Imagen en blanco y negro

El contenido generado por IA puede ser incorrecto.

**Universidad Nacional Mayor de San Marcos**

**Facultad de Ingeniería de Sistemas e Informática**

E.P. de Ingeniería de Software

**GUÍA N° 2**

**Asignatura:** Programación Concurrente y Paralela

**Docente:** Javier Antonio Prudencio Vidal

**Estudiante:** Bruno, Pumapillo Sarmiento

**Procesamiento paralelo en Java**

**Tabla de contenido**

* Diferencia entre el procesamiento paralelo y secuencial
* Procesamiento paralelo en Java usando parallelStream()
* Procesamiento paralelo en Java usando parallel()
* Procesamiento paralelo en Java usando CompletableFuture

**Procesamiento paralelo**

El procesamiento paralelo es una nueva tendencia en la programación moderna. Este procesamiento nos permite ejecutar múltiples tareas simultáneamente, disminuyendo el tiempo de procesamiento, pero las tareas deben ser independientes.

En este artículo, veremos el procesamiento paralelo en Java, junto con los ejemplos y explicaciones necesarios para facilitar el tema.

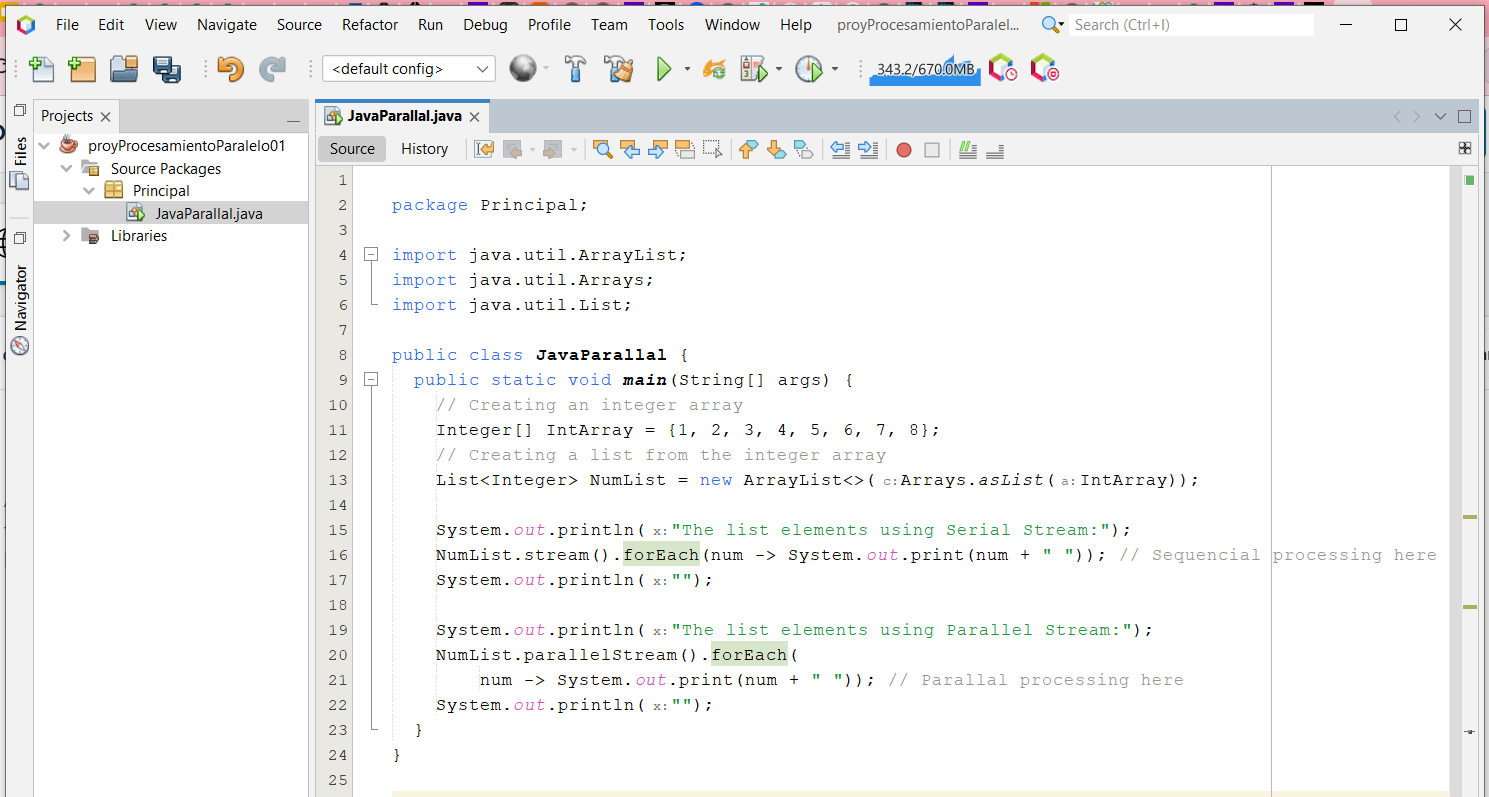
**Diferencia entre el procesamiento paralelo y secuencial**

