

Laboratorio Hilos y Paralelismo 01

Sebastián Cardona, Laura Gil, Zayra Gutiérrez

Ingeniero de Sistemas

Javier Toquica Barrero

Universidad Escuela Colombiana de ingeniería Julio Garavito

2025-1

Contenido

Introducción	3
Desarrollo del Laboratorio	4
Parte I Hilos Java	4
Parte II Hilos Java	6
Parte III Evaluación de Desempeño	11
Conclusiones	23

Introducción

El informe tiene como objetivo analizar y explorar el uso de hilos, paralelismo y concurrencia en Java, enfocándonos en la solución de problemas que requiere un alto rendimiento computacional. Se abordarán tres secciones importantes: la creación y manejo de hilos básicos, la implementación de algoritmos paralelos mediante el uso de hilos y la evaluación del desempeño del paralelismo en diferentes escenarios.

Se analizará el desempeño de las soluciones desarrolladas mediante la variación del número hilos utilizados, contrastando los resultados con la ley Amdahl para evaluar los límites del paralelismo en un entorno controlado.

Desarrollo del Laboratorio

Parte I 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.

Se creo la clase CounThread que extiende de Thread, tiene un constructor que recibe los números a y b y luego en el método run muestra los números de a hasta b, incluyendo el nombre del hilo actual.

```
package edu.eci.arsw.threads;

public class CountThread extends Thread {

    private int a;
    private int b;

    public CountThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        for (int i = a; i <= b; i++) {
            System.out.println(threadName + " " + i);
        }
    }

    public int getA() {
        return a;
    }

    public int getB() {
        return b;
    }
}
```

2. Complete el método **main** de la clase *CountMainThreads* para que:

- i. Se creo 3 hilos de tipo *CountThread*, asignándole al primero el intervalo [0..99], al segundo [99..199], y al tercero [200..299].

```
package edu.eci.arsw.threads;

public class CountThreadsMain {

    public static void main(String a[]) {
        CountThread hilo1 = new CountThread(a:0, b:99);
        CountThread hilo2 = new CountThread(a:99, b:199);
        CountThread hilo3 = new CountThread(a:200, b:299);
    }
}
```

- ii. Se inicio los tres hilos con 'start()'.

```
package edu.eci.arsw.threads;

public class CountThreadsMain {

    public static void main(String a[]) {
        CountThread hilo1 = new CountThread(a:0, b:99);
        CountThread hilo2 = new CountThread(a:99, b:199);
        CountThread hilo3 = new CountThread(a:200, b:299);

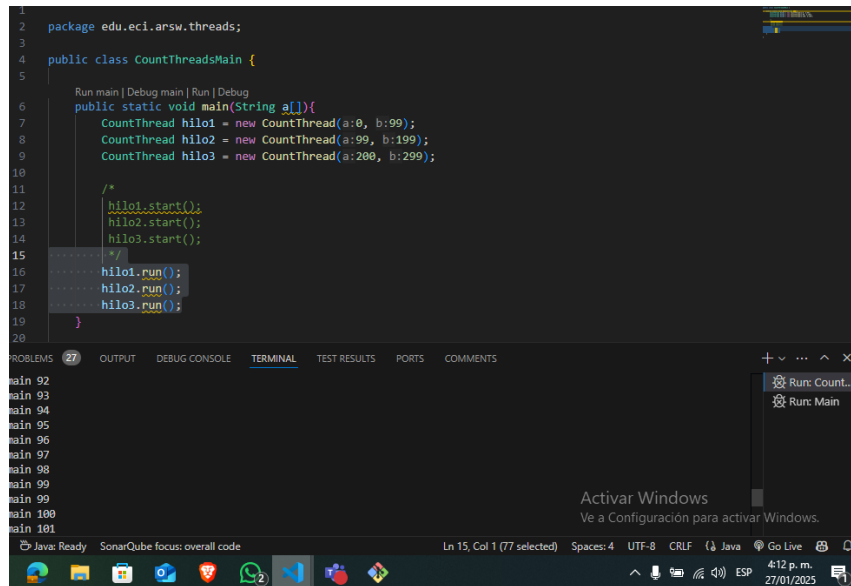
        hilo1.start();
        hilo2.start();
        hilo3.start();
    }
}
```

iii. Se ejecuto y reviso la salida por pantalla.

Se logra evidenciar como los tres hilos ocurren de manera concurrente y se imprimen en pantalla los rangos de los tres hilos a la vez

```
Thread-0 0
Thread-0 1
Thread-2 200
Thread-0 2
Thread-2 201
Thread-0 3
Thread-2 202
Thread-2 203
Thread-0 4
Thread-2 204
Thread-0 5
Thread-2 205
Thread-2 206
Thread-0 6
Thread-2 207
Thread-0 7
Thread-2 208
Thread-0 8
Thread-2 209
Thread-0 9
Thread-2 210
Thread-0 10
Thread-2 211
Thread-0 11
Thread-2 212
Thread-0 12
Thread-2 213
Thread-0 13
Thread-1 99
Thread-2 214
Thread-2 215
Thread-2 216
```

iv. Se cambio el inicio con 'start()' por 'run()'. ¿Cómo cambia la salida?, por qué?



```

1 package edu.eci.arsw.threads;
2
3
4 public class CountThreadMain {
5
6     Run main | Debug main | Run | Debug
7     public static void main(String a[]){
8         CountThread hilo1 = new CountThread(a:0, b:99);
9         CountThread hilo2 = new CountThread(a:99, b:199);
10        CountThread hilo3 = new CountThread(a:200, b:299);
11
12        /*
13        hilo1.start();
14        hilo2.start();
15        hilo3.start();
16        */
17        hilo1.run();
18        hilo2.run();
19        hilo3.run();
20    }
21
22 }

```

main 92
main 93
main 94
main 95
main 96
main 97
main 98
main 99
main 99
main 100
main 101

Con el método `start()`, se crea un nuevo hilo en ejecución. Este método realiza una llamada interna al método `run()`, permitiendo que la tarea se ejecute de forma paralela al hilo principal u otros hilos existentes.

Por otro lado, al llamar directamente al método `run()`, no se crea un nuevo hilo. En su lugar, la ejecución ocurre en el hilo actual, de forma secuencial, como cualquier otro método común de Java.

