JAVA Benchmarking

Relatório do Trabalho Prático 2

Diogo Silva (a76407)
João Barreira (a73831)
Rafael Costa (a61799)

Janeiro 2018

Universidade do Minho
Escola de Engenharia

Processamento de Dados com Streams de JAVA
Mestrado Integrado em Engenharia Informática – 4.º Ano
Universidade do Minho

1 Introdução

O presente projeto corresponde a um programa de benchmarking de construções específicas da linguagem Java.

O objetivo consiste em medir a performance de Arrays, Coleções e Streams (sequenciais e paralelas) usando para o efeito Java 7 e Java 8 através de comparações diretas e, ainda, Java 9.

Desta forma, pretende-se entender quais as vantagens e desvantagens das várias alternativas disponíveis para igual solução. Assim, será possível compreender as situações mais adequadas para usar Java 7 ou Java 8, assim como as suas estruturas e tipo de dados.

2 Condições de Teste

Os benchmarks correram todos no mesmo computador com o perfil de máxima performance. O hardware utilizado em todos os testes deste projeto foi o seguinte:

- **Central Processing Unit (CPU)**: Intel® Core™ i7-67000HQ 2.60 GHz
- **Random Access Memory (RAM)**: DDR4 2133 MHz SDRAM, 32 GB, (16 GB x 2)
- **Graphics Processing Unit (GPU)**: NVIDIA® GeForce® GTX980M com 8GB GDDR5
- **Hard Disk Drive (HDD)**: 1 TB 7200 RPM

3 Testes

3.1 Teste 1

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 e da média desses valores.

3.1.1 Código

1. Usando double[] e for():

```
public static Supplier<double[]>
t1_7_1(final List<Transaction> transactions) {
    return () -> {
        final int size = transactions.size();
```

```
            final double[] values = new double[size];

            for (int i = 0; i < size; i++) {
                values[i] = transactions.get(i)
                    .getValue();
            }

            return values;
        };
    }
```

2. Usando double[] e forEach():

```
    public static Supplier<double[]>
    t1_7_2(List<Transaction> transactions) {
        return () -> {
            final double[] values =
                new double[transactions.size()];
            int i = 0;

            for (Transaction t : transactions) {
                values[i++] = t.getValue();
            }
            return values;
        };
    }
```

3. Usando DoubleStream e stream():

```
    public static <T> Supplier<DoubleStream>
    t1_8_1_1(List<T> list, Function<T, Double> f) {
        return () -> list.stream().mapToDouble(f::apply);
    }
```

4. Usando DoubleStream e parallelStream():

```
    public static <T> Supplier<DoubleStream>
    t1_8_1_2(List<T> list, Function<T, Double> f) {
        return () -> list.parallelStream()
            .mapToDouble(f::apply);
    }
```

5. Usando Stream<Double> e stream():

```
    public static <T> Supplier<Stream<Double>>
    t1_8_2_1(List<T> list, Function<T, Double> f) {
        return () -> list.stream().map(f);
    }
```

6. Usando Stream<Double> e parallelStream():

```
public static <T> Supplier<Stream<Double>>
t1_8_2_2(List<T> list, Function<T, Double> f) {
    return () -> list.parallelStream().map(f);
}
```

3.1.2 Resultados

Tabela 1: Resultado do Teste 1 em segundos

	J7 for	J7 forEach	J8 Double Stream Sequential	J8 Double Stream Parallel	J8 Stream< Double> Sequential	J8 Stream <Double> Parallel
1000000	0.010095783	0.009283932	3.79E-05	2.21E-05	1.82E-05	2.01E-05
2000000	0.02051828	0.017092312	4.98E-05	3.52E-05	1.70E-05	1.70E-05
4000000	0.031403	0.032787688	2.05E-05	2.05E-05	1.82E-05	3.28E-05
6000000	0.043793691	0.045573441	2.29E-05	2.09E-05	3.12E-05	1.74E-05

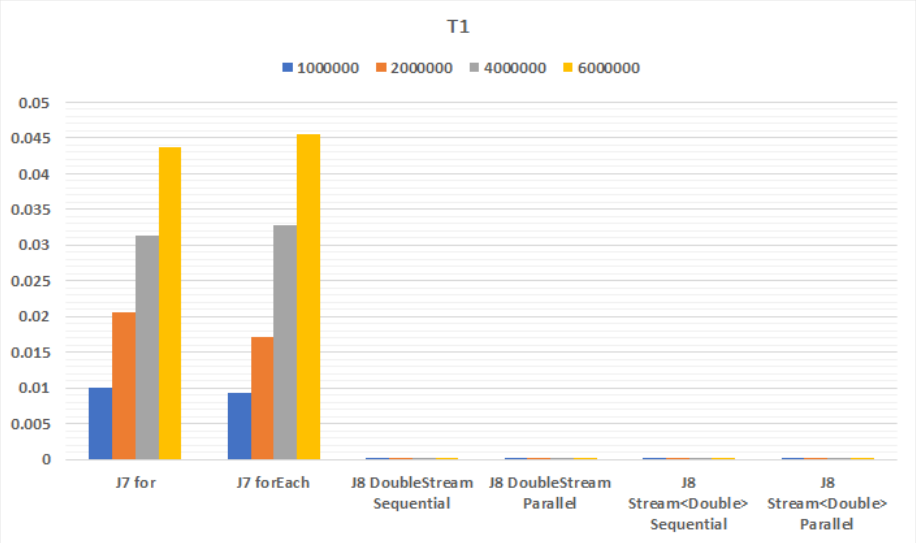

Figura 1: Gráfico do teste 1 em segundos

3.1.3 Análise e Conclusões

É possível observar de imediato que o recurso a Java 7 foi claramente mais custoso computacionalmente. Não existiu uma diferença significativa entre usar

for e forEach em Java 7. Relativamente a Java 8 o paralelismo para o domínio em questão não inseriu impacto relevante computacionalmente dado que os valores são muito aproximados. Podemos concluir ainda que a Stream<Double> é ligeiramente mais eficiente que DoubleStream.

3.2 Teste 2

Considere o problema típico de a partir de um data set de dada dimensão se pretenderem criar dois outros data sets correspondentes aos 30% primeiros e aos 30% últimos do data set original segundo um dado critério. Defina sobre TransCaixa um critério de comparação que envolva datas ou tempos e use-o neste teste, em que se pretende comparar a solução com streams sequenciais e paralelas às soluções usando List<> e TreeSet<>.

