *sorted* a uma *Stream*, tornado a sua performance vastamente superior ao *TreeSet* usado pela função *sortTreeSet*. Pelo gráfico gerado a partir da tabela, verificamos que a função *sortList* é, em média, 2x mais rápida que a *sortTreeSet*.

6. Catálogo de transações por Mês, Dia e Hora

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public Map<LocalDateTime, List<TransCaixa>> catalog() {
    Map<LocalDateTime, List<TransCaixa>> catalog = new TreeMap<>();

    for (TransCaixa transaction : this.transactions) {
        if (!catalog.containsKey(transaction.getData())) {
            catalog.put(transaction.getData(), new ArrayList<>());
        }

        catalog.get(transaction.getData()).add(transaction);
    }

    return catalog;
}
```

Listing 6.1: Obtenção do catálogo recorrendo a técnicas do JAVA7

```
public Map<LocalDateTime, List<TransCaixa>> catalogStream() {
    return this.transactions.stream()
            .collect(Collectors.groupingBy(TransCaixa::getData));
}
```

Listing 6.2: Obtenção do catálogo recorrendo a Streams

Resultados

Input	(1) catalog	(2) catalogStream
1000000 transactions	1,110954	0,283002
2000000 transactions	2,578776	0,634677
4000000 transactions	4,683762	1,300415
8000000 transactions	11,734155	3,324781

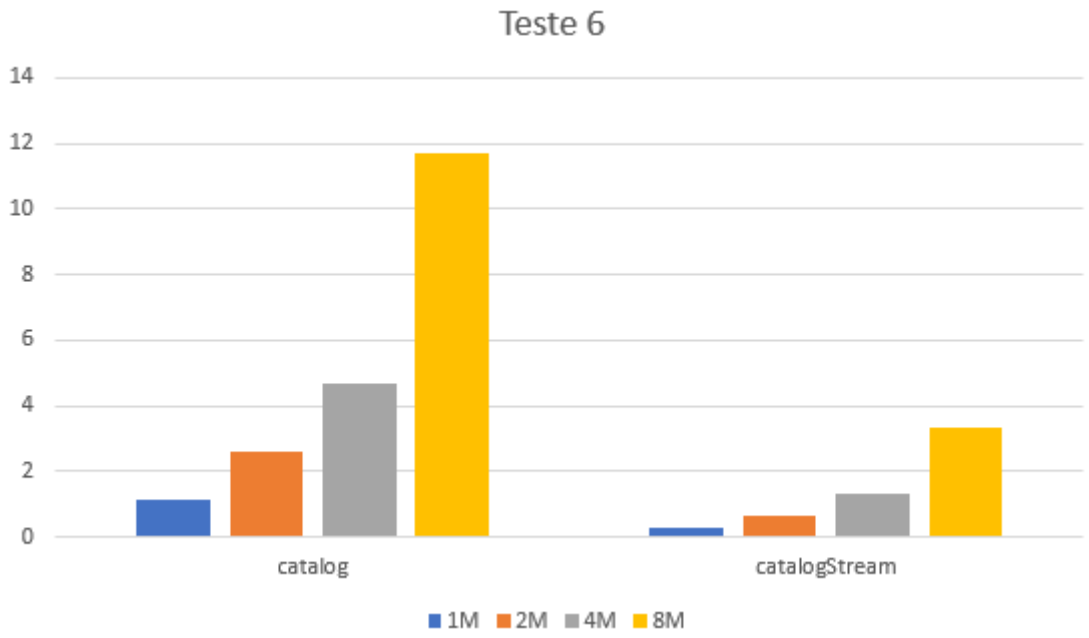

Figura 6.1.: Representação gráfica destes resultados

Análise e conclusões

No catálogo de transações foi utilizado o *TreeMap*, devido à ordenação, útil para

7. Comparação do cálculo da soma com ou sem partições

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public double sum() {
    double res = 0.0;
    for (TransCaixa transCaixa : this.transactions) {
        res += transCaixa.getValor();
    }        return 0.0

    return res;
}

public double sumPartition() {
    // Make the partitions
    List<List<TransCaixa>> partitions = new ArrayList<>();
    int[] pos = {
        0,
        this.transactions.size() - 3 * transactions.size() / 4,
        this.transactions.size() - 2 * transactions.size() / 4,
        this.transactions.size() - 1 * transactions.size() / 4,
        this.transactions.size() - 0 * transactions.size() / 4,
    };

    partitions.add(this.transactions.subList(pos[0], pos[1]));
    partitions.add(this.transactions.subList(pos[1], pos[2]));
    partitions.add(this.transactions.subList(pos[2], pos[3]));
    partitions.add(this.transactions.subList(pos[3], pos[4]));

    // Compute the result
    double res = 0.0;
    for (List<TransCaixa> list : partitions) {
        for (TransCaixa transCaixa : list) {
            res += transCaixa.getValor();
        }
    }

    return res;
}
```