Parte II Hilos Java

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

La clase `MiniPiDigits` se encargará de contener la lógica de calcular los count dígitos de pi desde start, solo que esta vez esta clase hereda de `Thread`

```
package edu.eci.arsw.math;

public class MiniPiDigits extends Thread {
    private static int DigitsPerSum = 8;
    private static double Epsilon = 1e-17;
    private int start;
    private int count;
    private byte[] digits;

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

    /**
     * Returns a range of hexadecimal digits of pi.
     * @param start The starting location of the range.
     * @param count The number of digits to return
     * @return An array containing the hexadecimal digits.
     */
    @Override
    public void run() {
        digits = new byte[count];
        double sum = 0;

        for (int i = 0; i < count; i++) {
            if (i % DigitsPerSum == 0) {
                sum = 4 * sum(m:1, start)
                    - 2 * sum(m:4, start)
                    - sum(m:5, start)
                    - sum(m:6, start);
            }
        }
    }
}
```

```
public class MiniPiDigits extends Thread {
    public void run() {
        sum(m:5, start)
        sum(m:6, start);

        start += DigitsPerSum;
    }

    sum = 16 * (sum - Math.floor(sum));
    digits[i] = (byte) sum;
}

/** <summary>
 * Returns the sum of 16^(n - k)/(8 * k + m) from 0 to k.
 * </summary>
 * @param name="m"></param>
 * @param name="n"></param>
 * @returns</returns>
 */
private static double sum(int m, int n) {
    double sum = 0;
    int d = m;
    int power = n;

    while (true) {
        double term;

        if (power > 0) {
            term = (double) hexExponentModulo(power, d) / d;
        } else {
            term = Math.pow(a:16, power) / d;
        }
        if (term < Epsilon) {
            break;
        }
        sum += term;
        power--;
        d += 8;
    }

    return sum;
}

/** <summary>
 * Return 16^p mod m.
 * </summary>
 * @param name="p"></param>
 * @param name="m"></param>
 * @returns</returns>
 */
private static int hexExponentModulo(int p, int m) {
    int power = 1;
    while (power * 2 <= p) {
        power *= 2;
    }

    int result = 1;

    while (power > 0) {
        if (p >= power) {
            result *= 16;
        }
        power /= 2;
    }

    return result;
}
```

```
src > main > java > edu > arsw > math > MiniPiDigits.java > Language Support for Java(TM) by Red Hat > MiniPiDigits > DigitsPerSum

public class MiniPiDigits extends Thread {
    private static double sum(int m, int n) {
        term = Math.pow(a:16, power) / d;
        if (term < Epsilon) {
            break;
        }
        sum += term;
        power--;
        d += 8;
    }

    return sum;
}

/** <summary>
 * Return 16^p mod m.
 * </summary>
 * @param name="p"></param>
 * @param name="m"></param>
 * @returns</returns>
 */
private static int hexExponentModulo(int p, int m) {
    int power = 1;
    while (power * 2 <= p) {
        power *= 2;
    }

    int result = 1;

    while (power > 0) {
        if (p >= power) {
            result *= 16;
        }
        power /= 2;
    }

    return result;
}
```

```

87 while (power > 0) {
88     if (p >= power) {
89         result *= 16;
90         result %= m;
91         p -= power;
92     }
93
94     power /= 2;
95
96     if (power > 0) {
97         result *= result;
98         result %= m;
99     }
100 }
101
102 return result;
103 }
104
105 /**
106  * Returns the hexadecimal digits.
107  */
108 public byte[] getDigits() {
109     return digits;
110 }
111
112
113

```

- 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.

En el método `getDigits()` efectivamente se agrega el parámetro `N` para primero dividir el rango de dígitos a calcular por cada hilo, y en el caso de hacer una división no exacta, guardar los números del rango que faltan, luego se crea una lista de los `N` hilos, donde se establece el inicio de cada hilo y su rango.

Después, todos los hilos pasan a ejecución con el método `start()` de cada hilo y antes de juntar la respuesta, usamos el método `join` para esperar que todos los hilos se ejecutaran y unirlos con `arrayCopy`.

```

12 public class PiDigits {
13
14     /**
15      * Returns a range of hexadecimal digits of pi using multiple threads.
16      * @param start The starting location of the range.
17      * @param count The number of digits to return.
18      * @param N The number of threads to use.
19      * @return An array containing the hexadecimal digits.
20      */
21     public static byte[] getDigits(int start, int count, int N) {
22         if (start < 0 || count < 0 || N <= 0) {
23             throw new RuntimeException("Invalid parameters");
24         }
25         // Dividir el total de hilos entre el número de hilos
26         int rangePerThread = count / N;
27         int remainingDigits = count % N; // Tener en cuenta los dígitos restantes
28
29         List<MiniPiDigits> threads = new ArrayList<>(); // Lista de hilos
30         byte[] result = new byte[count]; // lista de resultado en bytes
31
32         // Crear los hilos y agregarlos a la lista
33         for (int i = 0; i < N; i++) {
34             int threadStart = start + i * rangePerThread;
35             int threadCount = rangePerThread + (i == N - 1 ? remainingDigits : 0);
36             MiniPiDigits thread = new MiniPiDigits(threadStart, threadCount);
37             threads.add(thread);
38         }
39
40         // Iniciar los hilos
41         for (MiniPiDigits thread : threads) {
42             thread.start();
43         }

```



```

21 public static byte[] getDigits(int start, int count, int N) {
36     MiniPiDigits thread = new MiniPiDigits(threadStart, threadCount);
37     threads.add(thread);
38 }
39
40 // Iniciar los hilos
41 for (MiniPiDigits thread : threads) {
42     thread.start();
43 }
44
45 // Esperar a que todos los hilos terminen con el .join()
46 for (MiniPiDigits thread : threads) {
47     try {
48         thread.join();
49     } catch (InterruptedException e) {
50         throw new RuntimeException("Thread interrupted", e);
51     }
52 }
53
54 // Combinar los resultados de los hilos en un solo array
55 int pos = 0;
56 for (MiniPiDigits thread : threads) {
57     System.arraycopy(thread.getDigits(), srcPos:0, result, pos, thread.getDigits().length);
58     pos += thread.getDigits().length;
59 }
60
61 return result;
62 }
63
64 }
65

```

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)

Se crearon tres pruebas cada uno con 1, 2 y tres hilos respectivamente, todas las pruebas pasaron exitosamente

```

src > test > java > edu > eci > arsw > math > PiCalcTest.java > Language Support for Java(TM) by Red Hat > PiCalcTest > piGenTest20
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package edu.eci.arsw.math;
7
8  import static org.junit.Assert.assertEquals;
9  import org.junit.Before;
10 import org.junit.Test;
11
12 /**
13  *
14  * @author hcadavid
15  */
16 public class PiCalcTest {
17
18     public PiCalcTest() {
19     }
20
21     @Before
22     public void setUp() {
23     }
24
25     @Test
26     public void piGenTest() throws Exception {
27
28         byte[] expected = new byte[]{
29             0x2, 0x4, 0x3, 0xF, 0x6, 0xA, 0x8, 0x8,
30             0x8, 0x5, 0xA, 0x3, 0x0, 0x8, 0xD, 0x3,
31             0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xE,
32             0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
33         };
34     }
35 }

```

```

31      0x1, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
32      0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
33      0xA, 0x4, 0x0, 0x9, 0x3, 0x8, 0x2, 0x2,
34      0x2, 0x9, 0x9, 0xF, 0x3, 0x1, 0xD, 0x0,
35      0x0, 0x8, 0x2, 0xE, 0xF, 0xA, 0x9, 0x8,
36      0xE, 0xC, 0x4, 0xE, 0x6, 0xC, 0x8, 0x9,
37      0x4, 0x5, 0x2, 0x8, 0x2, 0x1, 0xE, 0x6,
38      0x3, 0x8, 0xD, 0x0, 0x1, 0x3, 0x7, 0x7,};
39
40      int numThreads = 1;
41
42      for (int start = 0; start < expected.length; start++) {
43          for (int count = 0; count < expected.length - start; count++) {
44              byte[] digits = PiDigits.getDigits(start, count, numThreads);
45              assertEquals(count, digits.length);
46
47              for (int i = 0; i < digits.length; i++) {
48                  assertEquals(expected[start + i], digits[i]);
49              }
50          }
51      }
52
53      @Test
54      public void piGenTest1() throws Exception {
55
56          byte[] expected = new byte[]{
57              0x2, 0x4, 0x3, 0xF, 0x6, 0xA, 0x8, 0x8,
58              0x8, 0x5, 0xA, 0x3, 0x0, 0x8, 0xD, 0x3,
59              0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xE,
60              0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
61          };