En el procesamiento secuencial, todos los procesos independientes de una tarea se ejecutan uno por uno, y cuando un proceso está funcionando, otro proceso debe esperar.

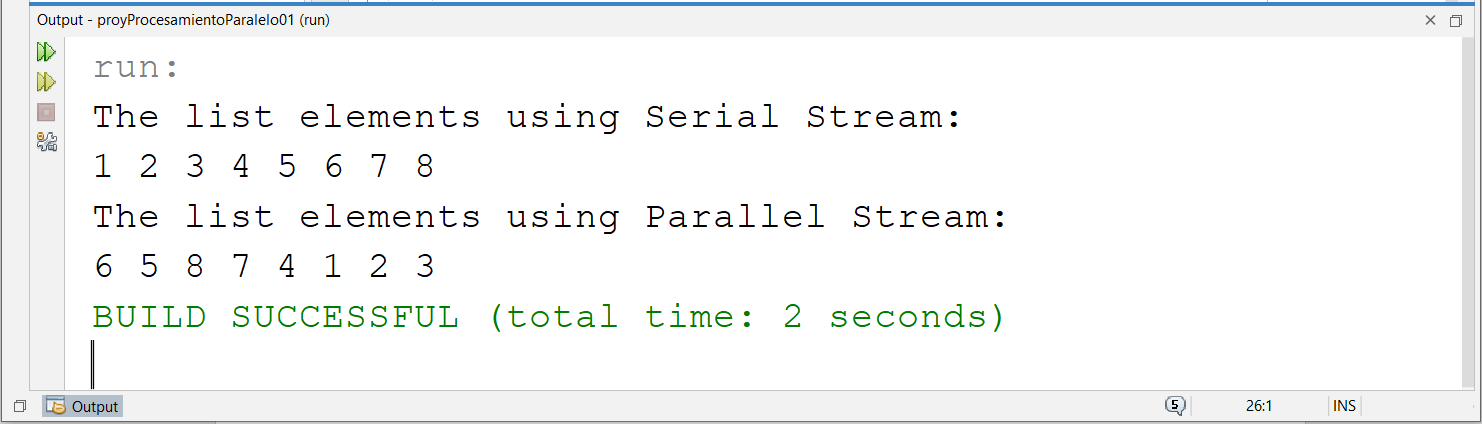
Por otro lado, en el procesamiento paralelo, se pueden ejecutar varios procesos a la vez y no es necesario esperar a que finalice un proceso. El procesamiento en paralelo es esencial para la multitarea.