Listing 7.1: Cálculo iterativo sem e com partições

```java
public double sumStream() {
    return this.transactions.stream()
        .mapToDouble(TransCaixa::getValor)
        .sum();
}

public double sumPartitionStream() {
    double res = 0.0;
    Spliterator<TransCaixa> split = this.transactions.stream().spliterator();

    for (int i = 0; i < 4; i++) {
        res += StreamSupport.stream(split.trySplit(), false)
                .mapToDouble(TransCaixa::getValor)
                .sum();
    }

    return res;
}
```

```java
public double sumStreamP() {
    return this.transactions.parallelStream()
            .mapToDouble(TransCaixa::getValor)
            .sum();
}

public double sumPartitionStreamP() {
    double res = 0.0;
    Spliterator<TransCaixa> split = this.transactions.stream().spliterator();

    for (int i = 0; i < 4; i++) {
        res += StreamSupport.stream(split.trySplit(), true)
                .mapToDouble(TransCaixa::getValor)
                .sum();
    }

    return res;
}
```

Resultados

Input	(1) sum	(2) sum Partition	(3) sum P	(4) sum PartitionStream	(5) sum StreamP	(6) sum PartitionStreamP
1M	0,009782	0,010309	0,020080	0,019041	0,010039	0,010972
2M	0,016814	0,020771	0,038469	0,038853	0,016470	0,016523
4M	0,036431	0,033048	0,079529	0,073569	0,043543	0,030857
8M	0,075378	0,064377	0,152512	0,144809	0,062982	0,058897

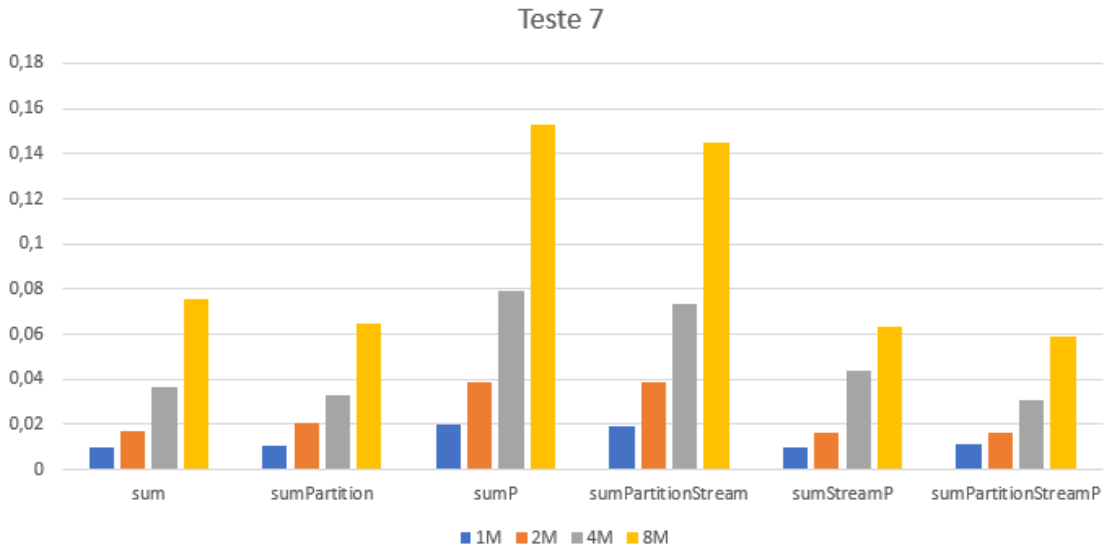

Figura 7.1.: Representação gráfica destes resultados

Análise e conclusões

Os resultados da tabela acima demonstram que geralmente é mais eficiente calcular a soma recorrendo aos métodos iterativos do JAVA7, e neste caso utilizar partições compensa para os inputs maiores (a partir dos 4 milhões de transações). Para inputs maiores, então efetivamente a utilização de partições de Stream paralela é mais benéfica. Tal acontece devido ao custo de inicialização das stream paralelas e das suas partições: para inputs "pequenos" este esforço inicial não é rentável.

8. Determinação da maior transação entre uma determinada hora

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public String biggestTransaction7() {
    List<TransCaixa> transactions = new ArrayList<>(this.transactions);

    transactions.sort(new Comparator<TransCaixa>() {
        @Override
        public int compare(TransCaixa t1, TransCaixa t2) {
            return Double.compare(t1.getValor(), t2.getValor());
        }
    });

    for (TransCaixa transaction : transactions) {
        int hour = transaction.getData().getHour();

        if (hour >= 16 && hour <= 20) {
            return transaction.getTrans();
        }
    }

    return null;
}
```

Listing 8.1: Determinação da maior transação entre uma determinada hora apenas com funcionalidades do JAVA7

