

**UNIVERSIDAD PERUANA DE CIENCIAS APLICADAS**

Programación Concurrente y Distribuida - CC65

Trabajo Parcial

**INTEGRANTES**

Flores Tenorio, Juan Diego Enrique u202014558

Goyas Ayllón, Leonardo André u202010206

Humbser Meza, Diego Fernando u202012711

**SECCIÓN**

CC72

**DOCENTE**

Carlos Alberto Jara Garcia

**CICLO 2024-01**

ÍNDICE

[1. INTRODUCCIÓN 3](#_Toc165485434)

[2. ALGORITMOS SECUENCIAL Y CONCURRENTE 3](#_Toc165485435)

[Implementación Secuencial 3](#_Toc165485436)

[Implementación Concurrente 7](#_Toc165485437)

[3. MECANISMOS DE PARALELIZACIÓN Y SINCRONIZACIÓN 8](#_Toc165485438)

[4. PRUEBAS DE COMPROBACIÓN Y RESULTADOS 9](#_Toc165485439)

[5. SIMULACIÓN CON PROMELA 10](#_Toc165485440)

[6. ANÁLISIS CON SPIN 14](#_Toc165485441)

[7. BIBLIOGRAFÍA 17](#_Toc165485442)

[8. ENLACE AL REPOSITORIO EN GITHUB 17](#_Toc165485443)

[9. ENLACE AL VIDEO 17](#_Toc165485444)

# INTRODUCCIÓN

La programación concurrente, fundamental en el desarrollo de software moderno, facilita la multitarea dentro de los sistemas, crucial para satisfacer las demandas de eficiencia y escalabilidad. Este proyecto utiliza el lenguaje de programación Go para implementar el algoritmo K-Means sin dependencias externas, enfatizando la paralelización y la sincronización, especialmente a través de semáforos. El objetivo es ofrecer una solución de alto rendimiento adaptada a los desafíos contemporáneos del software.

# ALGORITMOS SECUENCIAL Y CONCURRENTE

## Implementación Secuencial

Los paquetes importados son *fmt* para formato e impresión, *math* para operaciones matemáticas, *math/rand* para generación de números aleatorios y *time* para funciones relacionadas con el tiempo.

import (

    "fmt"

    "math"

    "math/rand"

    "time"

)

Se define una estructura llamada *KMeans* para contener parámetros y datos del algoritmo K-Means. Incluye campos como *nClusters* para el número de clústeres, *maxIters* para el máximo número de iteraciones, *centroids* para almacenar centroides de clústeres, *labels* para etiquetas de puntos de datos y *data* para el conjunto de datos.

type KMeans struct {

    nClusters   int

    maxIters    int

    centroids   [][]float64

    labels      [1000000]int

    data        [][]float64

}

La función *NewKMeans* inicializa una nueva instancia de la estructura *KMeans* con los parámetros proporcionados.

func NewKMeans(nClusters, maxIters int, X [][]float64) \*KMeans {

    return &KMeans {

        nClusters: nClusters,

        maxIters:  maxIters,

        data: X,

    }

}

La función *euclideanDistance* calcula la distancia euclidiana entre dos puntos en el espacio euclidiano usando la fórmula .

func euclideanDistance(a, b []float64) float64 {

    var sum float64

    for i := range a {

        temp := a[i] - b[i]

        sum += temp \* temp

    }

    return math.Sqrt(sum)

}

El método *assignLabels* asigna etiquetas a los puntos de datos basados en su proximidad a los centroides. Itera a través de cada punto de datos, calcula la distancia euclidiana a cada centroide llamando a la función *euclideanDistance* y asigna la etiqueta del centroide más cercano.

