

**LABORATORIO 1 – INTRODUCCION A PARALELISMO - HILOS**

**SANTIAGO OSPINA  
ISABELLA MANRIQUE**

**ARQUITECTURAS DE SOFTWARE**

**DIEGO ANDRES TRIVIÑO GONZALEZ**

**ESCUELA COLOMBIANA DE INGENIERIA JULIO GARAVITO  
INGENIERIA DE SISTEMAS  
2023-1  
BOGOTA D.C**

## **INTRODUCCION**

Mediante las siguientes prácticas de laboratorio, veremos los primeros conceptos y aplicaciones del paralelismo haciendo uso de hilos en diferentes ejemplos prácticos.

## PRACTICA

### PARTE 1 – HILOS JAVA

1. De acuerdo con lo revisado en las lecturas, complete las clases CountThread, para que las mismas definan el ciclo de vida de un hilo que imprima por pantalla los números entre A y B.

```
public class CountThread extends Thread{

    2 usages
    int a,b;
    ± unknown
    @Override
    public void run() {
        for (int i = getA()+1; i < getB(); i++ ) {
            System.out.println(i);
        }
    }
}
```

2. Complete el método main de la clase CountMainThreads para que:
  - a. Cree 3 hilos de tipo CountThread, asignándole al primero el intervalo[0..99], al segundo [99..199], y al tercero [200..299].

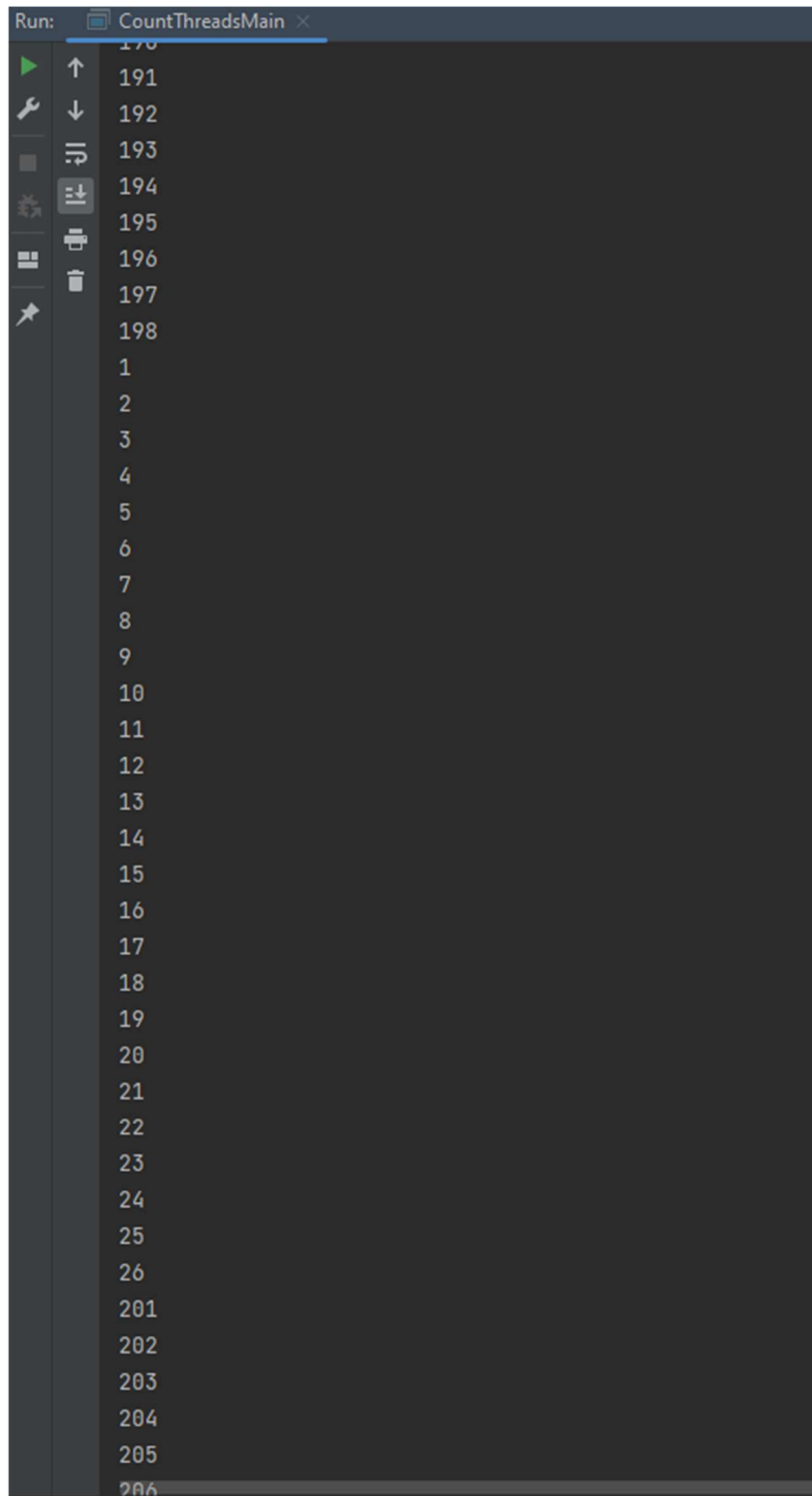
```
public class CountThreadsMain {

    ± unknown +1
    public static void main(String a[]){
        CountThread t1 = new CountThread();
        t1.setAB( a: 0, b: 99);
        CountThread t2 = new CountThread();
        t2.setAB( a: 100, b: 199);
        CountThread t3 = new CountThread();
        t3.setAB( a: 200, b: 299);
    }
}
```

