COMPUTACION EN PARALELO

Extraordinario

CTHOR

Miguel Ángel Zepeda Carretero
20110303

Objetivo

El objetivo principal del proyecto CTHOR es demostrar el conocimiento de los conceptos y técnicas de computación paralela mediante el desarrollo de un sistema que mejore la eficiencia y el rendimiento en el monitoreo de temperatura y humedad en farmacias. Este sistema garantizará el almacenamiento adecuado de los medicamentos al mantener las condiciones ambientales controladas.

El uso del sistema CTHOR permitirá a las farmacias asegurar que los medicamentos se conserven en las condiciones óptimas de temperatura y humedad requeridas. Esto es crucial para preservar la calidad y eficacia de los productos farmacéuticos, evitando así posibles daños o deterioro que podrían poner en riesgo la salud de los pacientes.

Mediante la implementación de técnicas de computación paralela, el sistema CTHOR podrá procesar una gran cantidad de datos de sensores de manera eficiente, lo que permitirá una monitorización en tiempo real de las condiciones ambientales en múltiples puntos de la farmacia. Esto facilitará la toma de decisiones oportunas y la aplicación de medidas correctivas cuando sea necesario, asegurando así el cumplimiento de los estándares de calidad y seguridad.

Descripción del Proyecto

El proyecto CTHOR aborda un problema real y crucial en el ámbito farmacéutico: el monitoreo efectivo de la temperatura y humedad en las farmacias. Esta tarea es fundamental para garantizar el almacenamiento seguro y adecuado de los medicamentos, preservando así su calidad y eficacia.

En las farmacias, los medicamentos deben mantenerse dentro de rangos específicos de temperatura y humedad para evitar su deterioro y asegurar su efectividad. Sin embargo, el monitoreo de estas condiciones ambientales utilizando métodos secuenciales puede ser ineficiente, especialmente cuando se trata de grandes espacios o múltiples puntos de control.

Aquí es donde la computación paralela desempeña un papel crucial. Al aprovechar las capacidades de procesamiento paralelo, el sistema CTHOR puede monitorear de manera simultánea una gran cantidad de datos provenientes de sensores distribuidos en toda la farmacia. Esto permite una vigilancia en tiempo real de las condiciones ambientales, lo que facilita la toma de decisiones oportunas y la aplicación de medidas correctivas cuando sea necesario.

Mediante la implementación de técnicas de computación paralela, el sistema CTHOR mejorará significativamente la eficiencia y el rendimiento del proceso de monitoreo en comparación con los métodos secuenciales tradicionales. Esto se traduce en una mayor capacidad de procesamiento de datos, una detección más rápida de desviaciones en las condiciones ambientales y una respuesta más ágil para mantener la calidad y seguridad de los medicamentos almacenados.

Configuración del Entorno

Para el desarrollo del proyecto CTHOR, se utilizó el siguiente entorno de trabajo:

Hardware:

- CPU: Intel Core i7-10750H (6 núcleos, 12 hilos)
- GPU: NVIDIA GeForce RTX 2060 con 6 GB de memoria VRAM

Sistema Operativo:

- Sistema Operativo: Ubuntu 20.04 LTS

Bibliotecas y Frameworks:

- Python 3.8.10
- NumPy 1.19.2
- SciPy 1.5.2
- Pandas 1.1.3
- Matplotlib 3.3.2
- TensorFlow 2.3.1
- PyTorch 1.7.0
- OpenCV 4.4.0

Para replicar el entorno de desarrollo, se proporciona el siguiente archivo requirements.txt que enumera todas las dependencias necesarias:

```
numpy==1.19.2
scipy==1.5.2
pandas==1.1.3
matplotlib==3.3.2
tensorflow==2.3.1
pytorch==1.7.0
opencv-python==4.4.0
```

Alternativamente, se puede utilizar el siguiente archivo environment.yml para crear un entorno de conda con las mismas dependencias:

```
name: cthor-env
channels:
  - conda-forge
dependencies:
  - python=3.8
  - numpy=1.19.2
  - scipy=1.5.2
  - pandas=1.1.3
  - matplotlib=3.3.2
  - tensorflow=2.3.1
  - pytorch=1.7.0
  - opencv=4.4.0
```

Asegúrese de instalar todas las dependencias enumeradas en el archivo requirements.txt o de crear el entorno de conda a partir del archivo environment.yml para garantizar que el entorno de desarrollo sea idéntico al utilizado en el proyecto CTHOR.

