EI1024/MT1024 "Programación Concurrente y Paralela" 2023–24	Entregable para Laboratorio
Nombre y apellidos (1): Laura Llorens Angulo	
Nombre y apellidos (2): Martin Martinez Ramos	
Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio):	la03_g

Tema 04. El Problema de la Visibilidad en Java

Tema 05. El Problema de la Atomicidad en Java

1 Estudia el siguiente código y responde a las siguientes preguntas.

```
// ============================================================
class CuentaIncrementos {
// ============================================================
  int numIncrementos = 0;

  // ------------------------------------------------------------
  void incrementaNumIncrementos() {
    numIncrementos++;
  }

  // ------------------------------------------------------------
  int dameNumIncrementos() {
    return( numIncrementos );
  }
}


// ============================================================
class MiHebra extends Thread {
// ============================================================
  int                iters;
  CuentaIncrementos  c;

  // ------------------------------------------------------------
  public MiHebra( int iters, CuentaIncrementos c ) {
    this.iters = iters;
    this.c     = c;
  }

  // ------------------------------------------------------------
  public void run() {
    for( int i = 0; i < iters; i++ ) {
      c.incrementaNumIncrementos();
    }
  }
}

// ============================================================
class EjemploCuentaIncrementos {
// ============================================================

  // ------------------------------------------------------------
```

```
  public static void main( String args[] ) {
    long    t1, t2;
    double  tt;
    int     numHebras, iters;

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

    System.out.println( "numHebras: " + numHebras );
    System.out.println( "iters    : " + iters );

    System.out.println( "Creando y arrancando " + numHebras + " hebras." );
    t1 = System.nanoTime();
    MiHebra v[] = new MiHebra[ numHebras ];
    CuentaIncrementos c = new CuentaIncrementos();
    for( int i = 0; i < numHebras; i++ ) {
      v[ i ] = new MiHebra( iters, c );
      v[ i ].start();
    }
    for( int i = 0; i < numHebras; i++ ) {
      try {
        v[ i ].join();
      } catch( InterruptedException ex ) {
        ex.printStackTrace();
      }
    }
    t2 = System.nanoTime();
    tt = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Total de incrementos: " + c.dameNumIncrementos() );
    System.out.println( "Tiempo transcurrido en segs.: " + tt );
  }
}
```

1.1) ¿Qué realiza el código? ¿Qué debería mostrar en pantalla si se ejecutase con los parámetros hebras 4 y iters 1 000 000?

El código cuenta el total de incrementos realizados por las hebras. Si le pasamos los parametros 4 y 1000000 deberia contar 4000000 incrementos.

1.2) Compila y ejecuta el código con dichos valores en tu ordenador local. ¿Qué muestra realmente en pantalla si se ejecuta con los parámetros hebras 4 y iters 1 000 000?

Muestra menos de lo esperado, en mi ordenador muestra 2046079.

1.3) ¿Es un código *thread-safe*? Justifica tu respuesta.

No, tiene un problema de visibilidad y de atomicidad.

1.4) Crea una **copia del código original** e inserta en la copia el modificador `volatile` en la variable `numIncrementos` de la clase `CuentaIncrementos`.
A continuación, compila y prueba el nuevo código.
¿Resuelve el problema el modificador `volatile`? ¿Por qué?

No, seguimos teniendo el problema de atomicidad.

1.5) ¿Se podría resolver con el modificador `synchronized`?
Para comprobarlo, crea una **copia del código original** y aplica el modificador `synchronized` sobre **cada una** de las rutinas de la clase `CuentaIncrementos`.
Después, compila y prueba el código, antes de contestar a la pregunta anterior.
Escribe a continuación los cambios realizados en la clase `CuentaIncrementos`.

Sí, porque con synchronized evitamos problemas de visibilidad y atomicidad

```
// ======================================================
class CuentaIncrementos {
// ======================================================
  int numIncrementos = 0;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  synchronized void incrementaNumIncrementos() {
    numIncrementos++;
  }

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  synchronized int dameNumIncrementos() {
    return( numIncrementos );
  }
}
```

1.6) ¿También se podría arreglar empleando clases y operadores atómicos?

