Wilmer Arley Rodríguez Ropero

Daniel Sebastián Ochoa Urrego

Laboratorio 1 ARSW

Parte 1 Hilos Java

1. Definimos el hilo haciendo uso de la interfaz Runnable de la siguiente manera

```
public class CountThread implements Runnable{
    2 usages
    private final int a;
    2 usages
    private final int b;

    5 usages  new *
    public CountThread(int a, int b){
        this.a = a;
        this.b = b;
    }
    new *
    @Override
    public void run() {
        for(int i = a; i <= b; i++){
            System.out.println(i);
        }
    }
}
```

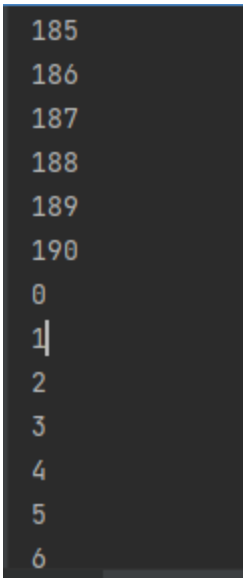
2. El método main de la clase quedo de la siguiente manera

```
 * @author hcadavid
 */
Héctor Cadavid *
public class CountThreadsMain {

    Héctor Cadavid *
    public static void main(String a[]){
        Thread A = new Thread(new CountThread( a: 0,  b: 99));
        Thread B = new Thread(new CountThread( a: 99,  b: 199));
        Thread C = new Thread(new CountThread( a: 200,  b: 299));
        A.start();
        B.start();
        C.start();
    }

}
```

Y al correr el programa vimos que los hilos corren concurrentemente en el procesador cada uno con cierto tiempo de ejecución imprimiéndose los números de a bloques internamente ordenados, pero sin ningún orden en particular de estos



Y al cambiar la llamada de los hilos por el método run en el main, lo que empieza a suceder al correr el programa es que los “hilos” se corren secuencialmente, esto porque en realidad no se están creando hilos, simplemente se están llamando los métodos como si fueran uno normal, por eso es que para crear hilos concurrentes debemos llamar el método start, pues este primero los crea y después llama el método run en cada uno de ellos.

Parte 2 Hilos Java

1. Creamos la clase PiThread, en donde además de sobrescribir el método run de la superclase como se muestra en la siguiente foto, movimos la lógica de la clase PiDigits como originalmente estaba, esto para sea la orquestadora de hilos mas adelante

```
public class PiThread extends Thread{
    2 usages
    private final int start;
    2 usages
    private final int count;
    2 usages
    private static int DigitsPerSum = 8;
    1 usage
    private static double Epsilon = 1e-17;

    1 usage   unknown
    public PiThread(int start, int count){
        this.start = start;
        this.count = count;
    }

    unknown *
    @Override
    public void run(){
        this.getDigits(start, count);
        // System.out.println(Arrays.toString(this.getDigits(start, count)));
    }
```

2. La función orquestadora de los hilos quedo de la siguiente manera

```java
public static byte[] getDigits(int start, int count, int N) throws InterruptedException {
    ArrayList<PiThread> threads = new ArrayList<>();
    byte[] answer = new byte[count];
    divideThreads(count, start, N, threads);
    waitThreads(threads);
    sortThreads(threads, answer);
    return answer;
}
```

En donde nos ayudamos de funciones internas para dividir el trabajo

```java
private static void divideThreads(int count, int start, int N, ArrayList<PiThread> threads){
    int module = (count) % N;
    int range = count / N;
    int threadStart = start;
    for(int i = 0;  i < N; i++){
        if(module > 0){
            range++;
            module--;
        }
        PiThread piThread = new PiThread(threadStart, range);
        threads.add(piThread);
        piThread.start();
        threadStart += range;
        range = count / N;
    }
}
```

```java
private static void waitThreads(ArrayList<PiThread> threads) throws InterruptedException {
    for(PiThread thread : threads){
        thread.join();
    }
}
```

```java
private static void sortThreads(ArrayList<PiThread> threads, byte[] answer){
    int offset = 0;
    for (PiThread thread : threads) {
        byte[] bytes = thread.getDigits();
        System.arraycopy(bytes,  srcPos: 0, answer, offset, bytes.length);
        offset += bytes.length;
    }
}
```

