


Aplicaciones para Ambientes Distribuidos

Laboratorio # 5 – Prof. Regis Rivera

Objetivo: Programación paralela en Java

Nota: Los nombres de archivos de cada aplicación están dados en el icono 

Programación paralela en Java

La programación paralela es un proceso de dividir un problema complejo en tareas más pequeñas y simples, que se pueden ejecutar al mismo tiempo utilizando varios recursos informáticos. En el proceso de programación paralela, las tareas que son independientes entre sí se ejecutan en paralelo utilizando diferentes computadoras o varios millones de millones presentes en la CPU de una computadora. La programación paralela es una necesidad esencial para que las empresas manejen proyectos pesados y de gran escala, ya que necesitan mantener sus estándares económicos. Debido a la programación paralela, la velocidad del proyecto aumenta y su probabilidad de error disminuye.

La programación paralela es bastante diferente del subproceso múltiple ya que sus tareas no necesitan seguir un orden de ejecución. Cada tarea de Programación Paralela está diseñada según la función que necesitan realizar. De ahí que se le conozca ampliamente como Paralelismo Funcional o Paralelismo de Datos.

¿Por qué se utiliza la programación paralela?

A medida que las CPU multinúcleo han avanzado en los últimos tiempos, se puede lograr una programación eficiente al máximo mediante la implementación de la programación paralela. Básicamente, la programación paralela se refiere a la ejecución paralela de procesos debido a la disponibilidad de múltiples recursos, como los núcleos de procesamiento. La programación paralela se considera un método mucho más eficiente que el subproceso múltiple. Las tareas paralelas que se resuelven simultáneamente se combinan para dar la solución final a un problema mayor.

El entorno estándar de Java (Java SE) proporciona al programador el "Marco Fork/Join". Fork/Join Framework ayuda a un programador a implementar la programación paralela fácilmente en sus aplicaciones. Pero la complejidad que surge con el marco Fork/Join es que el programador es quien necesita especificar cómo se divide el programa en tareas. Utilizando operaciones agregadas, el tiempo de ejecución de Java realizará esta división por nosotros.

El marco Fork/Join en Java SE:

`java.util.concurrent` es el paquete responsable de definir Fork/Join Framework en Java. El paquete `java.util.concurrent` contiene varias clases e interfaces que nos ayudan a lograr la programación paralela utilizando el lenguaje de programación Java. En el procesamiento paralelo, la parte de procesamiento está optimizada para el uso de múltiples procesadores al mismo tiempo. Cuando se trata de subprocesos múltiples, el tiempo de inactividad de una sola CPU se optimiza en función del tiempo compartido. Las tareas particionadas de un programa grande se envuelven en una

subclase ForkJoinTask haciendo uso de sus tareas abstractas. Las dos tareas abstractas de una subclase Fork/Join son " RecursiveTask " y " RecursiveAction ".

El marco Fork/Join utiliza una estrategia basada en la regla de divide y vencerás para lograr el procesamiento paralelo. En este proceso, una tarea enorme se divide en tareas pequeñas, que a su vez se dividen en tareas más simples. De esta manera, cada tarea se reduce a tareas más pequeñas hasta que sean lo suficientemente pequeñas como para manejarlas de forma secuencial.

Clases del marco Fork/Join:

1. RecursiveTask: es útil cuando deseamos devolver un resultado de nuestra tarea. Por ejemplo, al ordenar una matriz enorme, es necesario comparar el resultado de cada una de las subtareas entre sí. Esta tarea abstracta es una tarea compleja que dificulta la codificación.
2. RecursiveAction: Es útil cuando deseamos no devolver ningún resultado del programa. Por ejemplo, si deseamos inicializar una matriz grande con valores personalizados, entonces cada subtarea funciona sola en su propia parte de la matriz. Se crea una nueva clase para utilizar RecursiveAction.
La clase que creamos actúa como una clase secundaria que se extiende desde java.util.concurrent.RecursiveAction. Se debe implementar una nueva instancia de RecursiveAction para llamar a RecursiveAction. La instancia recién creada de RecursiveAction se invocará mediante ForkJoinPool.
3. ForkJoinTask: ForkJoinTask es una clase abstracta que se utiliza para definir una tarea. Básicamente, se puede crear una tarea con la ayuda del método fork() que pertenece a esta clase abstracta. Esta tarea es más ligera que un hilo creado utilizando la clase Thread.
4. ForkJoinPool: La clase ForkJoinPool nos proporciona un grupo común que nos ayuda en la ejecución de la tarea de ForkJoinTask.

Métodos del marco Fork/Join:

1. Compute(): se realiza una llamada recursiva cuando llamamos al método Compute() en la tarea correcta.
2. Fork(): cuando se utiliza el método fork(), nuestra PrimeRecursiveAction recién construida se agrega a la lista de tareas para el hilo activo.
3. Join(): cuando llamamos al método join() en una tarea bifurcada, debería ser uno de los últimos pasos después de usar otros métodos.

Laboratorio 5.1: Generar números aleatorios aplicando paralelismo

```
1  import java.util.concurrent.ForkJoinPool;
2  import java.util.concurrent.TimeUnit;
3  import java.util.concurrent.RecursiveAction;
4
5  class Main {
6      public static void main(String[] args) {
7          final int SIZE = 10;
8          ForkJoinPool pool = new ForkJoinPool();
9          double na[] = new double [SIZE];
10         System.out.println("Valores aleatorios inicializados: ");
```



Lab51.java

```

11     for (int i = 0; i < na.length; i++) {
12         na[i] = (double) i + Math.random();
13         System.out.format("%.4f ", na[i]);
14     }
15     System.out.println();
16     CustomRecursiveAction task = new
17         CustomRecursiveAction(na, 0, na.length);
18     pool.invoke(task);
19     System.out.println("Valores cambiados: ");
20     for (int i = 0; i < 10; i++)
21         System.out.format("%.4f ", na[i]);
22     System.out.println();
23 }
24 }
25
26 class CustomRecursiveAction extends
27     RecursiveAction {
28     final int THRESHOLD = 2;
29     double [] numbers;
30     int indexStart, indexLast;
31     CustomRecursiveAction(double [] n, int s, int l) {
32         numbers = n;
33         indexStart = s;
34         indexLast = l;
35     }
36     @Override
37     protected void compute() {
38         if ((indexLast - indexStart) > THRESHOLD)
39             for (int i = indexStart; i < indexLast; i++)
40                 numbers[i] = numbers[i] + Math.random();
41         else
42             invokeAll(new CustomRecursiveAction(numbers,
43                 indexStart, (indexStart - indexLast) / 2),
44                 new CustomRecursiveAction(numbers,
45                     (indexStart - indexLast) / 2,
46                     indexLast));
47     }
48 }

```

Ahora bien, tomando en cuenta el concepto de paralelismo donde buscamos ejecutar de manera simultanea varias tareas que simbolizan que son parte de un gran problema que buscamos resolver, pero más eficientemente y rápido ¿Cómo podemos identificarlo de manera más simple?

Seria interesante ver en números fríos, cuanto tiempo demorarían ciertas tareas de manera independiente (en espera) y contar el tiempo total que se ejecutaron todas (si midiéramos el tiempo de manera secuencial), y al final, pudiésemos saber en cuanto mejoro la velocidad de la programación paralela en función de haber ejecutado dichas tareas de manera secuencial

Para esto, requeriremos también a utilizar Future

Future es una interface, por lo que no podemos instanciarla directamente. Una clase que implementa Future es FutureTask . Esta clase permite añadirle un Runnable o un Callable y luego lanzarla en un hilo.

Laboratorio 5.2: Contar tiempo de ejecución de tareas paralelas vs el tiempo total si se ejecutaran de manera secuencial.



