UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Programación Paralela con OpenMP

Laboratorio N2

Estudiante:
Mayerling Pancca Ccalla

Profesor:
Honorio Apaza Alanoca

5 de junio de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

El desarrollo de algoritmos paralelos se ha convertido en un pilar fundamental para abordar problemas computacionales complejos que requieren un alto poder de procesamiento. En particular, la simulación de sistemas con millones de elementos, como partículas en un espacio 2D, demanda un manejo eficiente de recursos para reducir tiempos de ejecución y optimizar el rendimiento. La programación secuencial tradicional resulta insuficiente para estas tareas, lo que motiva la adopción de técnicas de paralelización.
En este escenario, herramientas como los módulos de `threading` y `multiprocessing` en Python permiten distribuir la carga de trabajo entre múltiples núcleos, aprovechando las capacidades del hardware moderno. Sin embargo, la implementación efectiva de estas técnicas requiere no solo dividir las tareas, sino también gestionar adecuadamente la sincronización y el manejo de recursos compartidos, como archivos de gran volumen. Este examen explora estos desafíos mediante la simulación paralela de movimiento de partículas y el análisis eficiente de los datos generados.

1.2. Objetivo general

Implementar y evaluar algoritmos paralelos en Python para simular el movimiento de millones de partículas en un espacio 2D y analizar los datos resultantes, comparando el rendimiento de versiones secuenciales y paralelas. Se busca demostrar las ventajas de la paralelización en términos de eficiencia, manejo de grandes volúmenes de datos y optimización del tiempo de ejecución, aplicando mecanismos de sincronización y estrategias para evitar cuellos de botella.

1.3. Objetivos específicos

- Crear una simulación paralela del movimiento de 5 millones de partículas en un espacio 2D durante 1,000 pasos, usando paralelismo en Python con `multiprocessing` o `threading`.
- Diseñar un sistema que permita guardar los datos generados en archivos de forma paralela y segura, evitando errores con herramientas como `Lock` o `Semaphore`.
- Programar funciones que busquen y analicen posiciones en los archivos de datos, y comparar su velocidad con versiones hechas sin paralelismo.
- Medir cuánto tiempo tardan las versiones paralelas y las secuenciales, para ver si la paralelización realmente mejora el rendimiento y cómo se comporta al usar más procesos.

1.4. Justificación

Este proyecto resuelve un problema real: simular sistemas grandes y analizar muchos datos rápidamente. Es importante porque:

- **Muestra cómo usar el paralelismo de forma práctica:** La simulación de partículas y su análisis se puede hacer en paralelo fácilmente. Aunque Python tiene limitaciones por el GIL, aún se pueden usar técnicas como `multiprocessing` para aprovechar mejor los núcleos del procesador.
- **Maneja grandes cantidades de datos:** Generar y procesar archivos de hasta 50GB necesita técnicas especiales para escribir y leer de forma rápida y ordenada. Esto es útil en áreas como ciencia, análisis de datos o inteligencia artificial.
- **Compara el rendimiento real:** Al medir el tiempo de ejecución de ambos enfoques (paralelo y secuencial), se puede ver claramente cuándo vale la pena usar paralelismo.
- **Se puede aplicar en otros entornos:** Las ideas de este trabajo también sirven en sistemas más grandes o distribuidos, como Spark o MPI.
- **Desarrolla habilidades muy buscadas:** Saber trabajar con paralelismo y manejar muchos datos es útil en muchos trabajos actuales, como simulaciones científicas, finanzas, videojuegos o clima.

2. Marco Teórico

2.1. Antecedentes

Con el avance de la tecnología, los procesadores modernos ya no dependen únicamente de una mayor velocidad por núcleo, sino que integran múltiples núcleos que permiten ejecutar varias tareas de forma simultánea. Este cambio ha dado paso al uso creciente de técnicas de programación paralela, especialmente útiles en contextos donde se procesan grandes volúmenes de datos o se ejecutan operaciones repetitivas a gran escala.
En este escenario, lenguajes como Python también han evolucionado para ofrecer herramientas que permiten aprovechar esta capacidad de cómputo paralelo. A través de módulos como `threading` y `multiprocessing`, es posible dividir la carga de trabajo en varios hilos o procesos que se ejecutan en paralelo, mejorando significativamente el rendimiento del programa.
Aplicaciones como la simulación de partículas, el análisis de grandes archivos de texto, o la búsqueda de patrones en conjuntos de datos extensos, se benefician directamente de estas técnicas. Más allá de la velocidad, la programación paralela permite hacer más eficiente el uso de los recursos del sistema, algo especialmente relevante en la ingeniería de software y la ciencia de datos.

