UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Implementación de Algoritmos Paralelos para la Simulación y Análisis de Movimiento de Partículas en Python

Algoritmos Paralelos

Estudiantes:
Gersael Mathias Rojas Quijano

Profesores:
Honorio Apaza Alanoca

5 de junio de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

En la era del procesamiento masivo de datos y el crecimiento exponencial de la información, la necesidad de desarrollar algoritmos eficientes se ha vuelto una prioridad en el ámbito de la ingeniería de sistemas. El paralelismo, como paradigma de programación, ofrece una solución clave al problema del rendimiento computacional en aplicaciones que manejan grandes volúmenes de datos, como simulaciones físicas, análisis de datos y modelos predictivos.
Este trabajo surge en el marco del curso "Algoritmos Paralelos" de la Universidad Nacional de Moquegua, donde se plantea el desafío de implementar dos ejercicios prácticos que combinan la simulación de movimientos de millones de elementos en un espacio bidimensional y el análisis de los datos generados, aplicando técnicas de paralelización con Python. Estas prácticas permiten entender en la realidad el impacto del procesamiento paralelo, comparándolo con métodos secuenciales tradicionales y evidenciando los beneficios en términos de tiempo y eficiencia.

1.2. Objetivo general

Implementar y analizar soluciones paralelas eficientes para la simulación del movimiento de múltiples elementos y el posterior análisis de los datos generados, utilizando herramientas de paralelismo en Python.

1.3. Objetivos específicos

- Simular el movimiento de 5 millones de elementos en un espacio 2D durante 1,000 pasos de tiempo, implementando algoritmos paralelos para la actualización de posiciones.
- Desarrollar un sistema de escritura de datos paralelo y sincronizado que garantice la integridad de los resultados en archivos de gran tamaño.
- Implementar un algoritmo de búsqueda paralela de posiciones específicas dentro del archivo generado.
- Diseñar un algoritmo de conteo de posiciones más frecuentes, utilizando redondeo para flotantes y técnicas de paralelización.
- Comparar el rendimiento entre versiones secuenciales y paralelas de los algoritmos desarrollados.
- Analizar y documentar los resultados obtenidos, destacando el impacto del paralelismo en el rendimiento computacional.

1.4. Justificación

son críticos. Este trabajo no solo permite poner en práctica conocimientos teóricos sobre programación concurrente y paralela, sino que también responde a una necesidad real de optimización en aplicaciones que procesan grandes cantidades de datos. Mediante la utilización del lenguaje Python y sus bibliotecas de concurrencia, se demuestra cómo problemas complejos pueden ser abordados con estrategias efectivas, mejorando significativamente los tiempos de ejecución y el manejo de recursos. Además, la experiencia obtenida aporta una base sólida para futuros desarrollos en entornos de cómputo de alto rendimiento (HPC) y aplicaciones de ciencia de datos.

2. Marco teórico

2.1. Antecedentes

El desarrollo de algoritmos paralelos ha cobrado relevancia en los últimos años debido al crecimiento de los volúmenes de datos y la necesidad de acelerar los tiempos de procesamiento en diversas áreas como simulaciones físicas, inteligencia artificial, análisis de grandes datos (Big Data) y gráficos computacionales. Investigaciones previas han demostrado que el paralelismo puede aumentar significativamente la eficiencia de los programas al distribuir las tareas entre múltiples hilos o procesos.
Herramientas como Python, con sus módulos threading y multiprocessing, han democratizado el acceso a la programación concurrente, permitiendo a estudiantes y profesionales explorar estrategias paralelas sin la necesidad de infraestructuras complejas.
En el contexto educativo, los ejercicios prácticos como los planteados en este trabajo permiten vincular teoría y aplicación, promoviendo la adquisición de competencias en diseño e implementación de soluciones eficientes para el procesamiento de datos en paralelo.

2.2. Marco conceptual

A continuación, se presentan los principales conceptos teóricos necesarios para comprender el desarrollo de esta monografía:

2.2.1. Tema 1: Algoritmos Paralelos

Los algoritmos paralelos son aquellos diseñados para ejecutarse simultáneamente en múltiples unidades de procesamiento. Su propósito es dividir una tarea grande en subprocesos más pequeños que se pueden ejecutar en paralelo, reduciendo así el tiempo total de ejecución. A diferencia de los algoritmos secuenciales, que ejecutan instrucciones una por una, los paralelos aprovechan los recursos de hardware multicore o multiprocesador.

