Taller 6

MAYRA ALEJANDRA SANCHEZ SALINAS (2040506)

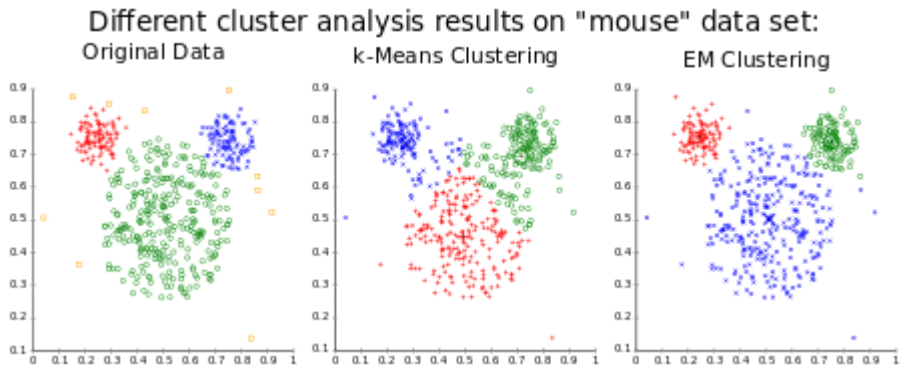
FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN FRANCISCO DIAZ FRIAS

UNIVERSIDAD DEL VALLE
SANTIAGO DE CALI, VALLE DEL CAUCA
2022

Algoritmo KMeans

K-medias es un método de agrupamiento, que tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo cuyo valor medio es más cercano. Es un método utilizado en minería de datos.



Implementar cada una de las fases del algoritmo, tanto en su versión para colecciones secuenciales como en su versi´on para colecciones concurrentes

1. **Clasificando los puntos**

   La clase Punto se usa para representar los vectores de entrada del algoritmo

```
class Punto(val x: Double, val y: Double, val z: Double) {
  private def cuadrado(v: Double): Double = v * v
  def distanciaAlCuadrado(that: Punto): Double = {
    cuadrado(that.x - x)  + cuadrado(that.y - y) + cuadrado(that.z - z)
  }
  private def round(v: Double): Double = (v * 100).toInt / 100.0
  override def toString = s"(${round(x)}, ${round(y)}, ${round(z)})"
}
```

   Se implementa la función de clasificación para la versión secuencuencial y para la versión concurrente

```
def clasificarPar(puntos: ParSeq[Punto], medianas: ParSeq[Punto]): ParMap[Punto, ParSeq[Punto]] = {
  val pointsGroupByMeans = puntos.groupBy(hallarPuntoMasCercano(_, medianas))
  medianas.map(mean => (mean, pointsGroupByMeans.getOrElse(mean, ParSeq()))).toMap
}

def clasificarSeq(puntos: Seq[Punto], medianas: Seq[Punto]): Map[Punto, Seq[Punto]] = {
  val pointsGroupByMeans = puntos.groupBy(hallarPuntoMasCercano(_, medianas))
  medianas.map(mean => (mean, pointsGroupByMeans.getOrElse(mean, Seq()))).toMap
}
```

   La ayuda que se utilizará es la siguiente función:

```
// clasificar
def hallarPuntoMasCercano(p: Punto, medianas: IterableOnce[Punto]): Punto = {
  val it = medianas.iterator
  assert(it.nonEmpty)
  var puntoMasCercano = it.next( )
  var minDistancia = p.distanciaAlCuadrado( puntoMasCercano )
  while ( it.hasNext ) {
    val point = it.next( )
    val distancia = p.distanciaAlCuadrado(point)
    if ( distancia < minDistancia ) {
      minDistancia = distancia
      puntoMasCercano = point
    }
  }
  puntoMasCercano
}
```

2. Actualizando medianas

Se implementará la fase de actualización de las medianas correspondientes a cada cluster

```
def actualizarPar ( clasif : ParMap [ Punto , ParSeq [ Punto ] ] , medianasViejas: ParSeq [ Punto ] ): ParSeq [ Punto ] = {
  medianasViejas.par.map(oldMean => hallarPuntoMasCercano(oldMean, clasif(oldMean)))
}

def actualizarSeq ( clasif : Map [ Punto , Seq [ Punto ] ] , medianasViejas: Seq [ Punto ] ): Seq [ Punto ] = {
  medianasViejas.map(oldMean => hallarPuntoMasCercano(oldMean, clasif(oldMean)))
}
```

Estas funciones, reciben la asociacion (map) de los puntos clasificados en el punto anterior como colecciones secuencial o paralela, y la colección secuencial o paralela de las medianas vigentes, y devuelven la nueva colección secuencial o paralela de medianas actualizadas según lo mencionado en la descripción de esa etapa.

