

Nombre y apellidos (1): Vicente Ventura Ninot

Nombre y apellidos (2): Víctor-Nesta Reig Buendía

Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio):Entregable
para
Laboratorio

la06_g

Tema 08. Concurrencia en Colecciones de Java

Tema 09. El Problema de la Coordinación en Java

Se dispone de un programa secuencial que calcula y muestra la palabra más usada en un vector de tiras de caracteres, y se desea desarrollar una solución paralela a dicho problema.

En esta práctica, debes emplear las hebras normales (sin *Thread Pools*) y puedes emplear una distribución cíclica del trabajo. Además, debes ir con cuidado en las versiones concurrentes para evitar que dos hebras intenten añadir la misma palabra al mismo tiempo. **También debes evitar el uso del método `merge`, salvo que se indique lo contrario.**

No olvides comprobar que todas las versiones paralelas funcionan correctamente. Para ello debes comparar los resultados (tanto la palabra como su número de veces) de las versiones paralelas con los de la versión secuencial. En las comprobaciones es conveniente que emplees el fichero `f0.txt`, dado que su contenido puede generar más errores.

Para realizar una comparación justa, debes emplear en todas las versiones **los mismos valores de capacidad inicial y factor de carga**. Así, deberías usar 1000 como capacidad inicial y 0.75F como factor de carga.

A continuación se muestra el código secuencial del que se dispone. Como puedes ver, el cálculo secuencial se realiza dos veces, pero sólo se cuenta el tiempo y se muestran resultados para la segunda ejecución. Ello se debe a que la primera ejecución es mucho más costosa (dado que debe cargar un montón de datos en antememoria), pero, obviamente, solo hace falta realizarlo una vez al principio del programa, por lo que **no debes repetirlo** para cada implementación paralela. **Tampoco se debe comentar**, aunque sea la parte más costosa del código, ya que desvirtuaría los resultados.

```

import java.io.*;
import java.util.*;

// import java.util.concurrent.*;
// import java.util.concurrent.atomic.*;
// import java.util.Map;
// import java.util.stream.*;
// import java.util.function.*;
// import static java.util.stream.Collectors.*;
// import java.util.Comparator.*;

// =====
class EjemploPalabraMasUsada {
// =====

    // -----
    public static void main( String args[] ) {
        long          t1, t2;
        double        ts, tp;
        int           numHebras;
    }
}
```

```

String nombreFichero , palabraActual;
Vector<String> vectorLineas;
HashMap<String , Integer> hmCuentaPalabras;

// Comprobacion y extraccion de los argumentos de entrada.
if( args.length != 2 ) {
    System.err.println( "Uso: java programa <numHebras> <fichero>" );
    System.exit( -1 );
}
try {
    numHebras      = Integer.parseInt( args[ 0 ] );
    nombreFichero = args[ 1 ];
    if( numHebras <= 0 ) {
        System.err.print( "Uso: [ java programa <numHebras> <fichero> ] " );
        System.err.println( "donde ( numHebras > 0 )" );
        System.exit( -1 );
    }
} catch( NumberFormatException ex ) {
    numHebras = -1;
    nombreFichero = "";
    System.out.println( "ERROR: Argumento numerico incorrectos." );
    System.exit( -1 );
}

// Lectura y carga de lineas en "vectorLineas".
vectorLineas = leeFichero( nombreFichero );
System.out.println( "Numero de lineas leidas: " + vectorLineas.size() );
System.out.println();

//
// Implementacion secuencial sin temporizar.
//
hmCuentaPalabras = new HashMap<String , Integer>( 1000, 0.75F );
for( int i = 0; i < vectorLineas.size(); i++ ) {
    // Procesa la linea "i".
    String[] palabras = vectorLineas.get( i ).split( "\W+" );
    for( int j = 0; j < palabras.length; j++ ) {
        // Procesa cada palabra de la linea "i", si es distinta de blanco.
        palabraActual = palabras[ j ].trim();
        if( palabraActual.length() > 0 ) {
            contabilizaPalabra( hmCuentaPalabras , palabraActual );
        }
    }
}

//
// Implementacion secuencial.
//
t1 = System.nanoTime();
hmCuentaPalabras = new HashMap<String , Integer>( 1000, 0.75F );
for( int i = 0; i < vectorLineas.size(); i++ ) {
    // Procesa la linea "i".
    String[] palabras = vectorLineas.get( i ).split( "\W+" );
    for( int j = 0; j < palabras.length; j++ ) {
        // Procesa cada palabra de la linea "i", si es distinta de blanco.
        palabraActual = palabras[ j ].trim();
        if( palabraActual.length() > 0 ) {
            contabilizaPalabra( hmCuentaPalabras , palabraActual );
        }
    }
}
}

