Actividad de programación Multiprocesadores:

Calculando Pi con threads en Java

José Ramón Romero Chávez (A01700318)
Tecnológico de Monterrey, Campus Querétaro
A01700318@itesm.mx

16 de septiembre de 2018

Resumen

Este documento se explicara de manera puntual la implementación del algoritmo de cálculo de PI usando el método de integración haciendo uso de paradigmas Single-Thread y Multi-Thread implementados en Java, que posteriormente serán analizados y evaluados con respecto a su velocidad.

1. Introducción

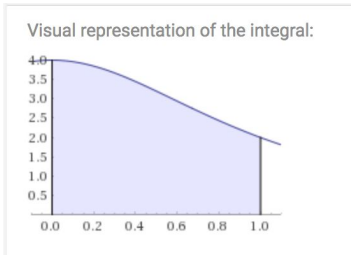
π (**pi**) es la relación entre la longitud de una circunferencia y su diámetro en geometría euclidiana. Es un número irracional y una de las constantes matemáticas más importantes. Se emplea frecuentemente en matemáticas, física e ingeniería.

El método de cálculo por integración de pi, está basado en la siguiente fórmula:

Definite integral:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi \approx 3.1416$$

Para calcular una aproximacion del area debajo de la curva de la función (Figura 1) , es necesario calcular el área de los rectángulos que conforman dicha curva, entre mayor sea la cantidad de rectángulos, la diferencia será menor y nos podrá otorgar un resultado más acertado (Figura 2)



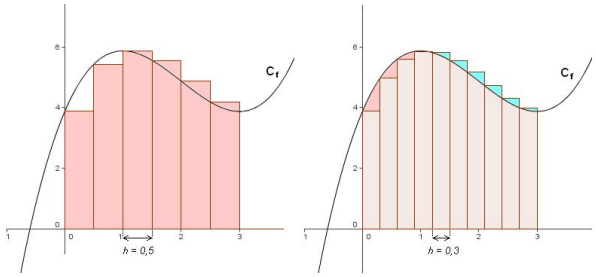


Figura 1 Figura 2

Ahora bien, dado este algoritmo, es posible realizarse de manera Single-Thread o Multi-Thread, para ello haremos uso de la clase Threads existente en Java para posteriormente explicar y comparar ambas implementaciones con fin de exponer los puntos importantes de cada uno con respecto a su contraparte.

2. Implementación

La implementación fue realizada en una computadora con las siguientes características:

- MacBook Pro (Retina, 13-inch, Early 2015)
  - macOS High Sierra 10.13.6 (17G65)
  - Processor Name: Intel Core i5
  - Processor Speed: 2.7 GHz
  - Number of Processors: 1
  - Total Number of Cores: 2 with hyperthreading
  - L2 Cache (per Core): 256 KB
  - L3 Cache: 3 MB
  - Memory: 8 GB 1867 MHz DDR3
  - javac -version: javac 9
  - java -version:
    - java version "9"
    - Java(TM) SE Runtime Environment (build 9+181)
    - Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode)

Versión Single-Thread

En esta primera implementación vemos todo el cálculo implementado en una sola clase Single.

El algoritmo va de lo siguiente, una vez determinado el número de rectángulos en los cuales de planea calcular el área debajo de la curva, es necesario calcular el ancho de los rectángulos con respecto al intervalo de integración [0,1] y el número de los mismos, y una vez con ello evaluar con respecto a la función para encontrar la altura que iremos sumando y al final usaremos para encontrar el área.

Es importante recalcar que la parte fundamental, es el ciclo iterativo que va desde el cero hasta el número de rectángulos deseado.

```
public class Single {
    public static void main(String[] args) {
        double num rect=0;
        if (args.length==0){
            num_rect= 100000;
        }else if (args.length==1){
            try {
            num_rect=Double.parseDouble(args[0]);
            }
            catch (Exception e) {
                System.err.println("Error: usage: java Single
[iterations]");
                    System.exit(-1);
                }
        }else{
                System.err.println("Error: usage: java Single
[iterations]");
            System.exit(-1);
        }
```

```
        double mid, height,width, area;
        double sum  = 0;
        width = 1.0/num_rect;

        for (double i = 0; i < num_rect ; i++) {
            mid=(i+.5)*width;
            height = 4.0 / (1.0 + mid* mid);
            double a=sum;
            sum= height;
        }
        area = width * sum;
        System.out.println("\n\nThreads = 1");
        System.out.println("Iterations = " + num_rect);
        System.out.println("PI = " + area"\n\n");
        System.exit(0);
    }
}
```

Versión Multi-Thread

En la versión multi-thread existen los siguientes puntos claves dentro de la implementación:

El algoritmo sigue siendo el mismo, sin embargo, es posible implementar la técnica de divide y conquista dentro de la iteración principal [0, SIZE], permitiendo segmentar dicho rango entre el número de threads implementados (SIZE/#THREADS).

