UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Curso: CC332, Programación paralela

Práctica calificada 05

Procesamiento de datos a gran escala

20210279F Calagua Mallqui Jairo Andre

20210035J Pisfil Puicon Angello Jamir

20210164D Salcedo Alvarez Guillermo Ronie

Diciembre 9, 2023

Resumen

El trabajo se centra en la Programación Paralela para abordar el Procesamiento de Datos a Gran Escala utilizando un dataset bidimensional. Se emplean medidas como la distancia Hamming, distancia euclidiana, correlación de Pearson y correlación de Spearman para analizar patrones y tendencias en los datos. A medida que las dimensiones del dataset aumentan, se destaca la complejidad computacional, abordada eficientemente mediante la implementación en Java con Threads para aprovechar la computación paralela y mejorar la eficiencia en el manejo de grandes conjuntos de datos.

Procesamiento de datos a gran escala

1. Objetivos

Se presenta los objetivos para esta práctica:

- Análisis de los algoritmos a utilizar para mostrar análisis de tendencias.
- Implementar las versiones secuencial y paralela para estos algoritmos.
- Análisis de los resultados a través de comparaciones de tiempos.

2. Análisis teórico de los algoritmos

2.1. Hamming

La distancia de Hamming es un coeficiente que mide la diferencia entre dos cadenas de igual longitud contando el número de posiciones en las que los bits o elementos correspondientes difieren.

La fórmula para calcular dicha distancia (H) entre dos cadenas A y B de longitud N es la siguiente:

$$H(A,B) = \sum_{i=1}^{N} \delta_i(A,B)$$

En donde la función δ_i es aquella que es igual a cero cuando los elementos en la posición i son iguales, y es uno cuando son iguales.

2.2. Distancia Euclidiana

La distancia euclidiana es un coeficiente que mide la longitud del segmento que conecta dos puntos en un espacio multidimensional, tratando los elementos como coordenadas.

La fórmula para calcular dicha distancia (D) entre dos puntos P_1 y P_2 se halla como:

$$D(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

En donde $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$. Luego, este mismo caso se puede generalizar para más dimensiones.

2.3.Correlación de Pearson

La correlación de Pearson, vista como el coseno del ángulo, es una herramienta poderosa para cuantificar la relación lineal entre variables continuas. Su aplicación se extiende a diversas áreas, siendo esencial en análisis de regresión, validación de modelos y exploración de la estructura de datos.

La fórmulam que representa dicha correlación es la siguiente:

$$\cos(\alpha) = \frac{\vec{X} \cdot \vec{Y}}{|\vec{X}| \cdot |\vec{Y}|}$$

En donde $\vec{X}$ e $\vec{Y}$ son nuestras columnas de los vectores dados.

2.4.Spearman

La distancia de Spearman, también conocida como coeficiente de correlación de rangos de Spearman, es una medida de la relación monotónica entre dos conjuntos de datos.

La fórmula para el coeficiente de correlación de rangos de Spearman es r_s:

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

En donde d_i son las diferencias entre los rangos de pares de observaciones entre X e Y.

3. Análisis de las funciones implementadas

3.1.Hamming

El código calcula la distancia de Hamming entre dos filas (observaciones) en un conjunto de datos representado por una matriz bidimensional. Utiliza un bucle para comparar elementos en cada posición de las filas, incrementando un contador cuando encuentra diferencias. La función devuelve la cantidad total de posiciones en las que las filas difieren, representando así la distancia de Hamming entre ellas.

```
//============================================================
public static int Hamming(int i, int j) {
    int distancia = 0;
    for (int k = 0; k < DS_Thread.M; k++) {
        if (DS_Thread.DS[i][k] != DS_Thread.DS[j][k]) {
            distancia++;
        }
    }
    return distancia;
}
```