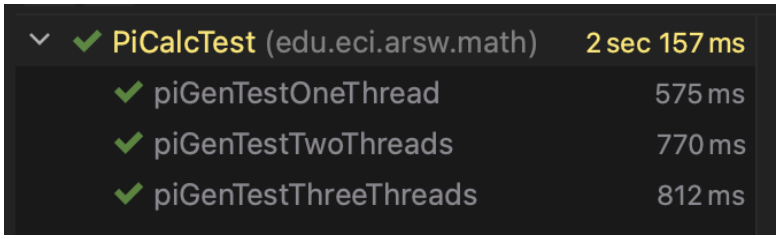
3. Copiamos la plantilla de prueba que había antes y usamos el nuevo método para crear los hilos, quedando de la siguiente manera

```java
@Test
public void piGenTestThreeThreads() 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, 0x8, 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.getDigits(start, count,  N: 3);
            assertEquals(count, digits.length);
            for (int i = 0; i < digits.length; i++) {
                assertEquals(expected[start + i], digits[i]);
            }
        }
    }
}
```

Y al correr las 3 pruebas, cada una con un nuero diferente de hilos podemos ver todas pasan

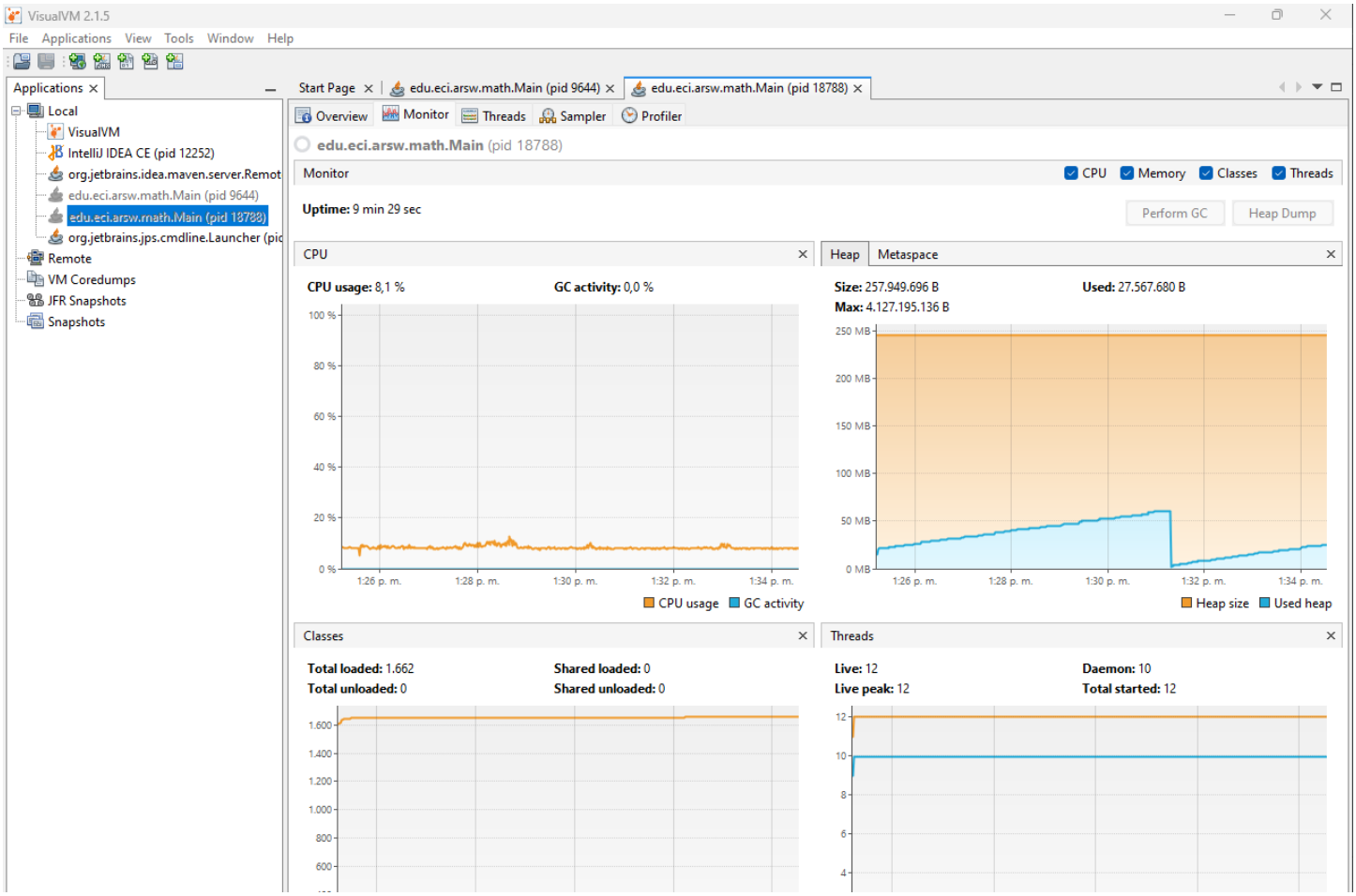


Parte 3 Evaluación de desempeño