3.2.1 Resultados

3.2.2 Código

1. Usando List e streams sequenciais

```
public static <T> Supplier<SimpleEntry<List<T>,
List<T>>> t2_list_1(final List<T> list ,
final double start, final double end,
final Comparator<T> comparator) {
    return () -> {
        final int size = list.size();

        if (start + end <= 1) {
            final int i = (int)Math
            .ceil(size*start);
            final int j = (int)Math
            .ceil(size*end);

            return new SimpleEntry<>(
                    IntStream.range(0, i)
                    .mapToObj(list::get)
                    .sorted(comparator)
                    .collect(toList()),
                    IntStream
                    .range(size-j, size)
                    .mapToObj(list::get)
                    .sorted(comparator)
                    .collect(toList())
            );
        }
```

```
        return null;
    };
}
```

2. Usando List e streams paralelas:

```
public static <T> Supplier<SimpleEntry<List<T>,
List<T>>> t2_list_2(final List<T> list,
final double start, final double end,
final Comparator<T> comparator) {
    return () -> {
        final int size = list.size();

        if (start + end <= 1) {
            final int i = (int)Math
            .ceil(size*start);
            final int j = (int)Math
            .ceil(size*end);

            return new SimpleEntry<>(
                    IntStream
                    .range(0, i).parallel()
                    .mapToObj(list::get)
                    .sorted(comparator)
                    .collect(toList()),
                    IntStream
                    .range(size - j, size)
                    .parallel()
                    .mapToObj(list::get)
                    .sorted(comparator)
                    .collect(toList())
            );
        }

        return null;
    };
}
```

3. Usando TreeSet e streams sequenciais:

```
public static <T> Supplier<SimpleEntry<Set<T>,
Set<T>>> t2_treeSet_1(final List<T> list,
final double start, final double end,
final Comparator<T> comparator) {
    return () -> {
        final int size = list.size();
```

```
            if (start + end <= 1) {
                final int i = (int)Math
                .ceil(size * start);
                final int j = (int)Math.ceil(size * end);

                return new SimpleEntry<>(
                        IntStream.range(0, i)
                        .mapToObj(list::get)
                        .collect(toCollection(() ->
                        new TreeSet<>(comparator))),
                        IntStream.range(size-j, size)
                        .mapToObj(list::get)
                        .collect(toCollection(() ->
                        new TreeSet<>(comparator)))
                );
            }

            return null;
        };
    }
```

4. Usando TreeSet e streams paralelas:

```
    public static <T> Supplier<SimpleEntry<Set<T>,
    Set<T>>> t2_treeSet_2(final List<T> list,
    final double start, final double end,
    final Comparator<T> comparator) {
        return () -> {
            final int size = list.size();

            if (start + end <= 1) {
                final int i = (int)Math
                .ceil(size * start);
                final int j = (int)Math
                .ceil(size * end);

                return new SimpleEntry<>(
                        IntStream.range(0, i)
                        .parallel()
                        .mapToObj(list::get)
                        .collect(toCollection(() ->
                        new TreeSet<>(comparator))),
                        IntStream
                        .range(size - j, size)
                        .parallel()
                        .mapToObj(list::get)
```

```
                        .collect(toCollection(() ->
                        new TreeSet<>(comparator)))
                );
            }

            return null;
        };
    }
```

Tabela 2: Resultado do Teste 2 em segundos

	List Sequential	List Parallel	TreeSet Sequential	TreeSet Parallel
1000000	0.421853587	0.146953192	0.679131893	0.419384456
2000000	1.002730266	0.330008632	1.643890877	0.849304396
4000000	2.289409009	0.765201451	3.840408414	1.922286376
6000000	4.481428382	1.35180691	6.780937124	3.659398994

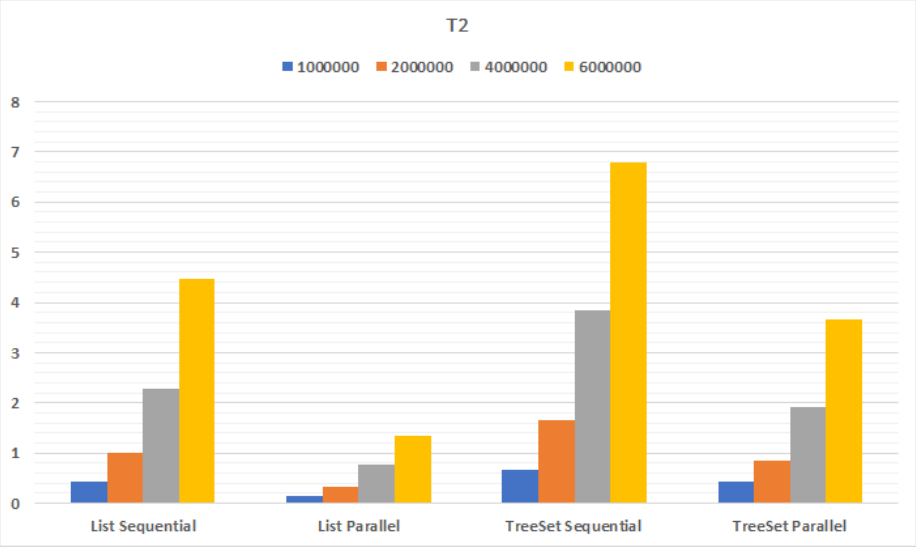

Figura 2: Gráfico do teste 2 em segundos

3.2.3 Análise e Conclusões

É possível observar que o uso de list compensa em relação ao uso de set dado que este elimina os objetos repetidos internamente usando para o efeito o

método equals, enquanto que a list não utiliza tal mecanismo. Para ambos os casos, denota-se que o paralelismo compensa dado que se observa um tempo de execução cerca de 2 a 3 vezes menor em relação à versão sequencial.

3.3 Teste 3

Crie uma IntStream, um int[] e uma List<Integer> com de 1M a 8M de números aleatórios de valores entre 1 e 9999. Determine o esforço de eliminar duplicados em cada situação.

3.3.1 Código

1. Usando IntStream

```
public static Supplier<IntStream>
t3_IntStream(int[] intStream) {
    return () -> {
        Supplier<IntStream> streamSupplier
        = () -> IntStream.of(intStream);
        return streamSupplier.get().distinct();
    };
}
```

2. Usando int[]

```
public static Supplier<int[]> t3(int[] ints) {
    return () -> {
        final int size = ints.length;
        final int[] unique = new int[size];
        int j;
        int k = 0;

        for (int anInt : ints) {
            for (j = 0; j < k; j++) {
                if (unique[j] == anInt) {
                    break;
                }
            }
            if (j == k) {
                unique[k++] = anInt;
            }
        }

        return Arrays.copyOf(unique, k);
    };
}
```

3. Usando List<Integer>

```
public static Supplier<List<Integer>>
t3(List<Integer> integers) {
    return () -> {
        final List<Integer> unique =
        new ArrayList<>();

        for (Integer integer : integers) {
            if (!unique.contains(integer)) {
                unique.add(integer);
            }
        }

        return unique;
    };
}
```

3.3.2 Resultados

Tabela 3: Resultado do Teste 3 em segundos

	IntStream	int[]	List<Integer>
1000000	2.37E-05	1.066090536	4.426220985
2000000	4.94E-05	2.050247012	9.00194449
3000000	2.57E-05	3.107330949	13.25618083
4000000	2.29E-05	4.125962418	17.79475083
5000000	2.57E-05	5.190922684	22.04943793
6000000	2.29E-05	6.23344349	26.91071267
7000000	3.71E-05	7.261678495	30.96094902
8000000	3.75E-05	8.260494498	35.10962112