```

```

t2 = System.nanoTime();
ts = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
System.out.print( "Implementacion secuencial: " );
imprimePalabraMasUsadaYVeces( hmCuentaPalabras );
System.out.println( " Tiempo(s): " + ts );
System.out.println( "Num. elems. tabla hash: " + hmCuentaPalabras.size() );
System.out.println();

/*
// Implementacion paralela 1: Uso de synchronizedMap y cerrojo
// ...
t1 = System.nanoTime();
// ...
t2 = System.nanoTime();
tp = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
System.out.print( "Implementacion paralela 1: " );
imprimePalabraMasUsadaYVeces( maCuentaPalabras );
System.out.println( " Tiempo(s): " + tp + " , Incremento " + ... );
System.out.println( "Num. elems. tabla hash: " + ... );
System.out.println();

// Implementacion paralela 2: Uso de Hashtable y cerrojo
// ...

// Implementacion paralela 3: Uso de ConcurrentHashMap y cerrojo
// ...

// Implementacion paralela 4: Uso de ConcurrentHashMap con merge
// ...

// Implementacion paralela 5: Uso de ConcurrentHashMap escalable
// ...

// Implementacion paralela 6: Uso de CHM escalable con AtomicInteger
// ...

// Implementacion paralela 7: Uso de CHM escalable con AtomicInteger y 256 niv.
// ...

// Implementacion paralela 8: Uso de Streams
// t1 = System.nanoTime();
// Map<String ,Long> stCuentaPalabras = vectorLineas.parallelStream()
//                               .filter( s -> s != null )
//                               .map( s -> s.split( "\\\W+" ) )
//                               .flatMap( Arrays::stream )
//                               .map( String::trim )
//                               .filter( s -> (s.length() > 0) )
//                               .collect( groupingBy (s -> s, counting()) );
// t2 = System.nanoTime();
// ...

*/
System.out.println( "Fin de programa." );
}