Las ayudas que se utilizaran son las siguientes dos funciones:

```
def calcularPromedioPar ( oldMean : Punto , points : ParSeq [ Punto ] ): Punto = {
  if ( points.isEmpty ) oldMean
  else {
    var x = 0.0
    var y = 0.0
    var z = 0.0
    points.seq.foreach{ p =>
      x += p.x
      y += p.y
      z += p.z
    }
    new Punto ( x / points.length, y / points.length , z / points.length )
  }
}

def calcularPromedioSeq ( oldMean : Punto , points : Seq [ Punto ] ): Punto = {
  if ( points.isEmpty ) oldMean
  else {
    var x = 0.0
    var y = 0.0
    var z = 0.0
    points.foreach{ p =>
      x += p.x
      y += p.y
      z += p.z
    }
    new Punto ( x / points.length, y / points.length , z / points.length )
  }
}
```

3. Detectando convergencia

Se implementa el criterio de detección de convergencia del algoritmo para los dos tipos de colecciones: secuencial y paralela, este algoritmo converge si dadas dos colecciones de secuencias, las viejas y las nuevas, la distancia entre cada mediana nueva y su correspondiente mediana antigua, es inferior a un valor eta dado.

```
def hayConvergenciaPar ( eta : Double , medianasViejas : ParSeq [ Punto ] , medianasNuevas : ParSeq [ Punto ] ) : Boolean ={
  (medianasViejas zip medianasNuevas).forall{
    case (medianasViejas, medianasNuevas) => medianasViejas.distanciaAlCuadrado(medianasNuevas) <=  eta
  }
}

def hayConvergenciaSeq ( eta : Double , medianasViejas: Seq [ Punto ] , medianasNuevas : Seq [ Punto ] ) : Boolean ={
  (medianasViejas zip medianasNuevas).forall{
    case (medianasViejas, medianasNuevas) => medianasViejas.distanciaAlCuadrado(medianasNuevas) <=  eta
  }
}
```

Las funciones que detectan la convergencia reciben entonces el parámetro eta y las dos colecciones de medianas (la antigua y la nueva) y devuelve un booleano indicando si hay convergencia o no

4. **Implementando el algoritmo KMeans**

   Implementar las funciones:

   - kMedianasSeq

```
@tailrec
final def  kMedianasSeq ( puntos : Seq [ Punto ] , medianas : Seq [ Punto ] , eta : Double ) : Seq [ Punto ] ={
  val classified = clasificarSeq(puntos, medianas)
  val newMeans = actualizarSeq(classified, medianas)
  if (!hayConvergenciaSeq(eta,medianas, newMeans)) kMedianasSeq(puntos, newMeans, eta) else newMeans
}
```

   - kMedianasPar

```
@tailrec
final def kMedianasPar ( puntos : ParSeq [ Punto ] , medianas : ParSeq [ Punto ] , eta: Double ) : ParSeq [ Punto ] ={
  val classified = clasificarPar(puntos, medianas)
  val newMeans = actualizarPar(classified, medianas)
  if (!hayConvergenciaPar(eta,medianas, newMeans)) kMedianasPar(puntos, newMeans, eta) else newMeans
}
```

   Estas funciones reciben una colección secuencial o paralela de puntos, y una colección secuencial o paralela de medianas iniciales, y devuelven la colección de medianas correspondientes a cada cluster, es decir las medianas correspondientes a aplicar el algoritmo kmeans correctamente, es decir hasta que se cumpla el criterio de convergencia.

5. **Corriendo el algoritmo**

   Finalmente para correr el algoritmo, y probarlo, y analizar el efecto de usar colecciones paralelas o no, incluya en su paquete las siguiente funciones:

```scala
def generarPuntosSeq(k: Int, num: Int): Seq[Punto] = {
  val randx = new Random(1)
  val randy = new Random(3)
  val randz = new Random(5)
  (0 ≤ until < num)
    .map({ i =>
      val x = ((i + 1) % k) * 1.0 / k + randx.nextDouble() * 0.5
      val y = ((i + 5) % k) * 1.0 / k + randy.nextDouble() * 0.5
      val z = ((i + 7) % k) * 1.0 / k + randz.nextDouble() * 0.5
      new Punto(x, y, z)
    }).to(mutable.ArrayBuffer)
}

def generarPuntosPar(k: Int, num: Int): ParSeq[Punto] = {
  val randx = new Random(1)
  val randy = new Random(3)
  val randz = new Random(5)
  (0 ≤ until < num)
    .map({ i =>
      val x = ((i + 1) % k) * 1.0 / k + randx.nextDouble() * 0.5
      val y = ((i + 5) % k) * 1.0 / k + randy.nextDouble() * 0.5
      val z = ((i + 7) % k) * 1.0 / k + randz.nextDouble() * 0.5
      new Punto(x, y, z)
    }).to(mutable.ArrayBuffer)
}
```

```scala
def inicializarMedianasPar ( k: Int , puntos : ParSeq [ Punto ] ) : ParSeq [ Punto ] ={
  val rand = new Random(7)
  ((0 ≤ until < k).map(_ => puntos(rand.nextInt(puntos.length))).to(mutable.ArrayBuffer))
}

def inicializarMedianasSeq ( k: Int , puntos : Seq [ Punto ] ) : Seq [ Punto ] ={
  val rand = new Random(7)
  (0 ≤ until < k).map(_ => puntos(rand.nextInt(puntos.length))).to(mutable.ArrayBuffer)
}
```