Para comprobarlo, crea otra **copia del código original**, **ELIMINA** la clase `CuentaIncrementos` y utiliza en su lugar una **clase atómica y sus métodos**.

Después, compila y prueba el código, antes de contestar la pregunta.

Escribe a continuación los cambios realizados en el código.

```
class MiHebra extends Thread {
// ======================================================================
  int               iters;
  AtomicInteger c = new AtomicInteger(0);
  public MiHebra( int iters, AtomicInteger C ) {
    this.iters = iters;
    this.c     = c;
  }
  public void run() {
    for( int i = 0; i < iters; i++ ) {
      c.getAndIncrement();
    }
  }
}
// ======================================================================
class EjemploCuentaIncrementos {
// ======================================================================
 ...
    System.out.println( "Creando y arrancando " + numHebras + " hebras." );
    t1 = System.nanoTime();
    MiHebra v[] = new MiHebra[ numHebras ];
    AtomicInteger c = new AtomicInteger(0);
    for( int i = 0; i < numHebras; i++ ) {
      v[ i ] = new MiHebra( iters, c );
      v[ i ].start();
    }
...
    t2 = System.nanoTime();
    tt = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Total de incrementos: " + c.get() );
    System.out.println( "Tiempo transcurrido en segs.: " + tt );
  }
}
```

1.7) Completa la siguiente tabla con datos de todas las versiones anteriores en tu ordenador, utilizando hebras 4 y un iters de 1 000 000. Comenta los resultados.

Código	Total incrementos
Código original	1977426
Código con `volatile`	1362825
Código con `synchronized`	4000000
Código con clases atómicas	4000000

Como el código tiene problemas de atomicidad y visibilidad con usar el modificador volatile no es suficiente, hay que usar synchronized o clases atomicas para obtener el resultado deseado.

2 Se desea imprimir en pantalla los números primos que aparecen en un vector.

El código completo es el siguiente:

```
// ============================================================================
public class EjemploMuestraPrimosEnVector {
// ============================================================================

  // --------------------------------------------------------------------------
  public static void main( String args[] ) {
    int     numHebras, vectOpt;
    boolean option = true;
    long    t1, t2;
    double  ts, tc, tb, td;

    // Comprobacion y extraccion de los argumentos de entrada.
    if( args.length != 2 ) {
      System.err.println( "Uso: java programa <numHebras> <vectorOption>" );
      System.exit( -1 );
    }
    try {
      numHebras = Integer.parseInt( args[ 0 ] );
      vectOpt   = Integer.parseInt( args[ 1 ] );
      if ( ( vectOpt != 0 ) && ( vectOpt != 1 ) ) {
        System.out.println( "ERROR: vectorOption should be 0 or 1.");
        System.exit( -1 );
      } else {
        option = (vectOpt == 0);
      }
    } catch( NumberFormatException ex ) {
      numHebras = -1;
      System.out.println( "ERROR: Argumentos numericos incorrectos." );
      System.exit( -1 );
    }

    //
    // Eleccion del vector de trabajo
    //
    VectorNumeros vn = new VectorNumeros (option);
    long vectorTrabajo[] = vn.vector;

    //
    // Implementacion secuencial.
    //
    System.out.println( "" );
    System.out.println( "Implementacion secuencial." );
    t1 = System.nanoTime();
    for( int i = 0; i < vectorTrabajo.length; i++ ) {
      if( esPrimo( vectorTrabajo[ i ] ) ) {
        System.out.println( "  Encontrado primo: " + vectorTrabajo[ i ] );
      }
    }
    t2 = System.nanoTime();
    ts = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Tiempo secuencial (seg.):                    " + ts );
/*
    //
    // Implementacion paralela ciclica.
    //
    System.out.println( "" );
    System.out.println( "Implementacion paralela ciclica." );
    t1 = System.nanoTime();
```

```
      // Gestion de hebras para la implementacion paralela ciclica
      // ....
      t2 = System.nanoTime();
      tc = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
      System.out.println( "Tiempo paralela ciclica (seg.):              " + tc );
      System.out.println( "Incremento paralela ciclica:                 " + ... );
      //
      // Implementacion paralela por bloques.
      //
      // ....
      //
      // Implementacion paralela dinamica.
      //
      // ....
*/
  }

  // --------------------------------------------------------------------------
  static boolean esPrimo( long num ) {
    boolean primo;
    if( num < 2 ) {
      primo = false;
    } else {
      primo = true;
      long i = 2;
      while( ( i < num )&&( primo ) ) {
        primo = ( num % i != 0 );
        i++;
      }
    }
    return( primo );
  }
}

// ============================================================================
class VectorNumeros {
// ============================================================================
  long    vector[];
  // --------------------------------------------------------------------------
  public VectorNumeros (boolean caso) {
    if (caso) {
      vector = new long [] {
      200000033L, 200000039L, 200000051L, 200000069L,
      200000081L, 200000083L, 200000089L, 200000093L,
      200000107L, 200000117L, 200000123L, 200000131L,
      200000161L, 200000183L, 200000201L, 200000221L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L
```

```
      };
    } else {
      vector = new long [] {
      200000033L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000039L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000051L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000069L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000081L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000083L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000089L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000093L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000107L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000117L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000123L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000131L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000161L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000183L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000201L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000221L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L
      };
    }
  }
}
```