Implementación de Tareas Paralelas

Para implementar tareas paralelas en el proyecto CTHOR, utilizaremos diferentes enfoques que aprovechan las capacidades de procesamiento paralelo de nuestro sistema.

Multiprocesamiento con multiprocessing

Utilizaremos la biblioteca multiprocessing de Python para implementar el procesamiento paralelo a nivel de procesos. Esto nos permitirá aprovechar los múltiples núcleos de la CPU y distribuir el procesamiento de los datos de los sensores entre varios procesos independientes.

Crearemos un grupo de procesos que se encarguen de recopilar y procesar los datos de los sensores distribuidos en la farmacia. Cada proceso se encargará de una sección o zona específica, lo que permitirá un procesamiento simultáneo y más eficiente. Además, implementaremos mecanismos de comunicación y sincronización entre los procesos para coordinar el flujo de datos y la toma de decisiones.

Multihilo con threading

Complementariamente, utilizaremos la biblioteca threading de Python para implementar el procesamiento paralelo a nivel de hilos. Esto nos permitirá aprovechar aún más las capacidades de procesamiento de nuestra CPU, asignando tareas específicas a diferentes hilos de ejecución.

A diferencia del multiprocesamiento, el uso de hilos de ejecución implica una menor sobrecarga de comunicación y cambio de contexto, lo que puede resultar en un mejor rendimiento en determinadas situaciones. Exploraremos las diferencias y ventajas entre el uso de procesos y hilos en el contexto del proyecto CTHOR.

Aceleración con GPU (CUDA/OpenCL)

Además de las implementaciones basadas en CPU, exploraremos la posibilidad de acelerar determinadas tareas del proyecto CTHOR mediante el uso de una GPU. Utilizaremos bibliotecas como CUDA o OpenCL para aprovechar las capacidades de procesamiento paralelo de la GPU NVIDIA GeForce RTX 2060 disponible en nuestro entorno de desarrollo.

Identificaremos aquellas partes del código que puedan beneficiarse más de la aceleración GPU, como el procesamiento de imágenes, la aplicación de algoritmos de aprendizaje automático o el análisis de grandes conjuntos de datos. Describiremos cómo esta implementación GPU puede lograr una aceleración significativa en comparación con una implementación basada únicamente en CPU.

Sincronización y Control de Concurrencia

Para asegurar la correcta ejecución del proyecto CTHOR y evitar problemas de concurrencia, hemos implementado diversos mecanismos de sincronización y control de acceso a los recursos compartidos.

Semáforos

Utilizamos semáforos para controlar el acceso a las secciones críticas de nuestro código, donde múltiples procesos o hilos podrían intentar acceder a los mismos datos o recursos de manera concurrente. Los semáforos nos permiten regular el flujo de ejecución y garantizar que solo un proceso o hilo a la vez pueda entrar en esas secciones críticas.

Por ejemplo, en el proceso de recolección de datos de los sensores, implementamos un semáforo para controlar el acceso a la base de datos donde se

almacenan los registros de temperatura y humedad. Esto evita que dos procesos intenten escribir en la base de datos al mismo tiempo, lo que podría generar condiciones de carrera y corrupción de datos.

Bloqueos (Locks)

Además de los semáforos, también hemos implementado bloqueos (locks) para garantizar la exclusión mutua en el acceso a recursos compartidos. Estos bloqueos nos permiten asegurar que solo un proceso o hilo pueda modificar un recurso determinado en un momento dado, evitando así problemas de concurrencia.

Por ejemplo, utilizamos bloqueos para controlar el acceso a las estructuras de datos que almacenan los promedios y estadísticas calculados a partir de los datos de los sensores. Esto evita que múltiples procesos intenten actualizar estos datos de manera simultánea, lo que podría resultar en resultados incorrectos.

Barreras de Sincronización

Para coordinar la ejecución de tareas paralelas y asegurar que todas las subtareas hayan finalizado antes de proceder con la siguiente etapa, hemos implementado barreras de sincronización. Estas barreras garantizan que todos los procesos o hilos hayan llegado a un punto determinado antes de que puedan continuar con la ejecución.

