TRABALHO PRÁTICO II

Arrays, Collections versus Streams
Streams sequenciais versus paralelas

Java Benchmarking

Benchmarking Java Streams

❶ Introdução e Objectivos.

- Na Engenharia de Software, áreas como software testing, profiling, benchmarking, performance testing, são, só por si, áreas especiais de estudo, com bases científicas, técnicas e metodologias próprias, em geral abordadas em disciplinas específicas, que cobrem desde aplicações desktop a aplicações web passando por aplicações móveis;

- O contexto deste trabalho é ainda mais fechado; Estaremos preocupados em medir a performance de porções de código Java desenvolvivo usando construções particulares como Arrays, Collecções e Streams, procurando compreender vantagens e desvantagens das várias alternativas disponíveis para igual solução;

- Neste trabalho iremos fazer o que, em geral, se designa por **Java Benchmarking for Massive Data**; Existem algumas boas ferramentas de benchmarking para Java, em particular **JMH** (Java Benchmark Harness) e Caliper, estando JMH já disponível como plugin de vários IDEs e, aparentemente, com JDK9;

- Porém, estudar e usar convenientemente JMH tem um tempo de aprendizagem incompatível com uma disciplina semestral com os objectivos de PDSJ, pelo que usaremos uma técnica mais próxima de **"Experimentation and measurement"**, o que, no nosso caso, não tem qualquer problema pois pretendemos resultados comparativos e não absolutos;

- O excelente pequeno artigo indicado a seguir estabelece muito bem os limites de credibilidade e interesse do benchmarking de aplicações por contraste com o benchmarking de porções de código:

  Microbenchmarking Comes to Java 9, Peter Verhas, Dzone, Sept, 2016
  https://dzone.com/articles/microbenchmarking-comes-to-java-9

Benchmarking Java Streams

- Clarificando ainda melhor o contexto e objectivos deste trabalho (um **estudo comparativo de performance**), outros dedicam-se mesmo, não a testar as existentes Colecções e Streams de Java, mas antes a desenvolver estruturas alternativas às existentes para lidarem com quantidades massivas de dados. No exemplo apresentado a seguir, uma equipa de programadores desenvolveu novas implementações de List e apresentam os testes de performance e de memória para 1_000_000 de valores;

BigList: a Scalable High-Performance List for Java, Tomas Mauch, 2014
https://dzone.com/articles/biglist-scalable-high

	BigList	GapList	ArrayList	LinkedList	TreeList	FastTable
Get random	8.8	1.1	1.0	23'912.0	21.4	2.5
Add random	1.0	402.0	1'066.0	543.0	1.7	4.1
Remove random	1.0	257.0	300.0	1'176.0	4.3	15.2
Get local	1.9	1.3	1.0	357.0	6.5	1.9
Add local	1.0	16.5	1'256.0	9'382.0	6.2	31.3
Remove local	1.0	1.4	2'945.0	29'455.0	15.2	92.0
Add multiple	1.0	3.3	61.8	6.8	10.3	79.5
Remove multiple	1.0	22.0	4.8	59.7	792.0	7'097.0
Copy	1.0	2.4	3.1	426.0	276.0	97.0

- No nosso caso, usaremos as classes standard de JDK7/8/9 e realizaremos testes comparativos de performance entre elas, em especial Collections versus Streams, para operações que irão processar de 1M a 8M de valores e de objectos, usando as construções de iteração actualmente disponíveis para tais processamentos, designadamente, Iterator<T>, iterador for e for-each de Java 7, forEach de Java 8, e lambdas e Streams de Java 8 e de Java 9, sequenciais e paralelas.

❷ Realização dos testes.

- Todo os testes serão realizados tendo por base os ficheiros de texto **transCaixa1M.txt** (usado nas aulas práticas), **transCaixa2M.txt**, **transCaixa4M.txt** e **transCaixa8M.txt**, respectivamente de 1, 2, 4 e 8 milhões de linhas de texto. Cada linha de texto representa uma transacção de caixa e, feito o seu *parsing*, gerará uma instância da classe **TransCaixa**;

- O método **static List<TransCaixa> setup(String ficheiro);**, cujo código está disponível, realiza a leitura e parsing das linhas do ficheiro indicado e criará a **List<TransCaixa>** que servirá de base à maioria dos testes a realizar;

- O método static genérico **testeBoxGenW(Supplier<? extends R> supplier);** será o método de referência a usar em todos os testes;

```
public static <R> SimpleEntry<Double,R> testeBoxGenW(Supplier<? extends R> supplier)  {
    // com warmup de 5 runs
    for(int i = 1 ; i <= 5; i++) supplier.get();
    System.gc();
    Crono.start();
    R resultado = supplier.get();
    Double tempo = Crono.stop();
    return new SimpleEntry<Double,R>(tempo, resultado);
}
```

Benchmarking Java Streams

- Por questões de clareza e referencial, deverá ser apresentado o ambiente software e hardware utilizado na realização dos testes, cf. OS, JDK, IDE, CPU (cores) e RAM.

- Como guideline para a apresentação de resultados e visando até alguma normalização, sugere-se o seguinte formato por teste:

Teste Nº _: <Descrição do teste realizado>

Obs: <Descrição de circunstâncias particulares>

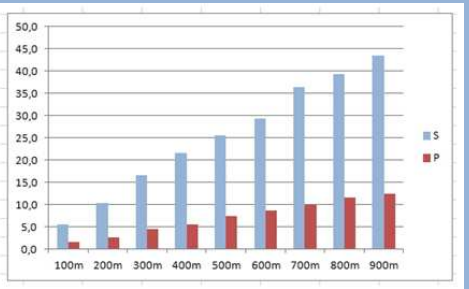
Código A

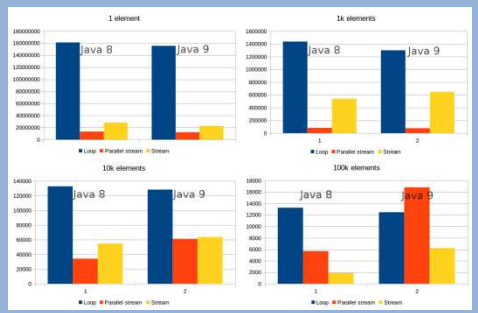
Código B

Código C

Res:

N	S	P
100m	5,5	1,5
200m	10,4	2,7
300m	16,5	4,4
400m	21,7	5,5
500m	25,6	7,4
600m	29,4	8,6
700m	36,4	10,0
800m	39,4	11,6
900m	43,5	12,5





Análise e Conclusões:

❸ Testes a realizar.

T1: 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.

T2: 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 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<>.

T3: 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.

T4: 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.

T5: 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>.

T6: 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.

T7: Usando List<TransCaixa> e Spliterator<TransCaixa> crie 4 partições cada uma com ¼ do data set. Compare os tempos de processamento de calcular a soma do valor das transacções com as quatro partições ou com o data set inteiro, quer usando List<> e forEach() quer usando streams sequenciais e paralelas.

T8: 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.

T9: 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.

T10: 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.

T11: 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.

T12: 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.