3.2.Distancia Euclidiana

Este código calcula la distancia euclidiana entre dos filas en un conjunto de datos representado por una matriz bidimensional. Utiliza un bucle para sumar las diferencias cuadradas entre elementos correspondientes en las filas. Luego, calcula la raíz cuadrada de la suma y devuelve este valor, que representa la distancia euclidiana entre las dos filas en el espacio multidimensional.

```
//============================================================
public static float DistEuclidiana(int i, int j) {
    int distancia = 0;
    for (int k = 0; k < DS_Thread.M; k++) {
        distancia += (DS_Thread.DS[i][k] - DS_Thread.DS[j][k]) *
                     (DS_Thread.DS[i][k] - DS_Thread.DS[j][k]);
    }
    return (float) Math.sqrt(distancia);
}
```

3.3.Correlación de Pearson

Este código calcula la correlación de Pearson entre dos filas en un conjunto de datos representado por una matriz bidimensional. Utiliza funciones auxiliares para realizar el producto escalar y calcular las normas de las filas. La correlación se obtiene dividiendo el producto escalar entre las normas. El resultado representa la fuerza y dirección de una posible relación lineal entre las dos filas.

```
//=========================================================
public static float Pearson(int i, int j) {
    return DotProduct(i, j) / (Norm(i) * Norm(j));
}

private static float DotProduct(int i, int j) {
    float acm = 0;
    for (int k = 0; k < DS_Thread.M; k++) {
        acm += DS_Thread.DS[i][k] * DS_Thread.DS[j][k];
    }
    return acm;
}

private static float Norm(int k) {
    float norm = 0;
    for (int i = 0; i < DS_Thread.M; i++) {
        norm += DS_Thread.DS[k][i] * DS_Thread.DS[k][i];
    }
    return norm;
}
```

3.4.Spearman

Este código calcula la distancia de Spearman entre dos filas en un conjunto de datos representado por una matriz bidimensional. Utiliza una función auxiliar para asignar rangos a los elementos de las filas y luego calcula la diferencia de rangos al cuadrado. La suma de estas diferencias se normaliza y se devuelve como la distancia de Spearman, proporcionando una medida de la relación monotónica entre las filas.

```java
//=========================================================
public static float Spearman(int i, int j) {
    int[] rankI = Rango(DS_Thread.DS[i]);
    int[] rankJ = Rango(DS_Thread.DS[j]);
    int n = rankI.length;

    int[] rankDiff = new int[n];
    for (int k = 0; k < n; k++) {
        rankDiff[k] = rankI[k] - rankJ[k];
    }

    float distancia = 0;
    for (int k = 0; k < n; k++) {
        distancia += rankDiff[k] * rankDiff[k];
    }
    distancia /= n;
    return distancia;
}

private static int[] Rango(int[] arr) {
    int n = arr.length;
    int[] rank = new int[n];

    for (int i = 0; i < n; i++) {
        rank[i] = 1;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (arr[i] < arr[j]) {
                rank[i]++;
            } else if (arr[i] > arr[j]) {
                rank[j]++;
            }
        }
    }
    return rank;
}
```

4. Resultados

A continuación se muestran los resultados del siguiente bloque de código para la ejecución de cada hilo más el algoritmo que quisiéramos ejecutar:

```
//-----------------------------------------------
new Thread(new Runnable() {
    public void run() {
        long inicio = System.currentTimeMillis();
        LBL1Start.setText("Thread #01");

        for (int k = 0; k < COMB; k += NUM_THREADS) {
            // float medida = Library.Pearson(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.DistEuclidiana(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
            float medida = Library.Spearman(LEFT_COL[k], RIGHT_COL[k]);

            TA1.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + " => val: " + medida + "\n");
        }

        AI1.set(0);
        long fin = System.currentTimeMillis() - inicio;
        LBL1Finish.setText("Time Execution: " + fin / 1000 + " segundos");
        System.out.println("\n");
    }
}).start();
```

Cabe destacar algunos puntos adicionales, como lo son las varibles iniciales, la creación y carga de nuestro dataset con elementos aleatorios.

```
private static final int NUM_THREADS = 4;
public static final int N = 1000;
public static final int M = 1000;
public static int[][] DS = new int[N][M];
private static final int COMB = N * (N - 1) / 2;
private static int[] LEFT_COL = new int[COMB];
private static int[] RIGHT_COL = new int[COMB];
private static Thread[] threads = new Thread[NUM_THREADS];
```