Esta implementación, consta de dos clases:

- *PiThread*: Una extensión de la superclase de Java Thread, encargada de realizar las iteraciones dado un intervalo y devolver la suma de las de las alturas de los rectángulos.
- *Main*: Donde se define el número de iteraciones, es creado y ejecutado el pool de PiThreads y finalmente recuperamos el resultado final de las ejecuciones.

```
public class Main {
     public static void main(String[] args) {
int MAXTHREADS = Runtime.getRuntime().availableProcessors();
         double SIZE=0;

        if (args.length==0){
            SIZE= 100000;
                                                MAXTHREADS    =
Runtime.getRuntime().availableProcessors();
        }else if (args.length==2){
            try {
                SIZE=Double.parseDouble(args[0]);
                int cores=Integer.parseInt(args[1]);
                    if (SIZE<MAXTHREADS){
                                System.err.println("Error: for a
propper execution the iteration number must be a bigger than
of the posible Processors -> ("+MAXTHREADS+")");
```

```
                        System.exit(-1);
                    }
                    if (cores<=MAXTHREADS){
                        MAXTHREADS=cores;

                    }else{
                            System.err.println("Error: you are
not allowed to create more than -> ("+MAXTHREADS+") due to
your amount of Processors");
                        System.exit(-1);
                    }
                } catch (Exception e) {
                       System.err.println("Error: usage: java
Main [iterations] [cores]");
                    System.err.println(e.toString());
                    System.exit(-1);

                }

        }
        else{
                 System.err.println("Error: usage: java Main
[iterations] [threads]");
            System.exit(-1);
        }
         PiThread [] hilos = new PiThread[MAXTHREADS];
        int a=(int)(SIZE/MAXTHREADS);

        for (int i = 0; i < MAXTHREADS; i++) {
```

```
                if (i != MAXTHREADS - 1) {
                                hilos[i]= new PiThread(SIZE,
(a*i),(a*(i+1)));
                } else {
                        hilos[i] = new PiThread(SIZE, (i * a),
SIZE);
                }
        }
        for (int i = 0; i < MAXTHREADS ; i++) {
            hilos[i].start();
        }
        try {
            for (int i = 0; i < MAXTHREADS ; i++) {
                hilos[i].join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        double s=0;
        for (int i = 0; i < MAXTHREADS; i++) {
            s=s+hilos[i].getSuma();
        }
        System.out.println("\n\nThreads = "+MAXTHREADS);
        System.out.println("Iterations = " + SIZE);
        System.out.println("PI = " + (s)+"\n\n" );
        System.exit(0);
    }
}
```

```
class PiThread extends Thread {
     double N;
     int start;
     int stop;
     double suma;
    public PiThread( double N, double start, double stop ) {
        super( "Calculo:"+start+"-"+stop ); // give a name to
the thread
        this.N      = N;
        this.start  = (int)start;
        this.stop   = (int)stop;
    }
    public double getSuma() {
        return suma;
    }
    public void run() {
        double mid, height,width;
        double sum  = 0;
        width = 1.0/(N);
        for (double i = start; i < stop ; i++) {
            mid=(i+.5)*width;
            height = 4.0 / (1.0 + mid* mid);
            sum+= height;
        }
        this.suma = width*sum;
    }

}
```

3. Resultados.

Nuestro caso inicial está basado en una ejecución de 100000 iteraciones,tanto single-thread como multi-thread (4), y haremos uso del comando `time` de unix para medir el tiempo.

Single-Thread 100000		Multi-Thread (4) 100000	
real	0,222	real	0,265
real	0,228	real	0,278
real	0,234	real	0,253
real	0,233	real	0,261
real	0,236	real	0,258
real	0,241	real	0,258
real	0,237	real	0,253
real	0,247	real	0,258
real	0,252	real	0,261
real	0,251	real	0,254
average	0,2381	average	0,2599
	AverageSpeedUp	0,9161215852	

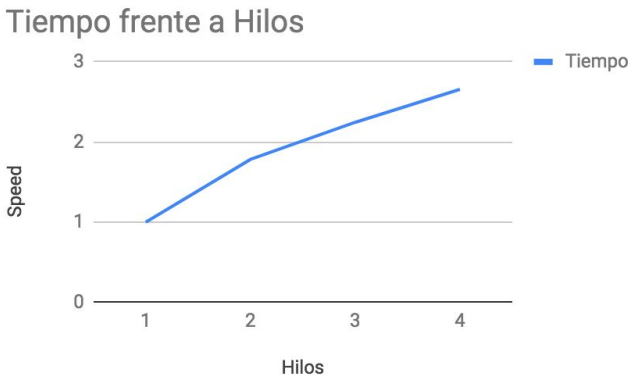
La evaluación promedio de 10 ejecuciones del caso arrojó el siguiente speedup:

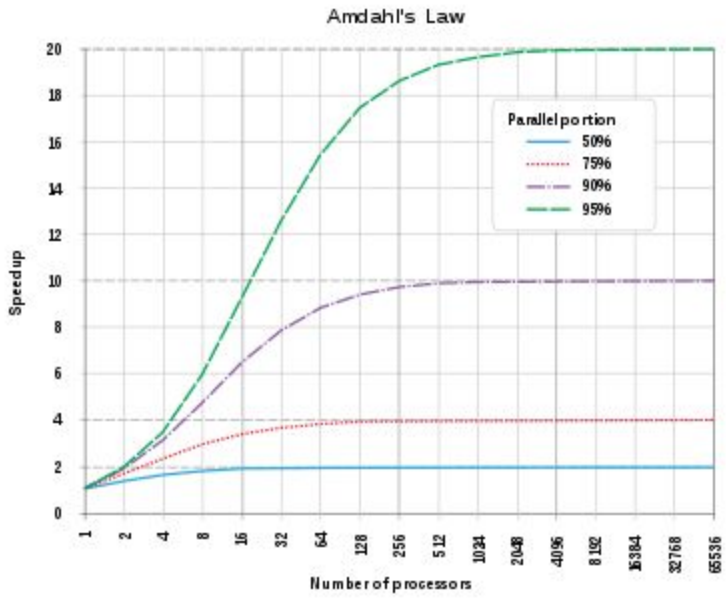
$S = \frac{T_s}{T_m}$ $S = \frac{0{,}2381\ s}{0{,}2599\ s}$ $S = .9161215852$

Es evidente que no ha existido mejora al ejecutar paralelismo con multi threads, esto debido a que el número de iteraciones no es significativamente grande es por ello que el proceso de creación de los hilos no llega a compensarse ante la ejecución de los mismos.

Ahora bien, teniendo eso en consideración y para poder observar verdaderamente las capacidades del cómputo en paralelo se ha decidido incrementar considerablemente el número de iteraciones y evaluar los tiempos.

MacBook-Pro-de-Ramon-2:multi-thread ramonromero$ time java Main 1000000000 1				
Threads	1			
Iterations	1.0E9			
PI	3,141592654		SpeedUp	1
real	0m3.811s			
user	0m3.883s			
sys	0m0.061s			
MacBook-Pro-de-Ramon-2:multi-thread ramonromero$ time java Main 1000000000 2				
Threads	2			
Iterations	1.0E9			
PI	3,141592654		SpeedUp	1,783341132
real	0m2.137s			
user	0m4.041s			
sys	0m0.063s			
MacBook-Pro-de-Ramon-2:multi-thread ramonromero$ time java Main 1000000000 3				
Threads	3			
Iterations	1.0E9			
PI	3,141592654		SpeedUp	2,245727755
real	0m1.697s			
user	0m4.425s			
sys	0m0.056s			
MacBook-Pro-de-Ramon-2:multi-thread ramonromero$ time java Main 1000000000 4				
Threads	4			
Iterations	1.0E9			
PI	3,141592654		SpeedUp	2,657601116
real	0m1.434s			
user	0m4.882s			
sys	0m0.057s			





Haciendo uso de la Ley de Amdahl, es posible observar que la porción de paralelización del es superior al 75 % del programa pero menor que del 90%.

Ahora es evidente la mejora que existe, ya que es fácil observar como la ejecución de la tarea de los hilos logra compensar su tiempo de creación. *Sin embargo, siempre hay que tener en cuenta la cantidad apropiada de threads, de acuerdo a la cantidad de cores de nuestro equipo, además de los tipos y la longitud de los datos que ocupamos para iterar.*

3. Referencias

[1]"Pi", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Pi. [Accessed: 17- Sep- 2018].

[2] [BRESHEARS] pp. 31 & 32.

[3]"Amdahl's law", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed: 15- Sep- 2018].

[4]"Computing Pi in Parallel with Java", Science.smith.edu, 2018. [Online]. Available: http://www.science.smith.edu/dftwiki/index.php/CSC352:_Computing_Pi_in_Parallel_with_Java. [Accessed: 15- Sep- 2018].

[5]"Wolfram|Alpha", M.wolframalpha.com, 2018. [Online]. Available: http://m.wolframalpha.com/. [Accessed: 15- Sep- 2018].

[6]"JDK 10 Documentation", Docs.oracle.com, 2018. [Online]. Available: https://docs.oracle.com/javase/10/. [Accessed: 15- Sep- 2018].