**Procesamiento paralelo en Java usando parallelStream()**

Este ejemplo ilustrará cómo podemos realizar un procesamiento paralelo en Java. Echemos un vistazo a nuestro siguiente ejemplo:

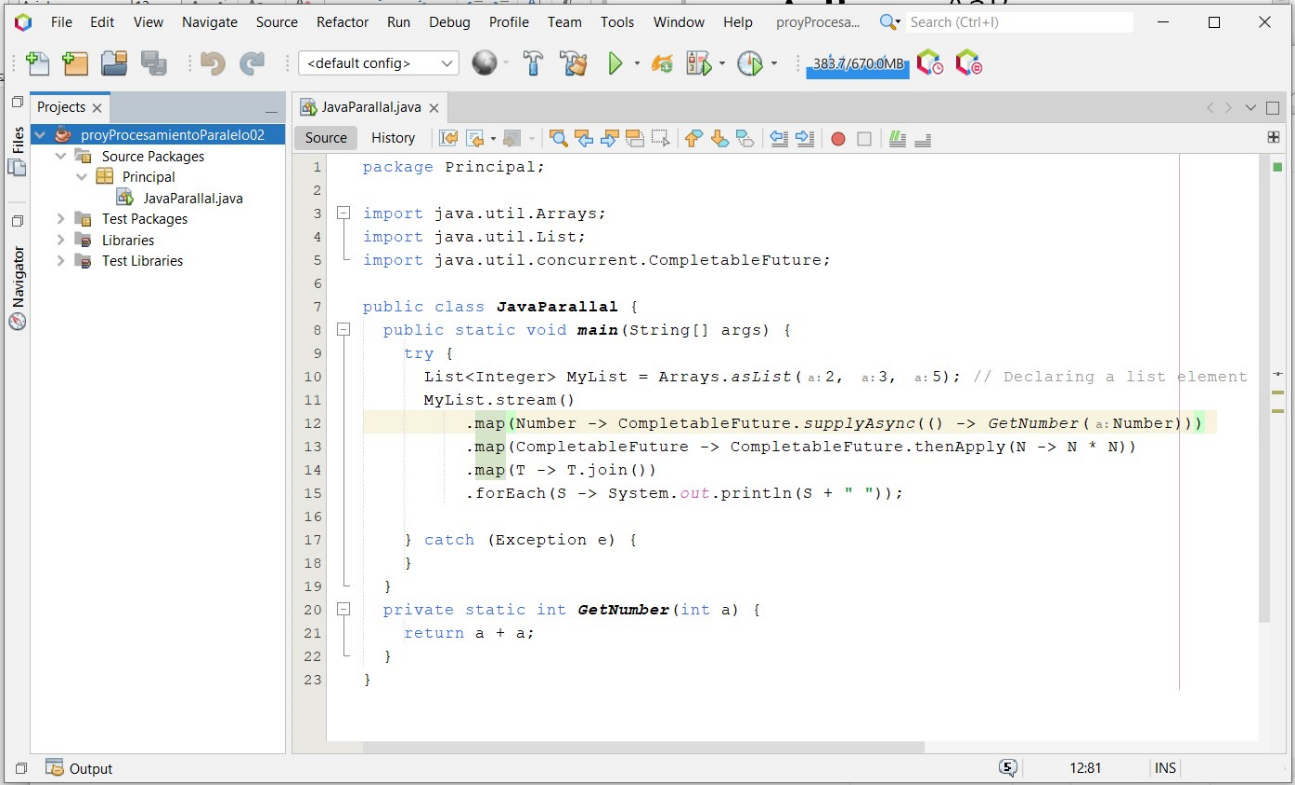


Observando las líneas de código y los dos escenarios comparados (secuencial y paralelo), estamos logrando los propósitos de cada una de las líneas de código. A continuación, se debe de ejecutar el código respectivo para obtener lo siguiente:



**Procesamiento paralelo en Java usando CompletableFuture**

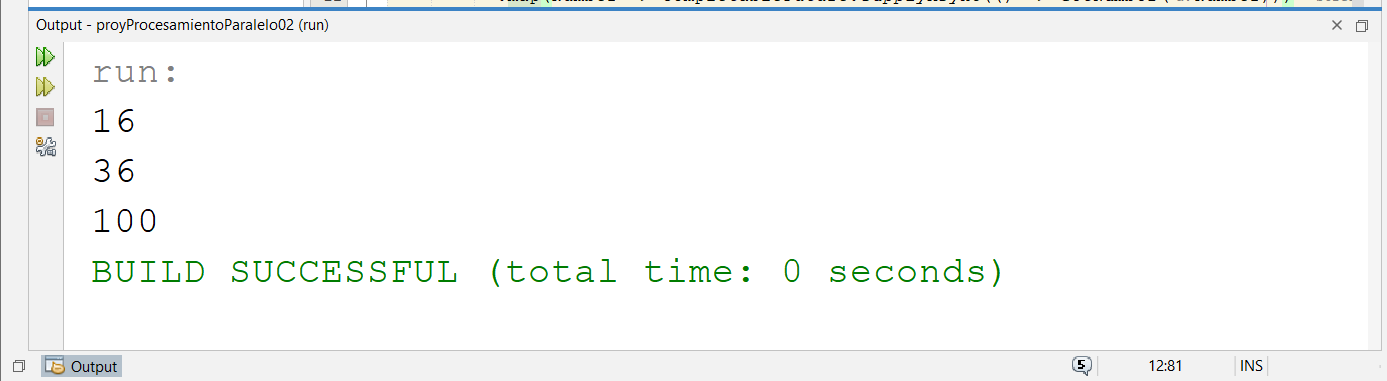
En nuestro ejemplo a continuación, demostramos cómo podemos realizar un procesamiento paralelo utilizando el método CompletableFuture. Vamos a ver el siguiente código de ejemplo:



En nuestro ejemplo anterior,

* **supplyAsync():** este método completa un trabajo de forma asíncrona. Por defecto, el método ForkJoinPool.commonPool() ejecuta la tarea desde el proveedor.
* **thenApply():** este método acepta una función como argumento y devuelve un nuevo CompletableStage después de que la etapa actual se complete normalmente.
* **join():** este método devuelve el resultado una vez que se ha completado.

Cuando ejecute el comando anterior, obtendrá un resultado como el siguiente en su consola:



**Ejercicios propuestos de procesamiento paralelo**

**Caso de Estudio 1:** Comparación entre Procesamiento Secuencial y Paralelo

Objetivo: Analizar la eficiencia del procesamiento paralelo comparado con el secuencial en la realización de tareas computacionalmente intensivas.

**Ejercicio Propuesto:**

Desarrollar un programa en Java que genere una lista de números grandes y realice la factorización de cada número.

Implementar dos versiones del programa: una utilizando un enfoque secuencial y otra utilizando parallelStream() para paralelizar la operación.

Medir y comparar el tiempo de ejecución y la utilización de CPU para ambos enfoques.

Métricas de Evaluación:

* Tiempo total de ejecución.
* Porcentaje de uso de CPU.
* Eficiencia de escalabilidad al aumentar el tamaño de la entrada.