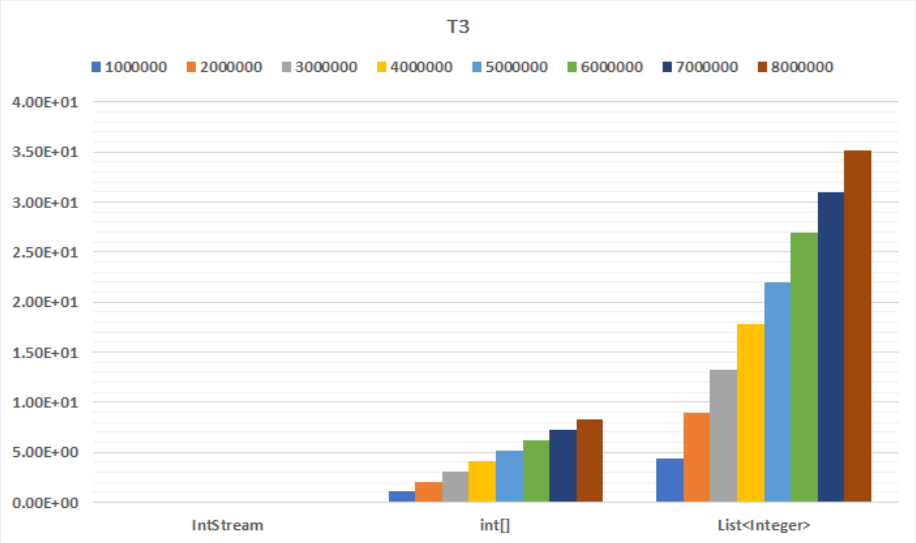

Figura 3: Gráfico do teste 3 em segundos

3.3.3 Análise e Conclusões

É possível concluir que é muito superior utilizar IntStream. Face às outras 2 propostas, utilizar um array é mais eficiente que utilizar o recurso a uma estrutura do tipo list. Podemos também verificar que os tempos de execução utilizando IntStream não aumentam na mesma proporção que nos restantes casos, ou seja, sempre que ao número de inteiros é acrescido 1 milhão, o novo overhead inserido não tem o mesmo impacto no tempo de execução quando o mesmo ocorre na utilização de um array ou de uma lista de inteiros.

3.4 Teste 4

Defina um método static, uma BiFunction e uma expressão lambda que dados dois números reais calculam o resultado da sua multiplicação. Crie em seguida um double[] com sucessivamente 1M, 2M, 4M e 8M de reais resultantes valores de caixa dos ficheiros de transações. Finalmente processe o array usando streams, sequenciais e paralelas, comparando os tempos de invocação e aplicação do método versus a bifunction e a expressão lambda explícita.

3.4.1 Código

1. Usando BiFunction e streams sequenciais

```
public static Supplier<Double> t4_8_1_1(
double[] values) {
```

```
        return () -> Arrays.stream(values)
        .reduce(0, (a, b) -> multiplyDouble
        .apply(a, b));
    }
```

2. Usando BiFunction e streams paralelas

```
    public static Supplier<Double> t4_8_1_2(
    double[] values) {
        return () -> Arrays.stream(values).parallel()
        .reduce(0, (a, b) -> multiplyDouble
        .apply(a, b));
    }
```

3. Usando expressão lambda e streams sequenciais

```
    public static Supplier<Double> t4_8_2_1(
    double[] values) {
        return () -> Arrays.stream(values)
        .reduce(0, (a, b) -> a * b);
    }
```

4. sando expressão lambda e streams paralelas

```
    public static Supplier<Double> t4_8_2_2(
    double[] values) {
        return () -> Arrays.stream(values).parallel()
        .reduce(0, (a, b) -> a * b);
    }
```

5. Função auxiliar

```
    private static
    BiFunction<Double, Double, Double>
    multiplyDouble = (a, b) -> a * b;
```

3.4.2 Resultados

Tabela 4: Resultado do Teste 4 em segundos

	BiFunction Sequential	BiFunction Parallel	Lambda Sequential	Lambda Parallel
1000000	0.001367701	7.93E-04	0.003092142	0.001084838
2000000	0.008590995	0.003029327	0.007833664	0.002413032
4000000	0.017232954	0.005189125	0.015484413	0.004592584
6000000	0.029934558	0.007666552	0.023052202	0.006816776

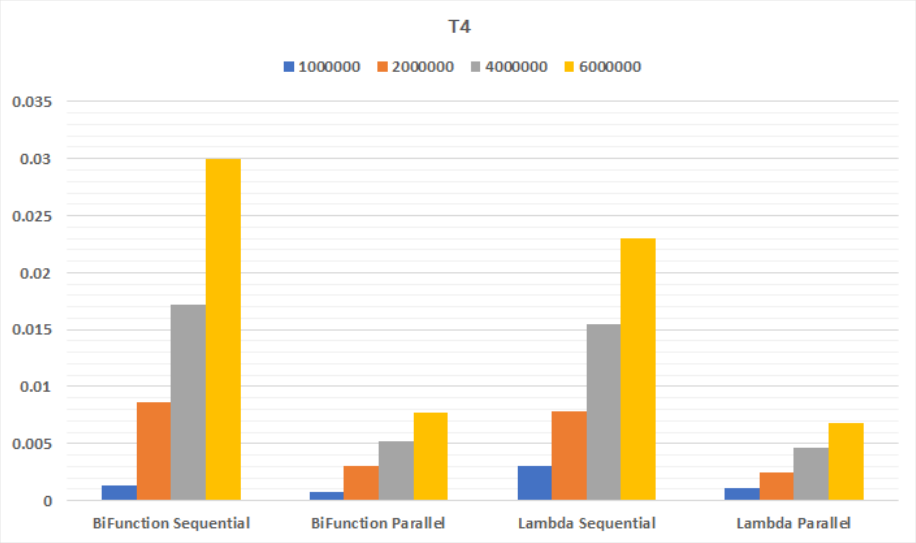

Figura 4: Gráfico do teste 4 em segundos

3.4.3 Análise e Conclusões

É possível observar que o uso de uma Lambda compensa em relação a uma BiFunction. Isto pode dever-se ao facto da Lambda ser aplicada diretamente enquanto que uma BiFunction encapsula uma função que deve ser aplicada através do método apply. Em ambos os casos, compensa utilizar paralelismo.

3.5 Teste 5

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>.

3.5.1 Código

1. Usar streams e efetuar o collect para TreeSet<Transaction>:

```
public static <T> Supplier<Set<T>> t5_1(List<T> list,
    Comparator<T> comparator) {
    return () -> list.stream().collect(toCollection(
        () -> new TreeSet<>(comparator)));
}
```

2. Usar a operação sorted() e efetuar o collect para List<Transaction>:

```
public static <T> Supplier<List<T>> t5_2(
List<T> list ,
    Comparator<T> comparator) {
        return () -> list.stream().sorted(comparator)
        .collect(toList());
}
```