Para la siguiente sección decidimos calcular solo 200000 dígitos de Pi, ya que con un millón nuestros computadores despegaban

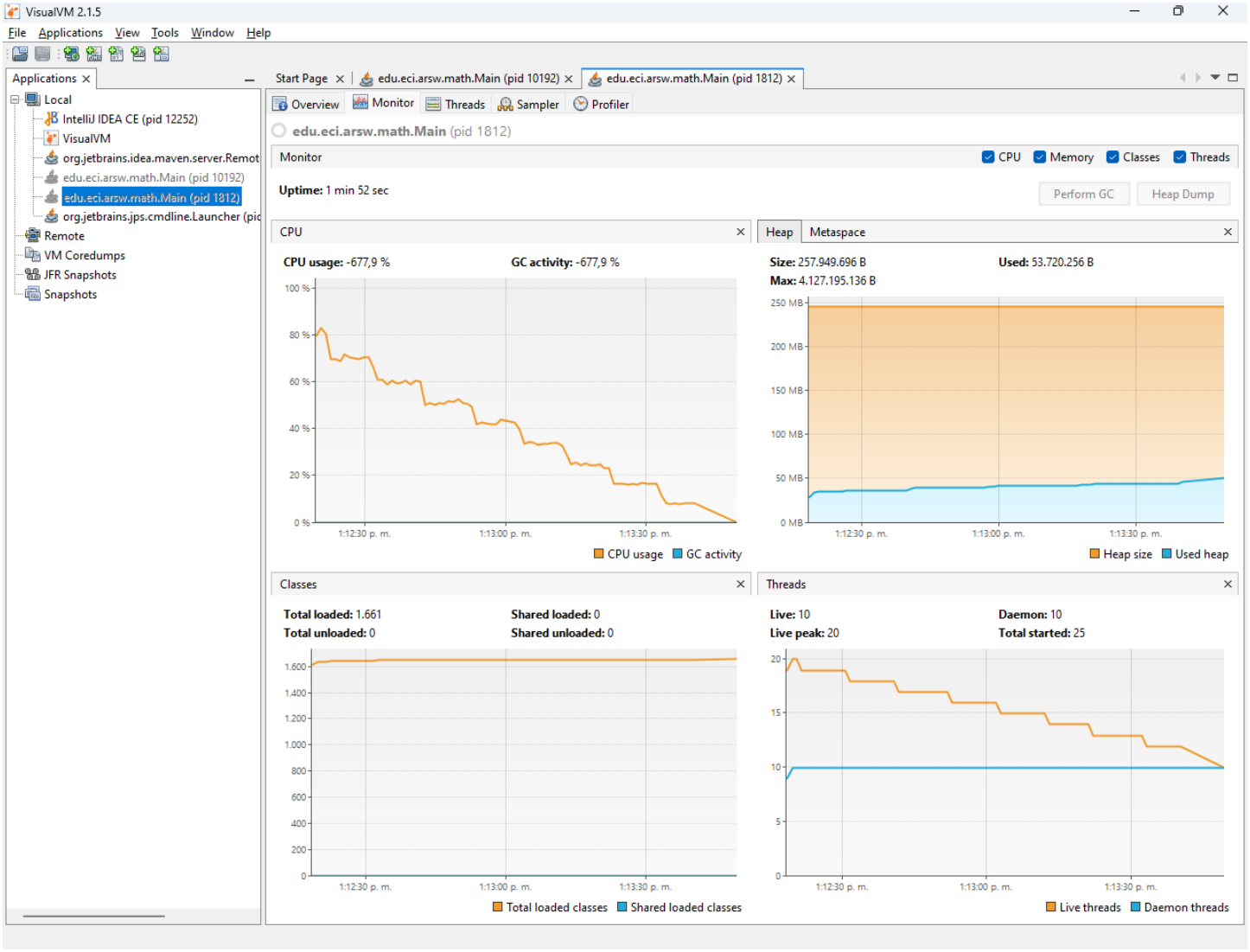
1. Al correr el programa haciendo uso de 1 hilo, esta fue la Memoria y CPU utilizada según el programa JVisualVM

```java
DanielOchoa +1 *
public static void main(String a[]) throws InterruptedException {
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors())));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors()*2)));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,200)));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,500)));
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0,  count: 200000, N: 1)));
}
```