Informe de uso de colecciones concurrentes

Tabla: Se usó colecciones paralelas en las siguientes funciones

Tipo de colección	Funciones	¿Por qué lo uso?
Secuencial	• KMedianasSeq • hayConvergenciaSeq • actualizarSeq • clasificarSeq	Se usaron colecciones secuenciales ya que estas son ordenadas, y pueden ser consultadas usando enteros, es una colección mutable
Paralela	• KMedianasPar • hayConvergenciaPar • actualizarPar • clasificarPar	Se hizo uso de las colecciones paralelas usando el método .par, esta colección es más eficiente que la secuencial

Apreciación: Me parece que las colecciones paralelas es un buen método ya que simplemente con un método se puede pasar una colección de scala a una colección paralela, algo que se tiene en cuenta es que no copian elementos y su conversión es rápida

Informe de corrección

- **Argumentación sobre la corrección**

  ➔ clasificarPar
  ➔ actualizarPar
  ➔ hayConvergenciaPar
  ➔ kMedianasPar

  Todas estas funciones son correctas porque usan el método de colección paralela, es decir, simplemente se copia lo secuencial sin cambiar nada de la estructura, solamente agregando el método .par

- **Casos de prueba**

  Se definen las variables de 5 puntos, 5 clusters y 5 eta's

```
//Puntos
val numPuntos1 = 500000
val numPuntos2 = 600000
val numPuntos3 = 100000
val numPuntos4 = 50000
val numPuntos5 = 90000

//Kluseh 2, 4, 8, 16 y 32
val k1 = 2
val k2 = 4
val k3 = 8
val k4 = 16
val k5 = 32

// Valores eta
val eta1 = 0.01
val eta2 = 0.02
val eta3 = 0.03
val eta4 = 0.04
val eta5 = 0.5
```

  Creando los puntos y medianas de forma secuencial y paralela

```
//Crear puntos seq
val puntosSeq1 = generarPuntosSeq (k1, numPuntos1)
val puntosSeq2 = generarPuntosSeq (k2, numPuntos2)
val puntosSeq3 = generarPuntosSeq (k3, numPuntos3)
val puntosSeq4 = generarPuntosSeq (k4, numPuntos4)
val puntosSeq5 = generarPuntosSeq (k5, numPuntos5)

//Crear medianas seq
val medianasSeq1 = inicializarMedianasSeq (k1, puntosSeq1)
val medianasSeq2 = inicializarMedianasSeq (k2, puntosSeq2)
val medianasSeq3 = inicializarMedianasSeq (k3, puntosSeq3)
val medianasSeq4 = inicializarMedianasSeq (k4, puntosSeq4)
val medianasSeq5 = inicializarMedianasSeq (k5, puntosSeq5)

//Crear puntos par
val puntosPar1 = generarPuntosPar ( k1 , numPuntos1 )
val puntosPar2 = generarPuntosPar ( k2 , numPuntos2 )
val puntosPar3 = generarPuntosPar ( k3 , numPuntos3 )
val puntosPar4 = generarPuntosPar ( k4 , numPuntos4 )
val puntosPar5 = generarPuntosPar ( k5, numPuntos5 )

//Crear medianas par
val medianasPar1 = inicializarMedianasPar(k1 , puntosPar1 )
val medianasPar2 = inicializarMedianasPar(k2 , puntosPar2 )
val medianasPar3 = inicializarMedianasPar(k3 , puntosPar3 )
val medianasPar4 = inicializarMedianasPar(k4 , puntosPar4 )
val medianasPar5 = inicializarMedianasPar(k5 , puntosPar5 )
```

Para clasificar de forma secuencial y paralela

```
//Clasificar
val calificar_S1 = clasificarSeq(puntosSeq1,medianasSeq1)
val calificar_S2 = clasificarSeq(puntosSeq2,medianasSeq2)
val calificar_S3 = clasificarSeq(puntosSeq3,medianasSeq3)
val calificar_S4 = clasificarSeq(puntosSeq4,medianasSeq4)
val calificar_S5 = clasificarSeq(puntosSeq5,medianasSeq5)

val calificar_P1 = clasificarPar(puntosPar1,medianasPar1)
val calificar_P2 = clasificarPar(puntosPar2,medianasPar2)
val calificar_P3 = clasificarPar(puntosPar3,medianasPar3)
val calificar_P4 = clasificarPar(puntosPar4,medianasPar4)
val calificar_P5 = clasificarPar(puntosPar5,medianasPar5)
```