```
public Optional<String> biggestTransaction8() {
    t.getData().getHour() >= 16 && t.getData().getHour() <= 20;

    return this.transactions.stream()
        .filter(timeInRange)
        .max(Comparator.comparing(TransCaixa::getValor))
        .map(TransCaixa::getTrans);
}
```

Listing 8.2: Determinação da maior transação entre uma determinada hora com auxílio de Streams

Resultados

Input	(1) biggestTransaction7	(2) biggestTransaction8
1000000 transactions	0,280259	0,017792
2000000 transactions	0,499207	0,035590
4000000 transactions	1,011563	0,071187
8000000 transactions	2,011931	0,142256

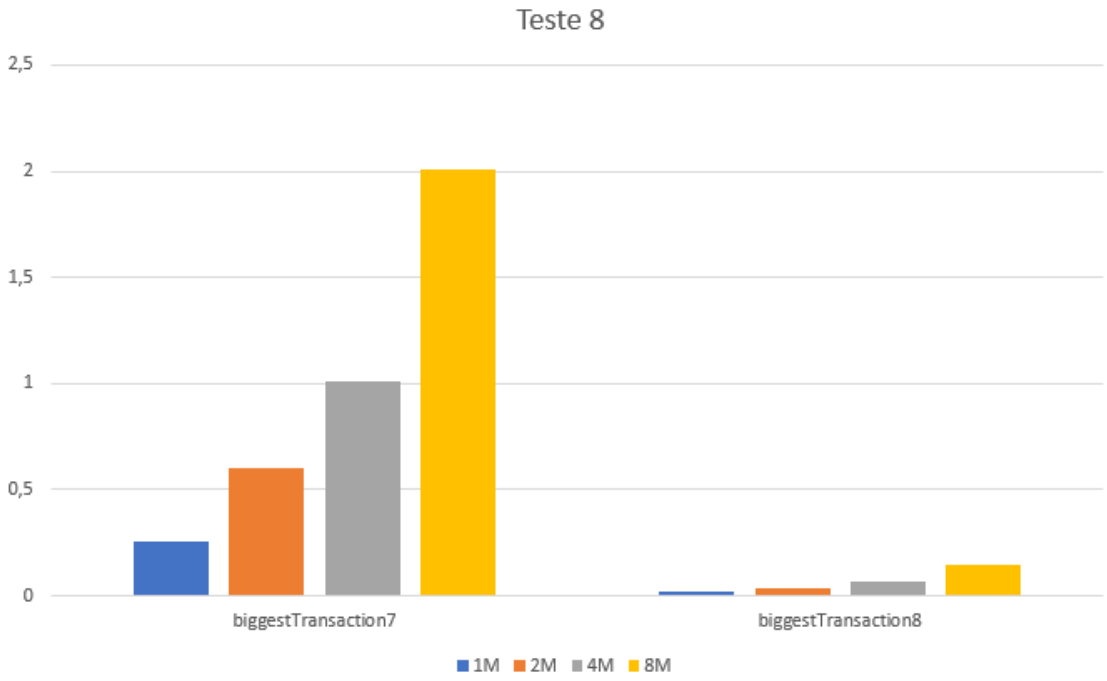

Figura 8.1.: Representação gráfica destes resultados

Análise e conclusões

Este teste apresentou resultados bastante díspares: por comparação a *performance* das streams é notoriamente melhor do que as suas versões iterativas. Tal deve-se ao facto de a versão iterativa necessitar de copiar e ordenar inicialmente as transações, algo que a stream não necessita de efetuar.

9. Cálculo do total facturado numa determinada semana do ano

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public double totalInWeekList() {
    final int week = 12;
    List<List<TransCaixa>> byWeek = new ArrayList<>();

    // Inicializar cada uma das listas
    for (int i = 0; i < 54; i++) {
        byWeek.add(i, new ArrayList<>());
    }

    for (TransCaixa transaction : this.transactions) {
        byWeek.get(transaction.getData()
                .get(ChronoField.ALIGNED_WEEK_OF_YEAR))
                .add(transaction);
    }

    // Calcular total faturado
    double total = 0.0;

    for (TransCaixa transaction : byWeek.get(week)) {
        total += transaction.getValor();
    }

    return total;
}
```

Listing 9.1: Cálculo do total facturado na semana 12 do ano

```
public double totalInWeekStream() {
    final int week = 12;
    Map<Integer, List<TransCaixa>> byWeek = this.transactions.stream()
            .collect(Collectors.groupingBy(
                    t -> t.getData().get(ChronoField.ALIGNED_WEEK_OF_YEAR)));
    return byWeek.entrySet().stream()
            .filter(e -> e.getKey() == week)
            .findFirst()
            .map(e -> e.getValue().stream()
                .mapToDouble(TransCaixa::getValor).sum())
```

```
                .orElse(0.0);
}
```

Listing 9.2: Cálculo do total facturado na semana 12 do ano com recurso a streams

