Práctica #3

Streams en Java

Rubén Escalante Chan (A01370880), Guillermo Pérez Trueba (A01377162)

24 de febrero, 2019.

Tabla de contenido

Este reporte fue elaborado para el curso de *Programación multinúcleo* del Tecnológico de Monterrey, Campus Estado de México.

1. Introducción

La práctica consta de 2 problemas. El primero se describe a continuación:

Ordenmiento por Conteo: asumimos que los elementos en la entrada son enteros no negativos en el rango {0, 1, . . . , k}. En ordenamiento por conteo primero contamos cuántos elementos son menores o iguales a x, para cada elemento x. Una vez que la información es calculada, cada elemento x es colocado directamente en su posici´on final en el arreglo de salida. Por ejemplo, si existen 2 copias de x y existen 12 elementos menores a x en la entrada, entonces las posiciones 13 y 14 de la salida deben contener a x. En esta práctica se ordenó un arreglo de números entre 1 y 100 que se generaron aleatoriamente.

El segundo problema consta de obtener el valor de PI según una 'n' dada. Posteriormente se generan 'n' puntos 2D aleatorios y se calcula si están dentro de la circunferencia. Para éste se utiliza el teorema de pitagoras y se compara el resultado, es decir {x * x + y * y < 1}. Si ésto se cumple, entonces se incrementa en uno el contador de puntos dentro de la circunferencia.

El objetivo consistió en resolver estos problema de manera secuencial y paralela usando *Streams* de Java.

Hardware y software utilizado

Los programas se probaron en una computadora de escritorio con las siguientes características:

- Procesador Intel Core i7-4770 de 3.40GHz con cuatro núcleos y ocho *hyperthreads*.
- 1 GiB de memoria RAM.
- Sistema operativo Ubuntu 14.04, kernel de Linux 3.13.0-107 de 64 bits.
- Compilador Java 1.8.0_51 de Oracle.

2. Solución

2.1. Ordenamiento Por Conteo

El siguiente listado muestra un programa completo en Java que ordena de forma secuencial y paralela un arreglo de tamaño 100:

OrdenamientoPorConteo.java

```
/*------------------------------------------------------------
 * Práctica #3: Streams en Java
 * Fecha: 24-Feb-2019
```

```
 * Autores:
 *          A01370880 Rubén Escalante Chan
 *          A01377162 Guillermo Pérez Trueba
 *-----------------------------------------------------------*/

import java.util.Arrays;
import java.util.Random;
import java.util.stream.IntStream;

public class OrdenamientoPorConteo {

    private static final int NUM_RECTS = 100_000;
    private static int[]a, result;
    private static int n;

    private static void mapper(int i) {

        int count = 0;
        for (int j = 0; j < n; j++) {
            if (a[j] < a[i]) {
                count++;
            } else if (a[i] == a[j] && j < i) {
                count++;
            }
        }
        result[count] = a[i];
    }

    public static void parallelCountSort() {
        IntStream.range(0, n).parallel().forEach(OrdenamientoPorConteoStreams::
mapper); //for (int i = 0; i < n; i++);
    }

    private static void sequentialCountSort() {
        IntStream.range(0, n).forEach(OrdenamientoPorConteoStreams::mapper); //for
(int i = 0; i < n; i++);
    }

    public static void main(String[] args) {

        a = new int[NUM_RECTS];

        for (int i = 0; i < NUM_RECTS; i++)
            a[i] = new Random().nextInt(100);

        n = a.length;

        result = new int[NUM_RECTS];
        long inicio = System.nanoTime();
        sequentialCountSort();
        long fin = System.nanoTime();
```

```
        double time = (fin - inicio)/1E9;
        System.out.printf("Array = %s T1 = %.4f%n", Arrays.toString(result), time);

        result = new int[NUM_RECTS];
        inicio = System.nanoTime();
        parallelCountSort();
        fin = System.nanoTime();
        double time8 = (fin - inicio)/1E9;
        System.out.printf("Array = %s T8 = %.4f%n", Arrays.toString(result), time8);

    }
}
```

El programa produce esta salida:

```
Array = [1, 2, 2, 2, 4, 4, 5, 7, 8, 9, 9, 9, 9, 9, 10, 11, 16, 16, 21, 22, 23, 25, 26,
27, 28, 28, 30, 31, 32, 32, 32, 33, 34, 34, 34, 35, 36, 37, 38, 39, 43, 44, 44, 45,
46, 47, 47, 47, 48, 50, 50, 51, 51, 52, 52, 52, 54, 56, 60, 61, 61, 62, 62, 65, 66,
66, 66, 66, 67, 67, 68, 71, 71, 71, 72, 73, 73, 74, 74, 74, 75, 76, 76, 77, 79, 83,
83, 84, 84, 86, 87, 87, 87, 89, 90, 91, 91, 92, 96, 97]
T1 = 0.0441

Array = [1, 2, 2, 2, 4, 4, 5, 7, 8, 9, 9, 9, 9, 9, 10, 11, 16, 16, 21, 22, 23, 25, 26,
27, 28, 28, 30, 31, 32, 32, 32, 33, 34, 34, 34, 35, 36, 37, 38, 39, 43, 44, 44, 45,
46, 47, 47, 47, 48, 50, 50, 51, 51, 52, 52, 52, 54, 56, 60, 61, 61, 62, 62, 65, 66,
66, 66, 66, 67, 67, 68, 71, 71, 71, 72, 73, 73, 74, 74, 74, 75, 76, 76, 77, 79, 83,
83, 84, 84, 86, 87, 87, 87, 89, 90, 91, 91, 92, 96, 97]
Tn = 0.0039

Speeddup = 1.13076
```

El arreglo es el mismo que en la versión secuencial, por lo que podemos suponer que nuestra versión paralela produce el resultado correcto.