- b. Inicie los tres hilos con 'start()'.

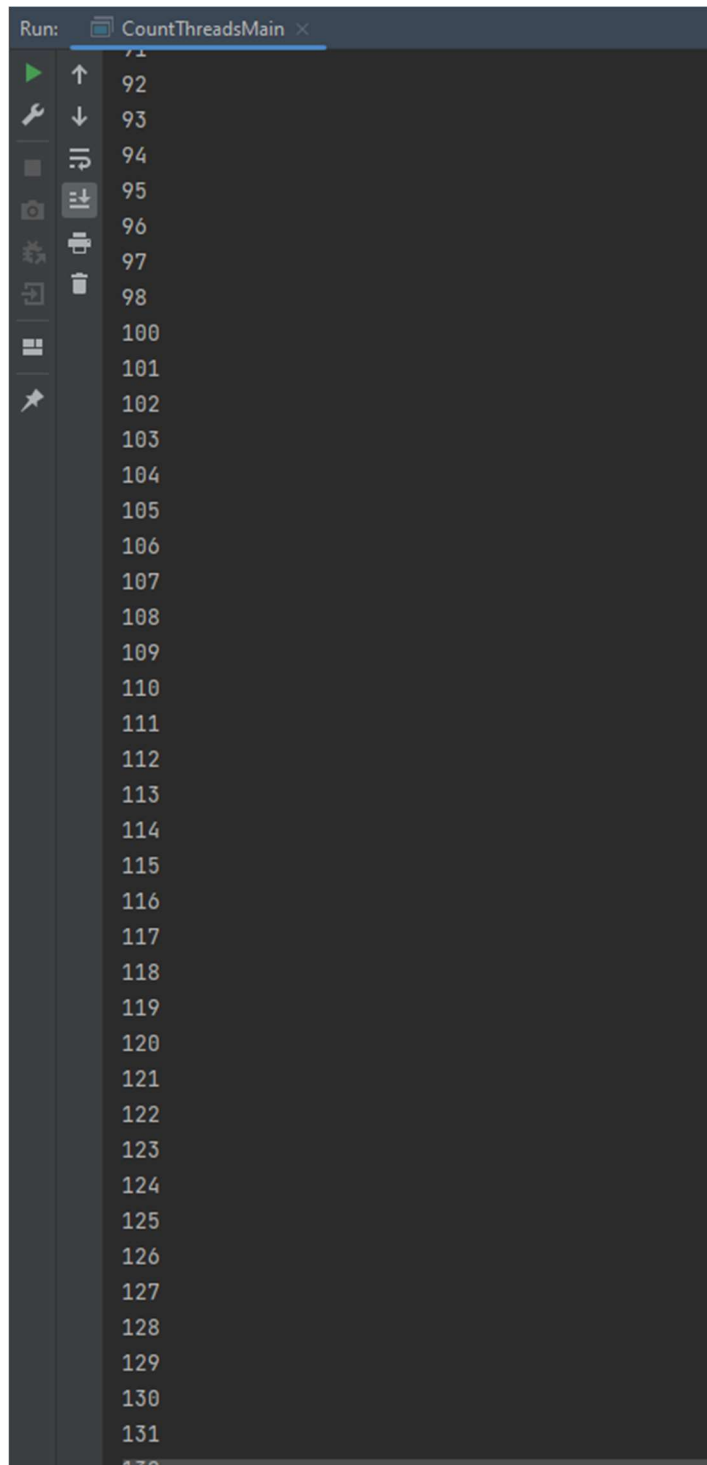
```
t3.start();
t1.start();
t2.start();
}
```