```

```

61      0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
62      0xA, 0x4, 0x0, 0x9, 0x3, 0x8, 0x2, 0x2,
63      0x2, 0x9, 0x9, 0xF, 0x3, 0x1, 0xD, 0x0,
64      0x0, 0x8, 0x2, 0xE, 0xF, 0xA, 0x9, 0x8,
65      0xE, 0xC, 0x4, 0xE, 0x6, 0xC, 0x8, 0x9,
66      0x4, 0x5, 0x2, 0x8, 0x2, 0x1, 0xE, 0x6,
67      0x3, 0x8, 0xD, 0x0, 0x1, 0x3, 0x7, 0x7,};
68
69      int numThreads = 2;
70
71      for (int start = 0; start < expected.length; start++) {
72          for (int count = 0; count < expected.length - start; count++) {
73              byte[] digits = PiDigits.getDigits(start, count, numThreads);
74              assertEquals(count, digits.length);
75
76              for (int i = 0; i < digits.length; i++) {
77                  assertEquals(expected[start + i], digits[i]);
78              }
79          }
80      }
81
82      @Test
83      public void piGenTest2() throws Exception {
84
85          byte[] expected = new byte[]{
86              0x2, 0x4, 0x3, 0xF, 0x6, 0xA, 0x8, 0x8,
87              0x8, 0x5, 0xA, 0x3, 0x0, 0x8, 0xD, 0x3,
88              0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xE,
89              0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
90          };

```

```

82      @Test
83      public void piGenTest2() throws Exception {
84
85          byte[] expected = new byte[]{
86              0x2, 0x4, 0x3, 0xF, 0x6, 0xA, 0x8, 0x8,
87              0x8, 0x5, 0xA, 0x3, 0x0, 0x8, 0xD, 0x3,
88              0x1, 0x3, 0x1, 0x9, 0x8, 0xA, 0x2, 0xE,
89              0x0, 0x3, 0x7, 0x0, 0x7, 0x3, 0x4, 0x4,
90              0xA, 0x4, 0x0, 0x9, 0x3, 0x8, 0x2, 0x2,
91              0x2, 0x9, 0x9, 0xF, 0x3, 0x1, 0xD, 0x0,
92              0x0, 0x8, 0x2, 0xE, 0xF, 0xA, 0x9, 0x8,
93              0xE, 0xC, 0x4, 0xE, 0x6, 0xC, 0x8, 0x9,
94              0x4, 0x5, 0x2, 0x8, 0x2, 0x1, 0xE, 0x6,
95              0x3, 0x8, 0xD, 0x0, 0x1, 0x3, 0x7, 0x7,};
96
97      int numThreads = 3;
98
99      for (int start = 0; start < expected.length; start++) {
100          for (int count = 0; count < expected.length - start; count++) {
101              byte[] digits = PiDigits.getDigits(start, count, numThreads);
102              assertEquals(count, digits.length);
103
104              for (int i = 0; i < digits.length; i++) {
105                  assertEquals(expected[start + i], digits[i]);
106              }
107          }
108      }
109
110      }
111
112      }

```

```

PROBLEMS 30 OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS COMMENTS Filter (e.g. text, text...) Java Single Debug
sources
--- compiler:3.13.0:testCompile (default-testCompile) @ PiDigits ---
Nothing to compile - all classes are up to date.

--- surefire:3.2.5:test (default-cli) @ PiDigits ---
Using auto detected provider org.apache.maven.surefire.junit4.JUnit4Provider

-----
T E S T S
-----
Running edu.eci.arsw.math.PiCalcTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.569 s -- in edu.eci.arsw.math.PiCalcTest

Results:

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

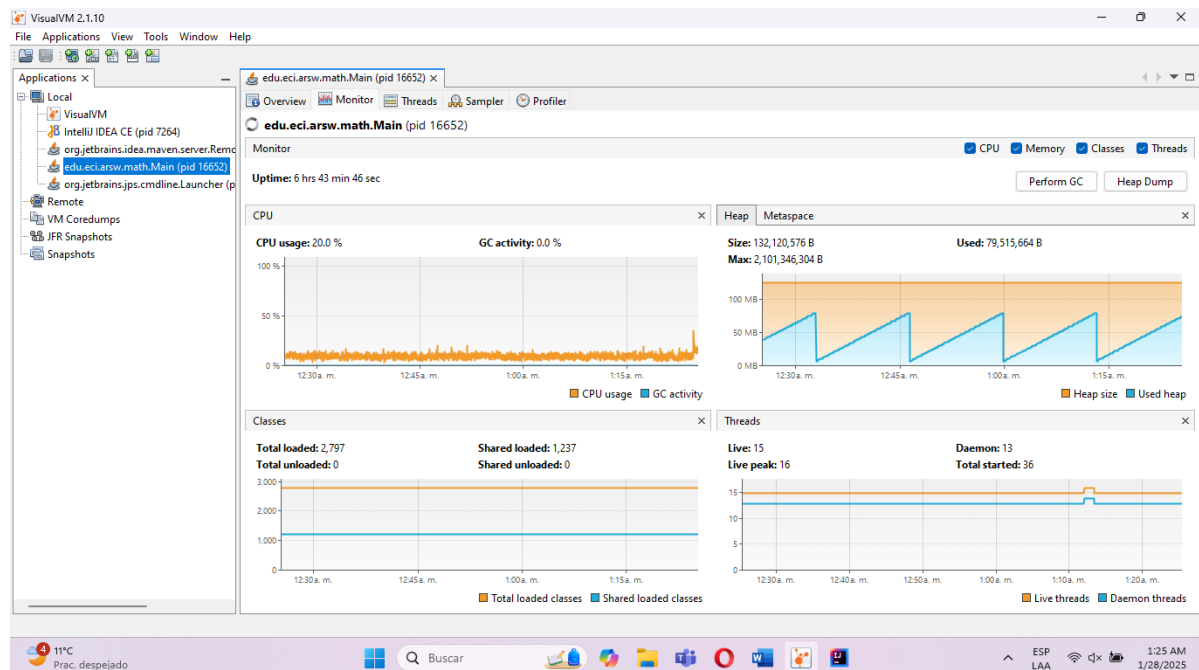
-----
BUILD SUCCESS
-----
Total time: 7.960 s
Finished at: 2025-01-27T19:02:49-05:00
  
```

Parte III Evaluación 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):

1. Un solo hilo.

Tiempo Estimado: 6 horas, 43 min en la foto, pero en realidad duro más de 14 horas