3.5.2 Resultados

Tabela 5: Resultado do Teste 5 em segundos

	TreeSet	List
1	1,513	0,899
2	3,559	2,086
4	8,343	4,683
6	13,991	8,895

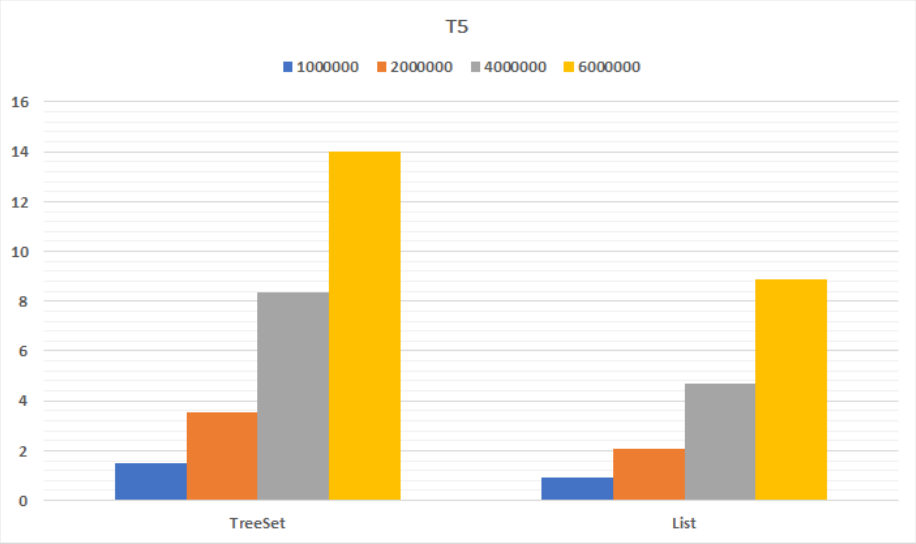

Figura 5: Gráfico do Teste 5 em segundos

3.5.3 Análise e Conclusões

É possível deduzir que compensa utilizar um collect para uma list após sofrer a operação de sorted() dado que a lista não necessita de eliminar repetidos ao inserir novos elementos. Ou seja, o overhead inserido com a operação de sorted() é inferior ao uso do TreeSet com o Comparator.

3.6 Teste 6

Considere o exemplo prático das aulas de streams em que se criou uma tabela com as transações catalogadas por Mês, Dia, Hora efetivos. Codifique em JAVA 7 o problema que foi resolvido com streams e compare tempos de execução. Faça o mesmo para um Map< DiaDaSemana, Hora>

3.6.1 Código

1. Catalogar em Mês, Dia e Hora em Java 7

```
public static Supplier<Map<Month,
Map<Integer, Map<Integer, List<Transaction>>>>>
t6_7_1(List<Transaction> transactions) {
    return () -> {
        Map<Month, Map<Integer, Map<Integer,
        List<Transaction>>>> map = new HashMap<>();
        Month month;
        int day;
        int hour;
        LocalDateTime dateTime;
        Map<Integer, Map<Integer,
        List<Transaction>>> dayMap;
        Map<Integer, List<Transaction>> hourMap;
        List<Transaction> list;

        for (Transaction t : transactions) {
            dateTime = t.getDate();
            month = dateTime.getMonth();
            day = dateTime.getDayOfMonth();
            hour = dateTime.getHour();

            dayMap = map.get(month);

            if (dayMap == null) {
                dayMap = new HashMap<>();
                map.put(month, dayMap);
            }

            hourMap = dayMap.get(day);

            if (hourMap == null) {
                hourMap = new HashMap<>();
                dayMap.put(day, hourMap);
            }
```

```
                list = hourMap.get(hour);

                if (list == null) {
                    list = new ArrayList<>();
                    hourMap.put(hour, list);
                }

                list.add(t);
            }
            return map;
        };
    }
```

2. Catalogar em Dia da Semana e Hora em Java 7

```
    public static Supplier<Map<DayOfWeek,
    Map<Integer, List<Transaction>>>>
    t6_7_2(List<Transaction> transactions) {
        return () -> {
            Map<DayOfWeek, Map<Integer,
                List<Transaction>>> map =
            new HashMap<>();
            DayOfWeek day;
            int hour;
            LocalDateTime dateTime;
            Map<Integer, List<Transaction>> hourMap;
            List<Transaction> list;

            for (Transaction t : transactions) {
                dateTime = t.getDate();
                day = dateTime.getDayOfWeek();
                hour = dateTime.getHour();

                hourMap = map.get(day);

                if (hourMap == null) {
                    hourMap = new HashMap<>();
                    map.put(day, hourMap);
                }

                list = hourMap.get(hour);

                if (list == null) {
                    list = new ArrayList<>();
                    hourMap.put(hour, list);
                }
```

```
                list.add(t);
            }
            return map;
        };
    }
```

3. Catalogar em Mês, Dia e Hora em Java 8:

```
public static Supplier<Map<Month, Map<Integer,
Map<Integer, List<Transaction>>>>>
t6_8_1(List<Transaction> transactions) {
    return () -> transactions.stream().collect(
    groupingBy(t -> t.getDate().getMonth(),
        groupingBy(t -> t.getDate().getDayOfMonth(),
        groupingBy(t -> t.getDate().getHour()))));
}
```

4. Catalogar em Dia da Semana e Hora em Java 8

```
public static Supplier<Map<DayOfWeek, Map<Integer,
List<Transaction>>>>
t6_8_2(List<Transaction> transactions) {
    return () -> transactions.stream().collect(
    groupingBy(t -> t.getDate().getDayOfWeek(),
        groupingBy(t -> t.getDate().getHour())));
}
```

3.6.2 Resultados

Tabela 6: Resultado do Teste 6 em milissegundos

	J7 Map<Month, Map<Day, Map<Hour List>>>	J7 Map<DayOfWeek, Map<Hour, List>>	J8 Map<Month, Map<Day, Map<Hour, List>>>	J8 Map<DayOfWeek, Map<Hour, List>>
1000000	88,031	56,126	126,964	74,109
2000000	193,180	113,897	272,102	147,463
4000000	371,223	209,437	536,044	273,333
6000000	491,443	338,215	692,240	427,595

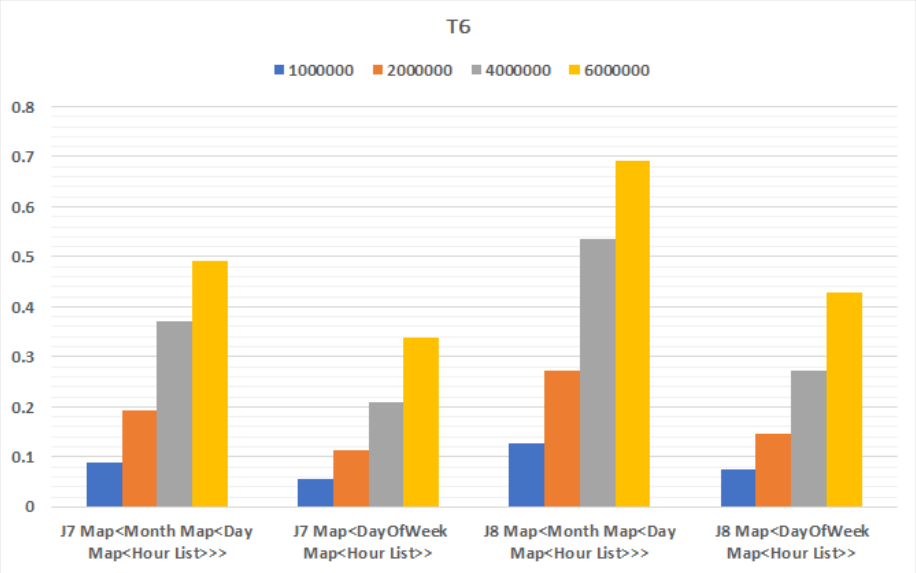

Figura 6: Gráfico do Teste 6 em segundos

3.6.3 Análise e Conclusões

Para este caso específico compensa utilizar Java 7 para ambos os casos e, aproximadamente, na mesma proporção. É possível que usando Java 8 seja menos eficiente devido ao método groupingBy. Contudo, a clareza do código em Java 8 é muito mais atraente e com a utilização de menos linhas de código que Java 7.