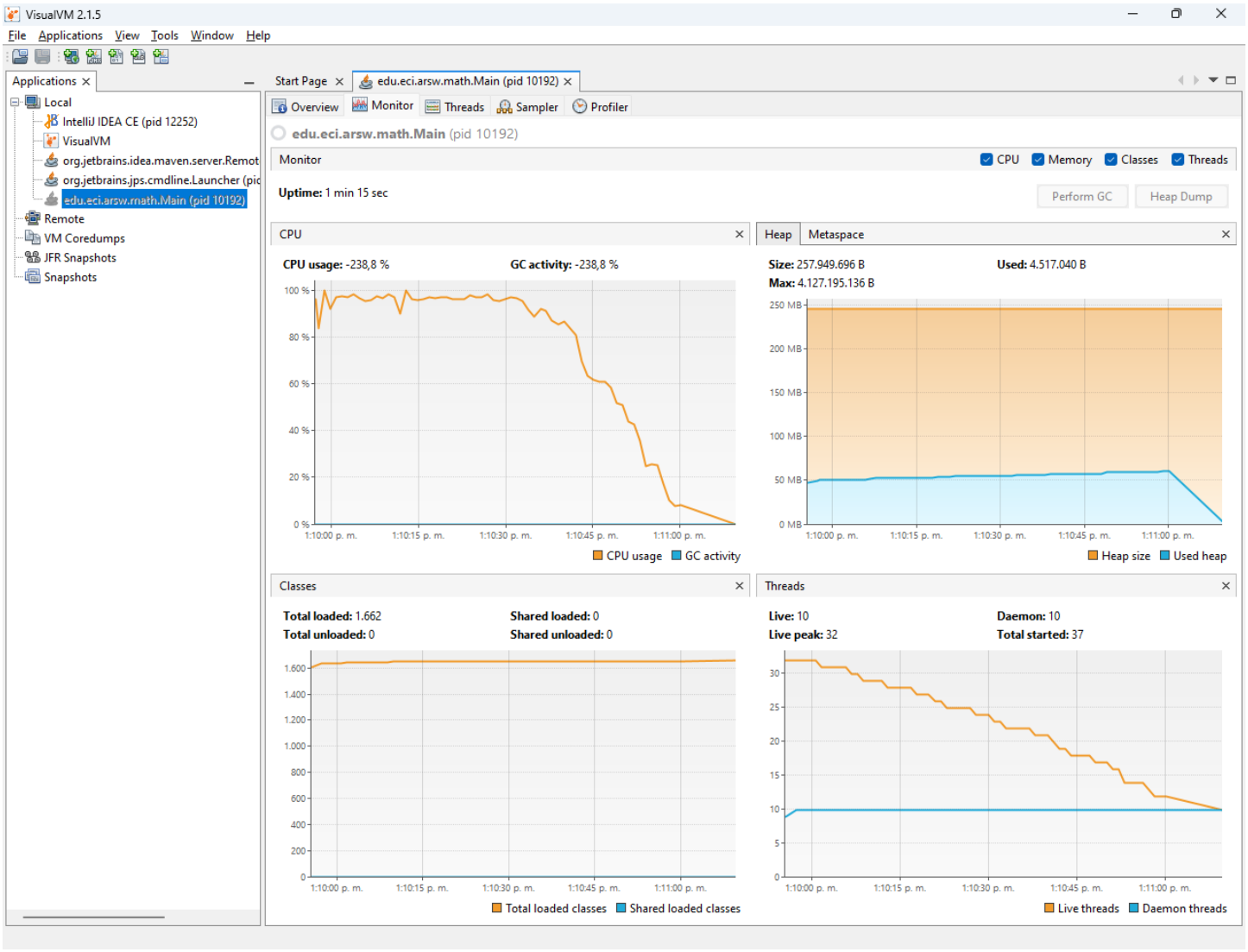
2. Al correr el programa haciendo uso de el mismo número hilos como de procesadores, esta fue la Memoria y CPU utilizada según el programa JVisualVM

```
DanielOchoa +1 *
public static void main(String a[]) throws InterruptedException {
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0,  count: 200000,Runtime.getRuntime().availableProcessors())));
}
```