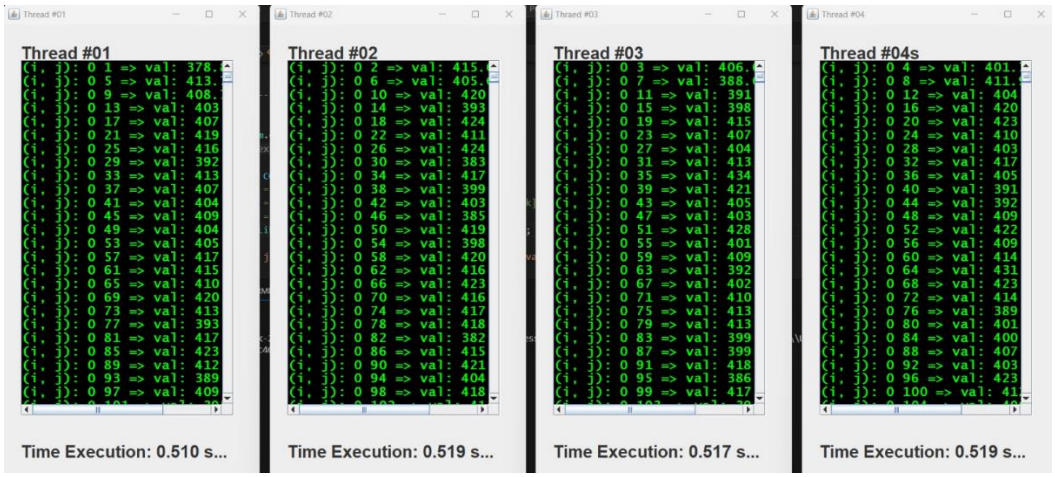
```
//===============================================
private static void LoadDataSet() {
    Random r = new Random();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            DS[i][j] = r.nextInt(50);
        }
    }
}

private static void LoadVectors() {
    int k = 0;
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            LEFT_COL[k] = i;
            RIGHT_COL[k] = j;
            k++;
        }
    }
}
```

Para ello, vamos a considerar inicialmente nuestro N = 400 y M = 400. Lo que se está realizando es la creación y carga de nuestros datasets. Y cada hilo se va a **balancear** equitativamente la cantidad de columnas. En donde cada Thread, va iterara sobre la cantidad total de combinaciones, e irá seleccionado los módulo (% i) del hilo con el que se trabaje. Adicionalmente, se han añadido los vectores LEFT y RIGHT, en los que se almacenan todas las combinaciones posibles entre las columnas.

Empezaremos a mostrar los resultados en tiempo para cada uno de nuestros coeficientes:

- Hamming:

- Distancia Euclidiana:



- Correlación de Pearson:

- Spearman:

Seguidamente, se realizarán las tablas de comparaciones de tiempo para valores de N y M diferentes, asimismo, se incluye la comparación para el **proceso serial** (equivalente a utilizar un solo hilo). Esta última implementación será de la forma:

```
//-----------------------------------------------
new Thread(new Runnable() {
    public void run() {
        long inicio = System.currentTimeMillis();
        LBL1Start.setText("Thread #01 (Serial)");

        for (int k = 0; k < COMB; k += NUM_THREADS) {
            float medida = Library2.Pearson(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.DistEuclidiana(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.Spearman(LEFT_COL[k], RIGHT_COL[k]);

            TA1.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + " => val: " + medida + "\n");
        }

        AI1.set(0);
        long fin = System.currentTimeMillis() - inicio;
        LBL1Finish.setText("Time Execution: " + fin / 1000 + " segundos");
        System.out.println("\n");
    }
}).start();
```