c. Ejecute y revise la salida por pantalla.



```
Run: CountThreadsMain x
190
191
192
193
194
195
196
197
198
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
201
202
203
204
205
206
```

- d. Cambie el inicio con 'start()' por 'run()'. Cómo cambia la salida?, por qué?.



The screenshot shows a dark-themed IDE window titled 'Run: CountThreadsMain'. On the left is a vertical toolbar with icons for running, debugging, and other actions. The main area of the window displays a list of numbers from 92 to 131, arranged in a single column. This indicates that the threads are executing sequentially, as they would if the `run()` method was used instead of `start()`.

Reemplazando el `start()` por un `run()`, vemos como con el `run()` los hilos parecen estar ejecutándose de manera secuencial mientras que con el `start` parecen ejecutarse al tiempo.

## PARTE 2 – HILOS JAVA

Para este ejercicio se quiere calcular, en el menor tiempo posible, y en una sola máquina (aprovechando las características multi-core de la mismas) al menos el primer millón de dígitos de PI (en base 16). Para esto:

1. Cree una clase de tipo Thread que represente el ciclo de vida de un hilo que calcule una parte de los dígitos requeridos.

```
public PiThread(int start, int count){
    this.start = start;
    this.count = count;
}

± unknown +1
@Override
public void run() {
    result = getDigitsPi(start, count);
}
```

Creamos la clase PiThread al que se le pasaban inicio y cantidad de dígitos a calcular y nos apoyamos en el método getDigits para crear getDigitsPi para calcular los dígitos requeridos por el usuario.

2. Haga que la función PiDigits.getDigits() reciba como parámetro adicional un valor N, correspondiente al número de hilos entre los que se va a paralelizar la solución. Haga que dicha función espere hasta que los N hilos terminen de resolver el problema para combinar las respuestas y entonces retornar el resultado. Para esto, revise el método join del API de concurrencia de Java.

```
public class PiDigits {

    4 usages ± isaeme23
    public static byte[] getDigits (int start, int amount, int n) throws InterruptedException {
        ArrayList<PiThread> threads = createThreads(start, amount, n);
        joinThreads(threads);
        return concatenateResults(threads, amount);
    }
}
```

Para esto, modificamos getDigits para que recibiera el parámetro de cuantos hilos se usarían para resolver el problema. Este método llama a otros 3:

- El primero de ellos crea los threads, los almacena en un ArrayList, establece el rango que necesita para cada uno junto con la posición desde la que cada uno de ellos necesita empezar a calcular los dígitos de pi y los inicializa con el método start()
- El segundo, recorre el ArrayList de hilos para usar el metodo join()

- El tercero, concatena los resultados de todos los hilos para mostrarlos en una sola línea y como un único resultado.

