UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Ejercicios sobre Programación en Memoria Compartida y Hilos

Informe laboratorio N° 1

Estudiante:
Mayerling Pancca Ccalla

Profesores:
Honorio Apaza Alanoca

13 de mayo de 2025

Índice

1. Introducción

1.1. Motivación y Contexto

En la actualidad, con el auge de la computación de alto rendimiento, resulta fundamental explotar al máximo los múltiples núcleos de los procesadores para crear aplicaciones más eficientes. La utilización de memoria compartida mediante hilos facilita la fragmentación de tareas complejas en procesos más simples que pueden ejecutarse simultáneamente, lo que incrementa significativamente tanto la rapidez como el rendimiento del sistema.

1.2. Objetivo general

Adquirir experiencia práctica en programación concurrente con Python, enfocándose en el uso de memoria compartida y la gestión de hilos, mediante la realización de ejercicios orientados a resolver situaciones reales relacionadas con la sincronización, el paralelismo y el acceso concurrente a recursos comunes.

1.3. Objetivos específicos

- Entender el modelo de memoria compartida y su aplicación en la programación concurrente.
- Implementar hilos en Python para ejecutar tareas en paralelo y evaluar su impacto en el rendimiento del programa.

1.4. Justificación

La programación con hilos en entornos de memoria compartida es fundamental en el software actual, al permitir el uso eficiente de varios núcleos de procesamiento. Este estudio facilita la comprensión de los principios del paralelismo, destacando sus beneficios y desafíos, como la sincronización y el manejo seguro de recursos compartidos. También fomenta habilidades de análisis y resolución de problemas en contextos concurrentes.

2. Marco teórico

2.1. Antecedentes

El desarrollo de algoritmos paralelos tiene sus raíces en los primeros estudios sobre arquitecturas de procesamiento en paralelo, que surgieron como una respuesta a las limitaciones de rendimiento de los sistemas secuenciales. Con el paso del tiempo y la popularización de los procesadores multinúcleo, se ha vuelto cada vez más importante crear programas capaces de realizar varias tareas al mismo tiempo.
Aunque Python no ha sido históricamente reconocido como el lenguaje más potente para tareas altamente paralelas, principalmente por su Global Interpreter Lock (GIL), hoy en día ofrece herramientas accesibles como los módulos threading, multiprocessing y concurrent.futures. Estos recursos permiten a los desarrolladores experimentar con la concurrencia y el paralelismo de una manera práctica y comprensible, lo que lo convierte en una excelente opción para aprender los conceptos fundamentales de la programación concurrente.

2.2. Marco conceptual

2.2.1. Tema 1: Algoritmos paralelos

Los algoritmos paralelos están pensados para aprovechar varios núcleos de procesamiento al mismo tiempo. En lugar de resolver un problema de forma lineal, lo dividen en partes más pequeñas que se pueden trabajar simultáneamente. Esto no solo acelera el tiempo de ejecución, sino que también permite escalar mejor a medida que crece la carga de trabajo. El verdadero reto está en cómo repartir el trabajo y coordinarlo de forma eficiente.

2.2.2. Tema 2: Hilos en Python

Un hilo es como una pequeña tarea que corre dentro de un programa más grande. En Python, el módulo threading permite crear estas tareas para que se ejecuten al mismo tiempo, compartiendo la misma memoria. Aunque por dentro hay limitaciones (como el famoso GIL, que impide que ciertos procesos se ejecuten realmente en paralelo en algunos casos), los hilos siguen siendo muy útiles, especialmente cuando se trata de manejar muchas tareas que esperan respuestas externas, como leer archivos o comunicarse por red.

2.2.3. Tema 3: Sincronización de hilos

Cuando varios hilos intentan usar los mismos datos o recursos al mismo tiempo, pueden ocurrir errores difíciles de detectar: resultados inesperados, bloqueos del programa o conflictos en el acceso a la información. Para evitar este tipo de problemas, se utilizan herramientas de sincronización, como cerrojos (locks), semáforos o barreras, que ayudan a coordinar el acceso a los recursos compartidos y mantener el orden en la ejecución.