3.7 Teste 7

Usando List<TransCaixa> e Spliterator<TransCaixa> crie 4 partições cada uma com 1/4 do data set. Compare os tempos de processamento de calcular a soma do valor das transações com as quatro partições ou com o data set inteiro, quer usando List<> e forEach() quer usando streams sequenciais e paralelas.

3.7.1 Código

1. Soma do valor das transações em Java 7:

```
public static Supplier<Double> t7_7(
List<Transaction> transactions) {
    return () -> {
        double sum = 0;
        for (Transaction t : transactions) {
```

```
            sum += t.getValue();
        }
        return sum;
    };
}
```

2. Soma do valor das transações em Java 8 usando Spliterator:

```
public static <T> Supplier<Double>
t7_8_1(List<T> list, Function<T, Double> f) {
    return () -> {
        Spliterator<T> spliterator0 =
            list.spliterator();
        Spliterator<T> spliterator1 =
            spliterator0.trySplit();
        Spliterator<T> spliterator2 =
            spliterator0.trySplit();
        Spliterator<T> spliterator3 =
            spliterator1.trySplit();

        ForkJoinPool pool = new ForkJoinPool(4);

        Function<Spliterator<T>, Double>
            sumFunction = spliterator -> {
            final DoubleWrapper d =
                new DoubleWrapper();
            while(spliterator
            .tryAdvance(t -> d.add(f.apply(t))));
            return d.get();
        };

        List<Future<Double>> futures =
        pool.invokeAll(Arrays.asList(
            () -> sumFunction.apply(spliterator0),
            () -> sumFunction.apply(spliterator1),
            () -> sumFunction.apply(spliterator2),
            () -> sumFunction.apply(spliterator3))
        );

        double sum = 0;
        for (Future<Double> future : futures) {
            try {
                sum += future.get();
            } catch (InterruptedException |
                ExecutionException e) {
                e.printStackTrace();
```

```
            }
        }
        return sum;
    };
}
```

3. Soma do valor das transações em Java 8 usando streams sequenciais:

```
public static <T> Supplier<Double>
    t7_8_2(List<T> list, Function<T, Double> f) {
    return () -> list.stream()
        .mapToDouble(f::apply).sum();
}
```

4. Soma do valor das transações em Java 8 usando streams paralelas:

```
public static <T> Supplier<Double>
    t7_8_3(List<T> list, Function<T, Double> f) {
    return () -> list.parallelStream()
        .mapToDouble(f::apply).sum();
}
```

3.7.2 Resultados

Tabela 7: Resultado do Teste 7 em milissegundos

	J7 List	J8 Spliterator	J8 Sequential	J8 Parallel
1000000	7,344	12,178	21,814	10,078
2000000	16,164	19,550	52,030	20,004
4000000	28,981	41,671	106,038	38,151
6000000	38,908	53,789	135,662	55,450

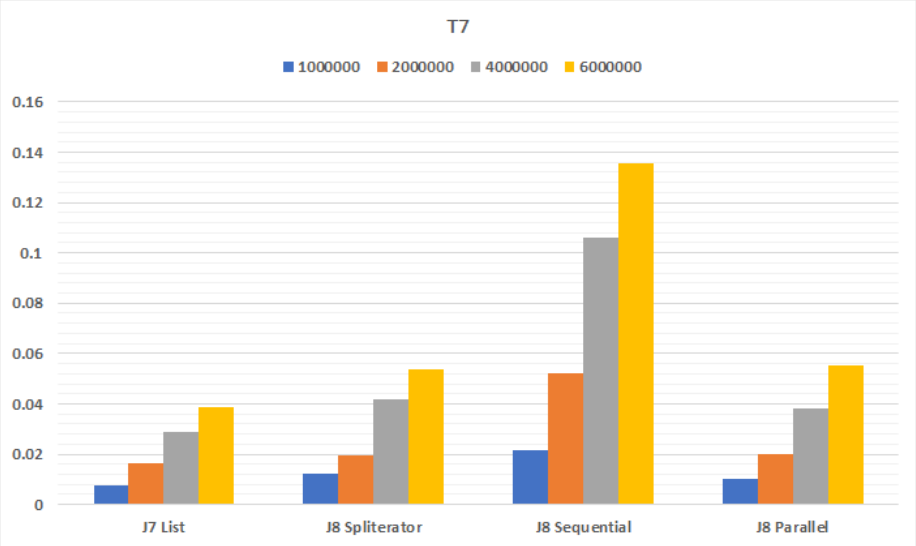

Figura 7: Gráfico do Teste 7 em segundos

3.7.3 Análise e Conclusões

É possível observar de imediato que usando Java 8 sequencial é menos eficiente que as outras 2 alternativas de Java 8 em todos os casos. Contudo, Java 7 é melhor alternativa para este tipo de testes visto que se pode criar apenas uma única variável que funciona como um acumulador sem usar boxing e unboxing.

3.8 Teste 8

Codifique em JAVA 7 e em JAVA 8 com streams, o problema de, dada a List<TransCaixa>, determinar o código da transação de maior valor realizada num a dada data válida entre as 16 e as 22 horas.

3.8.1 Código

1. Transição de maior valor entre as 16h e 22h em Java 7:

```
public static Supplier<String>
t8_7(List<Transaction> transactions) {
    return () -> {
        String id = "";
        double value = 0;

        for (Transaction t : transactions) {
```

```
                if (t.getDate().getHour() >= 16
                    && t.getDate().getHour() < 22
                    && t.getValue() > value) {
                    id = t.getId();
                    value = t.getValue();
                }
            }

            return id;
        };
    }
```

2. Transição de maior valor entre as 16h e 22h em Java 8:

```
public static <T> Supplier<String>
t8_8(List<T> list, Predicate<T> predicate,
Function<T, String> key, Function<T, Double> value) {
return () -> {
    final StringDoubleWrapper wrapper =
        new StringDoubleWrapper();

    list.forEach(t -> {
        if (predicate.test(t) &&
            value.apply(t) > wrapper.getValue()) {
            wrapper.set(key.apply(t),
            value.apply(t));
        }
    });

    return wrapper.getId();
};
}
```

3.8.2 Resultados

Tabela 8: Resultado do teste 8 em milissegundos

	Java 7	Java 8
1	13,63	24,23225
2	28,16587	47,96988
4	52,54745	81,02424
6	81,38335	121,821