```

```

// -----
public static Vector<String> leeFichero( String fileName ) {
    BufferedReader br;
    String linea;
    Vector<String> data = new Vector<String>();

    try {
        br = new BufferedReader( new FileReader( fileName ) );
        while( ( linea = br.readLine() ) != null ) {
            //// System.out.println( "Leida linea: " + linea );
            data.add( linea );
        }
        br.close();
    } catch( FileNotFoundException ex ) {
        ex.printStackTrace();
    } catch( IOException ex ) {
        ex.printStackTrace();
    }
    return data;
}

// -----
public static void contabilizaPalabra(
    HashMap<String , Integer> cuentaPalabras ,
    String palabra ) {
    Integer numVeces = cuentaPalabras.get( palabra );
    if( numVeces != null ) {
        cuentaPalabras.put( palabra , numVeces+1 );
    } else {
        cuentaPalabras.put( palabra , 1 );
    }
}

// -----
static void imprimePalabraMasUsadaYVeces(
    Map<String , Integer> cuentaPalabras ) {
    Vector<Map.Entry> lista =
        new Vector<Map.Entry>( cuentaPalabras.entrySet() );

    String palabraMasUsada = "";
    int numVecesPalabraMasUsada = 0;
    // Calcula la palabra mas usada.
    for( int i = 0; i < lista.size(); i++ ) {
        String palabra = ( String ) lista.get( i ).getKey();
        int numVeces = ( Integer ) lista.get( i ).getValue();
        if( i == 0 ) {
            palabraMasUsada = palabra;
            numVecesPalabraMasUsada = numVeces;
        } else if( numVecesPalabraMasUsada < numVeces ) {
            palabraMasUsada = palabra;
            numVecesPalabraMasUsada = numVeces;
        }
    }
    // Imprime resultado.
    System.out.print( "( Palabra: '" + palabraMasUsada + "' +
                      " veces: " + numVecesPalabraMasUsada + " )" );
}

// -----
static void printCuentaPalabrasOrdenadas(
```

```

        HashMap<String , Integer> cuentaPalabras ) {
    int i , numVeces;
    List<Map.Entry> list = new Vector<Map.Entry>( cuentaPalabras .entrySet() );
    // Ordena por valor.
    Collections.sort(
        list ,
        new Comparator<Map.Entry>() {
            public int compare( Map.Entry e1 , Map.Entry e2 ) {
                Integer i1 = ( Integer ) e1 .getValue();
                Integer i2 = ( Integer ) e2 .getValue();
                return i2 .compareTo( i1 );
            }
        }
    );
    // Muestra contenido .
    i = 1;
    System.out.println( "Veces Palabra" );
    System.out.println( "-----" );
    for( Map.Entry e : list ) {
        numVeces = ( ( Integer ) e.getValue () ).intValue ();
        System.out.println( i + " " + e.getKey() + " " + numVeces );
        i++;
    }
    System.out.println( "-----" );
}

```

- 1** Realiza una implementación paralela con la colección `HashMap`. Recuerda que esta colección es **no sincronizada**, por lo que puede tener problemas cuando es manejada por varias hebras. En el caso que necesites modificar el método `contabilizaPalabra`, crea una copia con otro nombre, para mantener el método utilizado en la versión secuencial. Debes asegurar que las versiones secuenciales no utilicen métodos sincronizados, para realizar análisis de costes equilibrados, Escribe a continuación el código que realiza tal tarea: la definición de la clase `MiHebra_1` y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

2] Realiza una implementación paralela con la colección Hashtable.

¿Sería posible reutilizar la clase MiHebra_1 en este ejercicio? Razona tu respuesta.

No, ya que dejaría de ser escalable

Escribe a continuación el código que realiza tal tarea: la definición de la clase `MiHebra_2`, si no se utiliza `MiHebra_1`, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

3 Realiza una implementación paralela con la colección ConcurrentHashMap utilizando un cerrojo adicional, es decir, empleando las cláusulas synchronized o static synchronized.

¿Sería posible reutilizar las clases MiHebra_1 o MiHebra_2 en este ejercicio? Razona tu respuesta.

Sí sería posible porque todas las clases implementan la misma interface Map.

Escribe a continuación el código que realiza tal tarea: la definición de la clase **MiHebra_3**, si no se utiliza **MiHebra_1** o **MiHebra_2**, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

- 4**] Realiza una implementación paralela con la colección `ConcurrentHashMap`, utilizando el método `merge`.

¿Es necesario incluir las cláusulas `synchronized` o `static synchronized`? Razona tu respuesta.

No será necesario ya que `merge` es atómico y está diseñado para este tipo de casos.

Escribe a continuación el código que realiza tal tarea: la definición de la clase `MiHebra_4`, si no se utiliza alguna clase anterior, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

5 Realiza una implementación paralela con la colección `ConcurrentHashMap`, pero sin utilizar cerrojos adicionales. En este caso, debes emplear los métodos `putIfAbsent`, `get` y `replace`. ¿Es necesario incluir las cláusulas `synchronized` o `static synchronized`? Razona tu respuesta.

No es necesario, esas operaciones también son atómicas.

Escribe a continuación el código que realiza tal tarea: la definición de la clase `MiHebra_5`, si no se utiliza alguna clase anterior, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

ATENCIÓN: Los ejercicios anteriores deben realizarse en casa. Los siguientes, en el aula.