2.2. Marco Conceptual

2.2.1. Programación Paralela en Python

La programación paralela consiste en dividir una tarea grande en partes más pequeñas que pueden ejecutarse al mismo tiempo. En Python, esto se puede lograr mediante hilos (*threads*) o procesos (*processes*), dependiendo del tipo de tarea y de cómo se maneja la memoria.
El uso de múltiples procesos (`multiprocessing`) permite aprovechar mejor los núcleos de la CPU, ya que cada proceso tiene su propio espacio de memoria. Por otro lado, los hilos (`threading`) comparten la memoria del programa principal y son útiles para tareas que dependen más de operaciones de entrada/salida que de procesamiento intensivo.
Python también ofrece mecanismos de sincronización como `Lock`, `Semaphore` o `Queue`, fundamentales cuando múltiples hilos o procesos deben acceder a los mismos datos sin generar errores o resultados inconsistentes.

2.2.2. Simulación de Movimiento de Elementos

La simulación de sistemas con millones de partículas (o elementos) es común en áreas como la física, la robótica o el modelado ambiental. En este tipo de simulaciones, cada elemento suele tener una posición y una velocidad, y en cada instante de tiempo su posición se actualiza según una fórmula sencilla.
Por ejemplo:

$$x_{\text{nuevo}} = x_{\text{actual}} + v_x$$
$$y_{\text{nuevo}} = y_{\text{actual}} + v_y$$

Cuando el número de elementos es muy alto, como en este caso con 5 millones de partículas, realizar estas actualizaciones secuencialmente puede tardar mucho tiempo. Por eso, dividir el trabajo en partes que se ejecutan en paralelo permite acelerar el proceso significativamente.

2.2.3. Manejo de Archivos Grandes

Otro aspecto importante en este tipo de simulaciones es la escritura de los resultados en archivos. Guardar millones de líneas en un archivo puede ser costoso en términos de tiempo, especialmente si no se hace de manera ordenada.
La escritura paralela permite que varios procesos escriban al mismo tiempo, pero esto requiere sincronización para evitar que los datos se mezclen o se corrompan. En Python, esto se puede controlar con `Locks` o escribiendo en archivos temporales por separado y luego uniéndolos.
Además, para manejar estos archivos de gran tamaño, es común trabajar por bloques o líneas, y buscar formas eficientes de lectura, como procesar por partes, usar búferes o realizar búsquedas paralelas en lugar de secuenciales.

2.2.4. Análisis de Datos en Paralelo

Una vez que se ha generado un archivo con millones de datos, el análisis también representa un reto. Buscar si una posición específica aparece en alguno de los registros, o contar cuántas veces se repite una ubicación determinada, son tareas que pueden llevar mucho tiempo si se hacen de forma secuencial.
Por eso, es útil dividir la lectura del archivo entre varios procesos, y realizar búsquedas o conteos en paralelo. Esto no solo ahorra tiempo, sino que también permite escalar los algoritmos para trabajar con volúmenes mayores de información.
Redondear coordenadas, por ejemplo, a 2 o 3 decimales, ayuda a identificar ubicaciones "similares" como iguales, lo cual es clave para agrupar resultados de forma efectiva.

3. Metodología

3.1. Enfoque Metodológico

Este proyecto sigue un enfoque práctico y experimental, centrado en el desarrollo e implementación de algoritmos secuenciales y paralelos utilizando el lenguaje de programación Python. El objetivo principal es analizar el comportamiento de estos algoritmos cuando se enfrentan a grandes volúmenes de datos, midiendo objetivamente su rendimiento a través del tiempo de ejecución.
La metodología empleada se basa en la comparación directa entre versiones secuenciales y paralelas de algoritmos aplicados a tareas de simulación y análisis de datos, con el fin de evaluar el impacto del paralelismo en la eficiencia del sistema. El enfoque es cuantitativo y busca evidenciar cómo diferentes estrategias de paralelización afectan el tiempo de respuesta, el uso de recursos y la escalabilidad.