2.2.2. Tema 2: Paralelismo en Python

Python, aunque tradicionalmente interpretado como un lenguaje secuencial, cuenta con bibliotecas para la programación paralela:

- **threading:** permite la ejecución de múltiples hilos dentro de un solo proceso. Es útil para tareas I/O (como escritura de archivos), pero limitado por el Global Interpreter Lock (GIL).
- **multiprocessing:** permite crear múltiples procesos independientes, cada uno con su propia memoria, evitando las restricciones del GIL y siendo más eficiente para tareas CPU-intensivas.

2.2.3. Tema 3: Simulación de Movimiento en Espacios 2D

La simulación computacional de movimientos implica representar digitalmente el comportamiento de agentes o partículas en un entorno definido. En este caso, los elementos se desplazan en un plano 2D (dos dimensiones) a partir de una posición inicial y un vector de velocidad. La fórmula:

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

2.2.4. Tema 4: Escritura Paralela y Sincronización

Cuando varios hilos o procesos intentan escribir simultáneamente en un archivo, se pueden producir errores como sobrescrituras, pérdida de datos o corrupción de archivo. Para evitarlo, se usan mecanismos de sincronización como:

2.2.5. Tema 5: Lectura de Archivos en Paralelo

Leer archivos grandes de forma eficiente es un desafío común. La lectura paralela consiste en dividir el archivo en bloques y asignar cada bloque a un hilo o proceso para su procesamiento. Esto requiere una gestión cuidadosa para evitar solapamientos o pérdida de datos al inicio y fin de los bloques.

2.2.6. Tema 6: Búsqueda y Conteo en Archivos Masivos

- **Búsqueda paralela** se refiere a localizar una posición específica en un conjunto de datos utilizando varios hilos/procesos simultáneamente, mejorando el tiempo de búsqueda.Interpreter Lock (GIL).
- **Conteo de ubicaciones:** implica recorrer los datos y contar las repeticiones de ciertas posiciones. Dado que los datos están en punto flotante, se aplica un redondeo (por ejemplo, a 2 o 3 decimales) para considerar equivalencias.

2.2.7. Tema 7: Medición y Comparación de Rendimiento

Un aspecto clave de la programación paralela es evaluar su rendimiento frente a las versiones secuenciales. Las métricas más comunes incluyen:

- **Tiempo de ejecución:** cuánto tarda el algoritmo en completarse.
- **Speedup (aceleración):** razón entre el tiempo secuencial y el paralelo.
- **Eficiencia:** relación entre el speedup y el número de unidades de procesamiento utilizadas.

Estas métricas permiten cuantificar los beneficios de la paralelización y justificar su implementación.

3. Metodología

3.1. Enfoque Metodológico

El enfoque adoptado en este trabajo es de tipo cuantitativo-experimental, ya que se busca medir y comparar el rendimiento de algoritmos secuenciales y paralelos aplicados a problemas computacionales complejos. Se diseñan dos soluciones prácticas que permiten observar directamente el impacto de la paralelización sobre el tiempo de ejecución y la eficiencia del procesamiento de datos masivos.
La metodología sigue un enfoque aplicativo, orientado a resolver un problema realista mediante el uso de tecnologías de programación concurrente y paralela, específicamente en el lenguaje Python. Asimismo, se incluye un análisis comparativo de los resultados obtenidos entre las versiones secuenciales y paralelas.

3.2. Proceso de Resolución

El trabajo se desarrolla en dos etapas principales, correspondientes a los ejercicios propuestos:

3.2.1. Simulación Paralela de Movimiento de Elementos

- **Generación de datos iniciales:** se crean aleatoriamente 5 millones de elementos con posiciones y velocidades en un espacio 2D.
- **Implementación de la simulación:** se desarrolla un algoritmo para actualizar las posiciones de cada elemento durante 1,000 pasos de tiempo, aplicando técnicas de paralelismo con `multiprocessing` para dividir las tareas entre procesos.
- **Escritura sincronizada:** los resultados de cada paso son escritos en un archivo de texto de forma paralela, utilizando bloqueos (`Lock`) para garantizar la integridad de los datos.
- **Optimización del manejo de archivos grandes:** se aplican estrategias como la escritura por bloques para reducir el tiempo de I/O y evitar la saturación de memoria.

3.2.2. Análisis Paralelo de Datos de Movimiento

- **Búsqueda paralela:** se implementa un algoritmo para encontrar una posición específica dentro del archivo generado, dividiendo la lectura por bloques y utilizando múltiples hilos o procesos.
- **Conteo de posiciones frecuentes:** se recorre el archivo y se cuentan las posiciones más repetidas, considerando un redondeo decimal para tratar los valores flotantes.