Y nuestra varible NUM_THREADS (en donde se indica la cantidad de hilos con la que vamos a trabajar) sería igual a uno. Sim embargo, la manera más directa para implementarlo secuencialmente será la siguiente:

```
//=============================================================================
Run | Debug
public static void main(String[] args) throws InterruptedException {
    AtomicInteger AI1 = new AtomicInteger(1);
    LoadDataSet();
    LoadVectors();
    ConfigurarControles(WDW1, WW:375, HH:800, LEFT:20, TOP:10, SP1, TA1, LBL1Start, LBL1Finish);

    long inicio = System.currentTimeMillis();
    LBL1Start.setText("Thread #01 (Serial)");

    for (int k = 0; k < COMB; k += NUM_THREADS) {
        float medida = Library2.Pearson(LEFT_COL[k], RIGHT_COL[k]);
        // float medida = Library.DistEuclidiana(LEFT_COL[k], RIGHT_COL[k]);
        // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
        // float medida = Library.Spearman(LEFT_COL[k], RIGHT_COL[k]);

        TA1.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + " => val: " + medida + "\n");
    }

    AI1.set(0);
    long fin = System.currentTimeMillis() - inicio;
    LBL1Finish.setText("Time Execution: " + fin / 1000 + " segundos");
    System.out.println("\n");
}
```

Por último se muestran las tablas de tiempo mencionadas anteriormente:

- Hamming

CORRELACIÓN DE HAMMING			PRALELOS (S)				
N	M	SERIAL (S)	HILO 1	HILO 2	HILO 3	HILO 4	Tprom
400	400	1.782	0.393	0.463	0.461	0.448	0.44125
1000	1000	2.328	0.582	0.684	0.661	0.675	0.6505
1000	2000	3.912	0.945	1.237	1.124	1.964	1.3175

- Distancia Euclidiana

CORRELACIÓN DE DISTANCIA EUCLIDIANA			PRALELOS (S)				
N	M	SERIAL (S)	HILO 1	HILO 2	HILO 3	HILO 4	Tprom
400	400	2.04	0.51	0.519	0.517	0.519	0.51625
1000	1000	6.276	1.001	1.026	1.031	0.997	1.01375
1000	2000	4.861	1.194	1.22	1.217	1.218	1.21225

- Correlación de Pearson

CORRELACIÓN DE PEARSON			PRALELOS (S)				
N	M	SERIAL (S)	HILO 1	HILO 2	HILO 3	HILO 4	Tprom
400	400	1.601	0.401	0.393	0.441	0.442	0.41925
1000	1000	5.574	1.393	1.41	1.399	1.396	1.3995
1000	2000	6.904	1.705	1.702	1.679	1.718	1.701

- Spearman

CORRELACIÓN DE SPEARMAN			PRALELOS (S)				
N	M	SERIAL (S)	HILO 1	HILO 2	HILO 3	HILO 4	Tprom
400	400	34.708	8.737	8.677	8.685	8.736	8.708
1000	1000	939.036	233.453	233.667	234.902	234.859	234.22
1000	2000	1502.457	375.614	375.503	375.567	775.682	475.5915

Podemos apreciar como para cada tabla, a medida que aumenta el **tamaño de nuestro dataset**, los tiempos también aumentarán. Aún más importante, se nota la diferencia entre los tiempos de ejecución cuando nuestro algoritmo se ejecuta, vemos que serial es mucho

mayor el tiempo que en paralelo debido a que los trabajos se realizan una tras otro; mientras que en paralelo, estamos optimizando esto y **dividiendo la carga** entre los Threads.

Otro aspecto que también podemos notar es que en nuestro coeficiente de correlación de Spearman, este toma mucho más tiempo que los demás, debido a la complejidad que tiene este algoritmo, aún así, la **proporción de tiempo** entre secuencial y paralelo se mantiene.

5. Conclusiones

En conclusión, los resultados obtenidos al comparar la ejecución secuencial y paralela de los algoritmos de distancia de Hamming, distancia euclidiana, correlación de Pearson y correlación de Spearman revelan un rendimiento significativamente mejor en la versión paralela, especialmente a medida que aumenta el tamaño del conjunto de datos.

