Informe Taller sobre paralelización de tareas y datos

> **Curso:** Fundamentos de Programación Funcional y Concurrente
> **Tema:** Paralelización de Algoritmos de Clustering (KMeans)

Estudiantes:

- Jorge Luis Junior Lasprilla Prada - 2420662
- Andrés Gerardo González Rosero - 2416541

---

1. Introducción

En este informe se presenta el análisis y resultados obtenidos en el taller de paralelización de tareas y datos aplicado al algoritmo de clustering KMeans. El objetivo principal fue implementar versiones paralelas de las funciones clave del algoritmo, evaluar su desempeño en comparación con las versiones secuenciales y analizar las ventajas y limitaciones de las diferentes estrategias de paralelización empleadas.

2. Informe de corrección

2.1 Funciones Secuenciales

2.1.1 `hallarPuntoMasCercano`

Especificación

Dado un punto p y medianas μ, la función retorna μ_i tal que:

$$\forall j,\ \ d(p, \mu_i) \leq d(p, \mu_j)$$

Implementación

Uso de `minBy` sobre la distancia al cuadrado.

Demostración de corrección

Sea f la función de costo (distancia al cuadrado).
El método `minBy` implementa el operador de minimización sobre un conjunto S:

$$\min_{\mu_j \in \mu} f(p, \mu_j)$$

Conclusión: `hallarPuntoMasCercano` es correcta por aplicación directa del axioma de minimización.

2.1.2 `calculePromedioSeq`

Especificación

Si $|C| > 0$, retorna el centroide c del conjunto C.
Si $|C| = 0$, retorna la mediana μ.

Implementación

Composición de `map`, `reduce` y división escalar.

Demostración de corrección

Para $|C| = 0$, el retorno cumple trivialmente.
Para $|C| > 0$:

$$c = \frac{1}{|C|} \sum_{p \in C} p$$

Conclusión: `calculePromedioSeq` es correcta por isomorfismo con la definición matemática del centroide.

2.1.3 `clasificarSeq`

Especificación

La función devuelve un mapeo M que constituye una partición de P tal que:

$$p \in M(\mu_i) \;\Rightarrow\; \mu_i = \arg\min_{\mu_j} d(p, \mu_j)$$

Implementación

Uso de `groupBy` con clave `hallarPuntoMasCercano`.

Demostración de corrección

`groupBy` genera clases exhaustivas y disjuntas.
La clave es $\arg\min$ sobre distancia, asignando el punto a su mediana más cercana.

Conclusión: `clasificarSeq` cumple la definición formal de cluster.

2.1.4 `actualizarSeq`

Especificación

La función produce μ' tal que:

a) Se preserva el orden posicional:

$$\mu'_i \leftrightarrow \mu_i$$

b) Cada valor es el centroide del cluster correspondiente:

$$\mu'_i = \text{centroide}(M(\mu_i))$$

Implementación

Uso de `map`.

Demostración de corrección

`map` preserva índices, y la función aplicada es correcta por definiciones previas.

Conclusión: `actualizarSeq` es correcta y preserva orden y valores.

2.1.5 `hayConvergenciaSeq`

Especificación

La función implementa:

$$\forall i,\ d(\mu_i, \mu_i') \leq \varepsilon$$

Implementación

Recursión de cola booleana.

Demostración de corrección

Una conjunción vacía es verdadera.
Cada paso evalúa el primer par y recurre al resto.

Conclusión: Se implementa inductivamente el cuantificador universal $\forall$.

2.1.6 `kMedianasSeq`

Especificación

El algoritmo converge a un estado final μ^* tal que:

$$\neg\ \mathrm{hayConvergenciaSeq}(\mu, \mu')$$

Implementación

Recursión de cola compuesta por clasificación, actualización y verificación.

Demostración de corrección

$$J(\mu^{(t+1)}) \leq J(\mu^{(t)})$$

La función de costo está acotada inferiormente, por lo que debe converger.

Conclusión: Su comportamiento es equivalente a la definición formal de KMeans.

2.2 Funciones Paralelas

2.2.1 `calculePromedioPar`

Proposición

$$\text{centroide}_{par}(C) = \text{centroide}_{seq}(C)$$

Demostración

División $C = C_1 \cup C_2$:

$$\sum C = \sum C_1 + \sum C_2$$

La suma es asociativa; la concurrencia no altera su valor.