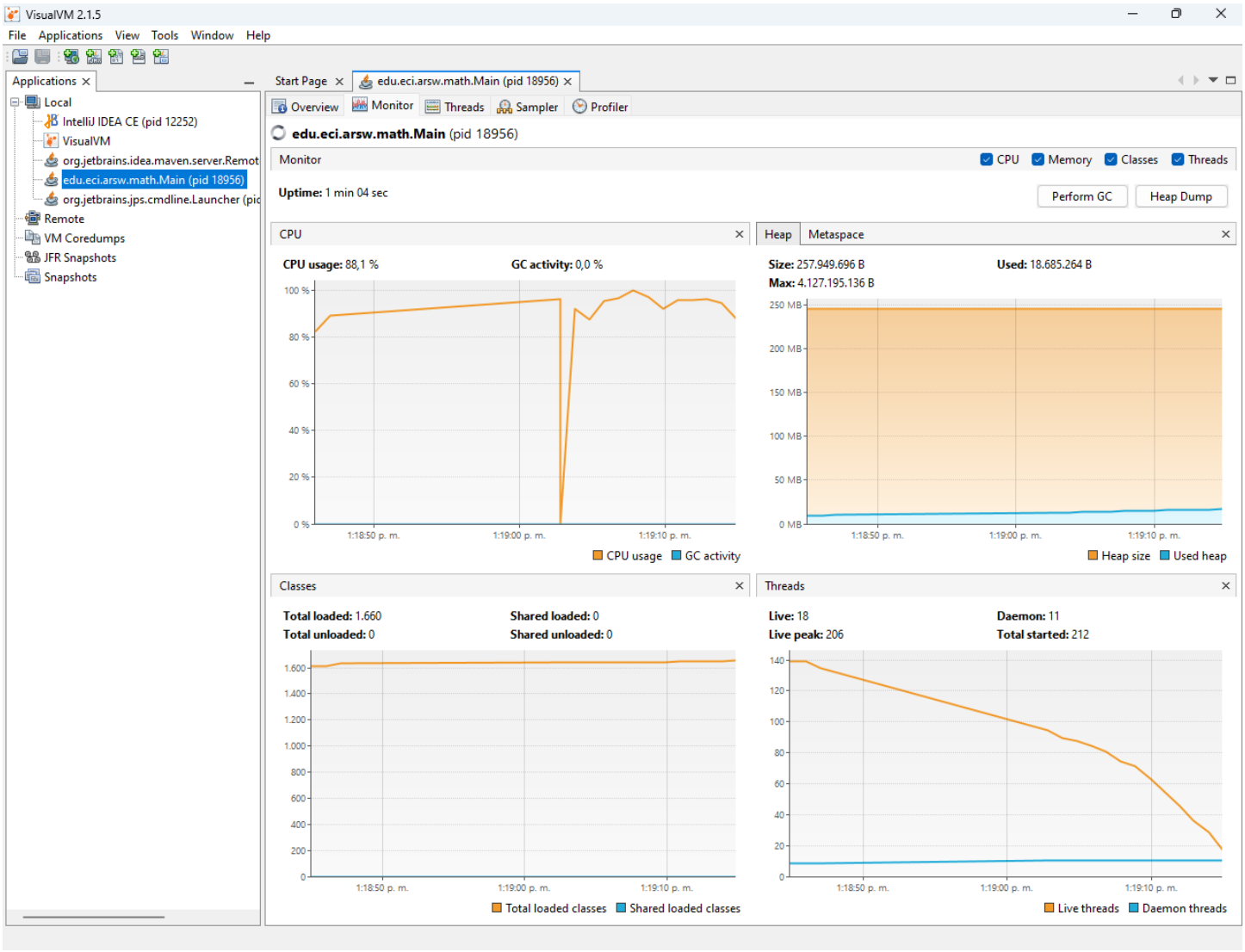
3. Al correr el programa haciendo uso del doble de hilos del número de procesadores del PC, esta fue la Memoria y CPU utilizada según el programa JVisualVM

```
DanielOchoa +1 *
public static void main(String a[]) throws InterruptedException {
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0,  count: 200000, N: Runtime.getRuntime().availableProcessors() * 2)));
}
```