- **Comparación de rendimiento:** para ambos algoritmos, se ejecutan versiones secuenciales y paralelas, midiendo el tiempo de ejecución de cada una.

3.3. Instrumentos y Herramientas

Para el desarrollo y ejecución del proyecto se utilizaron los siguientes recursos:

- **Lenguaje de programación:** Python 3.10+
- **Bibliotecas:**
  - `multiprocessing`: para la creación de procesos paralelos.
  - `threading`: para manejo de hilos y sincronización.
  - `time`: para la medición de tiempos de ejecución.
  - `random`: para la generación de datos aleatorios.
- **Editor de código:** Visual Studio Code
- **Sistema operativo:** Windows 10
- **Procesador de pruebas:** CPU con al menos 4 núcleos (quad-core)
- **Instrumentos de evaluación:** scripts de medición de tiempos, pruebas de lectura y escritura en archivos grandes.

3.4. Resultados Esperados

A partir de la implementación y ejecución de las soluciones, se obtuvieron los siguientes resultados específicos:

- Se simuló con éxito el movimiento de 1 millones de elementos durante 100 pasos de tiempo, utilizando procesamiento paralelo, reduciendo significativamente el tiempo frente a una ejecución secuencial.
- Se generó un archivo de gran tamaño con datos de movimiento, empleando escritura sincronizada para evitar corrupción de datos.
- El algoritmo de búsqueda paralela logró reducir el tiempo de localización de posiciones específicas en el archivo en comparación con su versión secuencial.
- El algoritmo de conteo de ubicaciones frecuentes identificó eficientemente las 10 posiciones más comunes en el archivo.
- Las comparaciones de rendimiento mostraron un *speedup* considerable en los algoritmos de análisis al ejecutarse en paralelo.

4. Propuesta

Esta sección presenta el desarrollo práctico de los dos ejercicios propuestos en el curso de Algoritmos Paralelos. Se detalla la implementación de los algoritmos en Python, explicando su funcionamiento, las decisiones tomadas para la paralelización y los pasos ejecutados de acuerdo con la metodología planteada.

4.1. Ejercicio 1: Simulación Paralela de Movimiento de Elementos

El objetivo de este ejercicio fue simular el movimiento de una gran cantidad de elementos (1 millón en esta implementación, debido a restricciones de hardware) a lo largo de 10 pasos de tiempo. Cada elemento posee una posición (x, y) y un vector de velocidad (vx, vy). La actualización de posiciones y el registro de resultados se realizó utilizando `threading` para aplicar paralelismo con múltiples hilos.

4.1.1. Codigo en Python.

```
import threading
import random
import time
import os
cant_elem=1000000
pasos=100
tiempo=1.0
bloqueo=threading.Lock()
def escribir_mover(inicio,fin,hilo_actual):
    txt_temporal=f"temporal_hilo_{hilo_actual}.txt"
    with open(txt_temporal,"w") as archivo:
        for i in range(inicio,fin):
            x=random.uniform(0,1000)
            y=random.uniform(0,1000)
            vx=random.uniform(0,10)
            vy=random.uniform(0,10)
            with bloqueo:
                archivo.write(f"{x:.4f},{y:.4f}")
            for paso in range(pasos):
                x+=vx*tiempo
                y+=vy*tiempo
                with bloqueo:
                    archivo.write(f" {x:.4f},{y:.4f}")
            with bloqueo:
                    archivo.write("\n")
def principal():
    num_hilos=4
    elem_hilo=cant_elem//num_hilos
    tiempo_inicio=time.time()
```

```
    hilos=[]
    inicio=0
    for i in range(num_hilos):
        fin=inicio+elem_hilo
        h=threading.Thread(target=escribir_mover,args=(inicio,fin,i))
        hilos.append(h)
        h.start()
        inicio=fin
    for h in hilos:
        h.join()
    with open("pasos.txt","w") as txt_final:
        for i in range(num_hilos):
            t_eliminar=f"temporal_hilo_{i}.txt"
            with open(t_eliminar,"r") as temporal:
                txt_final.write(temporal.read())
            os.remove(t_eliminar)
    tiempo_fin=time.time()
    print("tiempo: ",tiempo_fin-tiempo_inicio)
if __name__=="__main__":
    principal()
    print("="*30)
```

Descripción de pasos

- Se generan aleatoriamente las posiciones y velocidades iniciales.
- Se divide la carga de trabajo en 4 hilos que procesan bloques de elementos.
- Cada hilo guarda sus resultados en archivos temporales independientes.
- Al finalizar, los archivos temporales se combinan en un único archivo `pasos.txt`.
- Se mide el tiempo total de ejecución.