Conclusión: El resultado es idéntico al secuencial.

2.2.2 `clasificarPar`

Proposición

$$\text{clasificar}_{par}(P, \mu) = \text{clasificar}_{seq}(P, \mu)$$

Demostración

Dividir $P = P_1 \cup P_2$.
Concatenar:

$$M = M_1 \cup M_2$$

Los clusters son disjuntos y preservan claves.

Conclusión: No altera la partición semántica.

2.2.3 `actualizarPar`

Proposición

$$\mu'_{par} = \mu'_{seq}$$

Demostración

Cada elemento es independiente:

$$\mu'_i = \text{centroide}(M(\mu_i))$$

La paralelización sólo evalúa funciones puras.

Conclusión: El valor y el orden permanecen invariantes.

2.2.4 `hayConvergenciaPar`

Proposición

$$\bigwedge_i b_i$$

Demostración

La conjunción es:

- asociativa
- conmutativa
- idempotente

El orden de evaluación no altera el resultado.

Conclusión: El valor lógico es idéntico al secuencial.

2.2.5 `kMedianasPar`

Proposición

$$kMedianas_{par}(P, \mu_0) = kMedianas_{seq}(P, \mu_0)$$

Demostración

Sucesión de estados:

$$\mu^{(t+1)} = \text{update}(\text{clasificar}(P, \mu^{(t)}))$$

Los pasos paralelos son equivalentes a los secuenciales por las proposiciones anteriores.

Conclusión: Produce la misma convergencia al mismo punto fijo.

3. Informe de desempeño de las funciones secuenciales y paralelas

3.1. Metodología de evaluación

Para evaluar el rendimiento de las funciones `kMedianasSeq` y `kMedianasPar` se utilizaron las funciones `generarPuntos` e `inicializarMedianas` del paquete `kmedianas2D`, junto con `tiemposKmedianas` del paquete `Benchmark`. Las pruebas se realizaron variando el número de puntos y de clusters, siguiendo potencias de 2 para garantizar escalabilidad y consistencia en los resultados.

Cada prueba consistió en:

- Generar una secuencia de puntos aleatorios en el plano 2D mediante `generarPuntos(k, num)`.
- Inicializar las medianas de forma aleatoria mediante `inicializarMedianas(k, puntos)`.
- Ejecutar el algoritmo en sus versiones secuencial y paralela.
- Medir el tiempo de ejecución de cada versión usando `tiemposKmedianas`.
- Calcular la aceleración (speedup) como el cociente entre el tiempo secuencial y el tiempo paralelo.

Se registraron múltiples ejecuciones para cada configuración y se tomó, cuando fue pertinente, el tiempo mediano para reducir el efecto de ruido en las medidas.

3.2. Resultados obtenidos

Tamaño de entrada	Clusters	Tiempo Secuencial (ms)	Tiempo Paralelo (ms)	Aceleración
16	3	0.381	1.423	0.268
1024	8	3.480	4.655	0.747
32768	32	276.029	49.556	5.570
100000	64	2237.230	474.031	4.720

3.3. Análisis de los resultados

- Para entradas pequeñas (16 y 1024 puntos) la versión paralela no mejora el rendimiento. El sobrecoste asociado a la creación y gestión de tareas concurrentes y a la sincronización domina el tiempo total, anulando la ventaja del paralelismo.
- A partir de instancias con decenas de miles de puntos (por ejemplo 32768 puntos) se observa una mejora significativa: la versión paralela obtiene aceleraciones superiores a 5×. Esto indica que el paralelismo amortiza su sobrecoste cuando la cantidad de trabajo es grande.
- En el caso más grande medido (100000 puntos, 64 clusters) la aceleración se mantiene elevada (≈4.72×), confirmando que el enfoque paralelo escala bien para entradas masivas.
- La variación del número de clusters influye en la carga por cluster. Cuando hay muchos clusters con pocos puntos cada uno, la ganancia del paralelismo de datos disminuye debido a tareas más pequeñas por subtarea.
- Recomendación práctica: emplear la versión paralela para conjuntos de datos grandes (decenas de miles de puntos o más) y mantener la versión secuencial para pruebas pequeñas o entornos con bajo paralelismo disponible.