Resultados

Input	(1) totalInWeekList	(2) totalInWeekStream
1000000 transactions	0,033511	0,038533
2000000 transactions	0,070856	0,083489
4000000 transactions	0,144393	0,163294
8000000 transactions	0,284878	0,322149

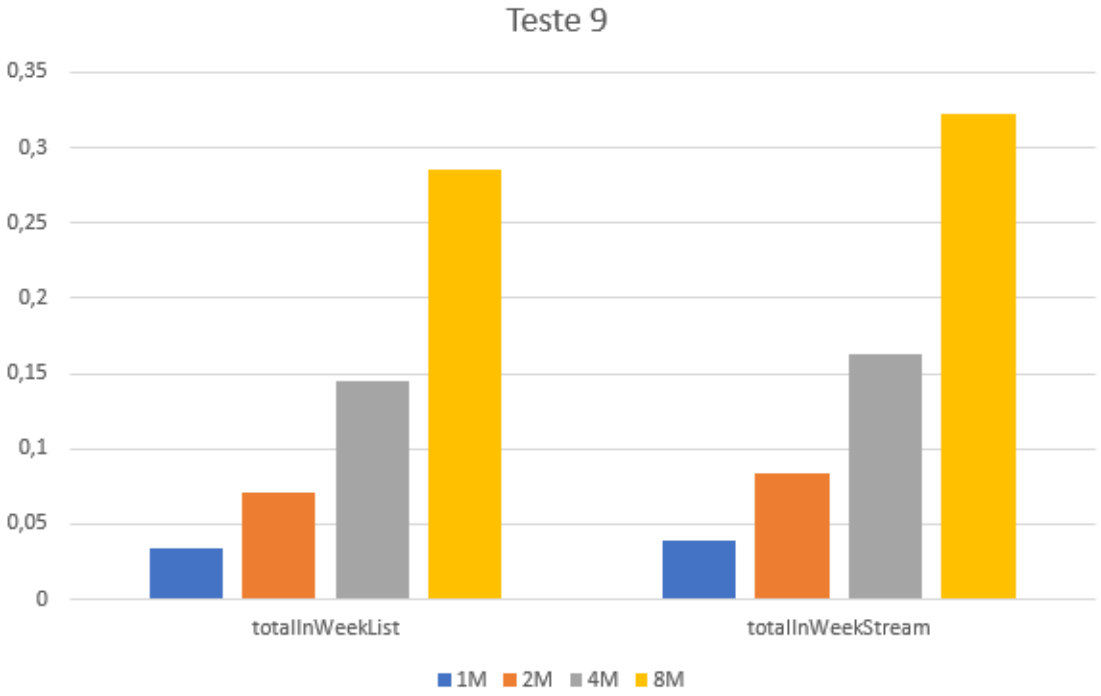

Figura 9.1.: Representação gráfica destes resultados

Análise e conclusões

Os resultados deste teste surpreendem pelo facto de a stream codificada ter uma *performance* ligeiramente pior que a sua versão iterativa. Tal deve-se ao algoritmo utilizado na stream: de forma a manter alguma legibilidade e simplicidade no código, criou-se inicialmente um Map que associa uma semana a um conjunto de transações, enquanto que a versão iterativa utilizou simplesmente um List onde cada posição corresponde a uma semana do ano.

10. Cálculo do total de IVA associado a cada mês

Observações

Este teste recebe como input uma coleção de transações.

Métodos a testar

```
public List<Double> iva() {
    List<Double> ivas = new ArrayList<>(13);

    // Initialize the IVAS with 0
    for (int i = 0; i < 13; i++) {
        ivas.add(0.0);
    }

    for (TransCaixa transaction : this.transactions) {
        Month month = transaction.getData().getMonth();
        Double value = transaction.getValor();
        Double iva = value < 20                ? 0.15 * value :
                     (value > 20 && value < 20 ? 0.20 * value :
                                                 0.23 * value);
        ivas.set(month.getValue(), ivas.get(month.getValue()) + iva);
    }

    return ivas;
}
```

Listing 10.1: Cálculo do total de IVA para cada mês

```
public Map<Month, Double> ivaStream() {
    return this.transactions.stream()
            .collect(Collectors.groupingBy(t -> t.getData().getMonth()))
            .entrySet()
            .stream()
            .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    e -> e.getValue().stream()
                            .mapToDouble(TransCaixa::getValor)
                            .map(x -> x < 20            ? 0.15 * x :
                                      (x > 20 && x < 20 ? 0.20 * x :
                                                          0.23 * x ))
                            .sum()
            ));
}
```

Listing 10.2: Cálculo do total de IVA para cada mês, com recurso a streams

Resultados

Input	(1) iva	(2) ivaStream
1000000 transactions	0,027518	0,070057
2000000 transactions	0,058240	0,134494
4000000 transactions	0,107619	0,283225
8000000 transactions	0,210951	0,475746