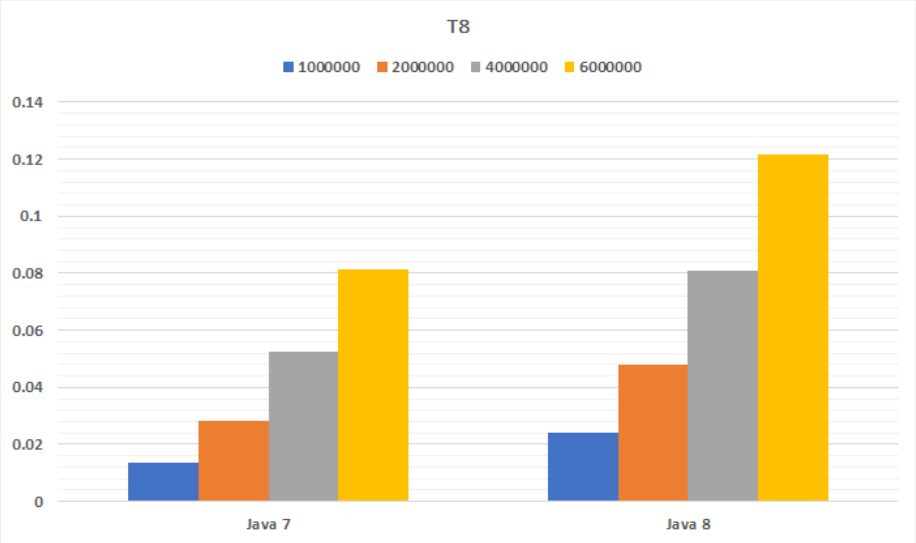

Figura 8: Gráfico do teste 8 em segundos

3.8.3 Análise e Conclusões

Neste tipo de teste o resultado foi mais proveitoso em Java 7 possivelmente por se ter usado uma abstração em Java 8 fornecendo duas Functions e um Predicate. Por outro lado em Java 8 é permitido reutilizar este código para se efetuar qualquer tipo de testes, bastando passar como argumento um predicado da classe correspondente.

3.9 Teste 9

Crie uma List<List<TransCaixa» em que cada lista elemento da lista contém todas as transações 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 faturado nessa semana.

3.9.1 Código

1. Em Java 7:

```
public static Supplier<Double> t9_7(
List<Transaction> transactions, int weekOfYear) {
    return t9_7(toWeekDayLists(transactions)
        .get(weekOfYear));
}
```

```
private static Supplier<Double> t9_7(
List<List<Transaction>> transactions) {
    return () -> {
        double sum = 0;
        for (List<Transaction> l : transactions) {
            for (Transaction t : l) {
                sum += t.getValue();
            }
        }
        return sum;
    };
}
```

2. Em Java 8:

```
public static Supplier<Double> t9_8(
List<Transaction> transactions, int weekOfYear) {
    return t9_8(toWeekDayLists(transactions)
        .get(weekOfYear));
}

private static Supplier<Double> t9_8(
List<List<Transaction>> transactions) {
    return () -> transactions.stream()
        .map(l -> l.stream()
        .map(Transaction::getValue)
        .reduce(0.0, Double::sum))
        .reduce(0.0, Double::sum);
}
```

3. Função auxiliar:

```
private static List<List<List<Transaction>>>
toWeekDayLists(List<Transaction> transactions) {
    int week;
    int day;
    List<List<Transaction>> days;

    final List<List<List<Transaction>>>
        weekDayList = new ArrayList<>();

    for (int i = 0; i < 53; i++) {
        days = new ArrayList<>();
        for (int j = 0; j < 7; j++) {
            days.add(new ArrayList<>());
        }
        weekDayList.add(days);
```

```
        }

        for (Transaction t: transactions) {
            week = t.getDate()
            .get(IsoFields.WEEK_OF_WEEK_BASED_YEAR) - 1;
            day = t.getDate()
            .getDayOfWeek().getValue() - 1;

            weekDayList.get(week).get(day).add(t);
        }

        return weekDayList;
    }
```

3.9.2 Resultados

Tabela 9: Resultado do teste 9 em segundos

	Java 7	Java 8
1000000	0.001163	0.001214
2000000	7.32E-04	0.002543
4000000	0.001502	0.004936
6000000	1.07E-05	4.39E-05

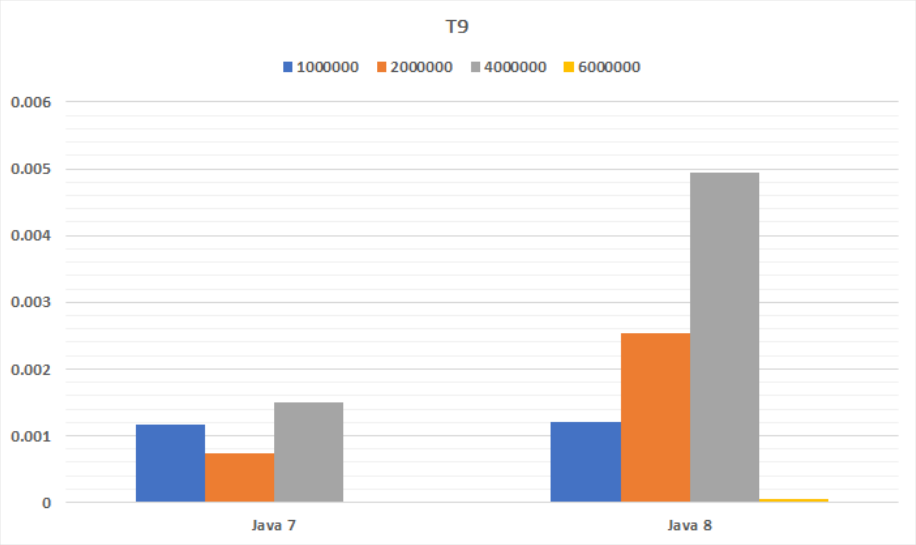

Figura 9: Gráfico do Teste 9 em segundos

3.9.3 Análise e Conclusões

Os tempos em Java 7 foram melhores que em Java 8 devido ao facto se poder efetuar uma acumulação do valor final numa variável sem ser necessário o boxing e o unboxing efetuado na versão de Java 8. Contudo, o grupo não conseguiu explicar a razão do desfasamento de tempo de execução para o ficheiro de 6 milhões, em ambos os casos, dado que este é inferior.

3.10 Teste 10

Admitindo que o IVA a entregar por transação é de 12% para transações 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.

3.10.1 Código

1. Em Java 7:

```
public static Supplier<List<Double>>
t10_7(List<Transaction> transactions) {
    return () -> {
        double value;
        int month;
        final List<Double> vat = new ArrayList<>();

        for (int i = 0; i < 12; i++) {
            vat.add(0.0);
        }

        for (Transaction t : transactions) {
            value = t.getValue();
            month = t.getDate()
            .getMonth().getValue() - 1;

            if (value > 29) {
                vat.set(month,
                    vat.get(month) + value * 0.23);
            } else if (value < 20) {
                vat.set(month,
                    vat.get(month) + value * 0.12);
            } else {
                vat.set(month,
                    vat.get(month) + value * 0.20);
            }
        }
```

```
            return vat;
        };
    }
```

2. Em Java 8:

```
public static <T> Supplier<List<Double>>
t10_8(List<T> list, Function<T, Integer> f,
Function<T, Double> g) {
    return () -> new ArrayList<>(list.stream()
        .collect(groupingBy(f, TreeMap::new,
        summingDouble(g::apply))).values());
}
```

3.10.2 Resultados

Tabela 10: Resultado do teste 10 em segundos

	Java 7	Java 8
1000000	0.024438	0.070644
2000000	0.053496	0.147124
4000000	0.096706	0.274338
6000000	0.148712	0.441982