Lab52.java

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.Random;
4  import java.util.concurrent.Callable;
5  import java.util.concurrent.ExecutionException;
6  import java.util.concurrent.ExecutorService;
7  import java.util.concurrent.Executors;
8  import java.util.concurrent.Future;
9
10 public class ExecutorServiceExample {
11     private static final Random PRNG = new Random();
12
13     private static class Result {
14         private final int wait;
15         public Result(int code) {
16             this.wait = code;
17         }
18     }
19
20     public static Result compute(Object obj) throws InterruptedException {
21         int wait = PRNG.nextInt(3000);
22         Thread.sleep(wait);
23         return new Result(wait);
24     }
25
26     public static void main(String[] args) throws InterruptedException,
27         ExecutionException {
28         List<Object> objects = new ArrayList<Object>();
29         for (int i = 0; i < 25; i++) {
30             objects.add(new Object());
31         }
32
33         List<Callable<Result>> tasks = new ArrayList<Callable<Result>>();
34         for (final Object object : objects) {
35             Callable<Result> c = new Callable<Result>() {
36                 @Override
37                 public Result call() throws Exception {
38                     return compute(object);
39                 }
40             };
41             tasks.add(c);
42         }
43
44         ExecutorService exec = Executors.newCachedThreadPool();
45
46         try {
47             long start = System.currentTimeMillis();
48             List<Future<Result>> results = exec.invokeAll(tasks);
49             int sum = 0;
50             for (Future<Result> fr : results) {
51                 sum += fr.get().wait;
52
53                 System.out.println(String.format("La tarea espero %d ms",
54                     fr.get().wait));
55             }
56             long elapsed = System.currentTimeMillis() - start;
57             System.out.println(String.format("Tiempo transcurrido: %d ms", elapsed));
58             System.out.println(String.format("... pero las tareas 'compute' esperaron un
59                 total de %d ms; mejorando la velocidad en %.2fx", sum, sum / (elapsed * 1d)));
60         } finally {
61             exec.shutdown();
62         }
63     }
64 }
```

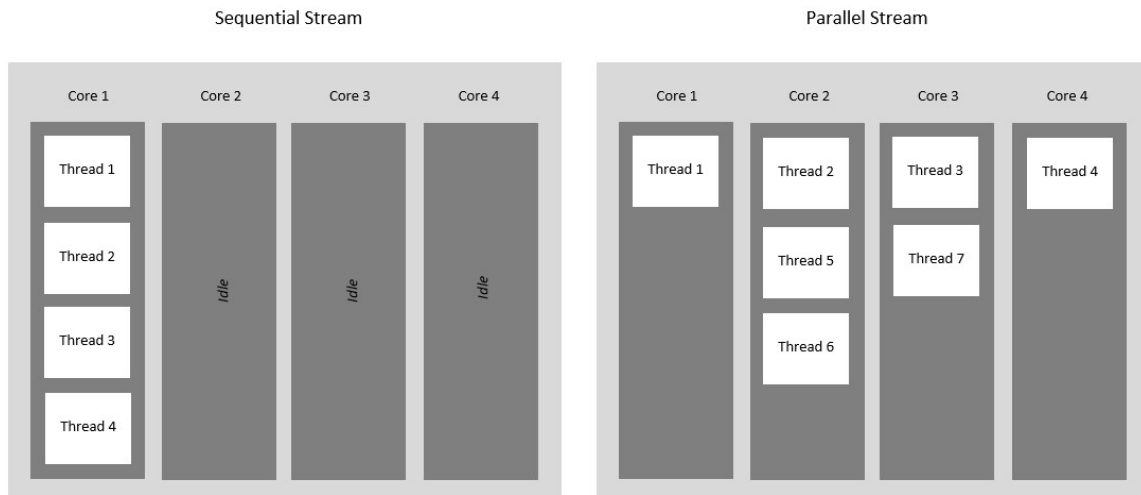
También Java nos permite trabajar mediante Parallel Streams (secuencias paralelas)

¿Qué son las secuencias paralelas de Java?

Java Parallel Streams es una característica de Java 8 y superior, diseñada para utilizar múltiples núcleos del procesador. Normalmente, cualquier código Java tiene un flujo de procesamiento, donde se ejecuta de forma secuencial. Mientras que al usar flujos paralelos, podemos dividir el

código en múltiples flujos que se ejecutan en paralelo en núcleos separados y el resultado final es la combinación de los resultados individuales. La orden de ejecución, sin embargo, no está bajo nuestro control.

Por lo tanto, es aconsejable utilizar flujos paralelos en los casos en los que, independientemente del orden de ejecución, el resultado no se ve afectado y el estado de un elemento no afecta al otro, así como la fuente de los datos tampoco se ve afectada.