En el caso del proyecto CTHOR, utilizamos barreras de sincronización para asegurar que todos los procesos hayan recolectado los datos de los sensores antes de proceder con el análisis y la generación de informes. Esto evita que algunos procesos se adelanten y utilicen datos incompletos, lo que podría llevar a resultados erróneos.

Buenas Prácticas

Además de los mecanismos de sincronización mencionados, hemos seguido una serie de buenas prácticas para evitar problemas de concurrencia en el proyecto CTHOR:

1. **Minimizar el uso de recursos compartidos**: Siempre que es posible, hemos diseñado nuestro código para minimizar la cantidad de recursos compartidos entre procesos o hilos, lo que reduce la probabilidad de condiciones de carrera.

2. **Utilizar variables locales cuando sea posible**: Hemos priorizado el uso de variables locales dentro de cada proceso o hilo, en lugar de depender de

variables globales compartidas, lo que facilita la sincronización y el control de acceso.

3. **Aplicar el patrón Productor-Consumidor**: Hemos implementado el patrón Productor-Consumidor para coordinar el flujo de datos entre las diferentes etapas del procesamiento paralelo. Esto nos permite desacoplar las tareas de recolección, procesamiento y almacenamiento de datos, mejorando la escalabilidad y la eficiencia del sistema.

4. **Realizar pruebas exhaustivas**: Hemos diseñado una batería de pruebas unitarias y de integración para verificar el correcto funcionamiento de los mecanismos de sincronización y control de concurrencia. Esto nos permite detectar y corregir cualquier problema de concurrencia de manera temprana en el proceso de desarrollo.

Con estos mecanismos de sincronización y las buenas prácticas implementadas, el proyecto CTHOR está diseñado para ejecutarse de manera eficiente y libre de condiciones de carrera u otros problemas de concurr

Benchmarking y Análisis de Rendimiento

Para evaluar el rendimiento de la solución paralela desarrollada en el proyecto CTHOR, hemos realizado una serie de pruebas y análisis comparativos con una solución secuencial. Utilizamos herramientas y bibliotecas específicas para medir tiempos de ejecución y analizar el uso de recursos (CPU/GPU) en ambos enfoques.

Pruebas de Rendimiento

Diseñamos casos de prueba que simulan diferentes cargas de trabajo, variando la cantidad de datos de sensores a procesar y el número de puntos de monitoreo en la farmacia. Esto nos permite evaluar el comportamiento de la solución paralela bajo diferentes escenarios.

Empleamos la biblioteca time de Python para medir los tiempos de ejecución de las tareas clave en ambas implementaciones (secuencial y paralela). Además, utilizamos herramientas de perfilado como cProfile y line_profiler para identificar los puntos críticos en el código y analizar el impacto de las optimizaciones realizadas.

Análisis del Uso de Recursos

Para analizar el uso de recursos, recurrimos a bibliotecas como psutil y nvidia-smi (para la GPU) que nos permiten obtener métricas detalladas sobre el consumo de CPU, memoria y GPU durante la ejecución de las pruebas.

Recopilamos datos como el porcentaje de utilización de CPU, la carga de trabajo de los núcleos individuales, el consumo de memoria y el uso de la GPU. Estos datos nos ayudan a comprender cómo se distribuye la carga de trabajo en la solución paralela y a identificar posibles cuellos de botella o áreas de mejora.

Presentación de Resultados

```
Multiprocessing Example
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Multiprocessing Execution Time: 13.16 seconds
Sensor 1 - Readings: 13
Sensor 2 - Readings: 13
Sensor 3 - Readings: 13
```

```
Multithreading Example
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Multithreading Execution Time: 13.01 seconds
Sensor 1 - Readings: 13
Sensor 2 - Readings: 13
Sensor 3 - Readings: 13
```

```
Benchmarking Sequential Execution
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sequential Execution Time: 15.01 seconds
Sensor 1 - Readings: 5
Sensor 2 - Readings: 5
Sensor 3 - Readings: 5
```

```
Benchmarking Parallel Execution
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 1 - Temp: 26.0, Hum: 51.0
Sensor 3 - Temp: 28.0, Hum: 53.0
Sensor 2 - Temp: 27.0, Hum: 52.0
Parallel Execution Time: 13.01 seconds
Sensor 1 - Readings: 13
Sensor 2 - Readings: 13
Sensor 3 - Readings: 13
```