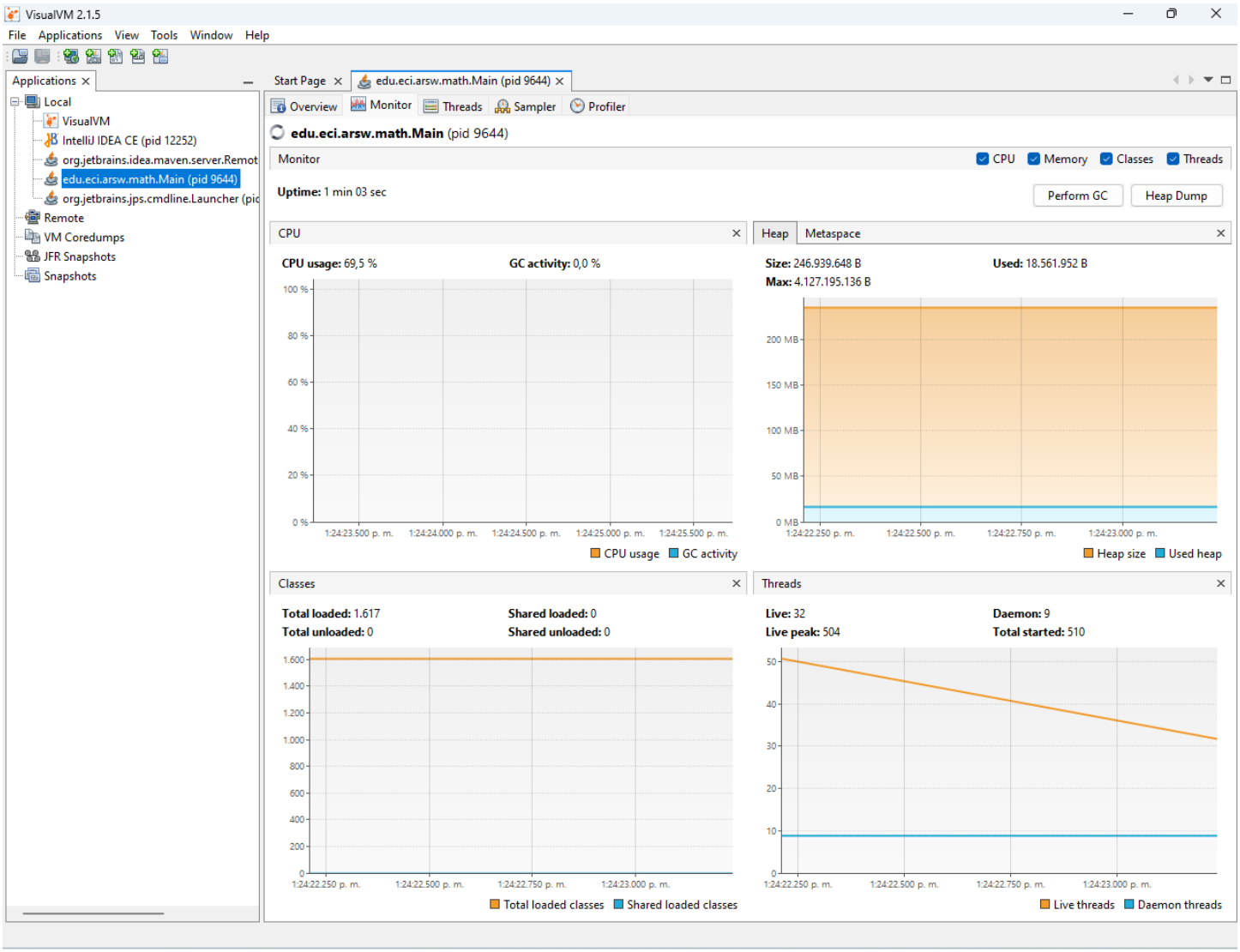
4. Al correr el programa haciendo uso de 200 hilos, esta fue la Memoria y CPU utilizada según el programa JVisualVM

```
DanielOchoa +1 *
public static void main(String a[]) throws InterruptedException {
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors())));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors()*2)));
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0,  count: 200000, N: 200)));
}
```