- 6** Realiza una implementación paralela con la colección `ConcurrentHashMap` y sin uso de cerrojos adicionales, pero manejando clases atómicas. En este caso, debes emplear los métodos `putIfAbsent` y `get`, así como la clase `AtomicInteger`.

¿Sería posible reutilizar alguna de las clases anteriores en este ejercicio? Razona tu respuesta.

No se podría porque las clases anteriores trabajaban con Integer, no con AtomicInteger.

Escribe a continuación el código que realiza tal tarea: la definición de la clase `MiHebra_6`, si no se utiliza alguna clase anterior, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

7 Realiza una implementación paralela idéntica a la del apartado anterior, pero con mayor número de niveles de concurrencia, empleando 256 niveles.

¿Sería posible reutilizar alguna de las clases anteriores en este ejercicio? Razona tu respuesta.

Si que sería posible, dado que los niveles de concurrencia solo se especifican al crear el ConcurrentHashMap y no cambia nada de la hebra

Escribe a continuación el código que realiza tal tarea: la definición de la clase **MiHebra_7**, si no se utiliza alguna clase anterior, y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

8 Este apartado tiene como objetivo, desarrollar una implementación paralela basada en **Streams**. Seguidamente se muestra el código implementa esta operación mediante un **Parallel Streams**

```
Map<String ,Long> stCuentaPalabras = vectorLineas.parallelStream()
    .filter( s -> s != null )
    .map( s -> s.split( "\W+" ) )
    .flatMap( Arrays::stream )
    .map( String::trim )
    .filter( s -> (s.length() > 0) )
    .collect( groupingBy( s -> s , counting())));

```

9 Completa la siguiente tabla y justifica los resultados. Obtén los resultados para 4 hebras en el ordenador del aula y los resultados para 16 hebras en patan. Redondea los tiempos dejando sólo tres decimales y redondea los incrementos dejando dos decimales.

En ambas pruebas debes emplear el fichero `f3.txt`. Este fichero debe generarse en el script de lanzamiento utilizando el siguiente comando:

```
cat f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt > /tmp/f3.txt
```

y debe ser borrado al final del script.

```
rm /tmp/f3.txt
```

| | 4 hebras | | 16 hebras | |
|---|----------|------------|-----------|------------|
| | Tiempo | Incremento | Tiempo | Incremento |
| Secuencial | 0,0998 | — | 0,54 | — |
| Paralela con <code>HashMap</code> | 0,124 | 0,81 | 0,597 | 0,904 |
| Paralela con <code>Hashtable</code> | 0,133 | 0,75 | 0,60 | 0,9 |
| Paralela con <code>ConcurrentHashMap</code> y con cerrojo adicional | 0,154 | 0,65 | 0,585 | 0,92 |
| Paralela con <code>ConcurrentHashMap</code> y <code>merge</code> | 0,076 | 1,31 | 0,299 | 1,80 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional mediante <code>putIfAbsent</code> , <code>get</code> y <code>replace</code> | 0,116 | 0,86 | 0,356 | 1,52 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional, mediante <code>putIfAbsent</code> , <code>get</code> y <code>AtomicInteger</code> | 0,0647 | 1,54 | 0,237 | 2,28 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional mediante <code>putIfAbsent</code> , <code>get</code> y <code>AtomicInteger</code> y con más niveles | 0,0538 | 1,85 | 0,08 | 6,75 |
| Parallel Stream | 0,0898 | 1,14 | 0,176 | 3,078 |

Justifica los resultados obtenidos.

10 Completa la siguiente tabla y justifica los resultados. Obtén los resultados para 4 hebras en el ordenador del aula y los resultados para 16 hebras en patan. Redondea los tiempos dejando sólo tres decimales y redondea los incrementos dejando dos decimales.