func (kMeans \*KMeans) assignLabels() {

    for i := 0; i < len(kMeans.data); i++ {

        minDist := math.MaxFloat64

        var minIdx int

        for j, c := range kMeans.centroids {

            dist := euclideanDistance(kMeans.data[i], c)

            if dist < minDist {

                minDist = dist

                minIdx = j

            }

        }

        kMeans.labels[i] = minIdx

    }

}

El método *checkConvergence* verifica si los centroides han convergido comparándolos con los centroides recién calculados. La convergencia se logra si la diferencia entre los valores correspondientes de los centroides antiguos y nuevos está dentro de un umbral.

func (kMeans \*KMeans) checkConvergence(a [][]float64) bool {

    for i, c := range kMeans.centroids {

        var count int

        for j, v := range c {

            if math.Abs(v-a[i][j]) > 1e-2  {

                count++

            }

        }

        if count == len(kMeans.centroids[0]) {

            return false

        }

    }

    return true

}

El método *updateCentroids* actualiza los centroides basados en las asignaciones actuales de los puntos de datos. Calcula nuevos centroides tomando la media de todos los puntos de datos asignados a cada cluster.

func (kMeans \*KMeans) updateCentroids() bool {

    newCentroids := make([][]float64, kMeans.nClusters)

    for i := range newCentroids {

        newCentroids[i] = make([]float64, 2)

    }

    counts := make([]int, kMeans.nClusters)

    for i := range kMeans.data {

        for j:= range kMeans.data[i] {

            newCentroids[kMeans.labels[i]][j] += kMeans.data[i][j]

        }

        counts[kMeans.labels[i]]++

    }

    for i := range newCentroids {

        for j := range newCentroids[i] {

            newCentroids[i][j] /= float64(counts[i])

        }

    }

    if kMeans.checkConvergence(newCentroids) {

        return true

    }

    kMeans.centroids = newCentroids

    return false

}

El método *Fit* ejecuta el algoritmo K-Means. Inicializa centroides aleatoriamente, asigna etiquetas a los puntos de datos, actualiza centroides iterativamente hasta la convergencia o hasta que se alcance el número máximo de iteraciones.

func (kMeans \*KMeans) Fit() {

    kMeans.centroids = make([][]float64, kMeans.nClusters)

    for i := range kMeans.centroids {

        kMeans.centroids[i] = make([]float64, len(kMeans.data[i]))

        for j := range kMeans.centroids[i] {

            kMeans.centroids[i][j] = kMeans.data[rand.Intn(len(kMeans.data))][j]

        }

    }

    kMeans.assignLabels()

    for iter := 0; iter < kMeans.maxIters; iter++ {

        if kMeans.updateCentroids() {

            fmt.Println("Ha llegado a convergencia.")

            break

        }

        kMeans.assignLabels()

    }

}

La función *createArrayValues* genera un conjunto de datos con valores aleatorios dentro de un rango especificado.

func createArrayValues(min, max float64) [][]float64 {

    X := make([][]float64, 1000000)

    for i := range X {

        X[i] = make([]float64, 2)

        for j := range X[i] {

            X[i][j] = min + rand.Float64() \* (max - min)

        }

    }

    return X

}

La función *main* ejecuta el programa. Genera un conjunto de datos, inicializa K-Means con parámetros específicos, ejecuta el algoritmo e imprime los centroides finales y el tiempo de ejecución para cada iteración. También calcula e imprime el tiempo de ejecución promedio durante múltiples iteraciones.

func main() {

    rand.Seed(time.Now().UnixNano())

    var sum time.Duration

    total\_duration := make([]time.Duration, 1000)

    for i := 0; i < 1000; i++ {

        start := time.Now()

        X := createArrayValues(0.0, 1000.0)

        kmeans := NewKMeans(10, 100, X)

        kmeans.Fit()

        fmt.Println("Final Centroids:", kmeans.centroids)

        fmt.Println("Execution Time: ", time.Since(start))

        total\_duration = append(total\_duration, time.Since(start))

    }

    total\_duration = total\_duration[50:len(total\_duration)-51]

    for i := 0; i < len(total\_duration); i++ {

        sum += total\_duration[i]