Para actualizar de forma secuencial y paralela

```
//Actualizar
val Actualizar_S1 = actualizarSeq(calificar_S1,medianasSeq1)
val Actualizar_S2 = actualizarSeq(calificar_S2,medianasSeq2)
val Actualizar_S3 = actualizarSeq(calificar_S3,medianasSeq3)
val Actualizar_S4 = actualizarSeq(calificar_S4,medianasSeq4)
val Actualizar_S5 = actualizarSeq(calificar_S5,medianasSeq5)

val actualizar_P1 = actualizarPar(calificar_P1,medianasPar1)
val actualizar_P2 = actualizarPar(calificar_P2,medianasPar2)
val actualizar_P3 = actualizarPar(calificar_P3,medianasPar3)
val actualizar_P4 = actualizarPar(calificar_P4,medianasPar4)
val actualizar_P5 = actualizarPar(calificar_P5,medianasPar5)
```

Saber si hay convergencia

```
//hayConvergencia
val convergencia_S1 = hayConvergenciaSeq(eta1,medianasSeq1,Actualizar_S1)
val convergencia_S2 = hayConvergenciaSeq(eta2,medianasSeq2,Actualizar_S2)
val convergencia_S3 = hayConvergenciaSeq(eta3,medianasSeq3,Actualizar_S3)
val convergencia_S4 = hayConvergenciaSeq(eta4,medianasSeq4,Actualizar_S4)
val convergencia_S5 = hayConvergenciaSeq(eta5,medianasSeq5,Actualizar_S5)

val convergencia_P1 = hayConvergenciaPar(eta1,medianasPar1,actualizar_P1)
val convergencia_P2 = hayConvergenciaPar(eta2,medianasPar2,actualizar_P2)
val convergencia_P3 = hayConvergenciaPar(eta3,medianasPar3,actualizar_P3)
val convergencia_P4 = hayConvergenciaPar(eta4,medianasPar4,actualizar_P4)
val convergencia_P5 = hayConvergenciaPar(eta5,medianasPar5,actualizar_P5)
```

La convergencia arroja resultados booleanos

```
val convergencia_S1: Boolean = true
val convergencia_S2: Boolean = true
val convergencia_S3: Boolean = true
val convergencia_S4: Boolean = true
val convergencia_S5: Boolean = true

val convergencia_P1: Boolean = true
val convergencia_P2: Boolean = true
val convergencia_P3: Boolean = true
val convergencia_P4: Boolean = true
val convergencia_P5: Boolean = true
```

Finalmente la Kmedianas

```
//kMedianas
val tiempo_S1 = standardConfig measure { kMedianasSeq(puntosSeq1,medianasSeq1,eta1) }
val tiempo_S2 = standardConfig measure { kMedianasSeq(puntosSeq2,medianasSeq2,eta2) }
val tiempo_S3 = standardConfig measure { kMedianasSeq(puntosSeq3,medianasSeq3,eta3) }
val tiempo_S4 = standardConfig measure { kMedianasSeq(puntosSeq4,medianasSeq4,eta4) }
val tiempo_S5 = standardConfig measure { kMedianasSeq(puntosSeq5,medianasSeq5,eta5) }


val tiempo_P1 = standardConfig measure { kMedianasPar(puntosPar1,medianasPar1,eta1) }
val tiempo_P2 = standardConfig measure { kMedianasPar(puntosPar2,medianasPar2,eta2) }
val tiempo_P3 = standardConfig measure { kMedianasPar(puntosPar3,medianasPar3,eta3) }
val tiempo_P4 = standardConfig measure { kMedianasPar(puntosPar4,medianasPar4,eta4) }
val tiempo_P5 = standardConfig measure { kMedianasPar(puntosPar5,medianasPar5,eta5) }
```

Calcula los tiempos para arrojar el siguiente resultado

```
Unable to create a system terminal val tiempo_S1: org.scalameter.Quantity[D
val tiempo_S2: org.scalameter.Quantity[Double] = 143.38278248 ms
val tiempo_S3: org.scalameter.Quantity[Double] = 32.58166532 ms
val tiempo_S4: org.scalameter.Quantity[Double] = 22.17177288 ms
val tiempo_S5: org.scalameter.Quantity[Double] = 53.6509244 ms


val tiempo_P1: org.scalameter.Quantity[Double] = 74.54719096000001 ms
val tiempo_P2: org.scalameter.Quantity[Double] = 87.95430040000001 ms
val tiempo_P3: org.scalameter.Quantity[Double] = 23.593983840000007 ms
val tiempo_P4: org.scalameter.Quantity[Double] = 21.501392119999995 ms
val tiempo_P5: org.scalameter.Quantity[Double] = 34.41396248 ms
```