3. Ajuste las pruebas de JUnit, considerando los casos de usar 1, 2 o 3 hilos (este último para considerar un número impar de hilos!)

a. Prueba para uno, dos y tres hilos:

```

@Test
public void piGenTestOneThread() throws Exception {

    byte[] expected = new byte[] {
        0x2, 0x4, 0x3, 0xF, 0x6, 0x8, 0xA, 0x8, 0x8, 0x8,
        0x8, 0x5, 0xA, 0x3, 0x8, 0x8, 0x8, 0xD, 0x3,
        0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xF,
        0x0, 0x3, 0x7, 0x8, 0x7, 0x3, 0x4, 0x4,
        0xA, 0x4, 0x0, 0x9, 0x3, 0x8, 0x2, 0x2,
        0x2, 0x9, 0x9, 0xF, 0x3, 0x1, 0xD, 0x0,
        0x0, 0x8, 0x2, 0xE, 0xF, 0xA, 0x9, 0x8,
        0xE, 0xC, 0x4, 0x6, 0x8, 0xC, 0x8, 0x9,
        0x4, 0x5, 0x2, 0x8, 0x2, 0x1, 0xE, 0x6,
        0x3, 0x8, 0xD, 0x8, 0x1, 0x3, 0x7, 0x7, };

    for (int start = 0; start < expected.length; start++)
        for (int count = 0; count < expected.length - start; count++)
            byte[] digits = PDigits.getPDigits(start, count);
            assertEquals(count, digits.length);

            for (int i = 0; i < digits.length; i++)
                assertEquals(expected[start + i], digits[i]);
    }
}

```

```
@Test
public void piGenTestTwoThreads() throws Exception {

    byte[] expected = new byte[]{{
        0x2, 0x4, 0x3, 0xF, 0x6, 0xA, 0x8, 0x8,
        0x8, 0x5, 0xA, 0x3, 0x0, 0x8, 0xD, 0x3,
        0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xE,
        0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
        0xA, 0x4, 0x0, 0x9, 0x3, 0x8, 0x2, 0x2,
        0x2, 0x9, 0x9, 0xF, 0x3, 0x1, 0xD, 0x0,
        0x0, 0x8, 0x2, 0xE, 0xF, 0xA, 0x9, 0x8,
        0xE, 0xC, 0x4, 0xE, 0x6, 0xC, 0x8, 0x9,
        0x4, 0x5, 0x2, 0x0, 0x2, 0x1, 0xE, 0x6,
        0x3, 0x8, 0xD, 0x0, 0x1, 0x3, 0x7, 0x7,;
    }};

    for (int start = 0; start < expected.length; start++) {
        for (int count = 0; count < expected.length - start; count++) {
            byte[] digits = PiDigits.getPigits(start, count, m/2);
            assertEquals(count, digits.length);

            for (int i = 0; i < digits.length; i++) {
                assertEquals(expected[start + i], digits[i]);
            }
        }
    }
}
```

b. Resultado de las tres pruebas:

Run: **PiCalcTest** ×

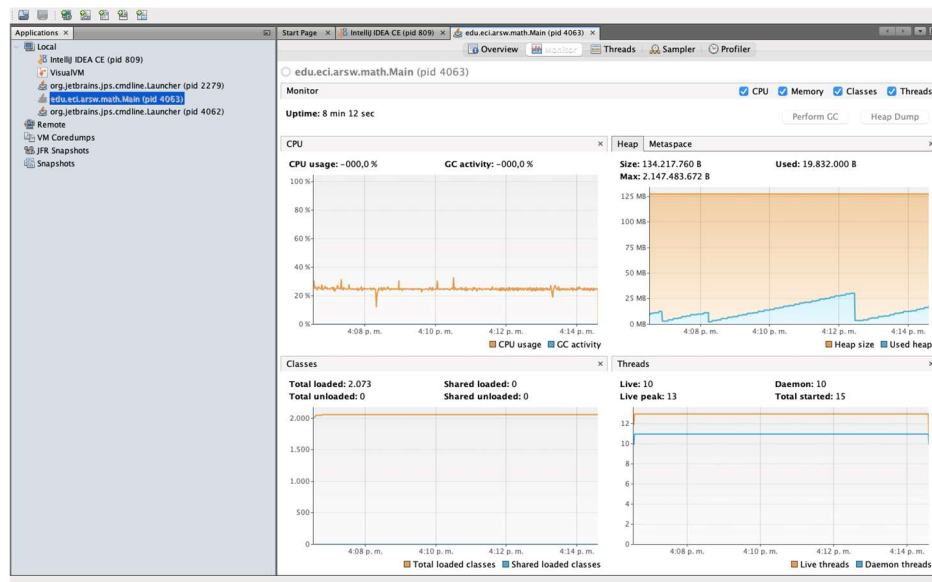
Test Name	Duration
<b>PiCalcTest (edu.eci.arsw.math)</b>	<b>4sec 453 ms</b>
✓ piGenTestOneThread	1sec 504 ms
✓ piGenTestTwoThreads	1sec 463 ms
✓ piGenTestThreeThreads	1sec 486 ms