**DESARROLLO**

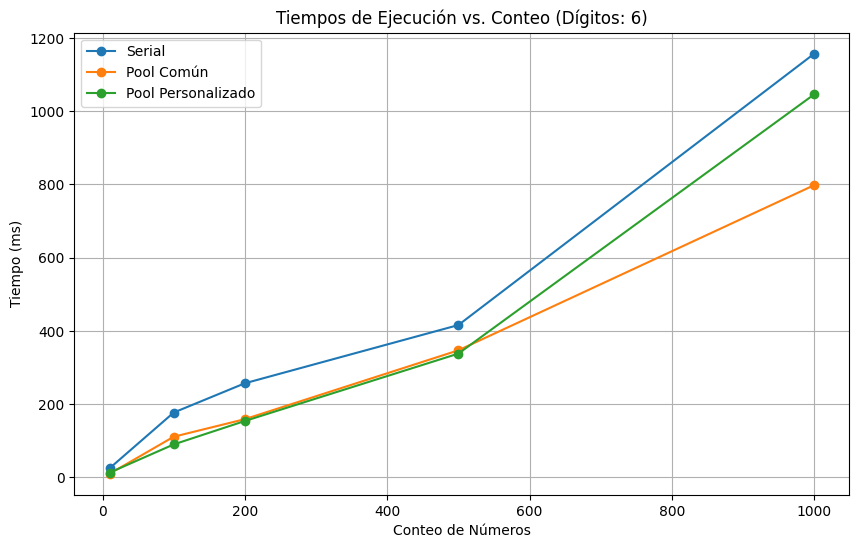
**Código Java**

|  |
| --- |
| import java.math.BigInteger;  import java.security.SecureRandom;  import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.ForkJoinPool;  public class BigNum2 {  SecureRandom sr = new SecureRandom();  // Configurar el número de hilos para el parallel stream  private static final int NUM\_THREADS = Runtime.getRuntime().availableProcessors();  private static List<BigInteger> factorization(BigInteger n) {  List<BigInteger> factors = new ArrayList<>();  BigInteger original = n;  // Factoriza por 2  while (n.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {  factors.add(BigInteger.TWO);  n = n.divide(BigInteger.TWO);  }  // Optimización: calcular sqrt una vez y actualizar cuando sea necesario  BigInteger sqrt = sqrt(n);  // Factoriza por impares  for (BigInteger i = BigInteger.valueOf(3);  i.compareTo(sqrt) <= 0;  i = i.add(BigInteger.TWO)) {  while (n.mod(i).equals(BigInteger.ZERO)) {  factors.add(i);  n = n.divide(i);  sqrt = sqrt(n); // Recalcular sqrt después de encontrar un factor  }  }  // Si queda un factor primo mayor  if (n.compareTo(BigInteger.TWO) > 0) {  factors.add(n);  }  // Debug: mostrar qué hilo está procesando  System.out.println("Factorizando " + original + " en hilo: " + Thread.currentThread().getName());  return factors;  }  // Metodo auxiliar para calcular raíz cuadrada de BigInteger  private static BigInteger sqrt(BigInteger n) {  if (n.equals(BigInteger.ZERO)) {  return BigInteger.ZERO;  }  BigInteger x = n;  BigInteger y = n.add(BigInteger.ONE).divide(BigInteger.TWO);  while (y.compareTo(x) < 0) {  x = y;  y = x.add(n.divide(x)).divide(BigInteger.TWO);  }  return x;  }  public BigInteger random(int k) {  byte[] ba = new byte[k];  ba[0] = (byte) (sr.nextInt(9) + 49);  for (int d = 1; d < k; d++) {  ba[d] = (byte) (sr.nextInt(10) + 48);  }  return new BigInteger(new String(ba));  }  // Lista de n numeros aleatorios de k digitos  public List<BigInteger> listNum(int n, int k) {  return sr.ints(n, 0, 10).mapToObj(i -> random(k)).toList();  }  public static void main(String[] args) {  System.out.println("Núcleos disponibles: " + NUM\_THREADS);  System.out.println("Parallelism del ForkJoinPool común: " +  ForkJoinPool.commonPool().getParallelism());  int n = Integer.parseInt(args[0]); // Number of digits  int k = Integer.parseInt(args[1]); // Number of numbers  BigNum2 b = new BigNum2();  List<BigInteger> listNum = b.listNum(k, n);  System.out.println("\n=== PROCESAMIENTO SERIAL ===");  long startTime = System.currentTimeMillis();  listNum.forEach(S -> System.out.println("Numero: " + S));  System.out.println();  listNum.forEach(S -> System.out.println("FACTORIZACION: " + factorization(S) + "\n"));  long serialTime = System.currentTimeMillis() - startTime;  System.out.println("Tiempo serial: " + serialTime + " ms\n");  System.out.println("=== PROCESAMIENTO PARALELO (ForkJoinPool común) ===");  startTime = System.currentTimeMillis();  // Usar parallel stream con el pool común  listNum.parallelStream()  .map(number -> {  List<BigInteger> factors = factorization(number);  return "Numero: " + number + " -> FACTORIZACION: " + factors;  })  .forEach(System.out::println);  long parallelTime = System.currentTimeMillis() - startTime;  System.out.println("\nTiempo paralelo (pool común): " + parallelTime + " ms");  System.out.println("Speedup: " + String.format("%.2f", (double)serialTime / parallelTime) + "x\n");  System.out.println("=== PROCESAMIENTO PARALELO (ForkJoinPool personalizado) ===");  // Crear un ForkJoinPool personalizado con todos los núcleos  ForkJoinPool customThreadPool = new ForkJoinPool(NUM\_THREADS);  try {  startTime = System.currentTimeMillis();  customThreadPool.submit(() ->  listNum.parallelStream()  .map(number -> {  List<BigInteger> factors = factorization(number);  return "Numero: " + number + " -> FACTORIZACION: " + factors;  })  .forEach(System.out::println)  ).get();  long customParallelTime = System.currentTimeMillis() - startTime;  System.out.println("\nTiempo paralelo (pool personalizado): " + customParallelTime + " ms");  System.out.println("Speedup vs serial: " + String.format("%.2f", (double)serialTime / customParallelTime) + "x");  System.out.println("Speedup vs pool común: " + String.format("%.2f", (double)parallelTime / customParallelTime) + "x");  } catch (Exception e) {  e.printStackTrace();  } finally {  customThreadPool.shutdown();  }  }  } |