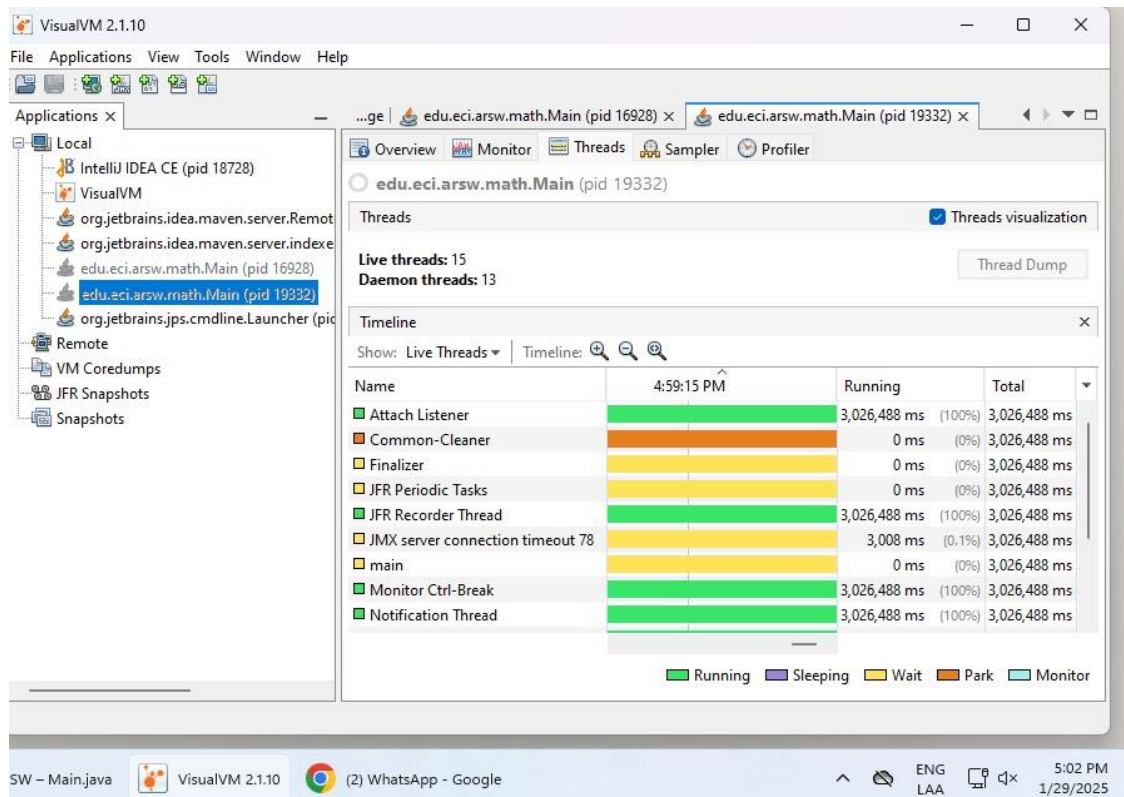
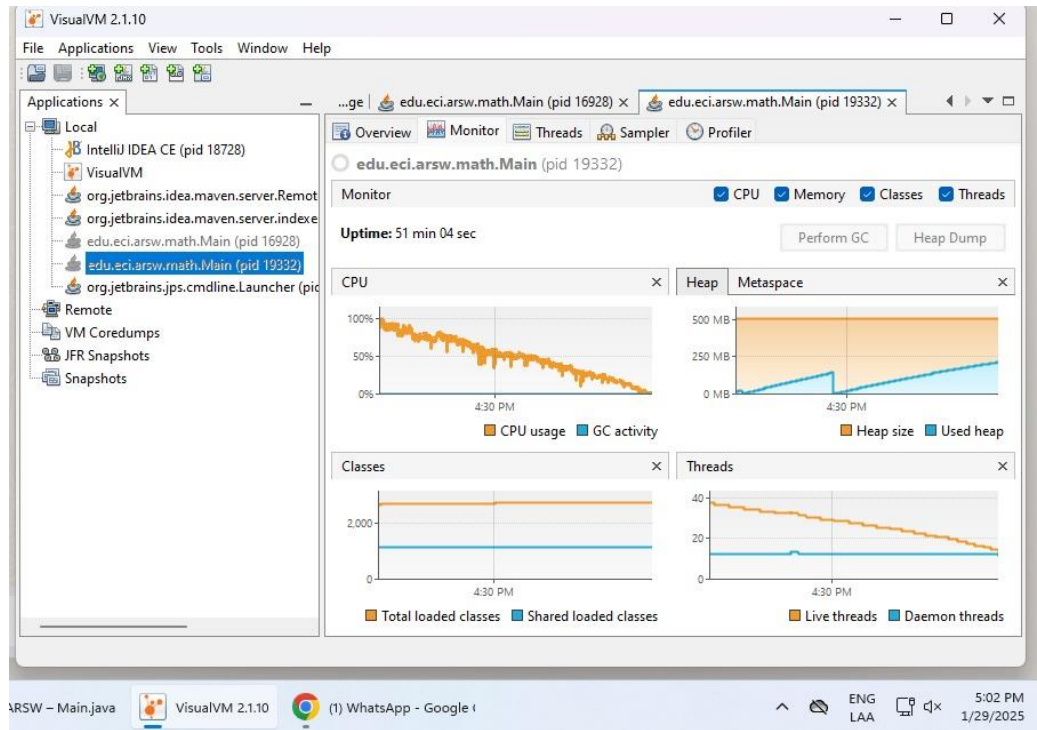




2. *Tantos hilos como núcleos de procesamiento (haga que el programa determine esto haciendo uso del [API Runtime](#)).*