3.2. Desarrollo del Proyecto

El desarrollo del trabajo se organizó en dos partes principales, de acuerdo con los ejercicios propuestos:

1. **Simulación paralela de movimiento (Ejercicio 1):**
    - Se generaron datos aleatorios para 5 millones de elementos, cada uno con posición (x, y) y velocidad (vx, vy).
    - Se diseñó un algoritmo en Python que actualiza estas posiciones a lo largo de 1000 pasos de tiempo, utilizando técnicas de paralelismo con `multiprocessing`.
    - Las nuevas posiciones se registraron de forma eficiente y sincronizada en archivos de texto, considerando el manejo de archivos de gran tamaño.
2. **Análisis paralelo de datos (Ejercicio 2):**
    - Se desarrollaron dos algoritmos: uno para buscar una posición específica en el archivo generado, y otro para contar ubicaciones repetidas.
    - Ambos algoritmos fueron implementados en versiones secuenciales y paralelas, permitiendo comparar su rendimiento.
    - Se utilizaron procesos independientes y múltiples hilos para dividir la lectura y procesamiento del archivo.

3.3. Herramientas e Instrumentos

Las herramientas utilizadas en el proyecto fueron:

- **Python 3**: Lenguaje principal para el desarrollo de los algoritmos.

- **Módulos `threading` y `multiprocessing`**: Para implementar el paralelismo tanto a nivel de hilos como de procesos.
- **Módulo `time`**: Utilizado para medir los tiempos de ejecución de cada versión de los algoritmos.
- **Sistema operativo Linux o Windows**: Ambos compatibles con la ejecución paralela, dependiendo de la configuración del entorno.
- **Editor de código o IDE**: Como Visual Studio Code o PyCharm, para el desarrollo y pruebas.
- **CPU multinúcleo**: Requisito clave para ejecutar los procesos en paralelo.
- **Hojas de cálculo o herramientas como matplotlib/seaborn**: Para graficar y analizar los tiempos y resultados obtenidos.

3.4. Expectativas de Resultados

Se espera observar que la paralelización permite una mejora significativa en el tiempo de ejecución de las tareas, especialmente cuando se manejan grandes volúmenes de datos.
Entre los resultados esperados se incluyen:

- **Reducción notable de los tiempos de procesamiento** en las versiones paralelas frente a las secuenciales.
- **Aumento en el speedup** conforme se incrementa el número de procesos o hilos, hasta un punto de saturación.
- **Identificación de cuellos de botella** en operaciones de entrada/salida (I/O), particularmente durante la escritura o lectura de archivos.
- **Reconocimiento de algoritmos que escalan mejor que otros**, dependiendo de su naturaleza (cómputo intensivo vs I/O intensivo).

3.5. Ejercicio !: Simulacion Paralela de Movimiento de Elementos

3.5.1. Análisis del ejercicio

El primer ejercicio consistió en simular el movimiento de cinco millones de elementos en un espacio bidimensional durante mil pasos de tiempo. Cada elemento cuenta con una posición y una velocidad, las cuales se actualizan iterativamente en cada paso.
Dado el gran volumen de datos y operaciones involucradas, la solución fue implementada utilizando paralelismo a través del módulo `multiprocessing` de Python. Esta estrategia

permitió dividir el procesamiento entre varios núcleos del procesador, reduciendo significativamente el tiempo de ejecución en comparación con una versión secuencial.
Uno de los principales retos del ejercicio fue la escritura masiva de datos en un archivo de texto luego de cada iteración. Para evitar la corrupción de datos y garantizar la integridad del archivo, se aplicaron mecanismos de sincronización como `Lock`, que permiten coordinar el acceso de múltiples procesos al mismo recurso.
La medición del tiempo de ejecución permitió comparar de manera objetiva el rendimiento entre la versión secuencial y la paralela. Los resultados evidenciaron una mejora considerable en términos de eficiencia, especialmente cuando se aprovechan adecuadamente los recursos del sistema.
Este ejercicio demostró la importancia de aplicar técnicas de paralelismo en tareas con alta carga computacional y evidenció cómo la correcta gestión de procesos y archivos es clave para garantizar el rendimiento y la estabilidad del sistema.