## PARTE 3 – EVALUACION DE DESEMPEÑO

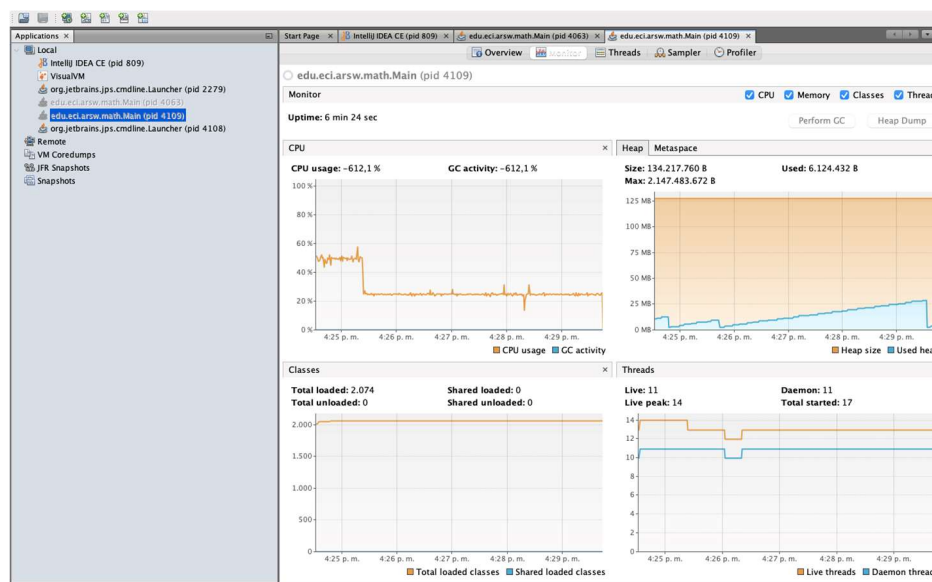
A partir de lo anterior, implemente la siguiente secuencia de experimentos para calcular el millón de dígitos (hex) de PI, tomando los tiempos de ejecución de los mismos (asegúrese de hacerlos en la misma máquina):

*(Las siguientes evaluaciones de desempeño se realizaron con 100.000 dígitos ya que por limitaciones de nuestras maquinas no se pudo más :())*