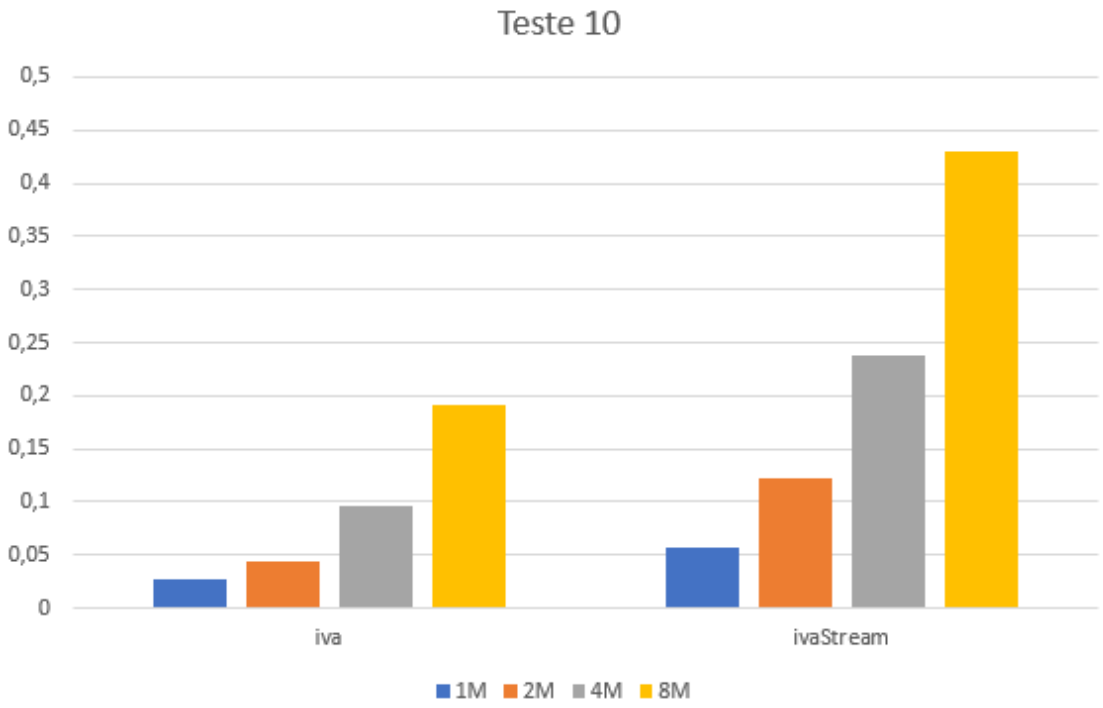

Figura 10.1.: Representação gráfica destes resultados

Análise e conclusões

Tal como o teste anterior este é um dos poucos casos em que a codificação do algoritmo numa única stream é mais complexa e menos eficiente. O algoritmo iterativo utiliza uma

List para armezanar os IVAs de cada mês, onde cada posição corresponde a um mês do ano. Esta é pois a única estrutura auxiliar utilizada, que vai sendo modificada, calculando-se pois o IVA em cada mês através da soma do valor que já lá se encontrava com o atual. Já a versão com Stream é um pouco mais complexa pois primeiro cria um Map onde associa um mês a um conjunto de transações e só depois é que efetua a sua soma.

11. Comparação de performance entre JDK8 e JDK9

Observações

Este teste recebe como input uma coleção de transações.

Para obter os resulados consoante a JDK desejada, alterou-se nas definições do IntelliJ de modo a compilar e a executar quer com a JDK8, quer com a JDK9, tendo sido utilizados os seguintes comandos (abreviados):