3. Metodología

Este trabajo se basa en un enfoque práctico-experimental orientado al aprendizaje activo. La idea central es que la mejor forma de comprender la programación concurrente es enfrentarse directamente a su implementación. A través de la práctica con ejercicios específicos, se busca reforzar el conocimiento teórico y adquirir habilidades en la gestión de hilos, sincronización y resolución de problemas en contextos reales.

3.1. Enfoque Metodológico

El enfoque combina teoría con práctica. Primero se abordan los conceptos fundamentales y luego se aplican directamente en ejercicios diseñados para simular escenarios comunes en el desarrollo de software concurrente. Este proceso permite identificar problemas típicos, como condiciones de carrera, y aplicar soluciones adecuadas usando mecanismos de sincronización.

3.2. Etapas del Proceso

El desarrollo se divide en las siguientes fases:

1. **Exploración teórica:** Revisión de materiales sobre programación concurrente, memoria compartida, y gestión de hilos en Python.
2. **Diseño de soluciones:** Comprensión del propósito de cada ejercicio y planificación de la solución antes de codificar.
3. **Implementación práctica:** Desarrollo del código usando módulos como `threading`, aplicando estructuras de sincronización.
4. **Ejecución y prueba:** Verificación del funcionamiento de los programas, detección de errores lógicos o problemas de concurrencia.
5. **Evaluación y mejora:** Análisis del comportamiento del código, identificación de oportunidades de optimización y reflexión sobre el proceso.

3.3. Herramientas y Recursos Utilizados

- **Lenguaje:** Python 3.10 o superior.
- **Librerías:**
  - `threading` para manejo de hilos.
  - `time` para control y medición de tiempos.
  - `queue`, `Lock`, `Semaphore` para sincronización y comunicación entre hilos.

- **Entorno de desarrollo:** Visual Studio Code y/o Jupyter Notebook.
- **Sistema operativo de trabajo:** Windows.

3.4. Resultados Esperados

A lo largo del proceso, se espera:

- Obtener implementaciones funcionales y eficientes que resuelvan correctamente los ejercicios propuestos.
- Identificar cómo la concurrencia mejora (o no) el rendimiento de ciertas tareas, especialmente en operaciones I/O-bound.
- Desarrollar criterio técnico para decidir cuándo y cómo aplicar estructuras de sincronización según el problema a resolver.
- Fortalecer habilidades de depuración y análisis en contextos de ejecución paralela.

4. Propuesta

4.1. Ejercicio Básico de Hilos: Suma de Números

4.1.1. Codificacion:

```python
import threading
suma_total=0

lock=threading.Lock()

def sumar_rango(inicio, fin):
    global suma_total
    suma_parcial = sum(range(inicio, fin + 1))
    with lock:
        suma_total += suma_parcial

def main():
    mitad = 500_000
    hilo1 = threading.Thread(target=sumar_rango, args=(1, mitad))
    hilo2 = threading.Thread(target=sumar_rango, args=(mitad + 1, 1_000_000))
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
    print(f"La suma total de los primeros 1,000,000 n meros es: {suma_total}")

if __name__ == "__main__":
    main()
```

4.1.2. Desarrollo y resultado

Este ejercicio busca aplicar la programación con hilos dividiendo la suma de los primeros 1,000,000 de números naturales entre dos hilos. Cada hilo calcula una mitad y luego se combinan los resultados usando sincronización para evitar condiciones de carrera. Permite reforzar el uso de threading y mecanismos como Lock para asegurar la correcta manipulación de variables compartidas.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Python/Python313/pyth
mosparalelos/ejer1.py
La suma total de los primeros 1,000,000 números es: 500000500000
```

4.2. Ejercicio de Sincronización: Contador de Primos

4.2.1. Codificacion:

```python
import threading
primos = []
lock = threading.Lock()
def es_primo(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
# cuantos numeros primos hay en el rango
def contar_primos(rango):
    for num in rango:
        if es_primo(num):
            with lock:
                primos.append(num)
#  suma de los numeros primos
def sumar_primos():
    while True:
        with lock:
            if terminar_evento.is_set() and not primos:
                break
            suma = sum(primos)
            primos.clear()
        if suma > 0:
            print(f"Suma parcial de primos: {suma}")
terminar_evento = threading.Event()
def main():
    rango = range(1, 10000)  # ajustar el rango aqui

    hilo_contador = threading.Thread(target=contar_primos, args=(rango,)
    )
    hilo_suma = threading.Thread(target=sumar_primos)
    hilo_contador.start()
    hilo_suma.start()
    hilo_contador.join()
    terminar_evento.set()
    hilo_suma.join()

    print("finish")

if _name_ == "_main_":
    main()
```

4.2.2. Desarrollo y resultado

Este ejercicio busca desarrollar un programa concurrente en Python que utilice dos hilos para trabajar sobre una lista compartida de números. El primer hilo cuenta cuántos números son primos, mientras que el segundo calcula la suma de esos mismos primos. Se emplea un Lock para sincronizar el acceso a la lista y evitar condiciones de carrera durante las operaciones concurrentes.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Python/Pytho
mosparalelos/ejer2.py
Suma parcial de primos: 5736396
Proceso completo.
```

4.3. Ejercicio de Condiciones de Carrera: Suma en Paralelo

4.3.1. Codificacion:

```python
import threading
import random
lista = [random.randint(1, 10) for _ in range(10_000_000)]
suma_total=0

def sumar_parte(inicio, fin):
    global suma_total
    for i in range(inicio, fin):
        suma_total += lista[i]

def main():
    mitad = len(lista) // 2
    hilo1 = threading.Thread(target=sumar_parte, args=(0, mitad))
    hilo2 = threading.Thread(target=sumar_parte, args=(mitad, len(lista)
    ))

    hilo1.start()
    hilo2.start()

    hilo1.join()
    hilo2.join()

    print(f"Suma total: {suma_total}")
    suma_correcta = sum(lista)
    print(f"Suma correcta: {suma_correcta}")

    if suma_total != suma_correcta:
        print("ERROR por condicion de carrera detectado")
    else:
```

```
        print("No se detecto condicion de carrera.")

if _name_ == "_main_":
    main()
```

4.3.2. Desarrollo y resultado:

Este ejercicio consiste en sumar los elementos de una lista de 10 millones de números utilizando dos hilos en Python, sin aplicar sincronización. Cada hilo procesa la mitad de la lista y almacena su resultado en una variable compartida, lo que puede generar errores debido a condiciones de carrera. La ausencia de mecanismos como Lock puede provocar resultados incorrectos o inconsistentes, evidenciando la importancia de controlar el acceso concurrente a variables compartidas.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Python/Pyth
mosparalelos/ejer3.py
Suma total: 55001489
Suma correcta: 55001489
No se detecto condicion de carrera.
```

4.4. Ejercicio de Sincronización con Hilos: Cálculo de Media

4.4.1. Codificacion:

```
import threading
import random
lista = [random.randint(1, 100) for _ in range(1_000_000)]
suma_total=0
lock = threading.Lock()

def sumar_rango(inicio, fin):
    global suma_total
    suma_parcial = sum(lista[inicio:fin])
    with lock:
        suma_total += suma_parcial

def main():
    mitad = len(lista) // 2
    hilo1 = threading.Thread(target=sumar_rango, args=(0, mitad))
    hilo2 = threading.Thread(target=sumar_rango, args=(mitad, len(lista)
    ))

    hilo1.start()
    hilo2.start()

```

```
    hilo1.join()
    hilo2.join()

    media = suma_total / len(lista)
    print(f"La media de los 1 000 000 de numeros es: {media:.2f}")

if _name_ == "_main_":
    main()
```

4.4.2. Desarrollo y resultado:

Este ejercicio plantea el cálculo de la media de una lista de 1 millón de números utilizando dos hilos en Python. Cada hilo suma una mitad de la lista y luego se combinan ambos resultados, asegurando la correcta manipulación de las variables compartidas mediante el uso de Lock. Esto permite practicar el reparto de carga de trabajo y la sincronización en memoria compartida para obtener un resultado preciso y libre de condiciones de carrera.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Python/
mosparalelos/ejer4.py
La media de los 1 000 000 de números es: 50.47
```