    }

    fmt.Println("\nAverage time: ", float64(sum) / float64(len(total\_duration)))

}

## Implementación Concurrente

Los paquetes importados son los mismos que se utilizaron en la versión secuencial. En esta implementación, se utiliza el paquete *sync,* para primitivas de sincronización como grupos de espera y mutexes.

import (

    "fmt"

    "math"

    "math/rand"

    "time"

    "sync"

)

La estructura *KMeans* se define para contener parámetros y datos para el algoritmo K-Means, igual que en la versión secuencial. Además, incorpora primitivas de sincronización como *sync.WaitGroup* y *sync.Mutex* para gestionar la ejecución concurrente de forma segura.

type KMeans struct {

    nClusters   int

    maxIters    int

    centroids   [][]float64

    labels      [1000000]int

    wait\_group  sync.WaitGroup

    mutex       sync.Mutex

    data        [][]float64

}

El método *assignLabels* utiliza *goroutines* para calcular de manera concurrente las distancias entre cada punto de datos y todos los centroides. Para cada punto de datos, se crea una goroutine para calcular las distancias y determinar el centroide más cercano, garantizando la computación paralela. Se emplea un *mutex* para sincronizar el acceso a recursos compartidos, como el arreglo de etiquetas que almacena las etiquetas asignadas, evitando condiciones de carrera de datos.

func (kMeans \*KMeans) assignLabels() {

    kMeans.wait\_group.Add(len(kMeans.data))

    for i := 0; i < len(kMeans.data); i++ {

        go func(i int, x [][]float64) {

            defer kMeans.wait\_group.Done()

            minDist := math.MaxFloat64

            var minIdx int

            kMeans.mutex.Lock()

            defer kMeans.mutex.Unlock()

            for j, c := range kMeans.centroids {

                dist := euclideanDistance(x[i], c)

                if dist < minDist {

                    minDist = dist

                    minIdx = j

                }

            }

            kMeans.labels[i] = minIdx

        }(i, kMeans.data)

    }

    kMeans.wait\_group.Wait()

}

# MECANISMOS DE PARALELIZACIÓN Y SINCRONIZACIÓN

En la implementación concurrente, se emplean dos mecanismos para lograr la paralelización y la sincronización: *goroutines* para la ejecución paralela y primitivas de sincronización como *sync.WaitGroup* y *sync.Mutex*.

**Goroutines**

Las *goroutines* son hilos livianos gestionados por el tiempo de ejecución de Go que permiten la ejecución concurrente de funciones. En el método *assignLabels*, se utilizan *goroutines* para paralelizar el cálculo de distancias entre puntos de datos y centroides. Al crear una *goroutine* para cada punto de datos, el algoritmo puede aprovechar de manera efectiva múltiples núcleos de CPU, acelerando el cálculo. Este mecanismo de paralelización permite al algoritmo procesar puntos de datos concurrentemente, mejorando el rendimiento general.

**Primitivas de Sincronización**

Se emplean primitivas de sincronización como *sync.WaitGroup* y *sync.Mutex* para coordinar la ejecución de *goroutines* y garantizar la seguridad de hilos al acceder a recursos compartidos. El *sync.WaitGroup* se utiliza en el método *assignLabels* para esperar a que todas las *goroutines* completen sus cálculos antes de proceder al siguiente paso del algoritmo. Esto asegura que todos los puntos de datos estén etiquetados correctamente antes de actualizar los centroides. Además, el *sync.Mutex* se utiliza para sincronizar el acceso a recursos compartidos, como el arreglo labels, evitando condiciones de carrera de datos cuando múltiples *goroutines* intentan modificar el arreglo simultáneamente. Al adquirir y liberar el *mutex*, el acceso concurrente a recursos compartidos se serializa, garantizando la consistencia y previniendo condiciones de carrera.

# PRUEBAS DE COMPROBACIÓN Y RESULTADOS

**Especificaciones del ambiente de pruebas**

* Procesador: Intel i9-10850k.
* Cantidad de hilos: 20.
* Frecuencia básica del procesador: 3.60 GHz.
* Memoria RAM: 16 GB DDR4.