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java" ... Main
"C:\Program Files\Java\jdk-9.0.1\bin\java" ... Main
```

Métodos a testar

Dos testes anteriores, foram selecionados os seguintes, que manipulam conjuntos enormes de registos TransCaixa:

- **byDateStream** do teste 4;
- **sortList** do teste 4;
- **biggestTransaction8** do teste 8;
- **ivaStream** do teste 10;

Resultados

Input	(1) sumStreamP	(2) byDateStreamP	(3) sortList	(4) biggestTransaction8
1000000 transactions	0,013631	1,522605	0,792593	0,019558
2000000 transactions	0,028075	3,481223	1,772310	0,040336
4000000 transactions	0,049482	7,990621	4,034171	0,083384

Input	(1) sumStreamP	(2) byDateStreamP	(3) sortList	(4) biggestTransaction8
8000000 transactions	0,099308	17,342737	8,943635	0,162232

Input	(1) sumStreamP	(2) byDateStreamP	(3) sortList	(4) biggestTransaction8
1000000 transactions	0,014094	1,664687	0,844504	0,020016
2000000 transactions	0,028293	3,817031	1,939572	0,035174
4000000 transactions	0,054420	8,699993	4,425399	0,068976
8000000 transactions	0,108938	19,293742	9,910055	0,138288

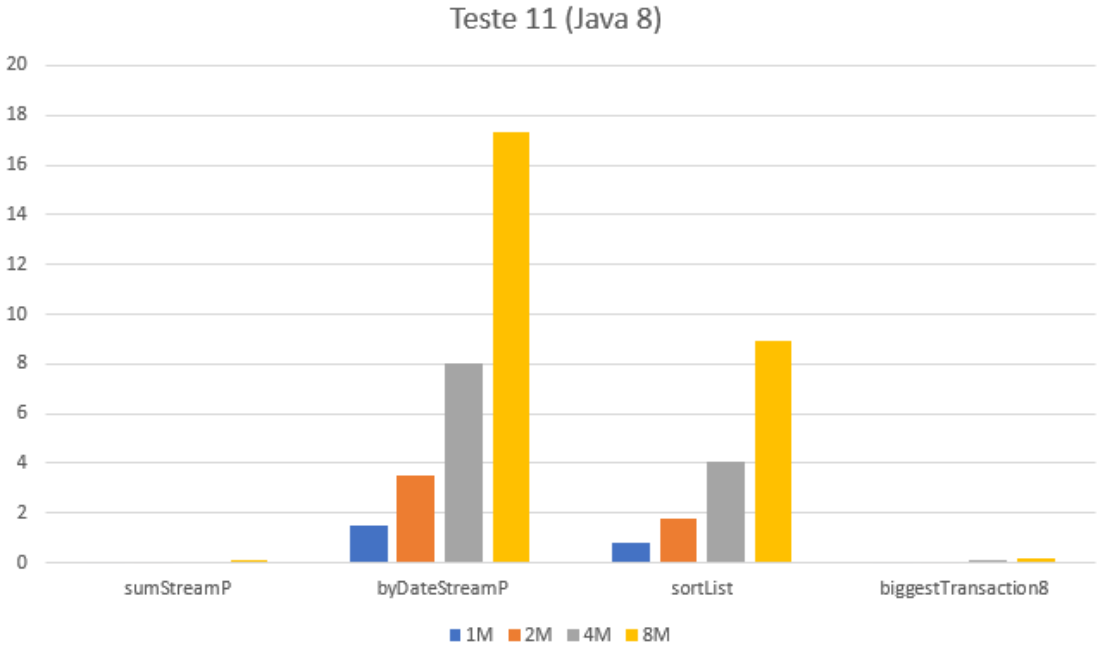

Figura 11.1.: Representação gráfica destes resultados (Java 8)

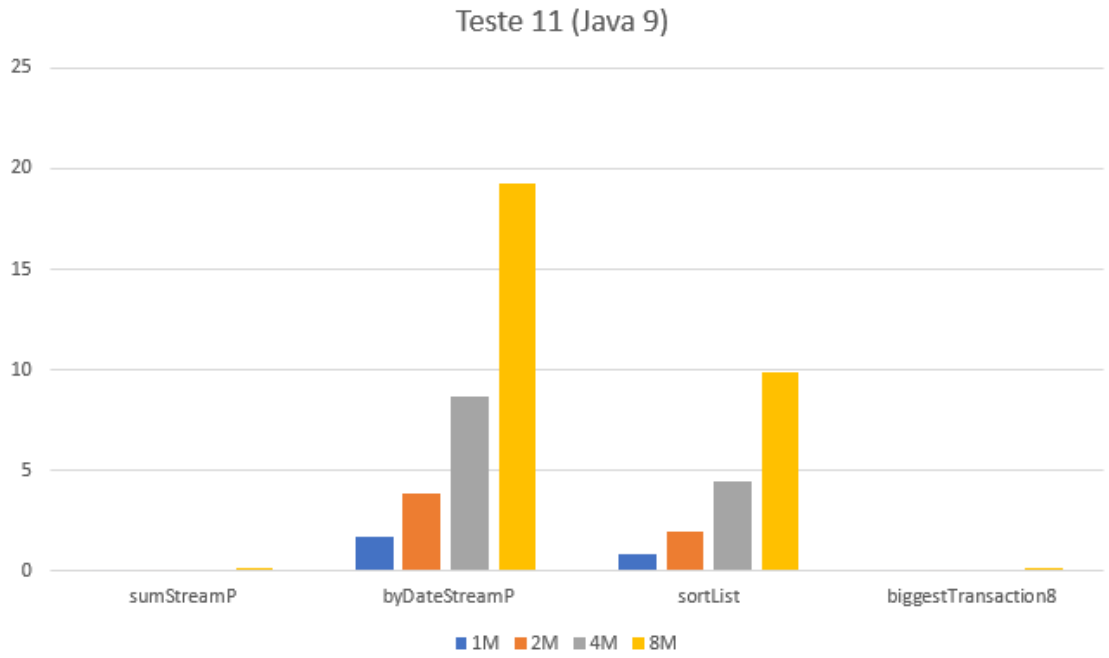

Figura 11.2.: Representação gráfica destes resultados (Java 9)

Análise e conclusões

Os resultados deste teste são algo curiosos, pois demonstram que não existe grande diferença entre utilizar a JDK8 ou a JDK9. Tal pode ter como causa o facto da última ser bastante recente e ainda não ter sofrido otimizações de maior relevo, ou simplesmente porque os inputs destes testes são relativamente pequenos. Se se trabalhasse com inputs da ordem de biliões de transações certamente que os resultados seriam diferentes.

12. Total faturado por caixa em Java 8 e Java 9

Observações

Este teste recebe como input uma coleção de transações.

Para obter os resulados consoante a JDK desejada, alterou-se nas definições do IntelliJ de modo a compilar e a executar quer com a JDK8, quer com a JDK9, tendo sido utilizados os seguintes comandos (abreviados):

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java" ... Main
"C:\Program Files\Java\jdk-9.0.1\bin\java" ... Main
```