Se destaca que la implementación paralela logra una distribución equitativa de la carga de trabajo entre los hilos, optimizando así el tiempo de ejecución. La diferencia más notable se observa en la correlación de Spearman, donde la complejidad de este algoritmo resulta en mayores tiempos de ejecución, pero la ventaja relativa de la implementación paralela se mantiene.

En términos generales, los resultados respaldan la eficacia de la programación paralela para el procesamiento de grandes conjuntos de datos, ofreciendo una mejora significativa en la eficiencia computacional. Además, la observación de que la proporción de tiempo entre las implementaciones secuencial y paralela se mantiene constante proporciona evidencia adicional de la eficacia de la paralelización, incluso en algoritmos más complejos.

Estos hallazgos sugieren que la implementación paralela es una estrategia efectiva para abordar la complejidad computacional asociada con el procesamiento de grandes

cantidades de datos, ofreciendo un rendimiento superior en comparación con la ejecución secuencial.

Anexos

DS_Thread.java

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JTextArea;


public class DS_Thread {

private static JFrame WDW1 = new JFrame("Thread #01");
private static JFrame WDW2 = new JFrame("Thread #02");
private static JFrame WDW3 = new JFrame("Thraed #03");
private static JFrame WDW4 = new JFrame("Thread #04");

private static JScrollPane SP1;
private static JScrollPane SP2;
private static JScrollPane SP3;
private static JScrollPane SP4;

private static JTextArea TA1 = new JTextArea();
private static JTextArea TA2 = new JTextArea();
private static JTextArea TA3 = new JTextArea();
private static JTextArea TA4 = new JTextArea();

private static Font fntLABEL = new Font("Arial", Font.BOLD, 24);
private static Font fntTEXT = new Font("Lucida Console", Font.BOLD, 18);

private static JLabel LBL1Start = new javax.swing.JLabel();
private static JLabel LBL1Finish = new javax.swing.JLabel();
private static JLabel LBL2Start = new javax.swing.JLabel();
private static JLabel LBL2Finish = new javax.swing.JLabel();
private static JLabel LBL3Start = new javax.swing.JLabel();
private static JLabel LBL3Finish = new javax.swing.JLabel();
private static JLabel LBL4Start = new javax.swing.JLabel();
private static JLabel LBL4Finish = new javax.swing.JLabel();

private static final int NUM_THREADS = 4;
```

```
public static final int N = 1000;
public static final int M = 1000;
public static int[][] DS = new int[N][M];
private static final int COMB = N * (N - 1) / 2;
private static int[] LEFT_COL = new int[COMB];
private static int[] RIGHT_COL = new int[COMB];
// private static Thread[] threads = new Thread[NUM_THREADS];


    //============================================================================
===
    public static void ConfigurarControles(JFrame WDW,
                                           int WW,
                                           int HH,
                                           int LEFT,
                                           int TOP,
                                           JScrollPane SP,
                                           JTextArea TA,
                                           JLabel LBLStart,
                                           JLabel LBLFinish
                                           ) {

        WDW.setSize(WW, HH);
        WDW.setLocation(LEFT, TOP);
        WDW.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        WDW.setLayout(null); //
        WDW.setVisible(true);

        LBLStart.setBounds(25, 20, 300, 40);
        LBLFinish.setBounds(25, 400, 300, 400);

        LBLStart.setFont(fntLABEL);
        LBLFinish.setFont(fntLABEL);

        TA.setEditable(false);
        TA.setBounds(35, 60, 300, 500);
        TA.setBackground(Color.WHITE);
        TA.setFont(fntTEXT);
        TA.setForeground(Color.GREEN);
        TA.setBackground(Color.BLACK);

        SP = new JScrollPane(TA);
        SP.setBounds(25, 50, 300, 500);

        WDW.add(LBLStart);
```

```
        WDW.add(SP);
        WDW.add(LBLFinish);
        WDW.setVisible(true);

    }

    //============================================================================
===
    private static void LoadDataSet() {
        Random r = new Random();
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                DS[i][j] = r.nextInt(50);
            }
        }
    }

    private static void LoadVectors() {
        int k = 0;
        for (int i = 0; i < N; i++) {
            for (int j = i + 1; j < N; j++) {
                LEFT_COL[k] = i;
                RIGHT_COL[k] = j;
                k++;
            }
        }
    }


