Versión secuencial

Para programar la versión secuencial, he utilizado dos acercamientos. Una versión iterativa normal de toda la vida para recorrer los números, junto a un método que comprueba los divisores para ver si un dado número es o no primo.

Los resultados se añaden a una lista hasta llegar al enésimo número.

Esta versión iterativa es bastante lenta, sobre todo a medida que el orden de magnitud del número a buscar aumenta.

Por ello, he implementado también de manera secuencial un algoritmo llamado "El tamiz de Eratóstenes" que funciona algo mejor.

(ihritik, s.f.)

Cómo en esta práctica el enfoque secuencial es solo con afán comparativo, no entraré en muchos detalles sobre su codificación.

Versión concurrente

Para programar la versión concurrente, he utilizado la clase ExecutorService. Los apuntes de la IOC son un cristo nazareno, así que he ido programando según he encontrado formas de hacerlo en internet.

Antes de empezar he hecho algunas pruebas a menor escala para ver como funcionan los hilos, que se pueden ver en el paquete probandoHilos.

Acercamiento a la concurrencia

He adaptado el algoritmo original secuencial a una versión que divide las tareas por hilos para intentar hacer más rápido la ejecución del programa, no obstante creo que aún hay margen de mejora porque no es muy rápido.

Dividendo la tarea en trozos más pequeños que se puedan ejecutar de forma concurrente

Para ello, he añadido una nueva clase, que implementa la interfaz Callable, ya que necesito el valor de retorno de estas tareas, si no, hubiera usado Runnable que es ligeramente más sencillo.

Se dividen por hilos las tareas, que en este caso es procesar por segmentos los números primos hasta llegar al enésimo, que le entra por parámetro.

Anotando el tiempo

He utilizado la clase stopwatch que me parece más bonita que usar los logs.

Conclusiones

La versión concurrente, empieza a ser interesante a medida que el orden de magnitud del primo a buscar aumenta.

(Nota: Las pruebas se han hecho con 3 hilos de mi ordenador)

Para 10.001:

```
Tiempo de ejecucion secuencial normal: 8.772 ms
Tiempo de ejecucion secuencial tamiz de Erastotenes: 2.819 s
Tiempo de ejecucion concurrente: 12.62 ms
El primo es: 104743
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Para 100.001:

```
Tiempo de ejecucion secuencial normal: 147.8 ms
Tiempo de ejecucion secuencial tamiz de Erastotenes: 2.715 s
Tiempo de ejecucion concurrente: 129.4 ms
El primo es: 1299721
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Para 1000.001

```
Tiempo de ejecucion secuencial normal: 4.809 s
Tiempo de ejecucion secuencial tamiz de Erastotenes: 2.685 s
Tiempo de ejecucion concurrente: 2.183 s
El primo es: 15485867
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Si empezamos a poner números altos:

10000001

```
Tiempo de ejecucion secuencial normal: 2.502 min
Tiempo de ejecucion secuencial tamiz de Erastotenes: 2.483 s
Tiempo de ejecucion concurrente: 1.096 min
El primo es: 179424691
------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------
Total time:  03:40 min
Finished at: 2024-10-22T16:00:05+02:00
```

Podemos observar que, a mayor número, la eficiencia del programa concurrente funciona mejor. No obstante, el tamiz de Eratóstenes secuencial tarda dos segundos, sospecho que tiene algo que ver con la memoria del programa o del IDE (NetBeans).