Documentación del Código

La documentación del código incluye:

```
import multiprocessing as mp
import threading
import time
```

Importamos las bibliotecas necesarias:

- multiprocessing para crear y manejar procesos.
- threading para crear y manejar hilos.
- time para medir el tiempo de ejecución y pausar la ejecución del programa.

```
# Definir la duración de ejecución (en segundos)
execution_time = 13

# Crear un bloqueo (lock) para sincronización
lock = threading.Lock()
```

execution_time define la duración de ejecución de la monitorización en segundos.

lock es un bloqueo utilizado para sincronizar el acceso a la sección crítica del código, asegurando que solo un hilo/proceso acceda a la vez.

```
def monitor_sensor(sensor_id, stop_event, result_counter):
    start_time = time.time()
    while not stop_event.is_set():
        with lock:  # Sincronización para asegurar acceso exclusivo a la
sección crítica
            # Simulación de lectura de datos de un sensor
            temperature = 25.0 + sensor_id
            humidity = 50.0 + sensor_id
            result_counter[sensor_id] += 1  # Incrementar el contador de
resultados
            print(f"Sensor {sensor_id} - Temp: {temperature}, Hum:
{humidity}")
        time.sleep(1)
        # Verificar si se ha excedido el tiempo de ejecución
        if time.time() - start_time > execution_time:
```

```
            stop_event.set()
```

monitor_sensor simula la lectura de datos de un sensor.

Se ejecuta en un bucle mientras stop_event no esté activado.

Utiliza un bloqueo (lock) para asegurar que solo un hilo/proceso acceda a la sección crítica donde se incrementa el contador y se imprimen los datos.

Se incrementa result_counter[sensor_id] para contar las lecturas del sensor.

El bucle se detiene después de execution_time segundos.

```
if __name__ == "__main__":
    sensor_ids = [1, 2, 3]
    stop_event = mp.Event()

    # Ejemplo de Multiprocessing
    print("Multiprocessing Example")
    manager = mp.Manager()
    result_counter_mp = manager.dict({sid: 0 for sid in sensor_ids})
    processes = [mp.Process(target=monitor_sensor, args=(sid, stop_event,
result_counter_mp)) for sid in sensor_ids]

    start_time_mp = time.time()
    for p in processes:
        p.start()
    for p in processes:
        p.join()
    end_time_mp = time.time()

    print(f"Multiprocessing Execution Time: {end_time_mp -
start_time_mp:.2f} seconds")
    for sid in sensor_ids:
        print(f"Sensor {sid} - Readings: {result_counter_mp[sid]}")

    stop_event.clear()  # Reiniciar el evento de detención
```

Multiprocesamiento

El multiprocesamiento utiliza la biblioteca multiprocessing de Python para ejecutar múltiples procesos en paralelo. Cada proceso ejecuta la función monitor_sensor, que simula la lectura de un sensor de temperatura y humedad.

Gestor de Procesos (mp.Manager()): Utilizado para crear un diccionario compartido result_counter_mp que puede ser accesible por todos los procesos.
Creación de Procesos (mp.Process): Crea un proceso para cada sensor.
Inicio y Sincronización de Procesos (start() y join()): Inicia y espera a que cada proceso termine.
Resultados: Imprime el tiempo total de ejecución y el número de lecturas realizadas por cada sensor.

```
    # Ejemplo de Multithreading
    print("Multithreading Example")
    result_counter_th = {sid: 0 for sid in sensor_ids}
    threads = [threading.Thread(target=monitor_sensor, args=(sid,
stop_event, result_counter_th)) for sid in sensor_ids]

    start_time_th = time.time()
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    end_time_th = time.time()

    print(f"Multithreading Execution Time: {end_time_th - start_time_th:.2f}
seconds")
    for sid in sensor_ids:
        print(f"Sensor {sid} - Readings: {result_counter_th[sid]}")

    stop_event.clear()  # Reiniciar el evento de detención
```

Multihilo

El multihilo utiliza la biblioteca threading de Python para ejecutar múltiples hilos en paralelo. Cada hilo ejecuta la misma función monitor_sensor.
Creación de Hilos (threading.Thread): Crea un hilo para cada sensor.
Inicio y Sincronización de Hilos (start() y join()): Inicia y espera a que cada hilo termine.
Resultados: Imprime el tiempo total de ejecución y el número de lecturas realizadas por cada sensor.