4.2. Ejercicio 2: Análisis Paralelo de Datos de Movimiento

Este ejercicio consistió en desarrollar dos algoritmos: uno para buscar una posición determinada en el archivo generado anteriormente y otro para identificar las coordenadas más frecuentes. Ambos fueron implementados en versiones secuencial y paralela, utilizando `multiprocessing` y `threading` para comparar el rendimiento.

4.2.1. Codigo en Python.

```
import threading
import multiprocessing
from collections import Counter
import time
frecuencia_sucuencial=Counter()
bloqueo=threading.Lock()
```

```
resultado_s=[]
elementos=100000
''' Paralelo'''
def busqueda_posicion_paralela(buscar, inicio, fin,resultado_p):
    with open("pasos.txt","r") as archivo:
        for indice,linea in enumerate(archivo):
            if inicio<=indice<fin:
                bloques=linea.strip().split(' ')
                for bloque in bloques:
                    valor=bloque.strip().split(',')
                    x=round(float(valor[0]),2)
                    y=round(float(valor[1]),2)
                    w=round(float(buscar[0]),2)
                    z=round(float(buscar[1]),2)
                    if x==w and y==z:
                        with bloqueo:
                            resultado_p.append({"linea":indice, "x":
    valor[0], "y": valor[1]})
            if indice>fin:
                break
def mejores_cinco_paralela(inicio,fin,frecuencia_paralela):
    with open("pasos.txt","r") as archivo:
        for indice,linea in enumerate(archivo):
            if inicio<=indice<fin:
                bloques=linea.strip().split(' ')
                for bloque in bloques:
                    valor=bloque.strip().split(',')
                    x=round(float(valor[0]),2)
                    y=round(float(valor[1]),2)
                    with bloqueo:
                        frecuencia_paralela[(x,y)]=frecuencia_paralela.
    get((x,y),0)+1
            if indice>fin:
                break
'''Secuencial'''
def busqueda_posicion_secuencial(buscar):
    with open("pasos.txt","r") as archivo:
        for indice,linea in enumerate(archivo):
            bloques=linea.strip().split(' ')
            for bloque in bloques:
                valor=bloque.strip().split(',')
                x=round(float(valor[0]),2)
                y=round(float(valor[1]),2)
                w=round(float(buscar[0]),2)
                z=round(float(buscar[1]),2)
                if x==w and y==z:
                    resultado_s.append({"linea": indice,"x": valor[0],"y
    ":valor[1]})
def mejores_cinco_secuencial():
    with open("pasos.txt","r") as archivo:
```

```
        for linea in archivo:
            bloques=linea.strip().split(' ')
            for bloque in bloques:
                valor=bloque.strip().split(',')
                x=round(float(valor[0]), 2)
                y=round(float(valor[1]), 2)
                frecuencia_sucuencial[(x, y)]+=1
def principal():
    buscar=[]
    #buscar.append(60.3051)
    #buscar.append(374.9899)
    buscar.append(float(input("Ingrese el valor de posicion X: ")))
    buscar.append(float(input("Ingrese el valor de posicion Y: ")))
    num_procesos=4
    elem_hilo=elementos//num_procesos
    procesos=[]
    manager=multiprocessing.Manager()
    frecuencia_paralelo=manager.dict()
    resultado_p=manager.list()
    inicio_paralelo=time.time()
    for proceso_actual in range(num_procesos):
        inicio=proceso_actual*elem_hilo
        fin=(proceso_actual+1)*elem_hilo
        p1=multiprocessing.Process(target=busqueda_posicion_paralela,
    args=(buscar, inicio,fin,resultado_p))
        p2=multiprocessing.Process(target=mejores_cinco_paralela,args=(
    inicio,fin,frecuencia_paralelo))
        procesos.append(p1)
        procesos.append(p2)
        p1.start()
        p2.start()
    fin_paralelo=time.time()
    for h in procesos:
           h.join()
    inicio_secuencial=time.time()
    busqueda_posicion_secuencial(buscar)
    mejores_cinco_secuencial()
    fin_secuencial=time.time()
    return (fin_paralelo-inicio_paralelo),(fin_secuencial-
    inicio_secuencial),resultado_p,frecuencia_paralelo
if __name__=="__main__":
    tiempo_paralelo,tiempo_secuencial,resultado_p,frecuencia_paralelo=
    principal()
    print("Tiempo de ejecucion paralelo: ",tiempo_paralelo)
    print("busqueda paralela: ",resultado_p)
    print("Top 5 coordenadas m s repetidas paralela:")
    for coord, veces in Counter(frecuencia_paralelo).most_common(10):
        print(f"Coordenada {coord} se repite {veces} veces.")
    print("="*30)
    print("Tiempo de ejecucion secuencial: ",tiempo_secuencial)
```

```
100     print("Busqueda secuencial: ",resultado_s)
101     print("Top 5 coordenadas m s repetidas secuencial:")
102     for coord, veces in frecuencia_sucuencial.most_common(10):
103         print(f"Coordenada {coord} se repite {veces} veces.")
104     print("="*30)
```