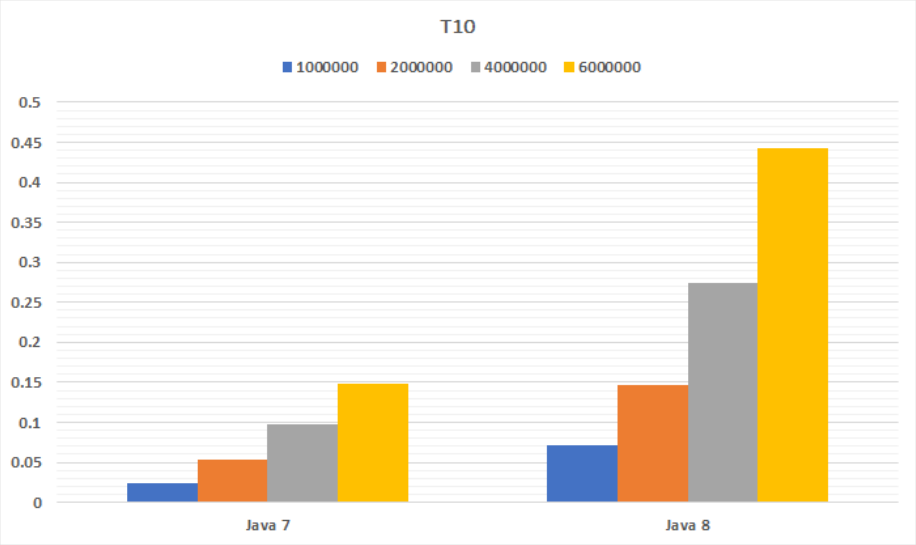

Figura 10: Gráfico do Teste 10 em segundos

3.10.3 Análise e Conclusões

Este teste foi mais eficiente em Java 7 possivelmente por se ter usado abstrações com recurso a Functions (e respetivos métodos apply) e por ser usado o boxing e unboxing de double em Java 8.

3.11 Teste 11

Selecione 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.

3.11.1 Resultados

Para este teste foram escolhidas as seguintes queries:

- *Teste 1*: DoubleStream sequencial;
- *Teste 6*: Map<Month, Map<Day, Map<Hour, List<Transaction»;
- *Teste 8*: Teste em Java 8;
- *Teste 10*: Teste em Java 8.

Tabela 11: Resultado do teste 11 com JDK 8 em segundos

	T1	T6	T8	T10
1000000	3.08E-05	0.162955	0.022184	0.07261
2000000	1.78E-05	0.345982	0.046813	0.150293
4000000	3.00E-05	0.68956	0.086764	0.281543
6000000	3.04E-05	0.786749	0.133301	0.451112

Tabela 12: Resultado do teste 11 com JDK 9 em segundos

	T1	T6	T8	T10
1000000	4.19E-05	0.236322	0.021797	0.066262
2000000	4.19E-05	0.459885	0.043341	0.151582
4000000	2.41E-05	0.841491	0.079077	0.265608
6000000	2.49E-05	0.860312	0.11701	0.418868

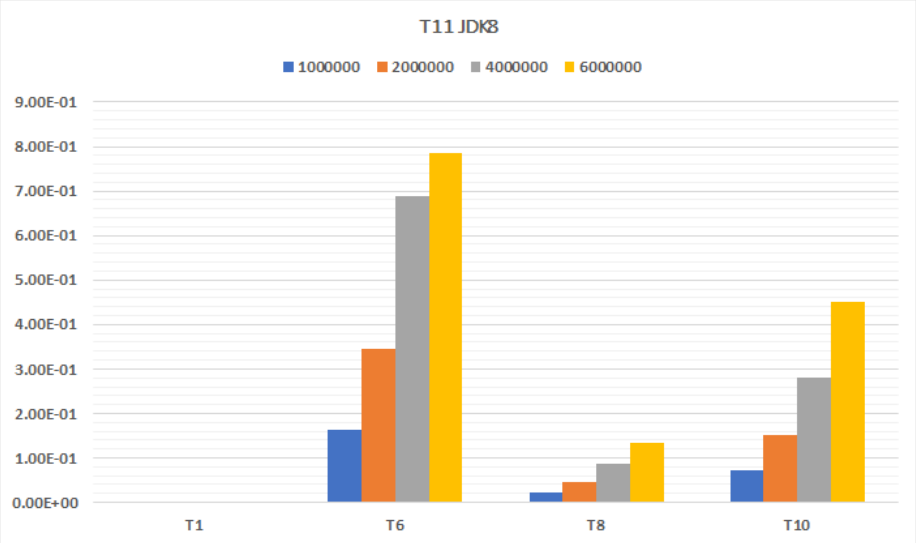

Figura 11: Gráfico do Teste 11 com JDK 8 em segundos

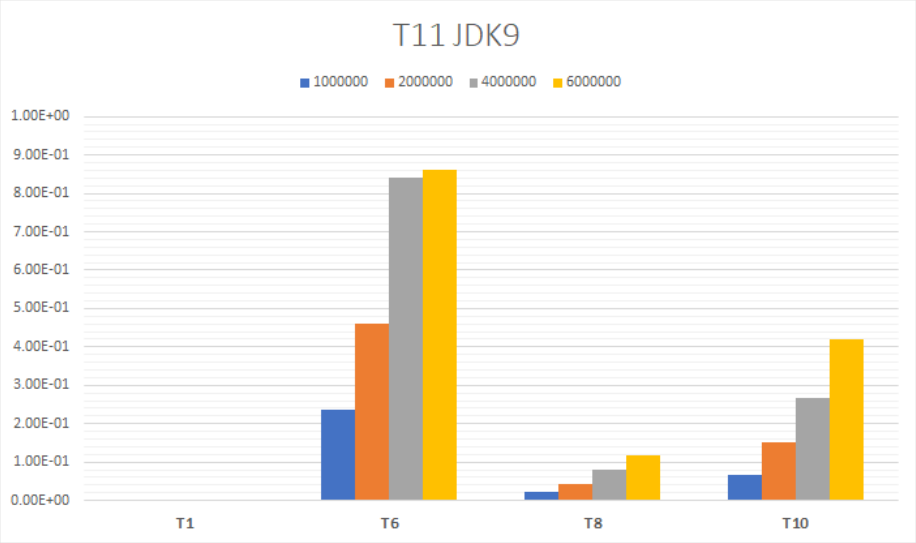

Figura 12: Gráfico do Teste 11 com JDK 9 em segundos

3.11.2 Análise e Conclusões

Para os testes efetuados para JDK 8 e JDK 9 os resultados revelaram-se inconclusivos relativamente à eficiência destes 2 development kits.

3.12 Teste 12

Considerando List<TransCaixa> criar uma tabela que associa a cada nº de caixa uma tabela contendo para cada mês as transações 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 faturado por caixa em JAVA 8 e em JAVA 9.