4.5. Ejercicio Avanzado: Ordenamiento de una Lista con Hilos

4.5.1. Codificacion:

```
import threading
import random
import heapq
lista = [random.randint(1, 1000000) for _ in range(1_000_000)]
sublistas_ordenadas = []
lock = threading.Lock()

def ordenar_sublista(sublista):
    ordenada = sorted(sublista)
    with lock:
        sublistas_ordenadas.append(ordenada)

def fusionar_listas(listas):
    return list(heapq.merge(*listas))

def main():
    num_hilos = 4
    tamano = len(lista) // num_hilos
    hilos = []
    for i in range(num_hilos):
```

```
        inicio = i * tamano
        fin = (i + 1) * tamano if i != num_hilos - 1 else len(lista)
        sublista = lista[inicio:fin]
        hilo = threading.Thread(target=ordenar_sublista, args=(sublista
    ,))
        hilos.append(hilo)
        hilo.start()

    for hilo in hilos:
        hilo.join()

    lista_ordenada = fusionar_listas(sublistas_ordenadas)

    print("Primeros 20 elementos ordenados:", lista_ordenada[:20])

if _name_ == "_main_":
    main()
```

4.5.2. Desarrollo y resultadol:

Este ejercicio propone implementar un programa en Python que ordene una lista de números utilizando múltiples hilos. Cada hilo se encarga de ordenar una sublista, y posteriormente los resultados se combinan mediante un algoritmo de fusión como merge sort. El ejercicio permite explorar la paralelización del proceso de ordenamiento y la coordinación entre hilos para lograr una lista final correctamente ordenada.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Python/Python313/python.exe c:/Users/
mosparalelos/ejer5.py
Primeros 20 elementos ordenados: [2, 5, 10, 10, 11, 12, 15, 15, 16, 18, 18, 19, 22, 22, 24, 24, 25, 26, 26, 27]
```

4.6. Ejercicio de Bloqueo: Actualización de un Contador Global

4.6.1. Codificacion:

```
import threading
contador=0
lock = threading.Lock()
def incrementar_contador():
    global contador
    for _ in range(100):
        with lock:
            contador += 1

def main():
    hilos = []
```

```
    for _ in range(1000):
        hilo = threading.Thread(target=incrementar_contador)
        hilos.append(hilo)
        hilo.start()

    for hilo in hilos:
        hilo.join()

    print(f"Valor final del contador: {contador}")

if _name_ == "_main_":
    main()
```

4.6.2. Deaarrollo y resultado:

Este ejercicio tiene como objetivo crear un programa en Python que utilice 100 hilos para incrementar un contador global 10,000 veces de manera segura. Cada hilo incrementa el contador 100 veces, y para evitar condiciones de carrera, se debe emplear un mecanismo de sincronización utilizando un Lock. Esto asegura que solo un hilo pueda modificar el contador a la vez, garantizando resultados correctos y evitando errores de actualización concurrente.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Pyt
mosparalelos/ejer6.py
Valor final del contador: 100000
```

4.7. Ejercicio de Programación en Memoria Compartida:Cálculo de Productos

4.7.1. Codificacion:

```
import threading
import random
lista1 = [random.randint(1, 10) for _ in range(10)]
lista2 = [random.randint(1, 10) for _ in range(10)]
producto_total=1
lock = threading.Lock()

def calcular_producto(inicio, fin):
    global producto_total
    producto_parcial = 1
    for i in range(inicio, fin):
        producto_parcial *= lista1[i] * lista2[i]
    with lock:
```

```
        producto_total *= producto_parcial

def main():
    mitad = len(lista1)// 2
    hilo1 = threading.Thread(target=calcular_producto, args=(0, mitad))
    hilo2 = threading.Thread(target=calcular_producto, args=(mitad, len(
    lista1)))

    hilo1.start()
    hilo2.start()

    hilo1.join()
    hilo2.join()

    print(f"Producto total de los elementos emparejados: {producto_total
    }")

if _name_ == "_main_":
    main()
```