2.2. Monte Carlo

El siguiente listado muestra un programa completo en Java que calcula de forma secuencial y paralela el número Π dónde 'n' es 1_000_000:

MonteCarlo.java

```
/*------------------------------------------------------
 * Práctica #3: Streams en Java
 * Fecha: 24-Feb-2019
 * Autores:
 *          A01370880 Rubén Escalante Chan
 *          A01377162 Guillermo Pérez Trueba
 *------------------------------------------------------*/
```

```java
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.IntStream;

public class StreamMonteCarlo {

    private static double mapper(int unused) {
        double x = ThreadLocalRandom.current().nextDouble() * 2 - 1;
        double y = ThreadLocalRandom.current().nextDouble() * 2 - 1;

        // Aplicar teorema de Pitágoras.
        return (x * x + y * y);
    }

    public static double sequentialMonteCarlo(int n) {
        long c = IntStream.range(0, n)
                .mapToDouble(StreamMonteCarlo::mapper)
                .filter(x -> x <= 1)
                .count();

        return 4 * ((double) c / n);
    }

    public static double parallelMonteCarlo(int n) {
        long c = IntStream.range(0, n)
                .parallel()
                .mapToDouble(StreamMonteCarlo::mapper)
                .filter(x -> x <= 1)
                .count();

        return 4 * ((double) c / n);
    }

    public static void main(String[] args) {

        final int N = 1_000_000;
        long inicio, fin;
        double result, t1, t8;

        // Secuencial
        inicio = System.nanoTime();
        result = sequentialMonteCarlo(N);
        fin = System.nanoTime();
        t1 = (fin - inicio) / 1.0e9;
        System.out.println("\nSecuencial");
        System.out.printf("Pi = %.10f T1 = %.2f%n", result, t1);

        // Paralelo
        inicio = System.nanoTime();
        result = sequentialMonteCarlo(N);
        fin = System.nanoTime();
        t8 = (fin - inicio) / 1.0e9;
```

```
        System.out.println("\nParalelo");
        System.out.printf("Pi = %.10f T1 = %.2f%n", result, t8);

        double speedup = t1 / t8;
        System.out.printf("\nSpeed Up = %.4f%n", speedup );
    }

}
```

El programa produce esta salida:

```
Secuencial
Pi = 3.1397640000 T1 = 0.09

Paralelo
Pi = 3.1435200000 T1 = 0.02

Speed Up = 4.9580
```

3. Resultados

3.1. Ordenamiento Por Conteo

A continuación se muestran los tiempos de ejecución de varias corridas de los dos programas:

Tabla 1. Tiempos de ejecución del ordenamiento secuencial

# de corrida	Tiempo T_1 (segundos)
1	25.5492
2	25.9551
3	25.5839
4	31.0371
5	26.5620
Media aritmética	26.9374

Tabla 2. Tiempos de ejecución del ordenamiento paralelo

# de corrida	Tiempo T_2 (segundos)
1	9.5628
2	10.3210
3	12.3339
4	11.4265
5	13.2698
Media aritmética	11.3826

A partir de las medias aritméticas calculadas, el *speedup* obtenido en un CPU es:

$S_2 = T_1 / T_2 = 26.9374 / 11.3826 = 2.3665$

El *speedup* obtenido es bastante bueno. Al haber una mejora en el tiempo, se puede concluir que el uso de *parallel* de la libreria de *java.util.stream* si nos ayuda a optimizar nuestros programas. Otra ventaja es que resulta bastante sencillo el uso de la misma.

3.2. Monte Carlo

A continuación se muestran los tiempos de ejecución de varias corridas de los dos programas:

Tabla 3. Tiempos de ejecución del cálculo secuencial

# de corrida	Tiempo T_1 (segundos)
1	0.09
2	0.08
3	0.08
4	0.08
5	0.07
Media aritmética	0.08

Tabla 4. Tiempos de ejecución del cálculo paralelo

# de corrida	Tiempo T_1 (segundos)	1
0.02	2	0.02
3	0.02	4
0.02	5	0.02

A partir de las medias aritméticas calculadas, el *speedup* obtenido en un CPU es:

$S_2 = T_1 / T_2 = 0.08 / 0.02 = 4$

El *speedup* obtenido es muy bueno. Para 'n' no muy altos, el uso de Streams paralelos es muy recomendable.

4. Agradecimientos

Se agradece al profesor Ariel Ortiz por sus enseñanzas y ayuda para solucionar este problema.

5. Referencias

- [Oracle] Oracle Corporation. *Module java.base from the Java 11 API Specification*

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html.

- Blai Bonet. (2018). CI2612: Algoritmos y Estructuras de Datos II. 16/02/2019+, de Universidad Simón Bolívar Sitio web: https://bonetblai.github.io/courses/ci2612/handouts/ci2612-lec10.pdf