    //============================================================================
===
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger AI1 = new AtomicInteger(1);
        AtomicInteger AI2 = new AtomicInteger(1);
        AtomicInteger AI3 = new AtomicInteger(1);
        AtomicInteger AI4 = new AtomicInteger(1);

        LoadDataSet();
        LoadVectors();

        ConfigurarControles(WDW1, 375, 800, 20, 10, SP1, TA1, LBL1Start,
LBL1Finish);
        ConfigurarControles(WDW2, 375, 800, 395, 10, SP2, TA2, LBL2Start,
LBL2Finish);
```

```
        ConfigurarControles(WDW3, 375, 800, 770, 10, SP3, TA3, LBL3Start,
LBL3Finish);
        ConfigurarControles(WDW4, 375, 800, 1145,10, SP4, TA4, LBL4Start,
LBL4Finish);

        //-------------------------------------------------
        new Thread(new Runnable() {
            public void run() {
                long inicio = System.currentTimeMillis();
                LBL1Start.setText("Thread #01");

                for (int k = 0; k < COMB; k += NUM_THREADS) {
                    float medida = Library.Pearson(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.DistEuclidiana(LEFT_COL[k],
RIGHT_COL[k]);
                    // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.Spearman(LEFT_COL[k],
RIGHT_COL[k]);

                    TA1.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + "
=> val: " + medida + "\n");
                }

                AI1.set(0);
                long fin = System.currentTimeMillis() - inicio;
                LBL1Finish.setText("Time Execution: " + fin / 1000 + "
segundos");
                System.out.println("\n");
            }
        }).start();

        //-------------------------------------------------
        new Thread(new Runnable() {
            public void run() {
                long inicio = System.currentTimeMillis();
                LBL2Start.setText("Thread #02");

                for (int k = 1; k < COMB; k += NUM_THREADS) {
                    float medida = Library.Pearson(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.DistEuclidiana(LEFT_COL[k],
RIGHT_COL[k]);
                    // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.Spearman(LEFT_COL[k],
RIGHT_COL[k]);
```

```
                    TA2.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + "
=> val: " + medida + "\n");
                }

                AI2.set(0);
                long fin = System.currentTimeMillis() - inicio;
                LBL2Finish.setText("Time Execution: " + fin / 1000 + "
segundos");
                System.out.println("\n");
            }
        }).start();

        //-------------------------------------------------
        new Thread(new Runnable() {
            public void run() {
                long inicio = System.currentTimeMillis();
                LBL3Start.setText("Thread #03");

                for (int k = 2; k < COMB; k += NUM_THREADS) {
                    float medida = Library.Pearson(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.DistEuclidiana(LEFT_COL[k],
RIGHT_COL[k]);
                    // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.Spearman(LEFT_COL[k],
RIGHT_COL[k]);

                    TA3.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + "
=> val: " + medida + "\n");
                }

                AI3.set(0);
                long fin = System.currentTimeMillis() - inicio;
                LBL3Finish.setText("Time Execution: " + fin / 1000 + "
segundos");
                System.out.println("\n");
            }
        }).start();

        //-------------------------------------------------
        new Thread(new Runnable() {
            public void run() {
                long inicio = System.currentTimeMillis();
                LBL4Start.setText("Thread #04s");

                for (int k = 3; k < COMB; k += NUM_THREADS) {
```

```
                    float medida = Library.Pearson(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.DistEuclidiana(LEFT_COL[k],
RIGHT_COL[k]);
                    // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
                    // float medida = Library.Spearman(LEFT_COL[k],
RIGHT_COL[k]);

                    TA4.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + "
=> val: " + medida + "\n");
                }

                AI4.set(0);
                long fin = System.currentTimeMillis() - inicio;
                LBL4Finish.setText("Time Execution: " + fin / 1000 + "
segundos");
                System.out.println("\n");
            }
        }).start();
    }
}
```

Library.java