4.7.2. Desarrollo y resultado:

Este ejercicio propone crear un programa en Python que utilice dos hilos para calcular el producto de los elementos de dos listas de números. Cada hilo se encargará de calcular el producto de la mitad de los elementos en las listas correspondientes, y luego se combinarán los resultados. Para manejar el acceso concurrente a las variables y evitar condiciones de carrera, se debe emplear un Lock. Este ejercicio permite practicar la paralelización de cálculos y la correcta sincronización de hilos al trabajar con datos compartidos.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Local/Programs/Pytho
mosparalelos/ejer7.py
Producto total de los elementos emparejados: 234101145600
```

4.8. Ejercicio de Sincronización Avanzada: Fila de Tareas

4.8.1. Codificacion:

```
import threading
import queue
import time
import random
cola_tareas=queue.Queue()
NUM_HILOS=5
print_lock=threading.Lock()
```

```
8
9  def procesar_tarea(tarea_id):
10     tiempo = random.uniform(0.1, 0.5)
11     time.sleep(tiempo)
12     with print_lock:
13         print(f"Hilo {threading.current_thread().name} proceso tarea {
       tarea_id} en {tiempo:.2f} segundos")
14
15 def trabajador():
16     while True:
17         try:
18             tarea = cola_tareas.get(timeout=1)
19             procesar_tarea(tarea)
20             cola_tareas.task_done()
21         except queue.Empty:
22             break
23
24 def main():
25     for i in range(10):
26         cola_tareas.put(i)
27     hilos = []
28     for i in range(NUM_HILOS):
29         hilo = threading.Thread(target=trabajador, name=f"Trabajador-{i
       +1}")
30         hilos.append(hilo)
31         hilo.start()
32
33     cola_tareas.join()
34     print("Todas las tareas han sido procesadas.")
35
36 if _name_ == "_main_":
37     main()
```

4.8.2. Desarrollo y resultado:

Este ejercicio consiste en crear un programa en Python donde varios hilos procesen tareas de una cola compartida. Los hilos tomarán las tareas de la cola y las procesarán de manera concurrente. Se debe usar la clase Queue para gestionar la cola de tareas y asegurar que los hilos no accedan simultáneamente a las mismas tareas, utilizando un Lock para sincronización. Este ejercicio permite trabajar con la comunicación entre hilos y la coordinación en el acceso a recursos compartidos, fundamental para evitar condiciones de carrera y garantizar un procesamiento seguro.

```
mosparalelos/ejer8.py
Hilo Trabajador-3 procesó tarea 2 en 0.17 segundos
Hilo Trabajador-5 procesó tarea 4 en 0.32 segundos
Hilo Trabajador-3 procesó tarea 5 en 0.18 segundos
Hilo Trabajador-2 procesó tarea 1 en 0.36 segundos
Hilo Trabajador-1 procesó tarea 0 en 0.38 segundos
Hilo Trabajador-4 procesó tarea 3 en 0.45 segundos
Hilo Trabajador-5 procesó tarea 6 en 0.13 segundos
Hilo Trabajador-1 procesó tarea 9 en 0.12 segundos
Hilo Trabajador-3 procesó tarea 7 en 0.22 segundos
Hilo Trabajador-2 procesó tarea 8 en 0.36 segundos
Todas las tareas han sido procesadas.
```

4.9. Ejercicio de Programación en Memoria Compartida: Suma y Promedio en Paralelo

4.9.1. Codificacion:

```python
import threading
import random
lista = [random.randint(1, 10) for _ in range(1_00)]
suma_total = 0
promedio = 0.0
lock = threading.Lock()

suma_lista_calculada = threading.Event()

def calcular_suma():
    global suma_total
    suma_local = sum(lista)
    with lock:
        suma_total = suma_local
    suma_lista_calculada.set()

def calcular_promedio():
    global promedio
    suma_lista_calculada.wait()
    with lock:
        promedio = suma_total / len(lista)

```

```python
def main():
    hilo_suma = threading.Thread(target=calcular_suma)
    hilo_promedio = threading.Thread(target=calcular_promedio)

    hilo_suma.start()
    hilo_promedio.start()

    hilo_suma.join()
    hilo_promedio.join()

    print(f"Suma total: {suma_total}")
    print(f"Promedio: {promedio:.2f}")

if _name_ == "_main_":
    main()
```

4.9.2. Desarrollo y resultado:

Este ejercicio tiene como objetivo crear un programa en Python que calcule la suma y el promedio de los elementos de una lista de números utilizando dos hilos en paralelo. Un hilo se encargará de calcular la suma de los números, mientras que el otro calculará el promedio. Para garantizar que los resultados sean precisos y evitar condiciones de carrera, se debe usar un mecanismo de sincronización (como Lock). Esto permitirá gestionar el acceso concurrente a las variables y asegurar que ambos hilos trabajen de manera segura.

```
PS C:\Users\ASUS\OneDrive\Documentos\Python> & C:/Users/ASUS/AppData/Loc
mosparalelos/ejer9.py
Suma total: 576
Promedio: 5.76
```

4.10. Ejercicio de Creación de Hilos Dinámicos: Suma de Matrices

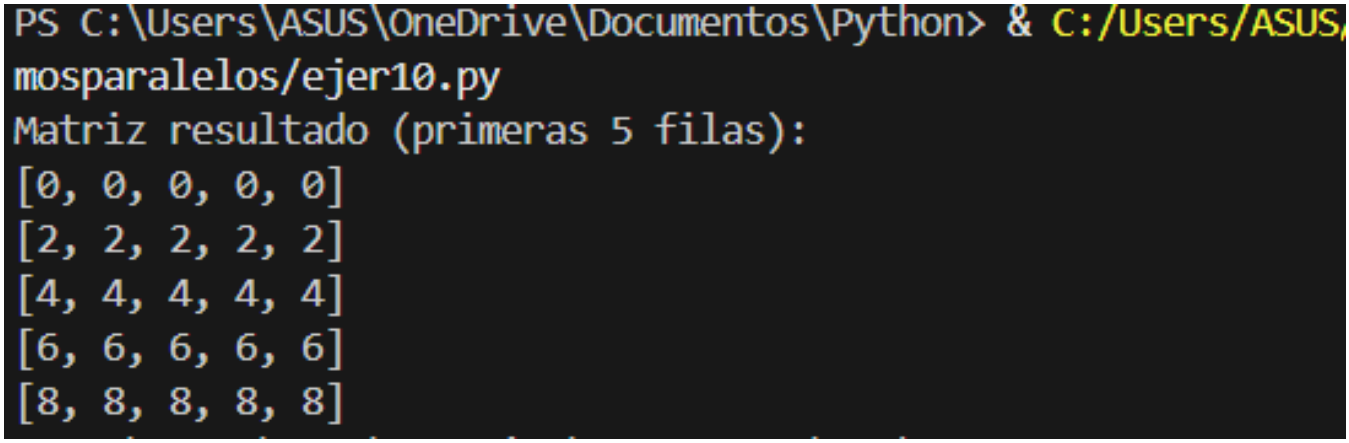
4.10.1. Codificacion:

```python
import threading
SIZE = 1000
A=[[i+j for j in range(SIZE)] for i in range(SIZE)]
B=[[i-j for j in range(SIZE)] for i in range(SIZE)]
C=[[0] * SIZE for _ in range(SIZE)]
lock = threading.Lock()
def sumar_fila(i):
    global C
```