Evaluación comparativa de las soluciones paralelas versus las secuenciales

Haga una tabla con los datos recolectados y analice los resultados

- Paralelo

#	puntoPar	medianaPar	eta	Resultado
1	((0.86, 0.86, 0.86), (0.2, 0.03, 0.04), (0.6, 0.53, 0.74), (0.16, 0.38, 0.23), (0.98, 0.61, 0.72), (0.0, 0.33, 0.34), (0.98, 0.9, 0.63), (0.46, 0.01, 0.37), (0.97, 0.9, 0.6), (0.46, 0.47, 0.46), (0.69, 0.57, 0.94), (0.17, 0.45, 0.11), (0.64, 0.99, 0.94), (0.25, 0.26, 0.21), (0.55, 0.98, 0.69), (0.38, 0.07, 0.49),	((0.76, 0.55, 0.67), (0.97, 0.94, 0.51))	0.01	74.54719096000001 ms

	(0.82, 0.77, 0.94), (0.07, 0.04, 0.14), (0.68, 0.72, 0.62), (0.06, 0.2, 0.29), (0.84, 0.64, 0.75), (0.4, 0.08, 0.03), (0.5, 0.74, 0.52), (0.26, 0.47, 0.34), (0.87, 0.94, 0.72), (0.07, 0.26, 0.17), (0.74, 0.78, 0.65), (0.27, 0.12, 0.06), (0.78, 0.84, 0.67), (0.1, 0.48, 0.34), (0.81, 0.66, 0.6), (0.09, 0.13, 0.01), (0.5, 0.7, 0.59), (0.08, 0.06, 0.11), (0.58, 0.87, 0.51), (0.27, 0.23, 0.34), (0.98, 0.58, ...			
2	((0.61, 0.61, 1.11), (0.7, 0.53, 0.04), (0.85, 0.78, 0.49), (0.16, 0.38, 0.73), (0.73, 0.36, 0.97), (0.5, 0.83, 0.34), (1.23, 1.15,	((0.51, 0.3, 0.92), (0.72, 0.69, 0.76), (0.65, 0.84, 0.26), (0.64, 0.29, 0.82))	0.02	87.95430040000001 ms

	0.38), (0.46, 0.01, 0.87), (0.72, 0.65, 0.85), (0.96, 0.97, 0.46), (0.94, 0.82, 0.69), (0.17, 0.45, 0.61), (0.39, 0.74, 1.19), (0.75, 0.76, 0.21), (0.8, 1.23, 0.44), (0.38, 0.07, 0.99), (0.57, 0.52, 1.19), (0.57, 0.54, 0.14), (0.93, 0.97, 0.37), (0.06, 0.2, 0.79), (0.59, 0.39, 1.0), (0.9, 0.58, 0.03), (0.75, 0.99, 0.27), (0.26, 0.47, 0.84), (0.62, 0.69, 0.97), (0.57, 0.76, 0.17), (0.99, 1.03, 0.4), (0.27, 0.12, 0.56), (0.53, 0.59, 0.92), (0.6, 0.98, 0.34), (1.06, 0.91, 0.35), (0.09, 0.13, 0.51), (0.25, 0.45, 0.84), (0.58, 0.56, 0.11), (0.83, 1.12, 0.26), (0.27,			

	0.23, 0.84), (0.73, 0...			
3	((0.49, 0.99, 1.24), (0.45, 0.78, 0.04), (0.47, 0.9, 0.36), (0.66, 0.38, 0.48), (1.1, 0.23, 0.59), (0.75, 0.58, 0.84), (1.35, 0.77, 0.76), (0.46, 0.51, 1.12), (0.59, 1.03, 0.98), (0.71, 1.22, 0.46), (0.57, 0.95, 0.56), (0.67, 0.45, 0.36), (0.77, 0.62, 0.81), (1.0, 0.51, 0.71), (0.93, 0.86, 0.82), (0.38, 0.57, 1.24), (0.45, 0.89, 1.31), (0.32, 0.79, 0.14), (0.56, 1.09, 0.24), (0.56, 0.2, 0.54), (0.97, 0.26, 0.63), (1.15, 0.33, 0.53), (0.87, 0.61, 0.64), (0.26, 0.97, 1.09), (0.49, 1.07, 1.1), (0.32, 1.01, 0.17), (0.61, 1.16,	((0.87, 0.28, 0.79), (0.65, 0.58, 0.53), (1.24, 0.25, 0.96), (0.68, 0.17, 0.43), (0.93, 0.25, 0.54), (1.17, 0.75, 0.75), (0.45, 0.99, 1.11), (0.35, 1.21, 0.39))	0.03	23.59398384 0000007 ms

