COMPUTACIÓN CONCURRENTE

PRÁCTICA 6 - THREAD POOLS Y FUTURES

Prof. Manuel Alcántara Juárez
manuelalcantara52@ciencias.unam.mx

Leonardo Hernández Cano
leonardohernandezcano@ciencias.unam.mx

Ricchy Alain Pérez Chevanier
alain.chevanier@ciencias.unam.mx

Fecha de Entrega: 20 de Noviembre de 2019 a las 23:59:59pm.

1. Objetivo

El objetivo de esta práctica es resolver problemas utilizando *thread pools* y *completable futures* en Java[1].

2. Introducción

Puedes utilizar directamenete la documentación del Java para un entendimiento más profundo de estas características del lenguaje. O recomendamos alguno de los siguientes enlances:

1. https://www.baeldung.com/java-completablefuture
2. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
3. https://www.callicoder.com/java-executor-service-and-thread-pool-tutorial/

En esta práctica trabajarás con una base de código construida con Java 8[2] y Maven 3, también proveemos infraestructura para escribir pruebas unitarias con la biblioteca `Junit 5.5.1`.

[1] Los mismo conceptos existen en otros lenguajes, por ejemplo en Javascript las promesas son esencialmente iguales a un completable future en Java

[2] De nuevo puedes utilizar cualquier versión de java que sea mayor o igual a Java 8 simplemente ajustando el archivo pom.xml

En el código que recibirás junto con este documento, prodrás encontrar la clase `App` que tiene un método *main* que puedes ejecutar como cualquier programa escrito en *Java*. Para eso primero tienes que empaquetar la aplicación y después ejecutar el *jar* generado. Utiliza comandos como los siguientes:

```
$ mvn package
...
...
$ java -jar target/practica06-1.0.jar
```

Recuerda que para ejecutar las pruebas unitarias de la misma es necesario ejecutar el siguiente comando:

```
$ mvn test
```

3. Ejercicios

3.1. Merge Sort

Escribe el algoritmo de ***merge sort*** de forma paralela, de tal forma que cada llamada recursiva pueda ser atendida por un thread distinto. La idea es que utilices un `Executor` para realizar la siguiente llamada recursiva.

3.2. Suma y Multiplicación de Polinomios

Sean $P(x) = \sum_{i=0}^{d} p_i x^i$ y $Q(x) = \sum_{i=0}^{d} q_i x^i$ polinomios de grado d, donde d es una potencia de 2. Podemos expresar:

$$P(x) = P_0(x) + (P_1(x)x^{d/2}),\ Q(x) = Q_0(x) + (Q_1(x)x^{d/2})$$

Donde $P_0(x)$, $P_1(x)$, $Q_0(x)$ y $Q_1(x)$ son polinomios de grado $d/2$.

La clase `Polynomial` provee métodos `put` y `get` para acceder a los coficientes, y provee un método *split* que en tiempo constante divide el polinomio $P(x)$ en dos polinomios de grado $d/2$ como explicamos anteriormente, donde los cambios en los sub polinomios se reflejan en el polinomio original y vice versa.

Tu tarea is escribir algoritmos paralelos para las operaciones de suma y multiplicación para esta clase de polinomios.

La suma puede descomponerse de la sigueinte manera $P(x) + Q(x) = (P_0(x) + Q_0(x)) + (Q_1(x) + P_1(x))x^{d/2}$. El producto puede descomponerse como sigue $P(x) * Q(x) = (P_0(x) * Q_0(x)) + (Q_0(x) * P_1(x) + Q_1(x) * P_0(x))x^{d/2} + (Q_1(x) * P_1(x))$.

Para implentar este algoritmo paralelo, ademas de utilizar un *thread pool* por medio de un `Executor`, cada sub polinomio debe de ser regresado por un `CompletableFuture`. En el capítulo 16 del libro de *The art of multiprocessor programming* puedes encontrar un ejemplo similar a este en el que suman y multiplican matrices de tamaño nxn.

4. Actividades

En esta práctica además de implementar la solución de cada problema, tambien tendrás que implementar tus propias pruebas unitarias para este par de problemas relativamente simples. Las pruebas unitarias tienen que ser significativas y estresar casos funcionales y casos extremos. La calificación de la práctica se dividirá en 6 puntos por resolver los problemas y 4 puntos por crear una buena suite de pruebas.