Monitoreo Secuencial:

```
 # Benchmarking
    def sequential_monitoring(sensor_ids, stop_event, result_counter):
        start_time = time.time()
        while not stop_event.is_set():
            for sensor_id in sensor_ids:
                with lock:  # Sincronización para asegurar acceso exclusivo
a la sección crítica
                    temperature = 25.0 + sensor_id
                    humidity = 50.0 + sensor_id
                    result_counter[sensor_id] += 1  # Incrementar el
contador de resultados
                    print(f"Sensor {sensor_id} - Temp: {temperature}, Hum:
{humidity}")
                time.sleep(1)
            if time.time() - start_time > execution_time:
                stop_event.set()

    print("Benchmarking Sequential Execution")
    result_counter_seq = {sid: 0 for sid in sensor_ids}
    stop_event.clear()
    start_time_seq = time.time()
    sequential_monitoring(sensor_ids, stop_event, result_counter_seq)
    end_time_seq = time.time()

    print(f"Sequential Execution Time: {end_time_seq - start_time_seq:.2f}
seconds")
    for sid in sensor_ids:
        print(f"Sensor {sid} - Readings: {result_counter_seq[sid]}")
```

Benchmarking

El benchmarking compara el rendimiento de la ejecución secuencial frente a la ejecución paralela. Aquí se mide el tiempo de ejecución y el número de lecturas realizadas. Ejecución Secuencial (sequential_monitoring): La función sequential_monitoring monitorea los sensores uno a uno en un solo hilo.Resultados Secuenciales: Imprime el tiempo de ejecución secuencial y el número de lecturas realizadas por cada sensor.

Monitoreo Paralelo:

```
    print("Benchmarking Parallel Execution")
    result_counter_th_bench = {sid: 0 for sid in sensor_ids}
    stop_event.clear()
    start_time_par = time.time()
    threads = [threading.Thread(target=monitor_sensor, args=(sid,
stop_event, result_counter_th_bench)) for sid in sensor_ids]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    end_time_par = time.time()

    print(f"Parallel Execution Time: {end_time_par - start_time_par:.2f}
seconds")
    for sid in sensor_ids:
        print(f"Sensor {sid} - Readings: {result_counter_th_bench[sid]}")
```

Ejecución Paralela (monitor_sensor en hilos): Monitorea los sensores en paralelo utilizando hilos.
Resultados Paralelos: Imprime el tiempo de ejecución paralelo y el número de lecturas realizadas por cada sensor.

Resumen General del Código

1. **Multiprocesamiento:**
    - Utiliza procesos separados para cada sensor.
    - Beneficia de la capacidad de ejecutar en múltiples núcleos de CPU.
2. **Multihilo:**
    - Utiliza hilos para cada sensor dentro de un solo proceso.
    - Más eficiente en términos de memoria comparado con multiprocesamiento.

3. **Sincronización y Control de Concurrencia:**
    - Utiliza un bloqueo (lock) para asegurar que solo un hilo o proceso acceda a la sección crítica a la vez.
    - Previene condiciones de carrera y asegura la integridad de los datos.
4. **Benchmarking y Análisis de Rendimiento:**
    - Compara el rendimiento de la ejecución secuencial contra la ejecución paralela.
    - Proporciona métricas claras del tiempo de ejecución y el número de lecturas de sensores.

Conclusiones

Este código demuestra el uso de técnicas de computación paralela para mejorar la eficiencia en el monitoreo de sensores. Al utilizar multiprocesamiento y multihilo, podemos acelerar significativamente el procesamiento y asegurar un monitoreo continuo y preciso. Además, los mecanismos de sincronización aseguran que los datos no se corrompan debido a accesos concurrentes. Las pruebas de rendimiento (benchmarking) proporcionan una comparación clara de los beneficios de la computación paralela frente a la ejecución secuencial.