En ambas pruebas debes emplear el fichero **f4.txt**. Este fichero debe generarse en el script de lanzamiento utilizando el siguiente comando:

```
cat f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt > /tmp/f3.txt  
cd /tmp  
cat f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt > f4.txt  
cd -
```

y debe ser borrado, junto con el resto de ficheros auxiliares, al final del script.

```
rm /tmp/f3.txt /tmp/f4.txt
```

| | 4 hebras | | 16 hebras | |
|---|----------|------------|-----------|------------|
| | Tiempo | Incremento | Tiempo | Incremento |
| Secuencial | 0,654 | — | 1,818 | — |
| Paralela con <code>HashMap</code> | 0,801 | 0,82 | 2,306 | 0,79 |
| Paralela con <code>Hashtable</code> | 0,767 | 0,86 | 2,659 | 0,68 |
| Paralela con <code>ConcurrentHashMap</code> y con cerrojo adicional | 0,596 | 1,1 | 1,795 | 1,012 |
| Paralela con <code>ConcurrentHashMap</code> y <code>merge</code> | 0,258 | 2,53 | 0,689 | 2,64 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional mediante <code>putIfAbsent</code> , <code>get</code> y <code>replace</code> | 0,328 | 1,99 | 0,864 | 2,104 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional, mediante <code>putIfAbsent</code> , <code>get</code> y <code>AtomicInteger</code> | 0,244 | 2,68 | 0,605 | 3,003 |
| Paralela con <code>ConcurrentHashMap</code> y sin cerrojo adicional mediante <code>putIfAbsent</code> , <code>get</code> y <code>AtomicInteger</code> y con más niveles | 0,236 | 2,78 | 0,442 | 4,12 |
| Parallel Stream | 0,286 | 2,21 | 0,545 | 3,34 |

Justifica los resultados obtenidos.

- 11** Completa la siguiente tabla y justifica los resultados. Obtén los resultados para 4 hebras en el ordenador del aula y los resultados para 16 hebras en patan. Redondea los tiempos dejando sólo tres decimales y redondea los incrementos dejando dos decimales.

En ambas pruebas debes emplear el fichero `f5.txt`. Este fichero debe generarse en el script de lanzamiento utilizando el siguiente comando:

```
cat f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt f1.txt f2.txt > /tmp/f3.txt
cd /tmp
cat f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt f3.txt > f4.txt
cat f4.txt f4.txt f4.txt f4.txt f4.txt f4.txt f4.txt f4.txt > f5.txt
cd -
```

y debe ser borrado, junto con el resto de ficheros auxiliares, al final del script.

```
rm /tmp/f3.txt /tmp/f4.txt /tmp/f5.txt
```

| | 4 hebras | | 16 hebras | |
|--|----------|------------|-----------|------------|
| | Tiempo | Incremento | Tiempo | Incremento |
| Secuencial | 5,379 | — | 11,515 | — |
| Paralela con HashMap | 6,193 | 0,86 | 17,953 | 0,64 |
| Paralela con Hashtable | 5,697 | 0,94 | 14,483 | 0,79 |
| Paralela con ConcurrentHashMap y con cerrojo adicional | 4,434 | 1,21 | 12,55 | 0,92 |
| Paralela con ConcurrentHashMap y merge | 1,991 | 2,70 | 3,935 | 2,93 |
| Paralela con ConcurrentHashMap y sin cerrojo adicional mediante putIfAbsent, get y replace | 2,203 | 2,44 | 4,447 | 2,59 |
| Paralela con ConcurrentHashMap y sin cerrojo adiciona, mediante putIfAbsent, get y AtomicInteger | 1,758 | 3,059 | 3,241 | 3,55 |
| Paralela con ConcurrentHashMap y sin cerrojo adicional mediante putIfAbsent, get y AtomicInteger y con más niveles | 1,809 | 2,97 | 3,129 | 3,68 |
| Parallel Stream | 1,952 | 2,85 | 3,157 | 3,65 |

Justifica los resultados obtenidos.

Las opciones con cerros son las más lentas ya que provocan bloqueos entre las hebras para acceder al recurso compartido.

Las mejores opciones son implementaciones de ConcurrentHashMap ya que esta estructura está diseñada para este tipo de situaciones en las que muchas hebras acceden al mismo recurso simultáneamente.