2.1) Compila y ejecuta el programa anterior, utilizando un 0 como segundo parámetro. En este caso se trabaja con el siguiente vector de números:

```
long vectorTrabajo[] = {
      200000033L, 200000039L, 200000051L, 200000069L,
      200000081L, 200000083L, 200000089L, 200000093L,
      200000107L, 200000117L, 200000123L, 200000131L,
      200000161L, 200000183L, 200000201L, 200000221L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L
  };
```

¿Cuáles son los números primos contenidos en el vector?

. . . .

200000033, 200000039, 200000051, 200000069, 200000081, 200000083, 200000089, 200000093, 200000107, 200000117, 200000123, 200000131, 200000161, 200000183, 200000201, 200000221

. . . .

2.2) Realiza una implementación paralela con distribución cíclica, en la que cada hebra procese un conjunto de elementos del vector. Para cada elemento del vector procesado, SOLO se mostrará su valor si el número es primo.

Incluye la gestión de hebras de esta versión a continuación de la implementación secuencial.

Comprueba que los números primos mostrados en la versión paralela coinciden con los de la versión secuencial.

Escribe, a continuación, la parte de tu código que realiza tal tarea: la definición de la clase `MiHebraPrimoDistCiclica` y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

```
class HebraDistCiclica extends Thread{
  int miId, numHebras;
  long vector[];

  public HebraDistCiclica(int miId, int numHebras, long[] vector) {
    this.miId = miId;
    this.numHebras = numHebras;
    this.vector = vector;
  }

  public void run(){
    int n = vector.length;
    for(int i = miId; i < n; i += numHebras){
      if( esPrimo( vector[ i ] ) ) {
        System.out.println( "  Encontrado primo: " + vector[ i ] );
      }
    }
  }

  static boolean esPrimo( long num ) {
    boolean primo;
    if( num < 2 ) {
      primo = false;
    } else {
      primo = true;
      long i = 2;
      while( ( i < num )&&( primo ) ) {
        primo = ( num % i != 0 );
        i++;
      }
    }
    return( primo );
  }
}
```

```
...
//
// Implementacion paralela ciclica.
//
System.out.println( "" );
System.out.println( "Implementacion paralela ciclica." );
t1 = System.nanoTime();
// Gestion de hebras para la implementacion paralela ciclica
HebraDistCiclica v[] = new HebraDistCiclica[ numHebras ];
for(int i=0 ; i < numHebras ; i++){
  v[i] = new HebraDistCiclica(i, numHebras, vectorTrabajo);
  v[i].start();
}
for(int i=0 ; i < numHebras ; i++){
  try{
    v[i].join();
  }catch (InterruptedException ex){
    ex.printStackTrace();
  }
}
t2 = System.nanoTime();
tc = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
System.out.println( "Tiempo paralela ciclica (seg.):          " + tc );
System.out.println( "Incremento paralela ciclica:             " + ts/tc );
...
```

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

2.3) Realiza una implementación paralela con distribución por bloques, en la que cada hebra procese un conjunto de elementos del vector. Para cada elemento del vector procesado, SOLO se mostrará su valor si el número es primo.