	0.27), (0.77, 0.12, 0.31), (0.91, 0.47, 0.54), (0.85, 0.73, 0.84), (1.18, 0.54, 0.72), (0.09, 0.63, 0.76), (0.13, 0.83, 0.97), (0.33, 0.81, 0.11), (0.46, 1.24, 0.14), (0.77, 0.23, 0.59), (1.11...			
4	((0.42, 0.67, 0.8), (0.33, 0.41, 0.54), (0.29, 0.47, 0.8), (0.41, 0.88, 0.85), (0.79, 0.67, 0.91), (0.37, 0.95, 1.09), (0.91, 1.09, 0.95), (0.96, 0.76, 1.25), (1.03, 1.21, 1.04), (1.09, 1.34, 0.46), (0.88, 1.01, 0.5), (0.92, 0.45, 0.23), (0.95, 0.56, 0.62), (1.12, 0.39, 0.46), (0.99, 0.67, 0.5), (0.38, 0.32, 0.87), (0.39, 0.58, 0.87), (0.2, 0.42, 0.64),	((1.14, 0.29, 0.23), (0.84, 0.52, 0.34), (1.36, 0.12, 0.71), (1.12, 0.39, 0.46), (0.49, 0.61, 1.1), (0.95, 0.57, 0.58), (0.06, 0.44, 0.69), (1.07, 1.15, 0.3), (0.4, 0.88, 0.82), (1.13, 0.42, 0.42), (0.53, 0.69, 0.58), (0.11, 0.68, 0.83), (0.35, 0.96, 0.99), (0.02, 0.51, 0.42), (0.06, 0.46, 0.63), (1.41, 0.25, 0.61))	0.04	21.501392119999995 ms

	(0.37, 0.65, 0.68), (0.31, 0.7, 0.92), (0.65, 0.7, 0.94), (0.77, 0.7, 0.78), (0.44, 0.93, 0.83), (0.76, 1.22, 1.21), (0.93, 1.25, 1.16), (0.69, 1.14, 0.17), (0.92, 1.22, 0.21), (1.02, 0.12, 0.18), (1.1, 0.4, 0.36), (0.97, 0.6, 0.59), (1.24, 0.35, 0.41), (0.09, 0.38, 0.39), (0.06, 0.52, 0.53), (0.2, 0.43, 0.61), (0.27, 0.81, 0.58), (0.52, 0.73, 0.96), (0.79, 0.64,...			
5	((0.39, 0.52, 0.58), (0.26, 0.22, 0.29), (0.19, 0.25, 0.52), (0.29, 0.63, 0.54), (0.64, 0.39, 0.56), (0.19, 0.64, 0.72), (0.7, 0.74, 0.54), (0.71, 0.38, 0.81), (0.75, 0.81, 0.57), (0.78,	((1.24, 0.25, 0.14), (0.65, 0.92, 1.05), (1.26, 0.1, 0.51), (0.87, 0.64, 0.76), (0.46, 0.4, 0.51), (0.85, 0.65, 0.68), (0.58, 1.1, 0.98), (0.62, 0.55, 0.72), (0.98, 1.09, 1.21), (0.66,	0.5	34.41396248 ms

	0.91, 0.96), (0.54, 0.54, 0.97), (0.54, 0.95, 0.67), (0.55, 1.03, 1.03), (0.69, 0.83, 0.84), (0.52, 1.08, 0.85), (0.88, 0.7, 1.18), (0.86, 0.93, 1.15), (0.64, 0.73, 0.89), (0.78, 0.94, 0.9), (0.69, 0.95, 1.1), (1.0, 0.92, 1.09), (1.09, 0.89, 0.91), (0.72, 1.08, 0.92), (1.01, 1.34, 1.27), (1.15, 1.35, 1.19), (0.88, 1.2, 0.17), (1.08, 1.25, 0.18), (1.14, 0.12, 0.12), (1.19, 0.37, 0.26), (1.03, 0.54, 0.47), (1.28, 0.26, 0.25), (0.09, 0.25, 0.2), (0.03, 0.36, 0.31), (0.14, 0.25, 0.36), (0.18, 0.59, 0.3), (0.39, 0.48, 0.65), (0.64, 0...	0.83, 0.76), (0.88, 1.14, 0.79), (0.1, 0.63, 0.48), (0.67, 0.83, 1.29), (0.67, 1.04, 0.87), (0.59, 0.66, 0.91), (1.43, 0.29, 0.63), (0.6, 1.02, 1.19), (0.27, 0.57, 0.6), (0.83, 0.5, 0.92), (0.46, 0.84, 0.64), (0.31, 0.66, 0.55), (0.67, 0.97, 1.03), (0.77, 1.19, 0.87), (1.29, 0.47, 0.36), (0.49, 0.78, 0.86), (0.79, 0.92, 0.98), (0.52, 0.73, 0.54), (0.99, 1.22, 1.45), (0.76, 1.07, 1.06), (0.96, 1.22, 0.87), (1.16, 0.28, 0.36), (0.38, 0.6, 0.87))		