**Ejecución de pruebas de condición de carrera en la implementación concurrente**

A screenshot of a computer

Description automatically generatedEn esta prueba se comprobó la existencia de la condición de carrera del código concurrente. Para ello, se utilizó la flag *-race* durante la ejecución del programa. Esta utilidad integrada de Go facilita la detección de condiciones de carrera al examinar el momento y la forma en que se accede a la memoria, así como el acceso no sincronizado a variables compartidas.

**Ejecución de prueba múltiple de la implementación concurrente**

En esta prueba se validó el código concurrente en 1000 ejecuciones con valores pseudoaleatorios. El tiempo de ejecución promedio es aproximadamente 20 segundos, siendo casi 470% más lento que su homólogo tradicional. Esta diferencia es ocasionada por la limitación en la velocidad de lectura del disco.

A screenshot of a computer program

Description automatically generated

**Ejecución de prueba múltiple de la implementación secuencial**

En esta prueba se validó el código tradicional en 1000 ejecuciones con valores pseudoaleatorios. El tiempo de ejecución es aproximadamente 3.5 segundos, siendo casi 82.5% más rápido que su homólogo concurrente.

**A screenshot of a computer

Description automatically generated**

# SIMULACIÓN CON PROMELA

La simulación PROMELA del algoritmo K-Means implica procesos concurrentes que representan la actualización de etiquetas para puntos de datos y el cálculo subsiguiente de centroides. La simulación comienza definiendo los centroides iniciales del clúster y los puntos de datos, junto con variables necesarias como *nClusters*, *nIters* y *nPoints*. Cada proceso se modela como un proctype de PROMELA, donde *UpdateLabel* representa la actualización concurrente de etiquetas para puntos de datos y *main* sirve como el proceso principal que orquesta la ejecución del algoritmo.

#define wait(S) atomic{S>0 -> S--}

#define signal(S) S++

#define nClusters 5

#define nIters 5

#define nPoints 30

// definiendo centroides (cluster) en puntos aleatorios

int centroidX[nClusters] = { 16, 26, 7, 27, 29 }

int centroidY[nClusters] = { 4, 16, 5, 6, 16 }

// contador de puntos por cluster

int clusterPoints[nClusters]

// label de cluster por punto del conjunto de datos

int labels[nPoints]

// definiendo un conjunto de datos aleatorio

int dataX[nPoints] = { 29, 9, 17, 28, 23, 21, 25, 13, 15, 15, 25, 7, 22, 6, 21, 15, 17, 21, 13, 27, 9, 4, 23, 23, 5, 9, 18, 22, 13, 11 }

int dataY[nPoints] = { 22, 13, 14, 27, 23, 17, 9, 25, 25, 27, 26, 14, 23, 0, 16, 11, 28, 16, 27, 27, 1, 6, 5, 20, 8, 6, 7, 29, 4, 7 }

// contador de puntos procesados para esperar

int count = nPoints

// semáforo

byte mutex = 1

En el proctype *UpdateLabel*, cada punto de datos itera sobre todos los clústeres para calcular la distancia de Manhattan a cada centroide. Esta distancia se calcula usando diferencias absolutas en las coordenadas X e Y, ya que PROMELA carece de aritmética de punto flotante y funciones de raíz cuadrada. Luego, el proceso actualiza la etiqueta del punto de datos al índice del centroide más cercano, decrementando la variable *count* para hacer seguimiento de los puntos procesados. Un semáforo binario *mutex* asegura la exclusión mutua, previniendo condiciones de carrera durante las actualizaciones de etiquetas.

proctype UpdateLabel(int i) {

    // NCS

    int minDist = nPoints

    int minIdx = 0

    // CS

    wait(mutex)

    int j

    for (j : 0 .. nClusters - 1) {

        int dist = 0

        // Calculando |X1i - X2i|

        int distX

        if

        :: centroidX[j] > dataX[i] ->

            distX = centroidX[j] - dataX[i]