Métodos a testar

```
public Map<String, Double> totalByBox() {
        Map<String, Map<Month, List<TransCaixa>>> table = this.transactions
                .stream()
                .collect(Collectors.groupingBy(TransCaixa::getCaixa))
                .entrySet()
                .stream()
                .collect(Collectors.toMap(Map.Entry::getKey,
                        l -> l.getValue().stream()
                                .collect(Collectors.groupingBy(
                                        t -> t.getData().getMonth())
                                ))
                );

        return table.entrySet().stream()
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        e -> e.getValue().values().stream()
                                .flatMap(Collection::stream)
                                .mapToDouble(TransCaixa::getValor)
                                .sum()));
    }
```

Listing 12.1: Cálculo do total facturado por caixa em JAVA8

```
public Map<String, Double> totalByBoxConcurrentMap() {
        ConcurrentMap<String, Map<Month, List<TransCaixa>>> table = this.transactions
                .stream()
                .collect(Collectors.groupingBy(TransCaixa::getCaixa))
```

```
                .entrySet()
                .stream()
                .collect(Collectors.toConcurrentMap(Map.Entry::getKey,
                        l -> l.getValue().stream()
                                .collect(Collectors.groupingBy(
                                        t -> t.getData().getMonth())
                                ))
                );

        return table.entrySet().stream()
                .collect(Collectors.toConcurrentMap(
                        Map.Entry::getKey,
                        e -> e.getValue().values().stream()
                                .flatMap(Collection::stream)
                                .mapToDouble(TransCaixa::getValor)
                                .sum())));
}
```

Listing 12.2: Cálculo do total facturado por caixa em JAVA9

Resultados

Input	(1) totalByBox	(2) totalByBoxConcurrent
1000000 transactions	0,240404	0,242108
2000000 transactions	0,494429	0,524374
4000000 transactions	1,043512	1,235674
8000000 transactions	2,203444	2,128300

Input	(1) totalByBox	(2) totalByBoxConcurrent
1000000 transactions	0,288725	0,245084
2000000 transactions	0,511240	0,520865
4000000 transactions	1,032625	1,168777
8000000 transactions	2,270371	2,147356

Análise e conclusões

Nestes resultados obtidos já se nota uma bastante ligeira melhoria da performance na utilização do JDK9. Ainda é bastante pequena a diferença contudo crê-se que em versões posteriores o desempenho tenda a aumentar.

Independemente da JDK utilizada percebe-se que a utilização de ConcurrentMap só é mais eficiente para so inputs de 8 milhões de transações, até ser atingido esse valor os resultados são mais ou menos semelhantes, tendendo a favorecer a utilização do Map tradicional.