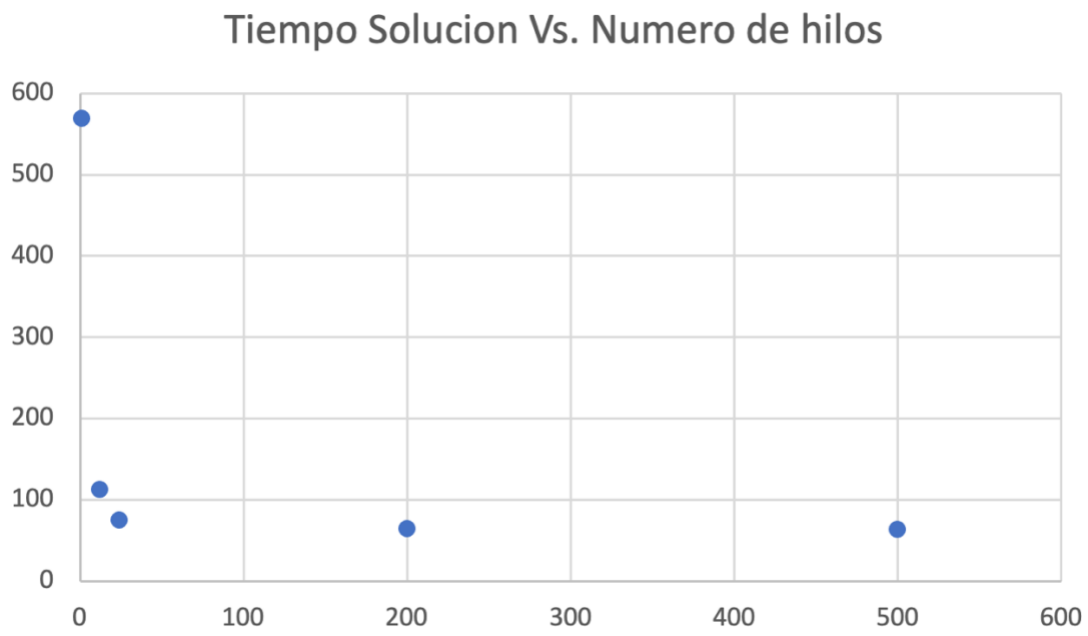
5. Al correr el programa haciendo uso de 500 hilos, esta fue la Memoria y CPU utilizada según el programa JVisualVM

```
DanielOchoa +1 *
public static void main(String a[]) throws InterruptedException {
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors())));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,Runtime.getRuntime().availableProcessors()*2)));
    //System.out.println(bytesToHex(PiDigits.getDigits(0, 200000,200)));
    System.out.println(bytesToHex(PiDigits.getDigits( start: 0,  count: 200000, N: 500)));
}
```



Análisis de resultados

Obtuvimos lo siguiente al graficar los tiempos de solución vs. el número de hilos



Hipótesis: A mayor cantidad de hilos se tendera a un tiempo de solución no necesariamente 0.

1. Como se puede observar, n está en el denominador y cuando este tiende a infinito, la función tendera al limite $\frac{1}{1-P}$ por lo que se puede decir que entre mayor cantidad de hilos se tendera a este valor, lo que significa que, al usar 200 hilos, ya estaremos muy cerca a este valor gastando menores recursos que si usáramos 500. Como también se puede evidenciar en la gráfica respecto al tiempo, este no da mejores resultados con una cantidad muy grande de hilos

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

2. Se puede observar que hay una diferencia de tiempo más considerable, correspondiente a una mejora de tiempo del 33%, esta mejora podría explicarse por qué en estos casos, estamos aprovechando al máximo los recursos del computador, optimizando el paralelismo pues en teoría hacemos uso de todos los núcleos del procesador además de que con ayuda de los hilos internos en los núcleos aprovechamos la concurrencia.

3. Al tener 500 computadores cada uno corriendo un hilo, se aplicaría mejor la ley de Amdahls ya que en este caso si tendríamos paralelismo real, mientras que corriéndolos todos en un mismo procesador obligaría al computador a hacer uso del Interleaving lo que afectaría la cantidad de hilos paralelos que es están corriendo.

   Al igual que con las 500 maquinas esto haría que se optimizara el paralelismo ya que cada núcleo se encargaría de su propio hilo lo que haría que hubiera Interleaving que afectara la estimación de la ley de Amdahls