¿Por qué transmisiones paralelas?

Las transmisiones paralelas se introdujeron para aumentar el rendimiento de un programa, pero optar por transmisiones paralelas no siempre es la mejor opción. Hay ciertos casos en los que necesitamos que el código se ejecute en un orden determinado y en estos casos, es mejor usar flujos secuenciales para realizar nuestra tarea a costa del rendimiento. La diferencia de desempeño entre los dos tipos de corrientes sólo es preocupante para programas a gran escala o proyectos complejos. En el caso de programas de pequeña escala, es posible que ni siquiera se note. Básicamente, debería considerar el uso de Parallel Streams cuando el flujo secuencial se comporta mal.

Hay dos formas que podemos crear, que se enumeran a continuación y se describen más adelante a continuación:

1. Usando el método `parallel()` en una secuencia
2. Usando `parallelStream()` en una colección

Método 1: usar el método `parallel()` en una secuencia

El método `parallel()` de la interfaz `BaseStream` devuelve un flujo paralelo equivalente.

En el código que se escribirá a continuación, creamos un objeto de archivo que apunta a un archivo 'txt' preexistente en el sistema. Luego creamos una secuencia que lee del archivo de texto una línea a la vez. Luego usamos el método `parallel()` para imprimir el archivo leído en la consola. El orden de ejecución es diferente para cada ejecución, puede observar esto en el resultado. Los dos resultados que se indican a continuación tienen diferentes órdenes de ejecución.

Laboratorio 5.3: Aplicar el método `parallel` en una secuencia

Nota: No funciona en compilador online

Nota: Crear ruta de carpeta o adaptarlo a alguna ruta disponible en su equipo



Lab53.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.nio.file.Files;
4  import java.util.stream.Stream;
5
6
7  public class GFG {
8
9      public static void main(String[] args) throws IOException {
10
11          File fileName = new File("C:\\carpeta\\Textfile.txt");
12
13          Stream<String> text = Files.lines(fileName.toPath());
14
15          text.parallel().forEach(System.out::println);
16
17          text.close();
18      }
19  }
```

La salida deberá mostrar el contenido de TextFile.txt

Método 2: usar `parallelStream()` en una colección

El método `parallelStream()` de la interfaz `Colección` devuelve un posible flujo paralelo con la colección como fuente. Expliquemos el funcionamiento con la ayuda de un ejemplo.

Laboratorio 5.4 Utilizar `parallel stream` mediante una Lista para leer el archivo de texto. Por lo tanto, necesitamos el método `parallelStream()`.

Nota: Aplican las mismas restricciones del Lab5.3



Lab54.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.nio.file.Files;
4  import java.util.*;
5
6  public class GFG {
7
8      public static void main(String[] args)
9          throws IOException
10     {
11
12         File fileName
13             = new File("C:\\Carpeta\\List_Textfile.txt");
14
15         List<String> text
16             = Files.readAllLines(fileName.toPath());
17
18         text.parallelStream().forEach(System.out::println);
19     }
20 }
```