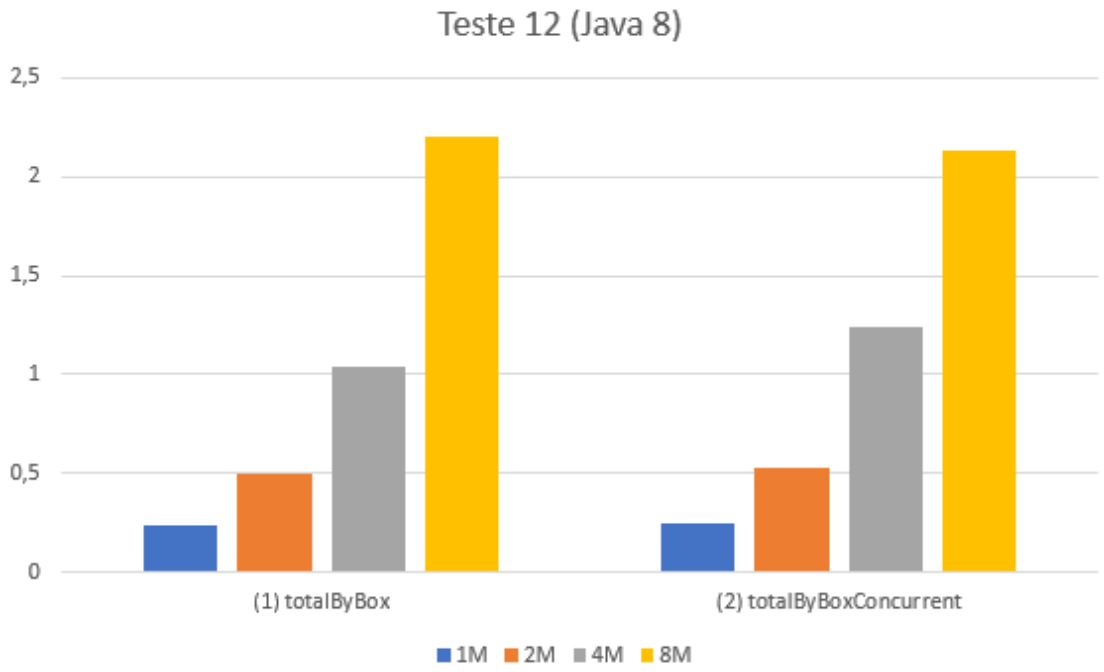

Figura 12.1.: Representação gráfica destes resultados (Java 8)

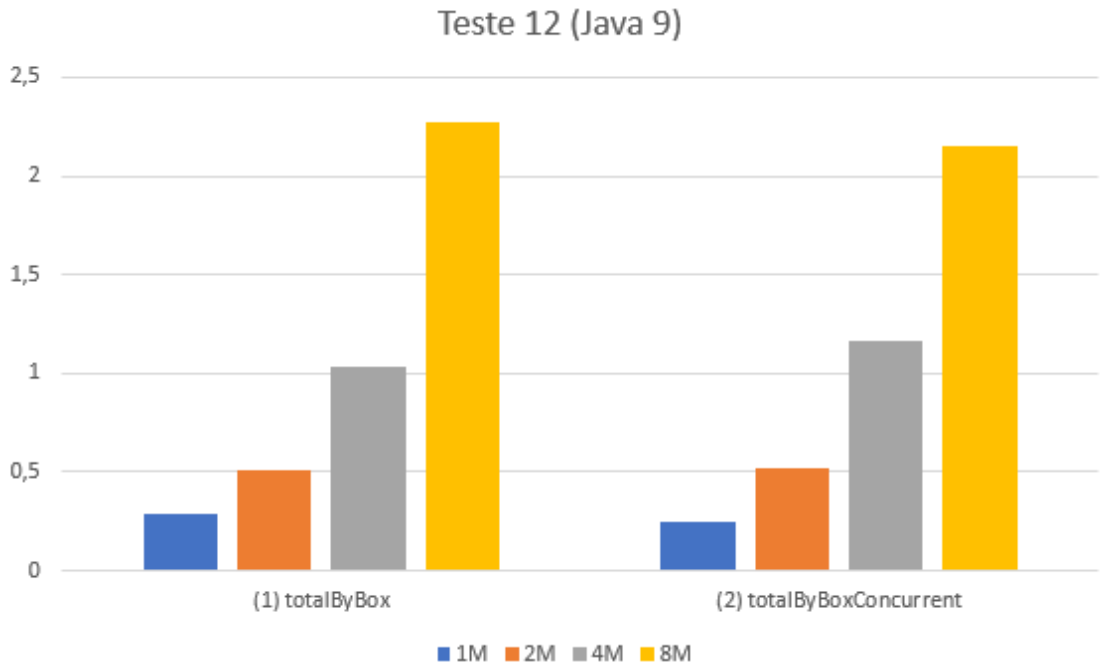

Figura 12.2.: Representação gráfica destes resultados (Java 9)

Parte III.

Conclusões e trabalho futuro

Conclusão

Após realizado este projeto é justo afirmar que o mesmo foi bem sucedido. Foram testadas todas as funcionalidades desejadas e estes testes foram ao encontro daquilo que era expectável.

De um modo geral a utilização das novas streams do JAVA8 compensa em termos de *performance* para a maioria dos problemas. A sua paralelização só começa a ser viável a partir de inputs bastante grandes, pois requere algum custo de inicialização para dividir a carga de trabalho. Contudo, existe um ponto em comum a praticamente todos os testes realiados: o código escrito com recurso a streams é bastante mais conciso e legível, não requere variáveis auxiliares, portanto de uma certa forma é *point-free* e dá a sensação ao programador de estar a escrever código funcional, que tal como é sabido garante uma boa robustez e minimiza a propenção de erro do mesmo.

Uma outra surpresa foi a pouca diferença de *performance* entre a JDK8 e a JDK9, provavelmente se deve ao facto de a última ser bastante recente, e de a máquina onde foram corridos os testes não beneficiar muito desta instalação.