3.12.1 Código

1. Map e ConcurrentMap de <String, Map<Month, List<Transaction»:

```
public static Supplier<Map<String, Map<Month,
List<Transaction>>>>
t12_Map_1(List<Transaction> transactions) {
    return () -> transactions
        .parallelStream()
        .collect(groupingBy(
        Transaction::getCounterId,
        groupingBy(t -> t.getDate().getMonth())));
}

public static Supplier<ConcurrentMap<String,
ConcurrentMap<Month, List<Transaction>>>>
t12_ConcurrentMap_1(List<Transaction> transactions) {
    return () -> transactions
        .parallelStream()
        .collect(groupingByConcurrent(
        Transaction::getCounterId,
        groupingByConcurrent(t -> t.getDate()
        .getMonth())));
}
```

2. Map e ConcurrentMap de <String, Double>:

```
public static Supplier<Map<String, Double>>
t12_Map_2(Map<String,
Map<Month, List<Transaction>>> map) {
    return () -> map.entrySet().stream()
    .collect(toMap(
    Map.Entry::getKey,
    e -> e.getValue().values()
```

```
            .stream().map(l -> l.stream()
            .map(Transaction::getValue)
            .reduce(0.0, Double::sum))
            .reduce(0.0, Double::sum), Double::sum)
        );
    }

    public static Supplier
    <ConcurrentMap<String, Double>>
    t12_ConcurrentMap_2(ConcurrentMap<String,
    ConcurrentMap<Month, List<Transaction>>> map) {
        return () -> map.entrySet().parallelStream()
            .collect(toConcurrentMap(
            ConcurrentMap.Entry::getKey,
            e -> e.getValue().values()
            .stream()
            .map(l -> l.stream()
            .map(Transaction::getValue)
            .reduce(0.0, Double::sum))
            .reduce(0.0, Double::sum), Double::sum)
        );
    }
```

3.12.2 Resultados

Tabela 13: Resultado do teste 12 com JDK 8 em segundos

	Map <CounterId Map <Month List>>	ConcurrentMap <CounterId ConcurrentMap <Month List>>	Map <CounterId Double>	ConcurrentMap <CounterId Double>
1000000	0.025782072	0.042849496	0.056139741	0.020937836
2000000	0.051660145	0.08546551	0.118305741	0.039550737
4000000	0.100388937	0.177639155	0.235739583	0.077616044
6000000	0.146619365	0.282298701	0.354207696	0.118128754

Tabela 14: Resultado do teste 12 com JDK 9 em segundos

	Map <CounterId Map<Month List>>	ConcurrentMap <CounterId ConcurrentMap <Month List>>	Map <CounterId Double>	ConcurrentMap <CounterId Double>
1000000	0.030887445	0.031997172	0.058653909	0.021487365
2000000	0.064432465	0.069661097	0.129949373	0.040947672
4000000	0.13878017	0.134054056	0.283456625	0.084649314
6000000	0.184193611	0.187335136	0.455469322	0.141091276

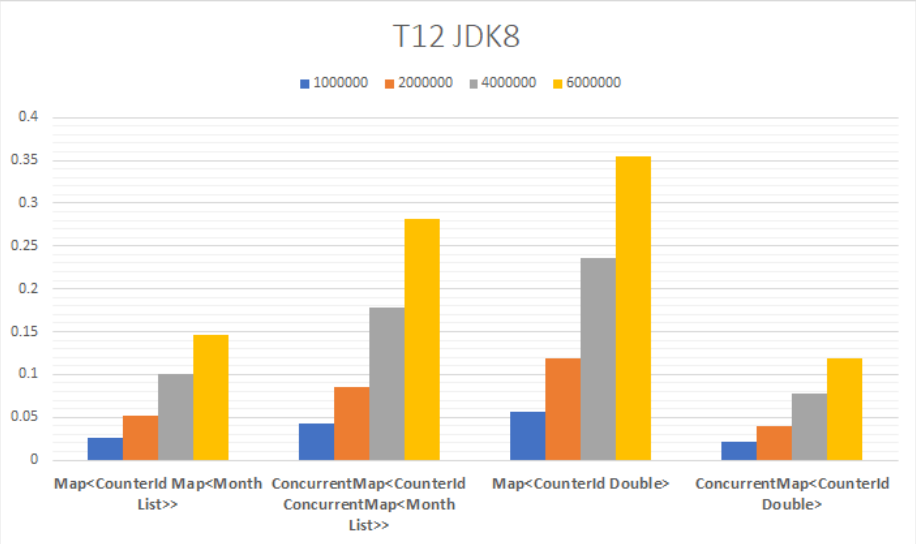

Figura 13: Gráfico do Teste 12 com JDK 8 em segundos

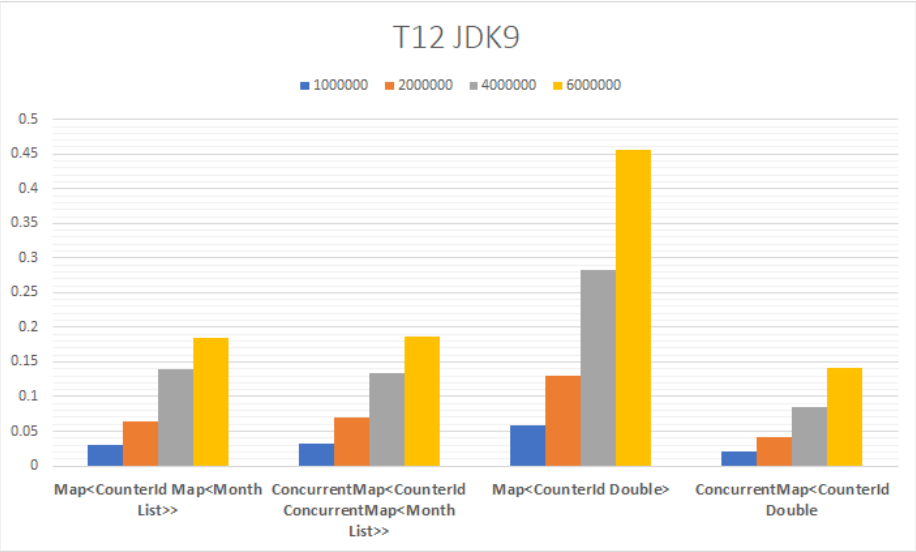

Figura 14: Gráfico do Teste 12 com JDK 9 em segundos

3.12.3 Análise e Conclusões

Para este caso pode-se verificar pelas figuras ilustradas acima que o ConcurrentMap sofreu uma otimização em JDK 9.

4 Conclusões e Análise Crítica

Este projeto permitiu aprofundar largamente os conhecimentos relacionados com streams de Java e o novo mecanismo de generalização introduzido em Java 8. E, neste seguimento, de forma a melhorar a reutilização do código desenvolvido, utilizaram-se tipos de dados genéricos sempre que possível, ou sempre que o grupo considerasse adequado.

Através da realização dos testes de benchmarking, foi possível obter um conhecimento concreto acerca da eficiência das estruturas de dados em diversos contextos. Aqui, naturalmente, ao utilizar a nova API de Java e comparar esta com Java 7 é possível entender em que contextos compensa utilizar um método ou outro. Da mesma maneira, também é importante determinar em que situações é proveitoso tirar partido do paralelismo.

Dos testes realizados, determinou-se que Java 8 nem sempre é mais eficiente. Em particular na utilização do método groupingBy que insere algum overhead no algoritmo. Neste tipo de situações revela-se mais eficiente recorrer a Java 7 e efetuar o agrupamento explícito. Outro caso relevante de salientar é o facto de IntStreams, DoubleStreams e Stream<Double> serem muito eficientes em comparação com as estruturas anteriores (arrays ou listas). Também foi interessante verificar que utilizar um collect para uma lista após um sorted é mais eficiente que utilizar um set com um comparator, algo que será tido em conta no futuro.

Por fim, o paralelismo tende a apenas compensar quando se tratam de operações completamente independentes tal como ocorreu no Teste 2.