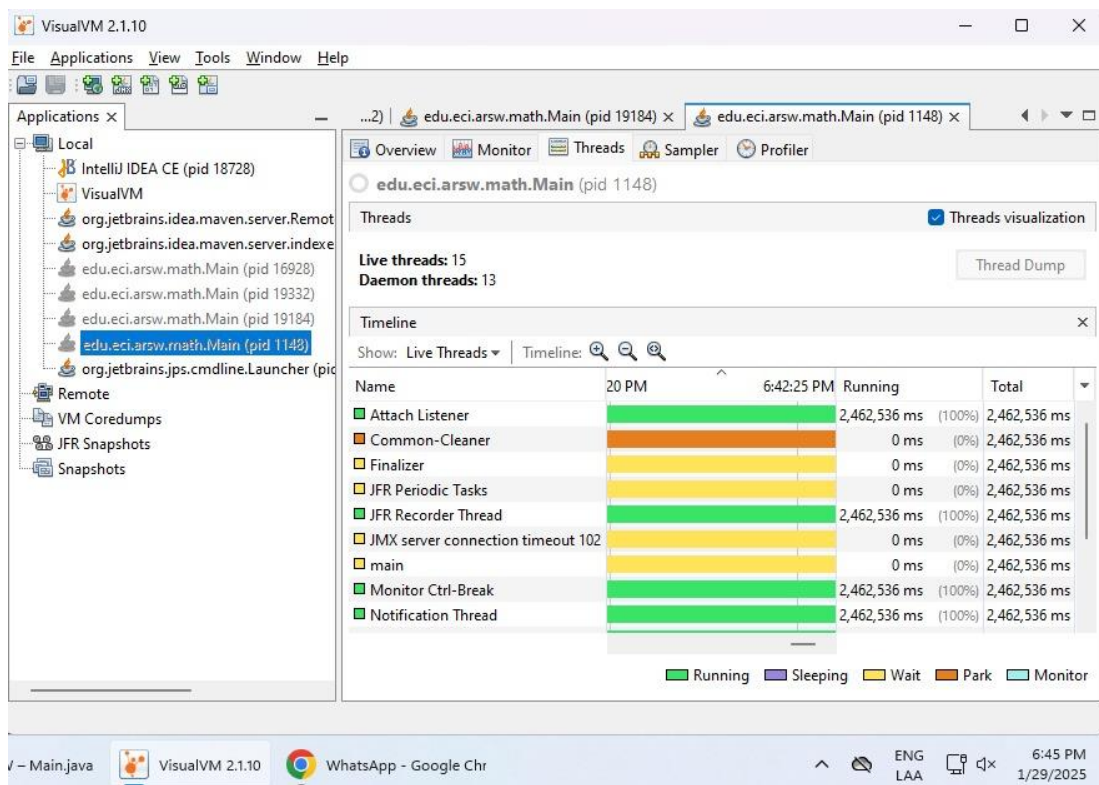
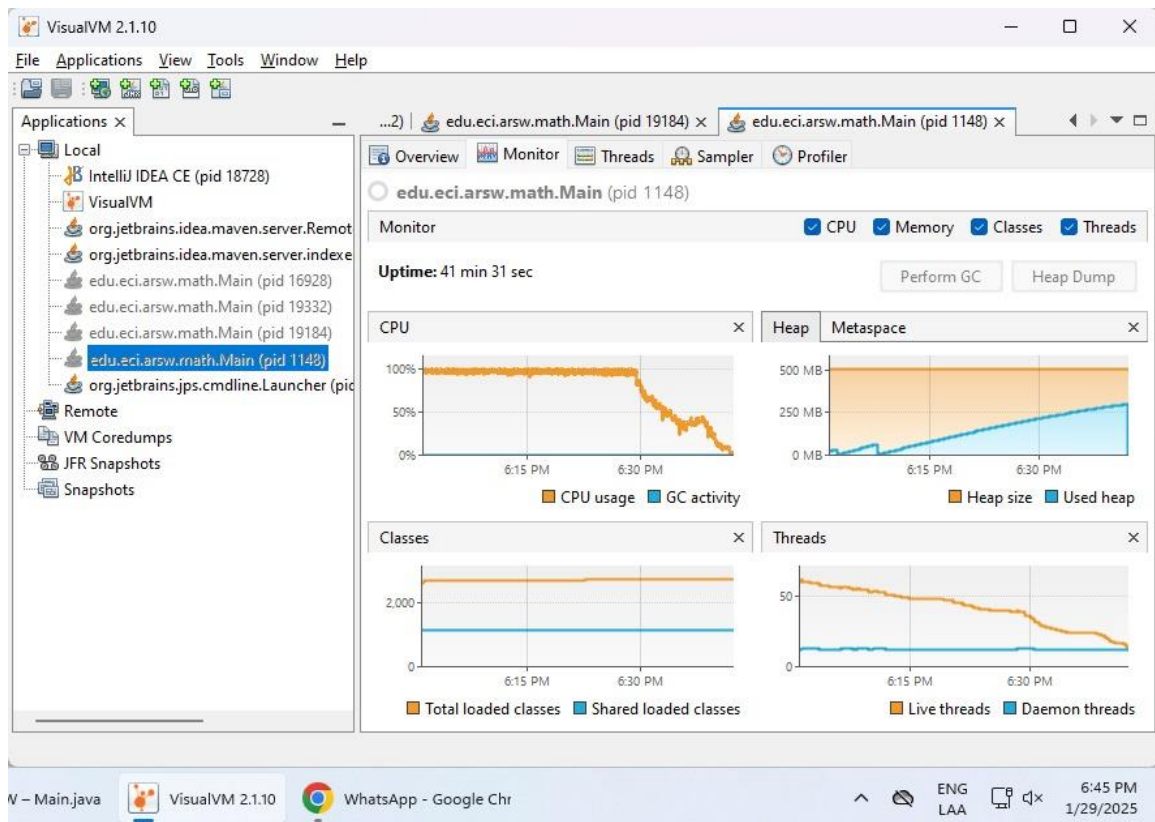
Tiempo Estimado: 51 min