```
public class Library {

    //============================================================
    public static int Hamming(int i, int j) {
        int distancia = 0;
        for (int k = 0; k < DS_Thread.M; k++) {
            if (DS_Thread.DS[i][k] != DS_Thread.DS[j][k]) {
                distancia++;
            }
        }
        return distancia;
    }


    //============================================================
    public static float DistEuclidiana(int i, int j) {
        int distancia = 0;
        for (int k = 0; k < DS_Thread.M; k++) {
            distancia += (DS_Thread.DS[i][k] - DS_Thread.DS[j][k]) *
```

```
                        (DS_Thread.DS[i][k] - DS_Thread.DS[j][k]);
        }
        return (float) Math.sqrt(distancia);
    }


    //============================================================
    public static float Pearson(int i, int j) {
        return DotProduct(i, j) / (Norm(i) * Norm(j));
    }

    private static float DotProduct(int i, int j) {
        float acm = 0;
        for (int k = 0; k < DS_Thread.M; k++) {
            acm += DS_Thread.DS[i][k] * DS_Thread.DS[j][k];
        }
        return acm;
    }

    private static float Norm(int k) {
        float norm = 0;
        for (int i = 0; i < DS_Thread.M; i++) {
            norm += DS_Thread.DS[k][i] * DS_Thread.DS[k][i];
        }
        return norm;
    }


    //============================================================
    public static float Spearman(int i, int j) {
        int[] rankI = Rango(DS_Thread.DS[i]);
        int[] rankJ = Rango(DS_Thread.DS[j]);
        int n = rankI.length;

        int[] rankDiff = new int[n];
        for (int k = 0; k < n; k++) {
            rankDiff[k] = rankI[k] - rankJ[k];
        }

        float distancia = 0;
        for (int k = 0; k < n; k++) {
            distancia += rankDiff[k] * rankDiff[k];
        }
        distancia /= n;
        return distancia;
```

```
    }

    private static int[] Rango(int[] arr) {
        int n = arr.length;
        int[] rank = new int[n];

        for (int i = 0; i < n; i++) {
            rank[i] = 1;
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (arr[i] < arr[j]) {
                    rank[i]++;
                } else if (arr[i] > arr[j]) {
                    rank[j]++;
                }
            }
        }
        return rank;
    }
}
```

DS_Serial.java

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JTextArea;


public class DS_Serial {

private static JFrame WDW1 = new JFrame("Thread #01 (Serial)");
private static JScrollPane SP1;
private static JTextArea TA1 = new JTextArea();


private static Font fntLABEL = new Font("Arial", Font.BOLD, 24);
```

```
private static Font fntTEXT = new Font("Lucida Console", Font.BOLD, 18);

private static JLabel LBL1Start = new javax.swing.JLabel();
private static JLabel LBL1Finish = new javax.swing.JLabel();

private static final int NUM_THREADS = 1;
public static final int N = 1000;
public static final int M = 1000;
public static int[][] DS = new int[N][M];
private static final int COMB = N * (N - 1) / 2;
private static int[] LEFT_COL = new int[COMB];
private static int[] RIGHT_COL = new int[COMB];
// private static Thread[] threads = new Thread[NUM_THREADS];


    //============================================================================
===
    public static void ConfigurarControles(JFrame WDW,
                                           int WW,
                                           int HH,
                                           int LEFT,
                                           int TOP,
                                           JScrollPane SP,
                                           JTextArea TA,
                                           JLabel LBLStart,
                                           JLabel LBLFinish
                                           ) {

        WDW.setSize(WW, HH);
        WDW.setLocation(LEFT, TOP);
        WDW.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        WDW.setLayout(null); //
        WDW.setVisible(true);

        LBLStart.setBounds(25, 20, 300, 40);
        LBLFinish.setBounds(25, 400, 300, 400);

        LBLStart.setFont(fntLABEL);
        LBLFinish.setFont(fntLABEL);

        TA.setEditable(false);
        TA.setBounds(35, 60, 300, 500);
        TA.setBackground(Color.WHITE);
        TA.setFont(fntTEXT);
        TA.setForeground(Color.GREEN);
```

```
        TA.setBackground(Color.BLACK);

        SP = new JScrollPane(TA);
        SP.setBounds(25, 50, 300, 500);

        WDW.add(LBLStart);
        WDW.add(SP);
        WDW.add(LBLFinish);
        WDW.setVisible(true);

    }

    //============================================================================
===
    private static void LoadDataSet() {
        Random r = new Random();
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                DS[i][j] = r.nextInt(50);
            }
        }
    }

    private static void LoadVectors() {
        int k = 0;
        for (int i = 0; i < N; i++) {
            for (int j = i + 1; j < N; j++) {
                LEFT_COL[k] = i;
                RIGHT_COL[k] = j;
                k++;
            }
        }
    }


    //============================================================================
===
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger AI1 = new AtomicInteger(1);
        LoadDataSet();
        LoadVectors();
        ConfigurarControles(WDW1, 375, 800, 20, 10, SP1, TA1, LBL1Start,
LBL1Finish);

        long inicio = System.currentTimeMillis();
```

```
        LBL1Start.setText("Thread #01 (Serial)");

        for (int k = 0; k < COMB; k += NUM_THREADS) {
            float medida = Library2.Pearson(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.DistEuclidiana(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.Hamming(LEFT_COL[k], RIGHT_COL[k]);
            // float medida = Library.Spearman(LEFT_COL[k], RIGHT_COL[k]);

            TA1.append("(i, j): " + LEFT_COL[k] + " " + RIGHT_COL[k] + " => val:
" + medida + "\n");
        }

        AI1.set(0);
        long fin = System.currentTimeMillis() - inicio;
        LBL1Finish.setText("Time Execution: " + fin / 1000 + " segundos");
        System.out.println("\n");
    }
}
```