### 1. Un hilo



### 2. Dos hilos

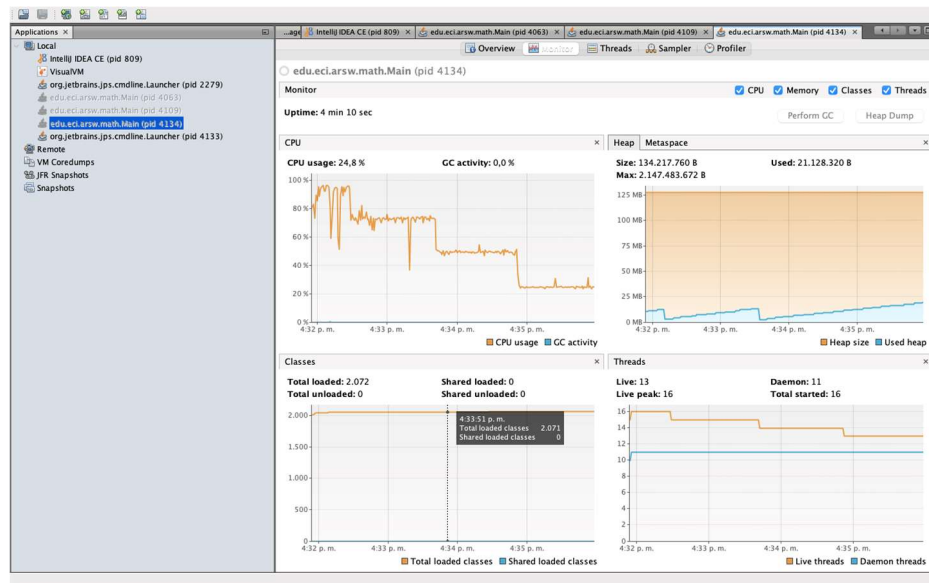




3. Tantos hilos como núcleos de procesamiento (haga que el programa determine esto haciendo uso del API Runtime)

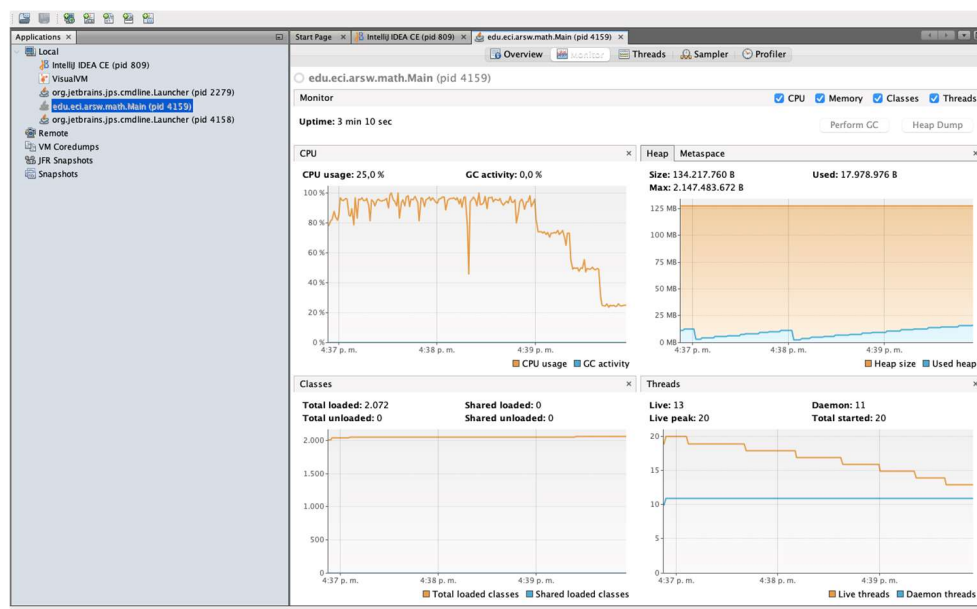
```
public static void main(String a[]) throws InterruptedException {  
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0, amount: 100000, Runtime.getRuntime().availableProcessors())));  
}
```