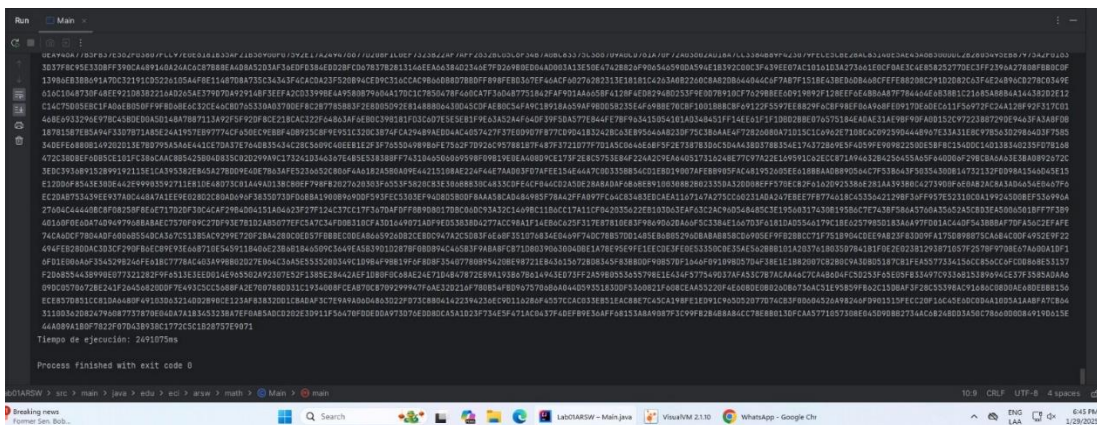




3. *Tantos hilos como el doble de núcleos de procesamiento (24 hilos).*

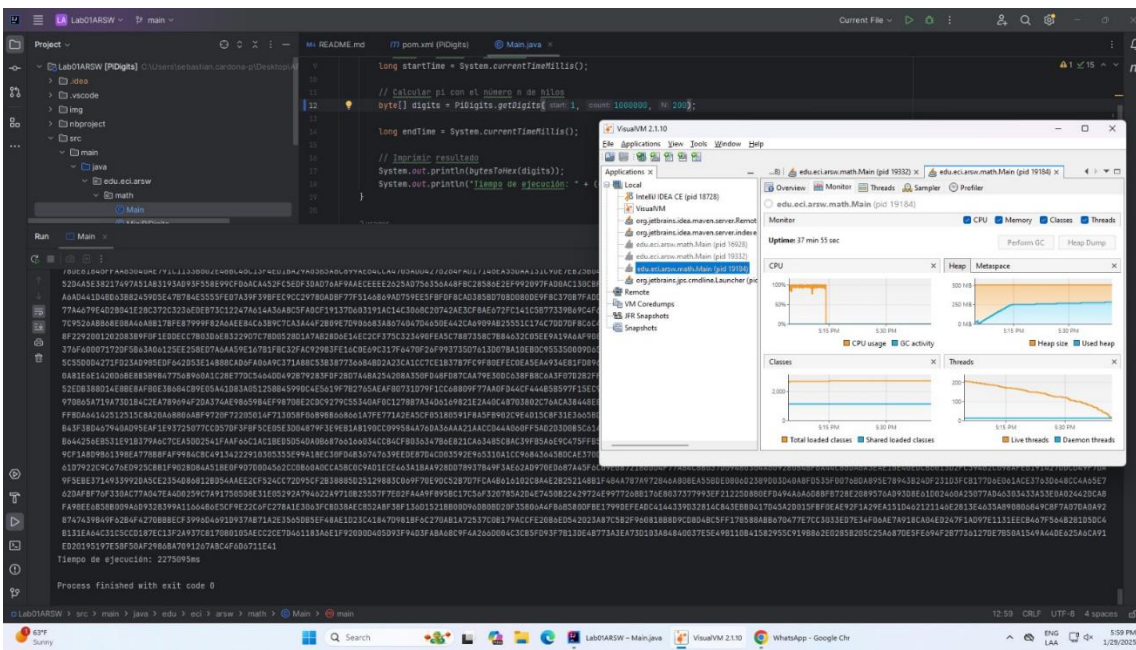
Tiempo Estimado: 41 min

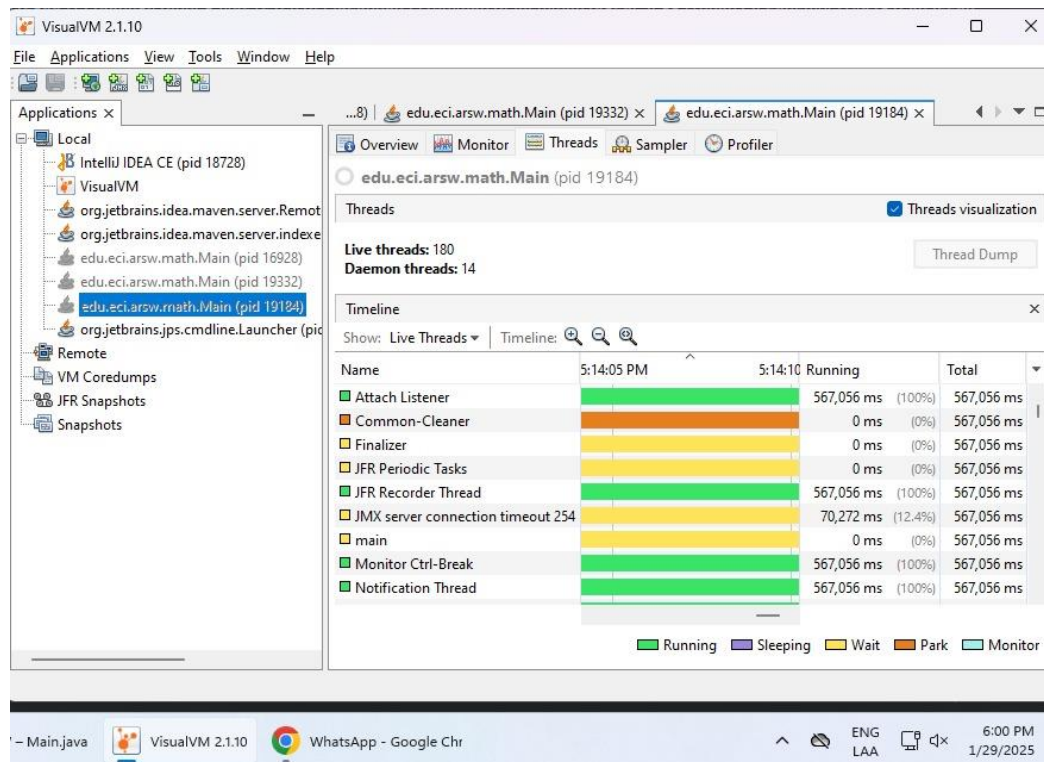
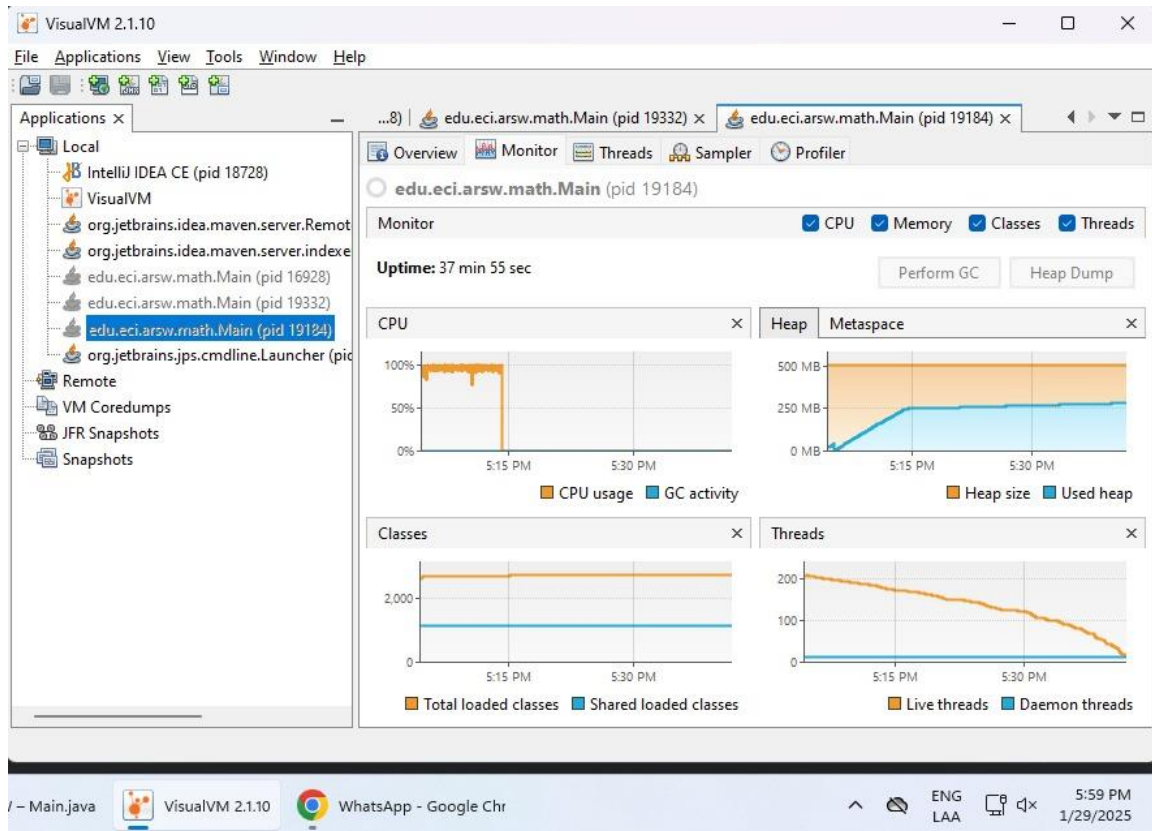


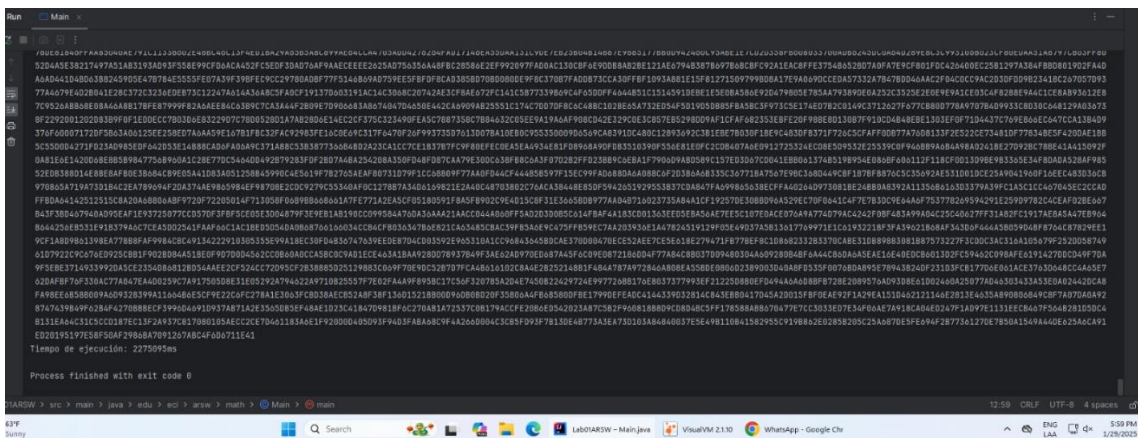


4. 200 hilos.