        :: else ->

            distX = dataX[i] - centroidX[j]

        fi

        // Calculando |Y1i - Y2i|

        int distY

        if

        :: centroidY[j] > dataY[i] ->

            distY = centroidY[j] - dataY[i]

        :: else ->

            distY = dataY[i] - centroidY[j]

        fi

        dist = distX + distY

        if

        :: dist < minDist ->

            minDist = dist

            minIdx = j

        :: else ->

        fi

    }

    labels[i] = minIdx

    count--

    signal(mutex)

}

El proceso *main* inicia el algoritmo imprimiendo los centroides iniciales del clúster y ejecutando un número fijo de iteraciones (*nIters*). Dentro de cada iteración, ejecuta concurrentemente el proceso *UpdateLabel* para cada punto de datos. Después de actualizar las etiquetas, espera a que todos los puntos de datos sean procesados antes de recalcular los centroides. Esta sincronización se logra usando un bloque atómico, asegurando que las actualizaciones de centroides ocurran después de que se hayan asignado todas las etiquetas. Finalmente, los centroides se calculan como la media de los puntos de datos asignados a cada clúster y se imprimen en la consola.

active proctype main() {

    printf("Clusters iniciales\n")

    int i

    for (i : 0 .. nClusters - 1) {

        printf("%d, %d\n", centroidX[i], centroidY[i])

    }

    int iter

    for (iter : 0 .. nIters - 1) {

        printf("Iteracion %d comenzada.\n", iter + 1)

        for (i : 0 .. nPoints - 1) {

            // Generando las funciones concurrentes

            run UpdateLabel(i)

        }

        // Waitgroup

        atomic { count < 1 -> count = nPoints }

        // Actualizar los centroides, inicializar en 0

        for (i : 0 .. nClusters - 1) {

            centroidX[i] = 0

            centroidY[i] = 0

            clusterPoints[i] = 0

        }

        // Recorrer los puntos y sumar los valores

        for (i : 0 .. nPoints - 1) {

            int a = labels[i]

            clusterPoints[a]++

            centroidX[a] = centroidX[a] + dataX[i]

            centroidY[a] = centroidY[a] + dataY[i]

        }

        // Actualizar centroides en el punto promedio

        for (i : 0 .. nClusters - 1) {

            centroidX[i] = centroidX[i] / clusterPoints[i]

            centroidY[i] = centroidY[i] / clusterPoints[i]

        }

        for (i : 0 .. nClusters - 1) {

            printf("%d, %d\n", centroidX[i], centroidY[i])

        }

        //printf("Iteracion %d terminada.\n", iter + 1)

    }

    printf("\nEntrenamiento terminado.\n")

    for (i : 0 .. nClusters - 1) {

        printf("%d, %d\n", centroidX[i], centroidY[i])

    }

}

**Resultados de la simulación del algoritmo K-Means en PROMELA**

**A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated**Los resultados impresos durante la ejecución de la simulación brindan información sobre el proceso de agrupación y la convergencia de los centroides durante múltiples iteraciones.

* **Centroides Iniciales:** La simulación comienza imprimiendo los centroides iniciales del clúster, que están predefinidos en el programa. Cada línea corresponde a las coordenadas X e Y de un centroide, representando los puntos de partida para la agrupación.
* **Actualizaciones de Iteraciones:** Para cada iteración, la simulación imprime los centroides actualizados después de asignar etiquetas a los puntos de datos. Cada línea representa las coordenadas X e Y de un centroide después de la iteración actual. Los centroides se recalculan en función de la posición media de los puntos de datos asignados a cada clúster.
* **Finalización del Entrenamiento:** Después de completar todas las iteraciones, la simulación imprime "Entrenamiento terminado" seguido de las coordenadas finales de los centroides. Estos centroides representan la solución de agrupación final después de que el algoritmo haya convergido.
* **Número de Procesos Creados:** Finalmente, la simulación muestra el número de procesos creados durante la ejecución. En este caso, "151 procesos creados" indica el número total de procesos generados durante la simulación, incluyendo el proceso principal y los procesos concurrentes para la actualización de etiquetas.