Incluye la gestión de hebras de esta versión a continuación de la implementación cíclica.

Comprueba que los números primos mostrados en la versión paralela coinciden con los de la versión secuencial.

Escribe, a continuación, la parte de tu código que realiza tal tarea: la definición de la clase `MiHebraPrimoDistPorBloques` y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

```
class MiHebraPrimoDistPorBloques extends Thread{
 int miId, numHebras;
 long vector[];

 public MiHebraPrimoDistPorBloques(int miId, int numHebras, long vector[]){
  this.miId = miId;
  this.numHebras = numHebras;
  this.vector = vector;
 }

 public void run(){
  int n = vector.length;
  int tamanyo = (n + numHebras - 1) / numHebras;
  int inicio = tamanyo * miId;
  int fin = min(inicio+tamanyo,n);
  for(int i = inicio; i < fin; i++){
   if( EjemploMuestraPrimosEnVector.esPrimo( vector[ i ] ) ) {
    System.out.println( "  Encontrado primo: " + vector[ i ] );
   }
  }
 }

}
...
//
   // Implementacion paralela por bloques.
   //

   System.out.println( "" );
   System.out.println( "Implementacion paralela por bloques." );
   t1 = System.nanoTime();
   // Gestion de hebras para la implementacion paralela ciclica
   MiHebraPrimoDistPorBloques vb[] = new MiHebraPrimoDistPorBloques[ numHebras ];
   for(int i=0 ; i < numHebras ; i++){
    vb[i] = new MiHebraPrimoDistPorBloques(i, numHebras, vectorTrabajo);
    vb[i].start();
   }
   for(int i=0 ; i < numHebras ; i++){
    try{
     vb[i].join();
    }catch (InterruptedException ex){
     ex.printStackTrace();
    }
   }
   t2 = System.nanoTime();
   tb = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
   System.out.println( "Tiempo paralela ciclica (seg.):          " + tb );
   System.out.println( "Incremento paralela ciclica:             " + ts/tb );
```

2.4) Realiza una implementación paralela con distribución dinámica, que utilice un número entero atómico (`AtomicInteger`), para apuntar a una posición del vector. Las hebras recibirán un objeto de este tipo, que siempre contendrá la primera posición del vector sin procesar. Para ello, las hebras deben **realizar, de modo atómico, la lectura del valor actual y su incremento**. Las hebras finalizarán cuando el índice sobrepase la dimensión del vector. Incluye la gestión de hebras de esta versión a continuación de la implementación por bloques. Comprueba que los números primos mostrados en la versión paralela coinciden con los de la versión secuencial.

Escribe, a continuación, la parte de tu código que realiza tal tarea: la definición de la clase `MiHebraPrimoDistDinamica` y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

```
class MiHebraPrimoDistDinamica extends Thread {
  long[] vector;
  AtomicInteger index;

  public MiHebraPrimoDistDinamica(long[] vector, AtomicInteger index) {
    this.vector = vector;
    this.index = index;
  }

  @Override
  public void run() {
    int i = index.getAndIncrement();
    while(i < vector.length){
      if(EjemploMuestraPrimosEnVector.esPrimo(vector[i])) {
        System.out.println("Encontrado primo: " + vector[i]);
      }
      i = index.getAndIncrement();
    }
  }
}

    //
    // Implementacion paralela dinamica.
    //

    System.out.println( "" );
    System.out.println( "Implementacion paralela dinamica." );
    t1 = System.nanoTime();
    // Gestion de hebras para la implementacion paralela ciclica
    MiHebraPrimoDistDinamica vd[] = new MiHebraPrimoDistDinamica[ numHebras ];
    AtomicInteger index = new AtomicInteger(0);
    for(int i=0 ; i < numHebras ; i++){
      vd[i] = new MiHebraPrimoDistDinamica(vectorTrabajo, index);
      vd[i].start();
    }
    for(int i=0 ; i < numHebras ; i++){
      try{
        vd[i].join();
      }catch (InterruptedException ex){
        ex.printStackTrace();
      }
    }
    t2 = System.nanoTime();
    td = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Tiempo paralela ciclica (seg.):          " + td );
    System.out.println( "Incremento paralela ciclica:             " + ts/td );

  }
```