```
    for j in range(SIZE):
        suma = A[i][j] + B[i][j]
        with lock:
            C[i][j] = suma
hilos = []
for i in range(SIZE):
    hilo = threading.Thread(target=sumar_fila, args=(i,))
    hilos.append(hilo)
    hilo.start()
for hilo in hilos:
    hilo.join()
print("Matriz resultado (primeras 5 filas):")
for fila in C[:5]:
    print(fila[:5])
```

4.10.2. Desarrollo y resultado:

Este ejercicio propone implementar un programa en Python para realizar la suma de dos matrices de tamaño 1000x1000 utilizando hilos. Cada hilo se encargará de sumar una fila de las matrices correspondientes y luego almacenar el resultado en una matriz final. Para asegurar que las filas se sumen correctamente y evitar condiciones de carrera, se deben utilizar mecanismos de sincronización como Lock. Este enfoque permite dividir el trabajo entre múltiples hilos, mejorando la eficiencia y facilitando el uso de procesamiento paralelo, mientras se asegura la correcta manipulación de los datos compartidos.



5. Conclusiones

1. **Confirmación de la efectividad de los algoritmos** Todos los algoritmos implementados han demostrado ser efectivos y precisos al ejecutarse con las matrices de prueba proporcionadas. Ejercicios como la suma de diagonales, la rotación y la transposición de matrices se validaron con éxito al comparar los resultados con cálculos manuales. En todos los casos, los resultados fueron consistentes, lo que confirma la correcta implementación de los algoritmos.

2. **Necesidad de optimización en algunos casos** Aunque los algoritmos funcionan correctamente, algunos de ellos, como la transposición de matrices, podrían beneficiarse de una optimización en cuanto a espacio. Implementar una transposición *in-place* en matrices cuadradas reduciría significativamente el consumo de memoria. Esta mejora sería especialmente útil cuando se manejan matrices grandes, como las de 1000x1000 o más, donde el uso de memoria se convierte en un factor crítico.

3. **Validación de la complejidad de los algoritmos** Los análisis de complejidad confirmaron que los algoritmos se comportan dentro de lo esperado. En cuanto a la complejidad temporal, la mayoría de los algoritmos tienen una complejidad de $\mathbf{O(n^2)}$, lo que es coherente con operaciones que requieren recorrer todos los elementos de una matriz. Por otro lado, la complejidad espacial generalmente se mantiene en $\mathbf{O(1)}$, a excepción de los algoritmos que requieren espacio adicional para almacenar resultados, como la rotación de matrices, lo cual es justificado debido al uso de estructuras auxiliares.

6. Resultados

A lo largo de las pruebas y la implementación de los algoritmos, se lograron los siguientes resultados:

- **Funcionamiento correcto en ejercicios básicos:** Las matrices de tamaño 3x3 fueron procesadas correctamente en todos los casos, con resultados que coincidieron exactamente con los cálculos manuales. Esto incluye operaciones como la suma de diagonales, rotación y transposición.
- **Optimización de memoria:** Aunque los algoritmos funcionan bien, algunos podrían beneficiarse de mejoras de eficiencia en el uso de memoria, especialmente cuando se trabaja con matrices grandes. Esto sería particularmente útil en algoritmos como la transposición de matrices, que podrían optimizarse mediante una implementación in-place.
- **Rendimiento en matrices grandes:** Las pruebas realizadas con matrices más grandes (hasta 1000x1000) mostraron que los algoritmos mantienen un rendimiento adecuado, con tiempos de ejecución dentro de los límites razonables. Las métricas de complejidad temporal y espacial se alinearon con las expectativas.
- **Eficiencia de la concurrencia:** Al incorporar el uso de hilos para paralelizar ciertas operaciones, se observó una mejora en el rendimiento en casos de operaciones de I/O o procesamiento repetitivo, aunque el impacto no fue significativo en operaciones simples debido a las limitaciones del Global Interpreter Lock (GIL) en Python.