Library2.java

```
public class Library2 {

    //============================================================
    public static int Hamming(int i, int j) {
        int distancia = 0;
        for (int k = 0; k < DS_Serial.M; k++) {
            if (DS_Serial.DS[i][k] != DS_Serial.DS[j][k]) {
                distancia++;
            }
        }
        return distancia;
    }


    //============================================================
    public static float DistEuclidiana(int i, int j) {
        int distancia = 0;
        for (int k = 0; k < DS_Serial.M; k++) {
            distancia += (DS_Serial.DS[i][k] - DS_Serial.DS[j][k]) *
                         (DS_Serial.DS[i][k] - DS_Serial.DS[j][k]);
        }
```

```
        return (float) Math.sqrt(distancia);
    }


    //============================================================
    public static float Pearson(int i, int j) {
        return DotProduct(i, j) / (Norm(i) * Norm(j));
    }

    private static float DotProduct(int i, int j) {
        float acm = 0;
        for (int k = 0; k < DS_Serial.M; k++) {
            acm += DS_Serial.DS[i][k] * DS_Serial.DS[j][k];
        }
        return acm;
    }

    private static float Norm(int k) {
        float norm = 0;
        for (int i = 0; i < DS_Serial.M; i++) {
            norm += DS_Serial.DS[k][i] * DS_Serial.DS[k][i];
        }
        return norm;
    }


    //============================================================
    public static float Spearman(int i, int j) {
        int[] rankI = Rango(DS_Serial.DS[i]);
        int[] rankJ = Rango(DS_Serial.DS[j]);
        int n = rankI.length;

        int[] rankDiff = new int[n];
        for (int k = 0; k < n; k++) {
            rankDiff[k] = rankI[k] - rankJ[k];
        }

        float distancia = 0;
        for (int k = 0; k < n; k++) {
            distancia += rankDiff[k] * rankDiff[k];
        }
        distancia /= n;
        return distancia;
    }
```

```
    private static int[] Rango(int[] arr) {
        int n = arr.length;
        int[] rank = new int[n];

        for (int i = 0; i < n; i++) {
            rank[i] = 1;
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (arr[i] < arr[j]) {
                    rank[i]++;
                } else if (arr[i] > arr[j]) {
                    rank[j]++;
                }
            }
        }
        return rank;
    }
}
```