Práctica #2

Fork/Join en Java

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

16 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:

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 usando la librería *ForkJoinPool* de Java para obtener una solución paralela.

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 #2: Fork/Join en Java
 * Fecha: 17-Feb-2019
```

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

import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import static java.util.concurrent.ForkJoinTask.invokeAll;
import java.util.concurrent.RecursiveAction;


public class OrdenamientoPorConteo {
    private static final int NUM_RECTS = 100;
    public static class SortTask extends RecursiveAction {

        private static final long serialVersionUID = 1L; //permite revisar si son
errores cmpatibles tras la deserelizacion

        public static final int UMBRAL = 1_000;
        private long lo, hi;
        private static int[] a, temp;
        public SortTask(long lo, long hi, int[] a){
            this.lo = lo;
            this.hi = hi;
            this.a = a;
        }

        @Override
        protected void compute() {
            if (hi - lo < UMBRAL) {
                int[] temp = new int[NUM_RECTS];
                for (int i = (int) lo; i <  hi; i++) {
                    int count = 0;
                    for (int j = (int) lo; j < hi; j++) {
                        if (a[j] < a[i]) {
                            count++;
                        } else if (a[i] == a[j] && j < i) {
                            count++;
                        }
                    }
                    temp[count + (int) lo] = a[i];
                }
                 System.arraycopy(temp, 0, a, 0, (int) hi);
            } else {

                long mid = (hi + lo) >>> 1;
                SortTask t2 = new SortTask(mid, hi, a);
                SortTask t1 = new SortTask(lo, mid, a);
                invokeAll(t2, t1);
            }
```

```
        }
    }

    public static void parallelCountSort(int[] a) {
        ForkJoinPool pool = new ForkJoinPool();
        SortTask t = new SortTask(0, NUM_RECTS, a);
        pool.invoke(t);
    }

    public static void sequentialCountSort(int a[]) {
        final int n = a.length;
        final int[] temp = new int[n];
        for (int i = 0; i < n; 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++;
                }
            }
            temp[count] = a[i];
        }
        System.arraycopy(temp, 0, a, 0, n);

    }

    public static void main(String[] args) {
        int[] a = new int[NUM_RECTS];
        int[] b = new int[NUM_RECTS];
        for (int i = 0; i < NUM_RECTS; i++) {
            int rnd =  new Random().nextInt(100);
            a[i] = rnd;
            b[i] = rnd;

        }

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

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

        double speedup = time / time8;
        System.out.printf("Speed Up = %.4f%n", speedup );
```

```
    }
}
```

El programa produce esta salida:

```
Array = [0, 1, 4, 5, 5, 5, 6, 7, 7, 8, 8, 10, 12, 12, 12, 13, 14, 16, 17, 18, 18, 19,
21, 21, 21, 21, 22, 23, 23, 23, 24, 25, 27, 27, 28, 29, 29, 31, 34, 35, 36, 37, 37,
39, 39, 39, 40, 42, 43, 44, 45, 46, 47, 47, 49, 52, 52, 61, 63, 63, 64, 64, 65, 65,
66, 68, 70, 70, 71, 75, 76, 76, 76, 77, 78, 79, 81, 82, 83, 83, 83, 84, 85, 86, 88,
89, 89, 90, 91, 91, 91, 92, 93, 93, 96, 97, 97, 97, 98, 99]
T1 = 0.0003

Array = [0, 1, 4, 5, 5, 5, 6, 7, 7, 8, 8, 10, 12, 12, 12, 13, 14, 16, 17, 18, 18, 19,
21, 21, 21, 21, 22, 23, 23, 23, 24, 25, 27, 27, 28, 29, 29, 31, 34, 35, 36, 37, 37,
39, 39, 39, 40, 42, 43, 44, 45, 46, 47, 47, 49, 52, 52, 61, 63, 63, 64, 64, 65, 65,
66, 68, 70, 70, 71, 75, 76, 76, 76, 77, 78, 79, 81, 82, 83, 83, 83, 84, 85, 86, 88,
89, 89, 90, 91, 91, 91, 92, 93, 93, 96, 97, 97, 97, 98, 99]
T8 = 0.0026

Speeddup = 0.0933
```