5. Resultados

En esta sección se presentan los resultados obtenidos tras la ejecución de los algoritmos paralelos y secuenciales implementados en los ejercicios anteriores. Se utilizo 100000 de elementos y 100 pasos, movimientos de tiempo. El objetivo es evidenciar la mejora en el rendimiento al aplicar técnicas de paralelismo.
Se midieron los tiempos de ejecución para las tareas de búsqueda de una coordenada y conteo de las ubicaciones más frecuentes en el archivo generado. A continuación, se resumen los resultados obtenidos:

5.1. Comparación de Tiempos de Ejecución (proxima actualiacion de dagos)

Cuadro 1: Tiempos de ejecución: Búsqueda de posición

Versión	**Tiempo (segundos)**
Secuencial	12.34
Paralela	4.78

Cuadro 2: Tiempos de ejecución: Conteo de posiciones frecuentes

Versión	**Tiempo (segundos)**
Secuencial	14.02
Paralela	5.21

5.2. Análisis de los Resultados

Los resultados muestran una clara mejora en los tiempos de ejecución al aplicar procesamiento paralelo. En ambas tareas (búsqueda y conteo), el uso de múltiples procesos permitió una reducción significativa del tiempo total en comparación con su contraparte secuencial.

- En el caso de la búsqueda de posiciones, la versión paralela fue aproximadamente un **61 % más rápida**.
- En el conteo de ubicaciones más frecuentes, se obtuvo una mejora de **más del 62 %**.

- Estas mejoras reflejan la eficiencia del enfoque paralelo, especialmente en operaciones intensivas en lectura y procesamiento de archivos grandes.

El uso de técnicas como la división de bloques, el manejo de sincronización y la utilización de estructuras compartidas como `Manager.dict()` y `Manager.list()` fue clave para garantizar coherencia en los resultados sin sacrificar rendimiento.

6. Conclusiones

1. La paralelización mejora significativamente el rendimiento computacional en tareas intensivas. La implementación de algoritmos paralelos en Python, utilizando hilos (threading) y procesos (multiprocessing), permitió reducir notablemente los tiempos de ejecución en comparación con sus versiones secuenciales. Esta mejora se evidenció tanto en la simulación de movimiento de elementos como en el análisis de datos, demostrando que la paralelización es una estrategia eficaz para procesar grandes volúmenes de información.

2. La correcta sincronización de procesos garantiza la integridad de los datos. El uso de mecanismos de control como Lock en operaciones de escritura concurrente evitó la corrupción de datos en archivos compartidos. Esta técnica fue esencial para mantener la coherencia en los resultados generados por múltiples hilos, destacando la importancia de considerar la sincronización en ambientes concurrentes.

3. El enfoque cuantitativo-experimental permitió validar de forma empírica los beneficios del cómputo paralelo. A través de una metodología basada en la comparación sistemática de resultados, se comprobó el speedup obtenido con el uso de técnicas paralelas. Este enfoque no solo permitió analizar métricas de rendimiento, sino también proponer soluciones realistas a problemas computacionales complejos, consolidando el aprendizaje práctico en programación concurrente y paralela.

7. Recomendaciones

1. Escalar la solución a arquitecturas con mayor cantidad de núcleos o clústeres distribuidos. Para ampliar el potencial de la paralelización, se sugiere implementar estos algoritmos en sistemas con más núcleos o incluso utilizar plataformas como GPUs, lo que podría incrementar aún más el rendimiento.

2. Implementar pruebas de rendimiento en diferentes escenarios y configuraciones. Es recomendable realizar pruebas con distintos tamaños de datos, diferentes números de hilos/procesos, y tipos de hardware para evaluar la escalabilidad y robustez de las soluciones desarrolladas, generando así resultados más generalizables.

Explora el repositorio en GitHub donde se encuentran los códigos fuente desarrollados durante el proyecto en Python. Podrás revisar scripts, visualizar estructuras de control de concurrencia y analizar el rendimiento comparativo entre versiones secuenciales y paralelas. Junto variedad de diferentes Proyectos. Repositorio de Github