4. Análisis comparativo de las diferentes soluciones

4.1. Equivalencia algorítmica entre versiones

Las implementaciones paralelas (`clasificarPar` , `actualizarPar` , `hayConvergenciaPar` , `kMedianasPar`) fueron diseñadas como versiones paralelizadas de sus homólogas secuenciales manteniendo la misma especificación funcional. La verificación empírica mostró equivalencia en los resultados finales de las medianas:

- Medianas secuenciales: (0.43, 0.87), (0.91, 0.10), (0.43, 0.87)
- Medianas paralelas: (0.43, 0.87), (0.91, 0.10), (0.43, 0.87)
- Resultados equivalentes: Sí

Esto respalda que las transformaciones paralelas no modifican la semántica del algoritmo.

4.2. Paralelismo de tareas

Descripción:

- Se aplicó paralelismo de tareas en funciones donde el trabajo se puede dividir en subtareas independientes (p. ej., particionar el conjunto de puntos para clasificación, comprobar convergencia por bloques).
  Ventajas:
- Permite explotar múltiples núcleos dividiendo el trabajo en unidades de tamaño ajustable mediante un umbral.
- Buena escalabilidad en problemas con gran cantidad de puntos.
  Limitaciones:
- Overhead de creación y sincronización de tareas puede sluperar la ventaja en entradas pequeñas.
- Rendimiento sensible al tamaño del granulo (granularity) y al umbral elegido.

4.3. Paralelismo de datos

Descripción:

- Se utilizó paralelismo de datos en operaciones de agregación sobre colecciones (p. ej., cálculo de promedios de clusters usando colecciones paralelas).
  Ventajas:
- Acelera operaciones de reducción y agregación sobre colecciones grandes.
- Implementación directa sobre colecciones paralelas del lenguaje.
  Limitaciones:
- Menor beneficio si la longitud de las colecciones es pequeña o muy fragmentada por cluster.
- Posibles costes adicionales por construcción y recorrido de iteradores paralelos.

4.4. Combinación de estrategias y conclusiones técnicas

- La versión `kMedianasPar` combina paralelismo de tareas y de datos: las fases gruesas (clasificación, comprobación de convergencia) se paralelizaban por tareas; las agregaciones internas (promedios) se paralelizaban por datos.
- La combinación resultó ser la más efectiva en instancias grandes, ya que explota paralelismo a dos niveles: particionado de la carga de trabajo y aceleración de operaciones internas.
- En problemas con muchos clusters y pocos puntos por cluster, conviene priorizar paralelismo de tareas con cuidado en el tamaño de las subtareas; en problemas con clusters densos (muchos puntos por cluster) el paralelismo de datos aporta un mayor beneficio.
- Es importante ajustar umbrales de paralelización y medir rendimiento en el hardware objetivo: el número de núcleos, la latencia de sincronización y la memoria compartida afectan la ganancia real.
- Recomendación de implementación: incluir parámetros configurables (umbral de paralelización, número máximo de tareas) y proporcionar perfiles de ejecución para seleccionar la configuración óptima según el tamaño del problema.

5. Conclusiones

- La paralelización de algoritmos de clustering como KMeans puede ofrecer mejoras significativas en rendimiento para conjuntos de datos grandes, siempre que se utilicen estrategias adecuadas de paralelismo.
- La elección entre paralelismo de tareas y de datos debe basarse en la naturaleza del problema, el tamaño de los datos y la arquitectura del sistema.
- La evaluación empírica es crucial para validar la efectividad de las implementaciones paralelas y ajustar parámetros de configuración.
- En general, la paralelización es una herramienta poderosa para mejorar el rendimiento en algoritmos (mas aun en entornos multicore y con una cantidad creciente de datos), pero requiere un análisis cuidadoso para maximizar sus beneficios.

Es muy importante comprender las características del problema y del entorno de ejecución para diseñar soluciones paralelas eficientes y efectivas. De esta forma, se pueden aprovechar al máximo las capacidades de hardware disponibles y mejorar significativamente los tiempos de procesamiento en aplicaciones de clustering, analítica de datos y cualquier otro dominio que requiera procesamiento intensivo.