Usamos el API Runtime con el método `getRuntime().availableProcessors()`

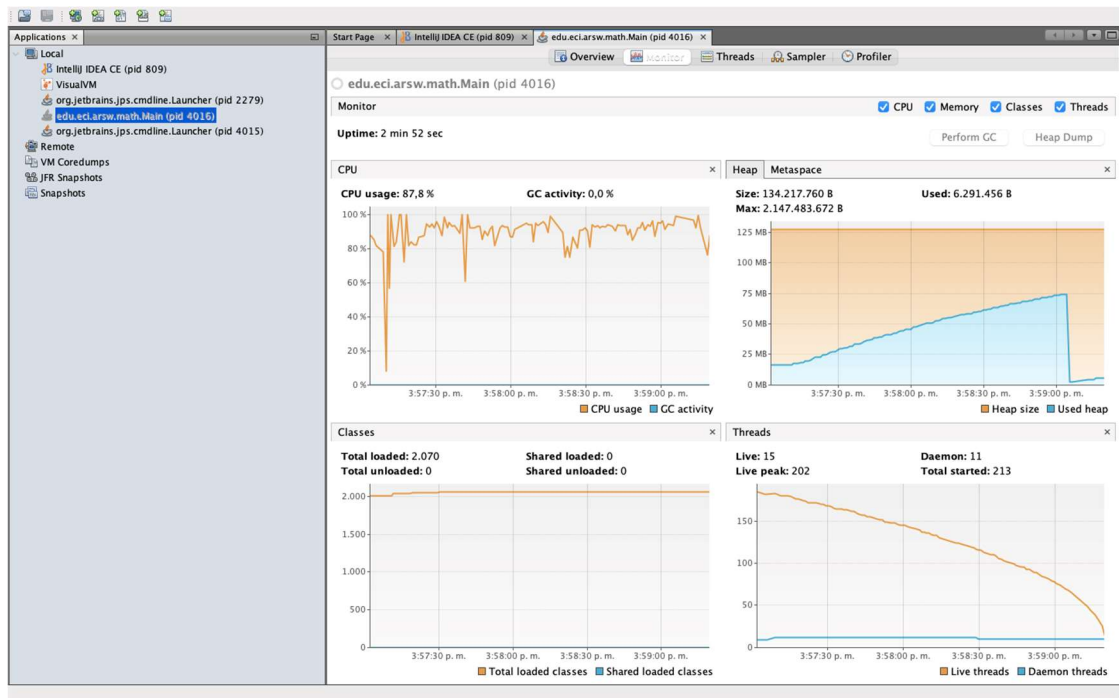


4. Tantos hilos como el doble de núcleos de procesamiento.

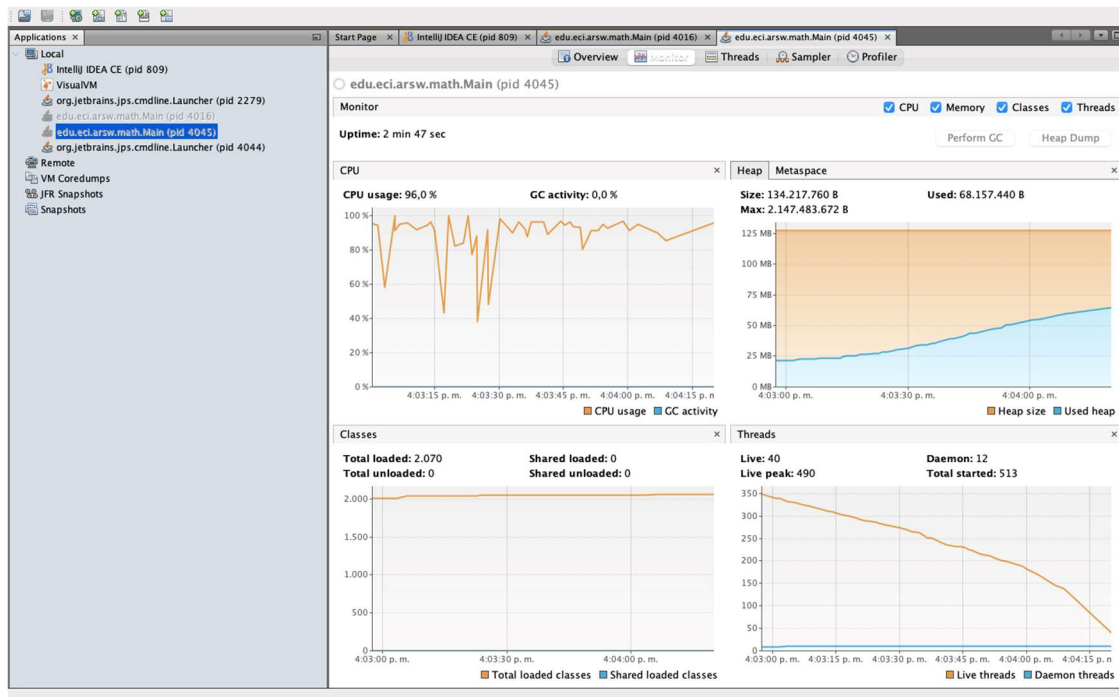
```
public static void main(String a[]) throws InterruptedException {  
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0, amount: 100000, n: Runtime.getRuntime().availableProcessors()*2)));  
}
```