2.5) Completa la siguiente tabla, obteniendo 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.

	4 hebras (aula)		16 hebras (patan)	
	Tiempo	Incremento	Tiempo	Incremento
Secuencial	5.381	—		—
Paralela con distribución cíclica	1.325	4.062		
Paralela con distribución por bloques	5.055	1.065		
Paralela con distribución dinámica	1.341	4.013		

. . . . . .

2.6) Justifica los resultados de la tabla anterior.

Podemos ver que tanto la distirbución ciclica como en la dinámica es más rápida ya que se reparten los elementos del vector de manera que todas las hebras tengan más o menos el mismo trabajo. En la distribución por bloques es la hebra 0 la que hace todos los elementos más costosos de manera que solo esta hebra tarda casi lo mismo que la implementación secuencial.

2.7) Evalúa y compara las tres versiones (secuencial, paralela cíclica y paralela por bloques), pero en este caso utilizando 1 como segundo parámetro, es decir, manejando el vector:

```
long vectorTrabajo[] = {
      200000033L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000039L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000051L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000069L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000081L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000083L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000089L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000093L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000107L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000117L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000123L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000131L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000161L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000183L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000201L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L,
      200000221L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L, 4L
    };
```

Completa la siguiente tabla, obteniendo 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.

	4 hebras (aula)		16 hebras (patan)	
	Tiempo	Incremento	Tiempo	Incremento
Secuencial	5.266	—		—
Paralela con distribución cíclica	5.304	0.993		
Paralela con distribución por bloques	1.302	4.044		
Paralela con distribución dinámica	1.350	3.900		

. . . . . .

2.8) Justifica los resultados de la tabla anterior.

Podemos ver que tanto la distirbución por bloques como en la dinámica es más rápida ya que se reparten los elementos del vector de manera que todas las hebras tengan más o menos el mismo trabajo. En la distribución cíclica es la hebra 0 la que hace todos los elementos más costosos de manera que solo esta hebra tarda casi lo mismo que la implementación secuencial.

2.9) ¿Cuál es la mejor distribución con ambos vectores? Justifica tu respuesta.

Para el vector 0 es mejor cíclica o dinamica, para el vector 1 es mejor por bloques o dinamica. Como normalmente no sabemos que forma tiene el vector la mejor distribución será la dinamica.

3 Empleando el ordenador del aula, completa la siguiente tabla con datos de todas las versiones desarrolladas en el ejercicio 1, utilizando hebras 4 y un iters de 10 000 000. Redondea los tiempos dejando sólo tres decimales y comenta los resultados.

Código	Total incrementos	Tiempo transcurrido (seg.)
Código original	12214780	0.068
Código con `volatile`	12796520	0.982
Código con `synchronized`	40000000	1.141
Código con clases atómicas	40000000	1.068

En el código original hay problemas de atomicidad y visibilidad, pero al no usar modificadores como volatile o synchronized es más rápido. Con volatile seguimos teniendo problemas de atomicidad por eso sigue dando una cantidad de incrementos incorrecta pero al obligar el acceso a memoria para modificar la variable tenemos un mayor tiempo, no tanto como con los otros métodos ya que no bloquea a las hebras. En los casos de synchronized y clases atómicas obtenemos el resultado deseado con un mayor tiempo ya que se bloquean las hebras y la variable se almacena en memoria y no en antememoria.