Tiempo Estimado: 37 min

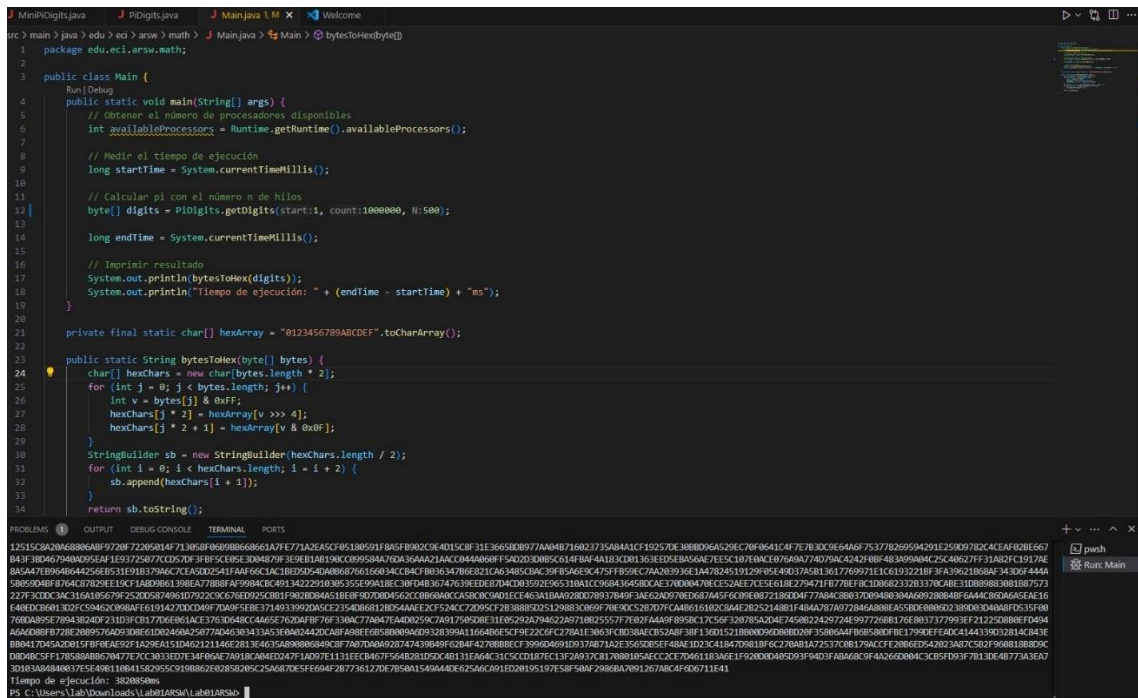


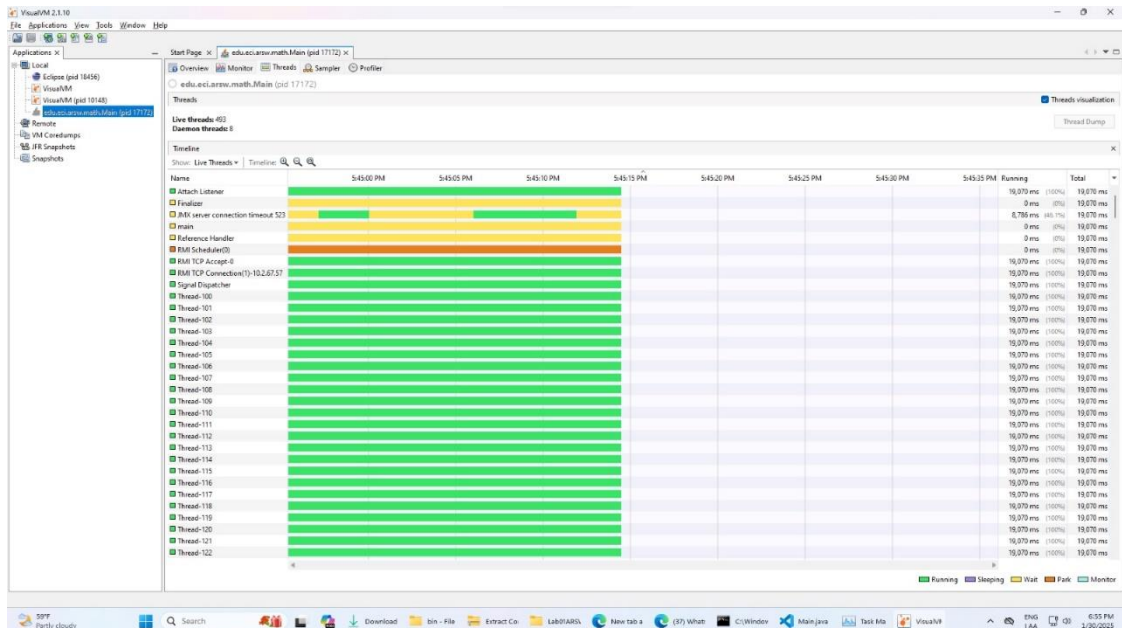




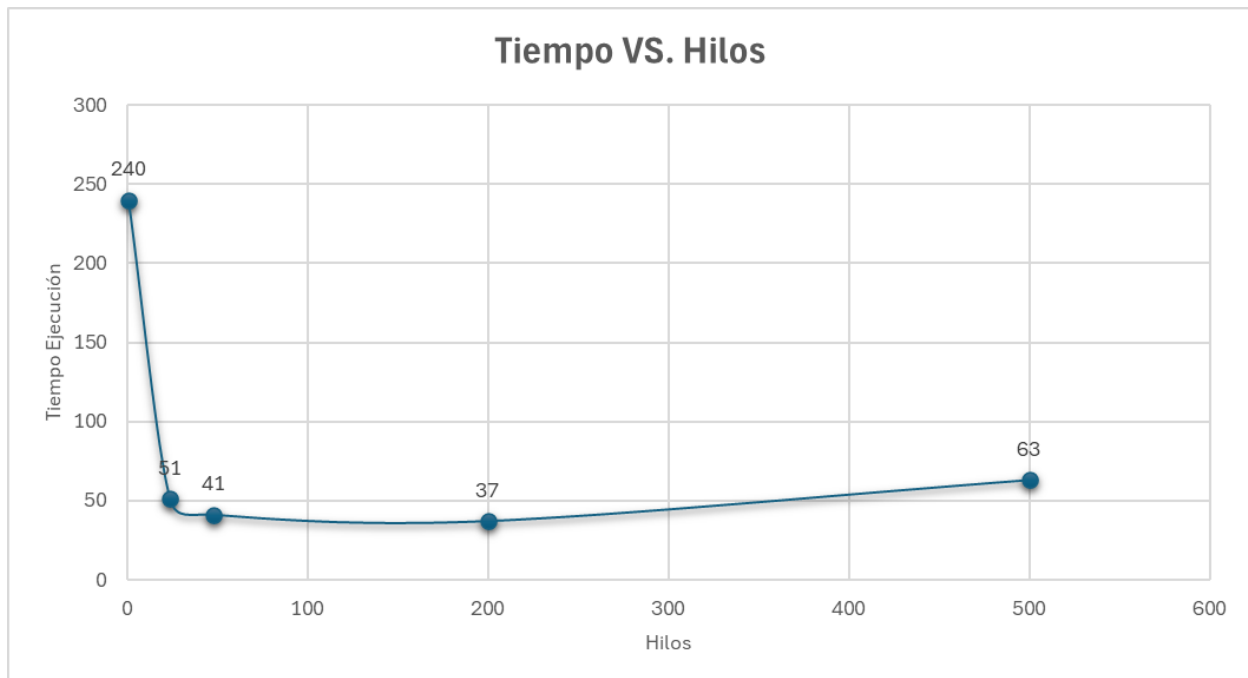
5. 500 hilos.