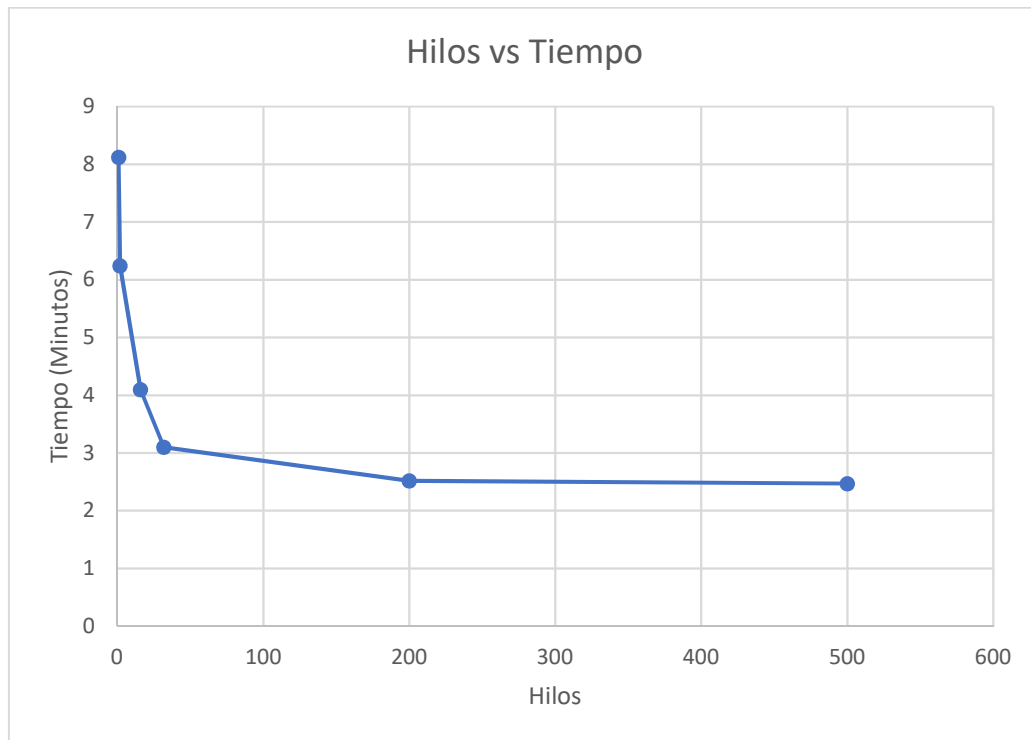
## 5. 200 hilos



## 6. 500 hilos



Grafica:



**Hipótesis:** A mayor cantidad de hilos se ira disminuyendo el tiempo de ejecución hasta llegar a un límite, que en este caso se puede ver que tiende a 2 minutos cuando el numero de hilos se aproxima a infinito

1. Se puede observar que  $n$  está en el denominador, y cuando vamos aumentando el valor vemos que en la función  $p/n$  tiende a 0 por lo que con valores muy grandes el denominador queda  $(1-P)$ , por lo que podemos afirmar que llegado un punto de hilos, no importa cuanto lo aumentemos no mejorara de manera que tenga algún efecto, por lo que pasa es que dado cierto hilos, estos pierden su importancia y ya no tienen el mismo efecto significativo en el aumento de la eficacia del algoritmo.
2. Se puede observar que hay una diferencia de tiempo de ejecución significativa, esta mejora podemos pensar que sucede por qué estamos aprovechando al máximo los recursos del computador, ya que estamos doblando el número de hilos, lo que hace que se produzca una mejora significativa y no perjudica de gran manera los recursos del computador.
3. Al tener 500 computadores cada uno corriendo un hilo, se aplicaría mejor la ley de Amdahls porque en este caso si tendríamos paralelismo real y una maquina solo tiene que correr un hilo, mientras que corriéndolos todos en una misma máquina, esta estaría obligaría a correr todos los hilos por si sola, lo que gasta recursos de la maquina y tiene un límite.