- Secuencial

#	puntoSeq	medianaSeq	eta	Resultado
1	((0.86, 0.86, 0.86), (0.2, 0.03, 0.04), (0.6, 0.53, 0.74), (0.16, 0.38, 0.23), (0.98, 0.61, 0.72), (0.0, 0.33, 0.34), (0.98, 0.9, 0.63), (0.46, 0.01, 0.37), (0.97, 0.9, 0.6), (0.46, 0.47, 0.46), (0.69, 0.57, 0.94), (0.17, 0.45, 0.11), (0.64, 0.99, 0.94), (0.25, 0.26, 0.21), (0.55, 0.98, 0.69), (0.38, 0.07, 0.49), (0.82, 0.77, 0.94), (0.07, 0.04, 0.14), (0.68, 0.72, 0.62), (0.06, 0.2, 0.29), (0.84, 0.64, 0.75), (0.4, 0.08, 0.03), (0.5, 0.74, 0.52), (0.26, 0.47, 0.34), (0.87, 0.94, 0.72), (0.07, 0.26, 0.17), (0.74, 0.78, 0.65), (0.27, 0.12, 0.06), (0.78, 0.84, 0.67), (0.1, 0.48, 0.34),	((0.76, 0.55, 0.67), (0.97, 0.94, 0.51))	0.01	92.81911967999999 ms

	(0.81, 0.66, 0.6), (0.09, 0.13, 0.01), (0.5, 0.7, 0.59), (0.08, 0.06, 0.11), (0.58, 0.87, 0.51), (0.27, 0.23, 0.34), (0.98, 0.58, 0.79), (0...			
2	((0.61, 0.61, 1.11), (0.7, 0.53, 0.04), (0.85, 0.78, 0.49), (0.16, 0.38, 0.73), (0.73, 0.36, 0.97), (0.5, 0.83, 0.34), (1.23, 1.15, 0.38), (0.46, 0.01, 0.87), (0.72, 0.65, 0.85), (0.96, 0.97, 0.46), (0.94, 0.82, 0.69), (0.17, 0.45, 0.61), (0.39, 0.74, 1.19), (0.75, 0.76, 0.21), (0.8, 1.23, 0.44), (0.38, 0.07, 0.99), (0.57, 0.52, 1.19), (0.57, 0.54, 0.14), (0.93, 0.97, 0.37), (0.06, 0.2, 0.79), (0.59, 0.39, 1.0), (0.9, 0.58, 0.03), (0.75, 0.99, 0.27), (0.26, 0.47, 0.84), (0.62, 0.69,	((0.51, 0.3, 0.92), (0.72, 0.69, 0.76), (0.65, 0.84, 0.26), (0.64, 0.29, 0.82))	0.02	143.38278248 ms

	0.97), (0.57, 0.76, 0.17), (0.99, 1.03, 0.4), (0.27, 0.12, 0.56), (0.53, 0.59, 0.92), (0.6, 0.98, 0.34), (1.06, 0.91, 0.35), (0.09, 0.13, 0.51), (0.25, 0.45, 0.84), (0.58,0.56, 0.11), (0.83, 1.12, 0.26), (0.27, 0.23, 0.84), (0.73, 0.33, 1.04...			
3	((0.49, 0.99, 1.24), (0.45, 0.78, 0.04), (0.47, 0.9, 0.36), (0.66, 0.38, 0.48), (1.1, 0.23, 0.59), (0.75, 0.58, 0.84), (1.35, 0.77, 0.76), (0.46, 0.51, 1.12), (0.59, 1.03, 0.98), (0.71, 1.22, 0.46), (0.57, 0.95, 0.56), (0.67, 0.45, 0.36), (0.77, 0.62, 0.81), (1.0, 0.51, 0.71), (0.93, 0.86, 0.82), (0.38, 0.57, 1.24), (0.45, 0.89, 1.31), (0.32, 0.79, 0.14), (0.56, 1.09, 0.24), (0.56,	((0.87, 0.28, 0.79), (0.65, 0.58, 0.53), (1.24, 0.25, 0.96), (0.68, 0.17, 0.43), (0.93, 0.25, 0.54), (1.17, 0.75, 0.75), (0.45, 0.99, 1.11), (0.35, 1.21, 0.39))	0.03	32.58166532 ms