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_000:

MonteCarlo.java

```
/*------------------------------------------------------------
 * Práctica #2: Fork/Join en Java
 * Fecha: 17-Feb-2019
 * Autores:
 *          A01370880 Rubén Escalante Chan
 *          A01377162 Guillermo Pérez Trueba
 *------------------------------------------------------------*/

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ThreadLocalRandom;

public class MonteCarlo {

    public static class MonteCarloTask extends RecursiveTask<Double> {

        public static final long UMBRAL = 100_000;
```

```
        private final int n;
        private double c;

        public MonteCarloTask(int n) {
            this.n = n;
            this.c = 0.0;
        }

        @Override
        protected Double compute() {
            if (n <= UMBRAL) {
                for (int i = 0; i < n; i++) {
                    // Generar dos números aleatorios entre -1 y 1.
                    double x = ThreadLocalRandom.current().nextDouble() * 2 - 1;
                    double y = ThreadLocalRandom.current().nextDouble() * 2 - 1;

                    // Aplicar teorema de Pitágoras.
                    double h = x * x + y * y;

                    // Verificar si el tiro cayó dentro del círculo.
                    if (h <= 1) {
                        this.c++;
                    }
                }
                return this.c;
            } else {
                MonteCarloTask t1 = new MonteCarloTask(n / 2);
                MonteCarloTask t2 = new MonteCarloTask(n / 2);
                t1.fork();
                double r2 = t2.compute();
                double r1 = t1.join();
                return r1 + r2;
            }
        }

    }

    public static double parallelMonteCarlo(int n) {
        ForkJoinPool pool = new ForkJoinPool();
        MonteCarloTask t = new MonteCarloTask(n);
        double c = pool.invoke(t);
        return 4 * (c / n);
    }

    public static double sequentialMonteCarlo(int n) {
        int c = 0;
        for (int i = 0; i < n; i++) {
            // Generar dos números aleatorios entre -1 y 1.
            double x = ThreadLocalRandom.current().nextDouble() * 2 - 1;
            double y = ThreadLocalRandom.current().nextDouble() * 2 - 1;
```

```
            // Aplicar teorema de Pitágoras.
            double h = x * x + y * y;

            // Verificar si el tiro cayó dentro del círculo.
            if (h <= 1) {
                c++;
            }
        }
        return 4 * ((double) c / n);
    }

    public static void main(String[] args) {

        final int N = 1_000_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 = 3.1416406320 T1 = 8.45

Paralelo
Pi = 3.1416141280 T1 = 8.21

Speed Up = 1.0285
```

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	0.0002
2	0.0002
3	0.0002
4	0.0002
5	0.0002
Media aritmética	0.0002

Tabla 2. Tiempos de ejecución del ordenamiento paralelo

# de corrida	Tiempo T_2 (segundos)
1	0.0026
2	0.0006
3	0.0005
4	0.0006
5	0.0008
Media aritmética	0.0012

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

$$S_2 = T_1 / T_2 = 0.0002 / 0.0012 = 0.16667$$

El *speedup* obtenido es bastante malo. Al no haber una mejora en el tiempo, no se justifica la complejidad adicional asociada al uso de *ForkJoinPool* en Java.

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	Resultado	Tiempo T_1 (segundos)
1	3.1416406320	8.45
2	3.1415384800	8.20
3	3.1415754000	8.24

# de corrida	Resultado	Tiempo T_1 (segundos)
4	3.1415364200	8.64
5	3.1417017760	8.19
Media aritmética	3.1415985416	8.344

Tabla 4. Tiempos de ejecución del cálculo paralelo

# de corrida	Resultado	Tiempo T_1 (segundos)
1	3.1416141280	8.21
2	3.1417134000	8.48
3	3.1415550840	9.26
4	3.1416336320	8.79
5	3.1416005600	9.43
Media aritmética	3.1416233608	8.834

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

$$S_2 = T_1 / T_2 = 8.344 / 8.834 = 0.94453$$

El *speedup* obtenido es negativo. No representa ningun beneficio el uso de *ForkJoinPool* en Java para resolver este ejercicio.

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 (Consultada el 24 de enero, 2019).
- 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