# ANÁLISIS CON SPIN

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedEl comando *spin -t* se utiliza para ejecutar el modelo SPIN en trail mode. Este proporciona un rastro detallado de la ejecución del sistema, mostrando la secuencia de estados y transiciones tomadas por cada proceso a lo largo de la simulación. Permite examinar paso a paso el comportamiento del modelo, lo que permite identificar errores, bloqueos o comportamientos inesperados.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

La salida indica la configuración inicial de la simulación del algoritmo K-Means, donde se definen los clústeres iniciales y los puntos de datos. Los centroides iniciales de los clústeres se imprimen, seguidos por las iteraciones del entrenamiento del algoritmo.

Las líneas posteriores proporcionan el estado de varias variables y matrices utilizadas en la simulación, incluidas las coordenadas de los centroides (*centroidX* y *centroidY*), el número de puntos asignados a cada clúster (*clusterPoints*), las etiquetas asignadas a cada punto de datos (*labels*) y las coordenadas de los puntos de datos (*dataX* y *dataY*). El valor de la variable *count* indica la cantidad de puntos procesados, y *mutex* representa el estado del semáforo utilizado para la sincronización.

A screenshot of a computer program

Description automatically generatedLa salida también muestra las transiciones de estado de los procesos individuales (*UpdateLabel*) y el proceso principal (*main*). Con un valor VECTORSZ=1024 se pueden encontrar 72 pasos, sin errores de afirmación (*assert*) o pausas inesperadas.

Al ejecutar *./pan* con anotación *-DVECTORSZ=2048* logra reconocer 9000+ estados, pero al tener un ambiente de bajo procesamiento (16 GB RAM), el sistema satura la memoria antes de verificar todos los pasos del modelo.

Sin embargo, cabe recalcar que la ejecución de la aplicación (*KMeansValidate.pml*) concluye rápidamente sin ningún error en consola o de las validaciones insertadas en el código. Como punto de mejora futura recomendamos experimentar con diferentes configuraciones de hardware, para poder evaluar de manera minuciosa modelos de inteligencia artificial.

# BIBLIOGRAFÍA

DATAtab. (2020, Noviembre 17). *k-Means Clustering: Simply explained & calculated* [Vídeo]. Youtube. <https://www.youtube.com/watch?v=jZs5rX8Kl3o>

Google (2024). Sync package. *GO*. Recuperado de: <https://pkg.go.dev/sync> [Consulta: 24/04/2024]

With Code Example. (2023). Synchronization Primitives in the sync Package. *WCE*. Recuperado de: <https://golang.withcodeexample.com/blog/synchronization-primitives-sync-package-go/> [Consulta: 27/04/2024]

Balachandran, A. (2021). Synchronization in Go using concurrency Primitives: A Case Study. *Medium*. Recuperado de: <https://medium.com/@anandpillai/synchronization-in-go-using-concurrency-primitives-a-case-study-535bb2a71c13> [Consulta: 27/04/2024]

Spin (s.f). Verifying Multi-threaded Software with Spin. *Spin*. Recuperado de: <https://spinroot.com/spin/whatispin.html> [Consulta: 28/04/2024]

# ENLACE AL REPOSITORIO EN GITHUB

<https://github.com/aaaaqa/Concurrent-KMeans>

# ENLACE AL VIDEO

One Drive:

<https://upcedupe-my.sharepoint.com/:v:/g/personal/u202010206_upc_edu_pe/EWt3u_B6eYRCtOqiXj0NV9YBSRAzp5I0ouLlZJZfMSCCxQ?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=9C0z46>

YouTube:

<https://youtu.be/q7xLgi6GPZo>