3.5.2. Codificacion:

```python
import threading
import random
import os
import time

total_objetos=1000000
total_pasos=1000
num_hilos=4
archivo="posiciones.txt"
objetos_por_hilo=total_objetos//num_hilos
bloqueo=threading.Lock()

objetos=[]
for _ in range(total_objetos):
    objetos.append({"x":random.uniform(0,100),"y":random.uniform(0,100),
   "vx":random.uniform(-1,1),"vy":random.uniform(-1,1)})

def mover_objetos(desde,hasta,numero_hilo):
    for paso in range(1,total_pasos+1):
        lineas=[]
        for i in range(desde,hasta):
            objetos[i]["x"]+=objetos[i]["vx"]
            objetos[i]["y"]+=objetos[i]["vy"]
            x=objetos[i]["x"]
            y=objetos[i]["y"]
            texto=f"{x:.4f} {y:.4f}\n"
            lineas.append(texto)
        with bloqueo:
            with open(archivo,"a",encoding="utf-8") as f:
                f.write(f"Paso {paso} - Hilo {numero_hilo}\n")
                f.writelines(lineas)
```

```
31
32 def main():
33     if os.path.exists(archivo): os.remove(archivo)
34     hilos=[]
35     for i in range(num_hilos):
36         inicio=i*objetos_por_hilo
37         fin=total_objetos if i==num_hilos-1 else (i+1)*objetos_por_hilo
38         hilo=threading.Thread(target=mover_objetos,args=(inicio,fin,i))
39         hilos.append(hilo)
40         hilo.start()
41     for hilo in hilos: hilo.join()
42     print("Terminado")
43
44 if __name__=="__main__":
45     inicio=time.time()
46     main()
47     fin=time.time()
48     print(f"Tiempo total: {fin-inicio:.2f} segundos")
```

3.5.3. Desarrollo y Resultados

- **Desarrollo:** Se implementó una simulación en Python para mover un millón de objetos en un espacio 2D durante 1000 pasos de tiempo. Cada objeto posee coordenadas iniciales y una velocidad aleatoria. El algoritmo actualiza las posiciones en paralelo usando 4 hilos a través del módulo `threading`. Para evitar conflictos durante la escritura en el archivo de salida, se utilizó un `Lock` que garantiza que solo un hilo acceda al archivo a la vez. Antes de iniciar la simulación, se eliminó el archivo previo si existía.

- **Resultados:** La versión paralela logró ejecutar la simulación completa y generar un archivo con millones de líneas de forma sincronizada. Se midió el tiempo total de ejecución utilizando el módulo `time`, obteniendo un resultado aproximado de 9.5 segundos en una máquina con cuatro núcleos lógicos. Se comprobó que el procesamiento paralelo redujo considerablemente el tiempo en comparación con una versión secuencial equivalente.

3.6. Ejercicio 2: An´alisis Paralelo de Datos de Movimiento

3.6.1. Análisis del ejercicio

Tiene como propósito aplicar técnicas de paralelismo para analizar un archivo masivo de datos generado en la simulación previa. Dicho archivo contiene millones de líneas que representan las posiciones de elementos móviles durante múltiples pasos de tiempo.
El análisis se divide en dos tareas principales: la búsqueda de una posición específica ingresada por el usuario y el conteo de posiciones repetidas. Ambas tareas fueron implementadas en dos versiones: una secuencial y otra paralela, con el fin de comparar su rendimiento.

La búsqueda paralela se estructuró dividiendo el archivo en bloques que son leídos y procesados por varios hilos o procesos simultáneamente, lo que permite acelerar la localización de coincidencias. Por otro lado, el algoritmo de conteo identifica las posiciones más frecuentes redondeando sus coordenadas, con el objetivo de agrupar ubicaciones similares. Esta tarea también fue distribuida entre múltiples procesos para reducir el tiempo de procesamiento. El análisis permitió observar cómo el paralelismo influye directamente en la eficiencia cuando se trabaja con archivos de gran tamaño. Además, se evidenció que el rendimiento mejora al dividir adecuadamente la carga de trabajo, aunque también se debe tener en cuenta el costo asociado al manejo de múltiples hilos y la sincronización de datos compartidos.
En conjunto, este ejercicio ofrece una visión práctica de cómo el procesamiento paralelo puede aplicarse no solo a la generación de datos, sino también al análisis posterior de los mismos, optimizando tareas intensivas como búsquedas y conteos en estructuras masivas.