**Resultados obtenidos**

| **Digits** | **Count** | **Serial\_ms** | **Comun\_ms** | **Speedup\_Comun** | **Personal\_ms** | **Speedup\_Pers\_vs\_Ser** | **Speedup\_Pers\_vs\_Com** | **User\_s** | **System\_s** | **CPU\_percent** | **Total\_s** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **6** | **10** | **25** | **10** | **2.50** | **13** | **1.92** | **0.77** | **0.17** | **0.14** | **91** | **0.338** |
| **6** | **100** | **177** | **111** | **1.59** | **90** | **1.97** | **1.23** | **0.40** | **0.16** | **98** | **0.566** |
| **6** | **200** | **257** | **159** | **1.62** | **154** | **1.67** | **1.03** | **0.59** | **0.16** | **91** | **0.817** |
| **6** | **500** | **416** | **347** | **1.20** | **338** | **1.23** | **1.03** | **1.40** | **0.25** | **126** | **1.305** |
| **6** | **1000** | **1157** | **798** | **1.45** | **1046** | **1.11** | **0.76** | **2.44** | **0.52** | **92** | **3.216** |
| **10** | **10** | **105** | **26** | **4.04** | **49** | **2.14** | **0.53** | **0.58** | **0.21** | **125** | **0.626** |
| **10** | **100** | **1922** | **121** | **15.88** | **158** | **12.16** | **0.77** | **1.34** | **0.34** | **58** | **2.885** |
| **10** | **200** | **639** | **163** | **3.92** | **298** | **2.14** | **0.55** | **1.02** | **0.67** | **123** | **1.371** |
| **10** | **500** | **1464** | **751** | **1.95** | **490** | **2.99** | **1.53** | **1.46** | **1.97** | **111** | **3.068** |
| **10** | **1000** | **2768** | **902** | **3.07** | **562** | **4.93** | **1.60** | **2.31** | **1.47** | **82** | **4.561** |
| **20** | **10** | **12464** | **7750** | **1.61** | **6812** | **1.83** | **1.14** | **50.00** | **0.98** | **186** | **27.294** |
| **20** | **100** | **1987264** | **1021473** | **1.95** | **975459** | **2.04** | **1.05** | **18223.62** | **74.01** | **459** | **3984.61** |

**Gráficos**



Gráfico, Gráfico de líneas

El contenido generado por IA puede ser incorrecto.

Gráfico, Gráfico de líneas

El contenido generado por IA puede ser incorrecto.

Gráfico, Gráfico de barras

El contenido generado por IA puede ser incorrecto.

**Promedios:**

**User\_s 1523.777500**

**System\_s 6.740000**

**CPU\_percent 136.833333**

**Total\_s 335.888083**