	0.2, 0.54), (0.97, 0.26, 0.63), (1.15, 0.33, 0.53), (0.87, 0.61, 0.64), (0.26, 0.97, 1.09), (0.49, 1.07, 1.1), (0.32, 1.01, 0.17), (0.61, 1.16, 0.27), (0.77, 0.12, 0.31), (0.91, 0.47, 0.54), (0.85, 0.73, 0.84), (1.18, 0.54, 0.72), (0.09, 0.63, 0.76), (0.13, 0.83, 0.97), (0.33, 0.81, 0.11), (0.46, 1.24, 0.14), (0.77, 0.23, 0.59), (1.11, 0.2, 0....			
4	((0.42, 0.67, 0.8), (0.33, 0.41, 0.54), (0.29, 0.47, 0.8), (0.41, 0.88, 0.85), (0.79, 0.67, 0.91), (0.37, 0.95, 1.09), (0.91, 1.09, 0.95), (0.96, 0.76, 1.25), (1.03, 1.21, 1.04), (1.09, 1.34, 0.46), (0.88, 1.01, 0.5), (0.92, 0.45, 0.23), (0.95, 0.56, 0.62), (1.12, 0.39, 0.46),	((1.14, 0.29, 0.23), (0.84, 0.52, 0.34), (1.36, 0.12, 0.71), (1.12, 0.39, 0.46), (0.49, 0.61, 1.1), (0.95, 0.57, 0.58), (0.06, 0.44, 0.69), (1.07, 1.15, 0.3), (0.4, 0.88, 0.82), (1.13, 0.42, 0.42), (0.53, 0.69, 0.58), (0.11, 0.68, 0.83), (0.35, 0.96, 0.99), (0.02, 0.51, 0.42),	0.04	22.17177288 ms

	(0.99, 0.67, 0.5), (0.38, 0.32, 0.87), (0.39, 0.58, 0.87), (0.2, 0.42, 0.64), (0.37, 0.65, 0.68), (0.31, 0.7, 0.92), (0.65, 0.7, 0.94), (0.77, 0.7, 0.78), (0.44, 0.93, 0.83), (0.76, 1.22, 1.21), (0.93, 1.25, 1.16), (0.69, 1.14, 0.17), (0.92, 1.22, 0.21), (1.02, 0.12, 0.18), (1.1, 0.4, 0.36), (0.97, 0.6, 0.59), (1.24, 0.35, 0.41), (0.09, 0.38, 0.39), (0.06, 0.52, 0.53), (0.2, 0.43, 0.61), (0.27, 0.81, 0.58), (0.52, 0.73, 0.96), (0.79, 0.64, 0.97), (...	(0.06, 0.46, 0.63), (1.41, 0.25, 0.61))		
5	(0.39, 0.52, 0.58), (0.26, 0.22, 0.29), (0.19, 0.25, 0.52), (0.29, 0.63, 0.54), (0.64, 0.39, 0.56), (0.19, 0.64, 0.72), (0.7, 0.74, 0.54), (0.71, 0.38, 0.81), (0.75, 0.81,	((1.24, 0.25, 0.14), (0.65, 0.92, 1.05), (1.26, 0.1, 0.51), (0.87, 0.64, 0.76), (0.46, 0.4, 0.51), (0.85, 0.65, 0.68), (0.58, 1.1, 0.98), (0.62, 0.55, 0.72), (0.98, 1.09,	0.5	53.6509244 ms

	0.57), (0.78, 0.91, 0.96), (0.54, 0.54, 0.97), (0.54, 0.95, 0.67), (0.55, 1.03, 1.03), (0.69, 0.83, 0.84), (0.52, 1.08, 0.85), (0.88, 0.7, 1.18), (0.86, 0.93, 1.15), (0.64, 0.73, 0.89), (0.78, 0.94, 0.9), (0.69, 0.95, 1.1), (1.0, 0.92, 1.09), (1.09, 0.89, 0.91), (0.72, 1.08, 0.92), (1.01, 1.34, 1.27), (1.15, 1.35, 1.19), (0.88, 1.2, 0.17), (1.08, 1.25, 0.18), (1.14, 0.12, 0.12), (1.19, 0.37, 0.26), (1.03, 0.54, 0.47), (1.28, 0.26, 0.25), (0.09, 0.25, 0.2), (0.03, 0.36, 0.31), (0.14, 0.25, 0.36), (0.18, 0.59, 0.3), (0.39, 0.48, 0.65), (0.64, 0.36, 0.63...	1.21), (0.66, 0.83, 0.76), (0.88, 1.14, 0.79), (0.1, 0.63, 0.48), (0.67, 0.83, 1.29), (0.67, 1.04, 0.87), (0.59, 0.66, 0.91), (1.43, 0.29, 0.63), (0.6, 1.02, 1.19), (0.27, 0.57, 0.6), (0.83, 0.5, 0.92), (0.46, 0.84, 0.64), (0.31, 0.66, 0.55), (0.67, 0.97, 1.03), (0.77, 1.19, 0.87), (1.29, 0.47, 0.36), (0.49, 0.78, 0.86), (0.79, 0.92, 0.98), (0.52, 0.73, 0.54), (0.99, 1.22, 1.45), (0.76, 1.07, 1.06), (0.96, 1.22, 0.87), (1.16, 0.28, 0.36), (0.38, 0.6, 0.87))		

Como se puede observar por los resultados de la tabla cuando se analiza de forma paralela corre el algoritmo mucho más rápido que la forma secuencial

¿Cuándo es aconsejable usar el paralelismo? ¿Cuándo no?

El paralelismo es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente, basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.

Por lo tanto, para datos demasiados grandes se aconseja usar paralelismo para datos muy grandes