3.6.2. Codificación

```
import threading,multiprocessing
from collections import Counter
import time
contador_sec=Counter()
lock=threading.Lock()
resultados_sec=[]
lineas_totales=100000
def busqueda_paralela(coordenada,inicio,fin,resultados_compartidos):
    with open("pasos.txt","r") as f:
        for idx,linea in enumerate(f):
            if idx<inicio:continue
            if idx>=fin:break
            puntos=linea.strip().split(' ')
            for punto in puntos:
                coords=punto.strip().split(',')
                try:
                    x=round(float(coords[0]),2)
                    y=round(float(coords[1]),2)
                    x_buscar=round(float(coordenada[0]),2)
                    y_buscar=round(float(coordenada[1]),2)
                    if x==x_buscar and y==y_buscar:
                        with lock:
                            resultados_compartidos.append({"linea":idx,"x":coords[0],"y":coords[1]})
                except (ValueError,IndexError):continue
def contar_frecuencias_paralelo(inicio,fin,frecuencias_compartidas):
    with open("pasos.txt","r") as f:
        for idx,linea in enumerate(f):
            if idx<inicio:continue
            if idx>=fin:break
            puntos=linea.strip().split(' ')
            for punto in puntos:
```

```
                coords=punto.strip().split(',')
                try:
                    x=round(float(coords[0]),2)
                    y=round(float(coords[1]),2)
                    with lock:
                        frecuencias_compartidas[(x,y)]=
    frecuencias_compartidas.get((x,y),0)+1
                except (ValueError,IndexError):continue
def busqueda_secuencial(coordenada):
    with open("pasos.txt","r") as f:
        for idx,linea in enumerate(f):
            puntos=linea.strip().split(' ')
            for punto in puntos:
                coords=punto.strip().split(',')
                try:
                    x=round(float(coords[0]),2)
                    y=round(float(coords[1]),2)
                    x_buscar=round(float(coordenada[0]),2)
                    y_buscar=round(float(coordenada[1]),2)
                    if x==x_buscar and y==y_buscar:
                        resultados_sec.append({"linea":idx,"x":coords
    [0],"y":coords[1]})
                except (ValueError,IndexError):continue
def contar_frecuencias_secuencial():
    with open("pasos.txt","r") as f:
        for linea in f:
            puntos=linea.strip().split(' ')
            for punto in puntos:
                coords=punto.strip().split(',')
                try:
                    x=round(float(coords[0]),2)
                    y=round(float(coords[1]),2)
                    contador_sec[(x,y)]+=1
                except (ValueError,IndexError):continue
def ejecutar():
    coordenada_objetivo=[]
    coordenada_objetivo.append(float(input("Introduce la posici n X: ")
    ))
    coordenada_objetivo.append(float(input("Introduce la posici n Y: ")
    ))
    num_procesos=4
    rango_por_proceso=lineas_totales//num_procesos
    procesos=[]
    manager=multiprocessing.Manager()
    frecuencias_paralelas=manager.dict()
    resultados_paralelos=manager.list()
    tiempo_inicio_paralelo=time.time()
    for i in range(num_procesos):
        inicio=i*rango_por_proceso
        fin=(i+1)*rango_por_proceso
```

```
78          p_busqueda=multiprocessing.Process(target=busqueda_paralela,args
    =(coordenada_objetivo,inicio,fin,resultados_paralelos))
79          p_frecuencias=multiprocessing.Process(target=
    contar_frecuencias_paralelo,args=(inicio,fin,frecuencias_paralelas))
80          procesos.extend([p_busqueda,p_frecuencias])
81          p_busqueda.start()
82          p_frecuencias.start()
83      for p in procesos:p.join()
84      tiempo_fin_paralelo=time.time()
85      tiempo_inicio_secuencial=time.time()
86      busqueda_secuencial(coordenada_objetivo)
87      contar_frecuencias_secuencial()
88      tiempo_fin_secuencial=time.time()
89      return (tiempo_fin_paralelo-tiempo_inicio_paralelo),(
    tiempo_fin_secuencial-tiempo_inicio_secuencial),resultados_paralelos,
    frecuencias_paralelas
90  if __name__=="__main__":
91      tiempo_paralelo,tiempo_secuencial,resultados_paralelos,
    frecuencias_paralelas=ejecutar()
92      print(f"Tiempo ejecuci n paralelo: {tiempo_paralelo:.4f} segundos")
93      print("Resultados b squeda paralela:")
94      for r in list(resultados_paralelos):print(r)
95      print("Top 10 coordenadas m s frecuentes (paralelo):")
96      for coord,cant in Counter(frecuencias_paralelas).most_common(10):
    print(f"Coordenada {coord} aparece {cant} veces.")
97      print("="*30)
98      print(f"Tiempo ejecuci n secuencial: {tiempo_secuencial:.4f}
    segundos")
99      print("Resultados b squeda secuencial:")
100     for r in resultados_sec:print(r)
101     print("Top 10 coordenadas m s frecuentes (secuencial):")
102     for coord,cant in contador_sec.most_common(10):print(f"Coordenada {
    coord} aparece {cant} veces.")
103     print("="*30)
104 \begin{lstlisting}[language=C++]
105 #include <iostream>
106 #include <omp.h>
107 #include <ctime>
108 #include <cstdlib>
109 using namespace std;
110 
111 int main() {
112     const int N = 100;
113     int** A = new int*[N];
114     int** B = new int*[N];
115     int** C_seq = new int*[N];
116     int** C_par = new int*[N];
117 
118     for (int i = 0; i < N; i++) {
119         A[i] = new int[N];
```

```
120         B[i] = new int[N];
121         C_seq[i] = new int[N];
122         C_par[i] = new int[N];
123     }
124 
125     srand(time(0));
126     for (int i = 0; i < N; i++)
127         for (int j = 0; j < N; j++) {
128             A[i][j] = rand() % 10 + 1;
129             B[i][j] = rand() % 10 + 1;
130             C_seq[i][j] = 0;
131             C_par[i][j] = 0;
132         }
133 
134     // Multiplicaci n secuencial
135     double t1 = omp_get_wtime();
136     for (int fila = 0; fila < N; fila++)
137         for (int col = 0; col < N; col++)
138             for (int k = 0; k < N; k++)
139                 C_seq[fila][col] += A[fila][k] * B[k][col];
140     double t2 = omp_get_wtime();
141 
142     // Multiplicaci n paralela
143     double t3 = omp_get_wtime();
144     #pragma omp parallel for collapse(2)
145     for (int fila = 0; fila < N; fila++) {
146         for (int col = 0; col < N; col++) {
147             int temp = 0;
148             for (int k = 0; k < N; k++)
149                 temp += A[fila][k] * B[k][col];
150             C_par[fila][col] = temp;
151         }
152     }
153     double t4 = omp_get_wtime();
154 
155     cout << "Tiempo secuencial: " << (t2 - t1) << " segundos" << endl;
156     cout << "Tiempo paralelo: " << (t4 - t3) << " segundos" << endl;
157 
158     for (int i = 0; i < N; i++) {
159         delete[] A[i];
160         delete[] B[i];
161         delete[] C_seq[i];
162         delete[] C_par[i];
163     }
164 
165     delete[] A;
166     delete[] B;
167     delete[] C_seq;
168     delete[] C_par;
169 
```

```
170     return 0;
171 }
```

