Práctica #1

Calculando Pi con threads en Java

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

27 de enero, 2019.

Tabla de contenido

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

1. Introducción

Según [Wolfram], el número Π puede ser obtenido con la fórmula Bailey-Borwein-Plouffe. Ésta consiste en lo siguiente:

```
Π=Σ(k=0, ∞){[4/(8k+1)]-[2/(8k+4)]-[1/(8k+5)]-[1/(8k+6)]}(1/16)^k
```

En esta práctica se calculó el número Π utilizando las clases `java.math.BigDecimal` y `java.lang.Math` (ver el API de Java en [Oracle]). El objetivo consistió en resolver este problema de manera secuencial y usando *threads* 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 secuencial

El siguiente listado muestra un programa completo en Java que calcula de forma secuencial el número Π dónde el limite de la sumatoria (k) es 100_000_000:

PiSecuencial.java

```
/*----------------------------------------------------------
 * Práctica #1: Calculando Pi con threads en Java
 * Fecha: 27-Ene-2019
 * Autores:
 *          A01370880 Rubén Escalante Chan
 *          A01377162 Guillermo Pérez Trueba
 *----------------------------------------------------------*/

import java.lang.Math;
import java.math.BigDecimal;

public class PiSecuencial {

    public static BigDecimal calcularPi(int upperLimit) {
        BigDecimal sum = new BigDecimal(0);
        double res = 0;
        for (int i = 0; i <= upperLimit; i++) {
            double a, b, c, d;
            a = 4.0 / ((8.0 * i) + 1.0);
            b = 2.0 / ((8.0 * i) + 4.0);
            c = 1.0 / ((8.0 * i) + 5.0);
            d = 1.0 / ((8.0 * i) + 6.0);
            res = (1 / Math.pow(16, i)) * (a - b - c - d);

            sum = sum.add(BigDecimal.valueOf(res));
        }

        return sum;
    }

    public static void main(String[] args) {
        long inicio = System.nanoTime();
        BigDecimal pi = calcularPi(1_000_000);
        long fin = System.nanoTime();

        System.out.printf("Bits = %d, ", pi.movePointRight(pi.precision() - 1
).toBigInteger().bitCount());
        System.out.printf("Pi = %.8f, ", pi);
        System.out.printf("Time = %.4f\n", (fin - inicio) / 1E9);
    }
}
```

Dado que el número de decimales del resultado es extremadamente grande para comprobarlo, se utilizó el método `bitCount` para solo contar el número de bits igual a uno que tiene el valor binario del resultado una vez que se recorre el punto decimal todos los lugares posibles a la derecha.

Esta es la salida del programa:

```
Bits = 537, Pi = 3.14159265, Time = 0.5825
```

3. Solución paralela

La solución paralela en Java involucra usar dos *threads*. El primer *thread* se encarga de calcular la primera mitad de la fórmula Bailey-Borwein-Plouffe: desde 0 hasta 500,000. El segundo *thread* se encarga de calcular la otra mitad de de la fórmula Bailey-Borwein-Plouffe: desde 500,001 hasta 1,000,000.

PiParalelo.java

```
/*-------------------------------------------------------
 * Práctica #1: Calculando Pi con threads en Java
 * Fecha: 27-Ene-2019
 * Autores:
 *          A01370880 Rubén Escalante Chan
 *          A01377162 Guillermo Pérez Trueba
 *-------------------------------------------------------*/

import java.math.BigDecimal;

public class PiParalelo implements Runnable{

    private int start, upperLimit;
    private BigDecimal sum = new BigDecimal(0);

        @Override
    public void run() {
        double res = 0;
        double a, b, c, d;
        for (int i = start; i <= upperLimit; i++) {
            a = 4.0 / ((8.0 * i) + 1.0);
            b = 2.0 / ((8.0 * i) + 4.0);
            c = 1.0 / ((8.0 * i) + 5.0);
            d = 1.0 / ((8.0 * i) + 6.0);
            res = (1 / Math.pow(16, i)) * (a - b - c - d);

            sum = sum.add(BigDecimal.valueOf(res));
        }
    }

    public PiParalelo(int start, int upperLimit) {
        this.start = start;
        this.upperLimit = upperLimit;
    }

    public static BigDecimal calcularPi(int n) {
        PiParalelo p1 = new PiParalelo(0, n / 2);
```

```
        PiParalelo p2 = new PiParalelo(n / 2 + 1, n);
        Thread t1 = new Thread(p1);
        Thread t2 = new Thread(p2);
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e){
            //Nunca pasa
        }

        return p1.sum.add(p2.sum);
    }
    public static void main(String[] args) {
        long inicio = System.nanoTime();
        BigDecimal pi = calcularPi(1_000_000);
        long fin = System.nanoTime();
        System.out.printf("Bits = %d, ", pi.movePointRight(pi.precision() - 1
).toBigInteger().bitCount());
        System.out.printf("Pi = %.8f, ", pi);
        System.out.printf("Time = %.4f\n", (fin - inicio) / 1E9);
    }
}
```

El programa produce esta salida:

```
Bits = 537, Pi = 3.14159265, Time = 0.3048
```

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

4. Resultados

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

Tabla 1. Tiempos de ejecución del factorial secuencial

# de corrida	Tiempo T_1 (segundos)
1	0.5825
2	0.5830
3	0.5856
4	0.5880
5	0.5867
Media aritmética	0.58516

Tabla 2. Tiempos de ejecución del factorial paralelo

# de corrida	Tiempo T_2 (segundos)
1	0.3048
2	0.3545
3	0.3526
4	0.3641
5	0.3565
Media aritmética	0.3465

A partir de las medias aritméticas calculadas, el *speedup* obtenido en un CPU que utiliza dos de sus núcleos (un *thread* corriendo en cada núcleo) es:

$$S_2 = T_1 / T_2 = 0.58516 / 0.3465 = 1.6887$$

El *speedup* obtenido es muy bueno, incluso superando al *speedup* ideal. La mejora obtenida en el tiempo compensa la complejidad adicional asociada al uso de *threads* en Java.

5. Agradecimientos

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

6. 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).
- [Wolfram] Wolfram MathWorld. *Factorial* http://mathworld.wolfram.com/BBPFormula.html (Consultada el 24 de enero, 2019).