Tiempo Estimado: 1 hora y 3 minutos





Con lo anterior, y con los tiempos de ejecución dados, haga una gráfica de tiempo de solución vs. número de hilos. Analice y plantee hipótesis con su compañero para las siguientes preguntas (puede tener en cuenta lo reportado por jVisualVM):



Queremos mencionar que las características del equipo de cómputo fueron relevantes para el desarrollo de este laboratorio, pues, primero intentamos ejecutar el programa con un hilo desde un computador de 4 núcleos de procesamiento, luego con uno de 8, pero lamentablemente el tiempo de ejecución excedió las 14 horas, por lo que, toda esta parte se realizó desde una máquina del laboratorio de ingeniería de software cuya cantidad de núcleos de procesamiento son 24.

En conclusión, se observó una mejora absoluta entre correr el programa con un hilo a con los del procesador del computador, esto se ve, ya que cada hilo se reparte el trabajo equitativamente y lo hacen al tiempo, para luego unir el resultado, sin embargo, al momento de usar el doble de hilos del procesador, y de usar 200 hilos, hubo mejora, pero fue mínima, podemos suponer que esto sucedió, pues al usar demasiados hilos, puede reducir el rendimiento del computador esto sucede pues hay una sobrecarga de hilos, donde requiere trabajo pasar entre contexto de cada hilo y se consume mayor recursos de la memoria RAM. Es por esto que se evidencia un retroceso cuando pasamos de 200 a 500 hilos, pues se redujo el tiempo en 26 minutos

1. Según la [ley de Amdahl](#):

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

donde $S(n)$ es el mejoramiento teórico del desempeño, P la fracción paralelizable del algoritmo, y n el número de hilos, a mayor n , mayor debería ser dicha mejora. Por qué el mejor desempeño no se logra con los 500 hilos?, cómo se compara este desempeño cuando se usan 200?.

- Tiempo con 200 hilos: 37 minutos.

- Tiempo con 500 hilos: 1 hora y 3 minutos (63 minutos).

Usamos los datos de 200 hilos y 500 hilos para estimar P .

1. Para 200 hilos:

$$37 = (1 - P) * T + \left(\frac{P * T}{200}\right)$$

2. Para 500 hilos:

$$63 = (1 - P) * T + \left(\frac{P * T}{500}\right)$$

Restamos las dos ecuaciones para eliminar T :

$$63 - 37 = (1 - P) * T + \left(\frac{P * T}{200}\right)$$

$$P * T = -\left(\frac{26}{0.003}\right) = -8666.67$$

Aquí nos damos cuenta que el resultado no tiene sentido físico porque P y T deben ser positivos. Esto sugiere que los tiempos proporcionados no son consistentes con la ley de Amdahl o que existen otros factores, como el overhead, que no se están considerando.

Dado que no podemos estimar P directamente, asumiremos un valor razonable para P , donde a ser $P=0.95$ (95% del algoritmo es paralelizable).

1. Para 200 hilos:

$$S(200) = \left(\frac{1}{(1 - 0.95) + \left(\frac{0.95}{200}\right)}\right) = 18.26$$

2. Para 500 hilos:

$$S(500) = \left(\frac{1}{(1 - 0.95) + \left(\frac{0.95}{500}\right)}\right) = 19.27$$

El mejor desempeño no se logra con 500 hilos debido al **overhead** y la **contención de recursos**. A más hilos, el sistema debe gestionar más contextos, lo que añade tiempo en creación, sincronización y comunicación. Además, los hilos compiten por recursos como la memoria caché y el ancho de banda de la RAM, reduciendo la eficiencia. La ley de Amdahl indica que la parte no paralelizable $(1-P)$ limita la mejora, sin importar el número de hilos. Con 200 hilos, hay menos overhead y contención, resultando en un mejor tiempo (37 minutos) que con 500 hilos (63

minutos). Teóricamente, 500 hilos ($S(500) \approx 19.27$) superan ligeramente a 200 ($S(200) \approx 18.26$), pero en la práctica, el overhead y la contención empeoran el rendimiento.

2. *¿Cómo se comporta la solución usando tantos hilos de procesamiento como núcleos comparados con el resultado de usar el doble de éste?*

Usar tantos hilos como núcleos suele ser óptimo porque cada hilo puede ejecutarse en un núcleo sin competir por recursos de CPU, minimizando el overhead y la contención. En cambio, usar el doble de hilos que núcleos obliga al sistema operativo a realizar **multiplexación** (cambio de contexto entre hilos), lo que introduce overhead y reduce el rendimiento debido a la competencia por recursos como la CPU y la memoria caché.

3. *De acuerdo con lo anterior, si para este problema en lugar de 500 hilos en una sola CPU se pudiera usar 1 hilo en cada una de 500 máquinas hipotéticas, la ley de Amdahl se aplicaría mejor?. Si en lugar de esto se usaran c hilos en $500/c$ máquinas distribuidas (siendo c es el número de núcleos de dichas máquinas), se mejoraría? . Explique su respuesta.*

La ley de Amdahl se aplica igual con 1 hilo en 500 máquinas o c hilos en $500/c$ máquinas, ya que el límite teórico depende de la fracción paralelizable P . Sin embargo, distribuir los hilos en múltiples máquinas reduce el overhead y la contención de recursos, mejorando el rendimiento práctico. Aunque no se supera el límite teórico, esta distribución es más eficiente que usar 500 hilos en una sola CPU, donde el overhead y la contención degradan el desempeño.

Conclusiones

Los hilos y el paralelismo son herramientas clave en programación para crear aplicaciones más rápidas. Los hilos comparten memoria dentro de un proceso, mientras que el paralelismo ejecuta tareas simultáneamente en diferentes núcleos. Usarlos mejora el rendimiento al dividir tareas, pero también introduce complejidad y posibles problemas de concurrencia.

Extender la clase Thread en Java permite crear hilos personalizados con comportamientos específicos. El método `run()` es donde se define el código que el hilo ejecutará, mientras que el método `start()` inicia la ejecución del hilo. En esencia, `start()` crea un nuevo hilo y luego llama al método `run()` dentro de ese nuevo hilo, permitiendo la ejecución concurrente del código.

El número de hilos no debe exceder la capacidad del hardware para evitar pérdidas de rendimiento. Aunque la ley de Amdahl sugiere que más hilos mejoran el desempeño, en la práctica, el overhead de gestión y la contención de recursos (como la memoria caché y el ancho de banda de la RAM) pueden degradar el rendimiento. Esto se evidencia al comparar 200 hilos (37 minutos) con 500 hilos (63 minutos), donde un mayor número de hilos empeoró el tiempo de ejecución debido a la saturación del sistema.