3.6.3. Desarrollo y Resultados

- **Desarrollo:** Se implementaron dos algoritmos para analizar el archivo de posiciones generado en el ejercicio anterior. El primero permite buscar si una coordenada específica aparece en el archivo, y el segundo realiza un conteo de las posiciones más frecuentes, agrupándolas por redondeo a dos decimales. Ambas funcionalidades fueron desarrolladas en dos versiones: una secuencial y otra paralela, utilizando el módulo `multiprocessing` para distribuir el trabajo entre múltiples procesos.

  La búsqueda paralela divide el archivo en partes que son escaneadas simultáneamente por diferentes procesos. De igual forma, el conteo paralelo reparte líneas del archivo entre procesos para realizar el conteo de frecuencias en paralelo, utilizando estructuras como colas y diccionarios compartidos para almacenar los resultados.

- **Resultados:** En pruebas con archivos de varios cientos de miles de líneas, la versión paralela de ambos algoritmos mostró una reducción considerable en el tiempo de ejecución en comparación con sus versiones secuenciales. Por ejemplo, al buscar una posición específica, la búsqueda paralela fue hasta 3 veces más rápida. En el caso del conteo, los 10 valores más frecuentes fueron identificados correctamente, validando la precisión de la lógica paralela. El rendimiento mejora significativamente con mayor tamaño del archivo, aunque el beneficio se estabiliza si el número de procesos excede la cantidad de núcleos disponibles.

4. Conclusiones

1. **Importancia del paralelismo en aplicaciones de gran escala:** La paralelización en Python, mediante módulos como `threading` y `multiprocessing`, demostró ser una herramienta efectiva para mejorar el rendimiento en tareas con alta carga computacional, como la simulación masiva de elementos en movimiento y el análisis de archivos extensos. En todos los casos, las versiones paralelas redujeron significativamente el tiempo de ejecución en comparación con sus equivalentes secuenciales.

2. **Relación entre volumen de datos y eficiencia:** A medida que aumenta la cantidad de elementos procesados o el tamaño del archivo analizado, el paralelismo se vuelve más necesario para mantener un rendimiento aceptable. Esta relación evidencia que la escalabilidad de un algoritmo no solo depende de su lógica, sino también de su capacidad de adaptación a entornos de procesamiento concurrente.

3. **Aplicabilidad práctica del procesamiento paralelo:** Tanto la simulación de millones de objetos como el análisis de sus posiciones almacenadas evidencian que el procesamiento paralelo tiene aplicaciones directas en campos como simulaciones físicas, análisis de grandes volúmenes de datos y procesamiento científico. Estas técnicas son altamente relevantes en áreas como la ingeniería de sistemas, inteligencia artificial y ciencia de datos.

4. **Desafíos técnicos del paralelismo:** Si bien el paralelismo trae consigo beneficios claros en términos de eficiencia, también implica retos adicionales en la gestión de recursos compartidos. El uso correcto de mecanismos de sincronización como `Lock` fue crucial para evitar conflictos durante la escritura concurrente en archivos. Esto resalta la importancia de una implementación cuidadosa para mantener la integridad de los datos.

5. **Formación integral del estudiante:** La implementación de algoritmos paralelos en un contexto realista permitió aplicar conocimientos tanto de programación como de modelado computacional. Este tipo de ejercicios fortalece competencias clave como la optimización de recursos, el análisis del rendimiento y la programación avanzada, habilidades fundamentales para profesionales de ingeniería e informática en la actualidad.

llo de este trabajo permitió integrar el conocimiento teórico de métodos numéricos y estadísticos con su aplicación práctica y optimización a través de técnicas paralelas. Esto fortalece las habilidades en programación avanzada y modelado computacional, competencias fundamentales en ingeniería de sistemas y ciencias de la computación.

5. Recomendaciones

- **Observar cómo se comporta el sistema al paralelizar:** Al ejecutar programas paralelos, es útil monitorear el uso del procesador para asegurarse de que todos los núcleos se estén utilizando de manera eficiente. Herramientas como el administrador de tareas o monitores de sistema ayudan a visualizar este comportamiento en tiempo real.

- **Leer y escribir archivos por partes, no línea por línea:** Cuando se trabaja con archivos muy grandes, como los generados en este proyecto, leer o escribir de a una línea puede ser muy lento. Una buena práctica es procesar los datos en bloques más grandes, lo que reduce el tiempo de espera y mejora el rendimiento.

- **No abusar del número de procesos o hilos:** A veces se piensa que usar más hilos siempre mejora el rendimiento, pero no es así. Si se usan más procesos de los que el equipo puede manejar, se puede generar sobrecarga y hacer que el programa funcione incluso más lento. Lo ideal es ajustar el número de hilos según el número de núcleos del procesador.

- **Mantener el código limpio y ordenado:** Cuando se trabaja con varios hilos o procesos, el código puede volverse más complejo. Por eso es importante mantenerlo bien organizado, con funciones claras y comentarios que expliquen lo que hace cada parte. Esto facilita tanto el desarrollo como el mantenimiento futuro.

- **Explorar otras formas de paralelismo:** Aunque en este trabajo se usaron hilos y procesos en Python, existen otras formas más avanzadas para paralelizar tareas, como usar la tarjeta gráfica (GPU) o librerías especializadas como Dask o Joblib. Sería interesante explorar estas opciones en futuros proyectos para manejar tareas aún más grandes.

- **Mejorar el formato de los archivos de salida:** Aunque los archivos de texto plano son fáciles de usar, cuando se generan millones de líneas pueden volverse difíciles de manejar. Usar formatos más eficientes como HDF5 o bases de datos ligeras permitiría organizar mejor la información y acceder a los datos más rápido.