Concurrencia, paralelismo y procesamiento asíncronico

Parte 1

```python
# Cliente
import dill as pickle
import socket
from time import sleep

def my_funs():
    def mapper(v):
        return v, 1

    def reducer(my_args):
        v, obs = my_args
        return v, sum(obs)
    return mapper, reducer

def do_request(my_funs, data):
    conn = None
    try:
        conn = socket.create_connection(('127.0.0.1', 1936))
        conn.send(b'\x00')
        my_code = pickle.dumps(my_funs.__code__)
        conn.send(len(my_code).to_bytes(4, 'little', signed=False))
        conn.send(my_code)
        my_data = pickle.dumps(data)
        conn.send(len(my_data).to_bytes(4, 'little'))
        conn.send(my_data)
        job_id = int.from_bytes(conn.recv(4), 'little')
        print(f'Obteniendo datos desde job_id {job_id}')
    finally:
        if conn:
            conn.close()

    result = None
    while result is None:
        try:
            conn = socket.create_connection(('127.0.0.1', 1936))
            conn.send(b'\x01')
            conn.send(job_id.to_bytes(4, 'little'))
            result_size = int.from_bytes(conn.recv(4), 'little')
            result = pickle.loads(conn.recv(result_size))
        finally:
            if conn:
                conn.close()
        sleep(1)
    print(f'El resultado es {result}')
```

```
if __name__ == '__main__':
    do_request(my_funs, 'Python rocks. Python es lo maximo'.split('
'))
```

Explicación

```
import dill as pickle
import socket
from time import sleep
```

Este bloque importa las librerías necesarias:

- dill as pickle: dill es una extensión de la librería pickle que permite serializar objetos de Python más complejos. Se usa aquí para serializar y enviar funciones a través de una conexión de red.
- socket: para la comunicación de red.
- sleep: pausa la ejecución del programa para esperar por la respuesta del servidor.

```
def my_funs():
    def mapper(v):
        return v, 1

    def reducer(my_args):
        v, obs = my_args
        return v, sum(obs)
    return mapper, reducer
```

my_funs es una función que define y retorna dos funciones internas, mapper y reducer:

- mapper: toma un valor v y retorna un par (v, 1).
- reducer: toma un par (v, obs) donde obs es una lista, y retorna (v, sum(obs)), sumando todos los elementos en obs.

```
def do_request(my_funs, data):
    conn = None
    try:
        conn = socket.create_connection(('127.0.0.1', 1936))
        conn.send(b'\x00')
        ...
        conn.send(my_code)
        ...
        conn.send(my_data)
        job_id = int.from_bytes(conn.recv(4), 'little')
        print(f'Obteniendo datos desde job_id {job_id}')
    finally:
        if conn:
            conn.close()
    ...
```

do_request maneja la conexión de red y la solicitud al servidor:

- Establece una conexión con el servidor en la dirección local (127.0.0.1) y puerto 1936.
- Envía un byte inicial como indicador (posiblemente para indicar el tipo de solicitud).
- Serializa y envía el código de las funciones (my_funs) y los datos.
- Recibe un job_id del servidor, que es un identificador para la tarea solicitada.

```
    result = None
    while result is None:
        try:
            conn = socket.create_connection(('127.0.0.1', 1936))
            conn.send(b'\x01')
            conn.send(job_id.to_bytes(4, 'little'))
            result_size = int.from_bytes(conn.recv(4), 'little')
            result = pickle.loads(conn.recv(result_size))
        finally:
            if conn:
                conn.close()
        sleep(1)
    print(f'El resultado es {result}')
```

Este bloque intenta obtener el resultado de la tarea:

- Se reconecta al servidor y envía un nuevo byte indicador (posiblemente para solicitar el resultado).
- Envía el job_id para identificar la tarea.
- Recibe el resultado, deserializa y lo imprime.

```
if __name__ == '__main__':
    do_request(my_funs, 'Python rocks. Python es lo maximo'.split('
'))
```

Si el script se ejecuta como programa principal, llama a do_request con las funciones definidas y un conjunto de datos de prueba que son palabras divididas de una frase.

```
# Servidor

import asyncio
#import marshal
import pickle
from random import randint
# Codigo de Tiago Rodriguez

results = {}


async def submit_job(reader, writer):
    job_id = max(list(results.keys()) + [0]) + 1
    writer.write(job_id.to_bytes(4, 'little'))
    writer.close()
```

```
    sleep_time = randint(1, 4)
    await asyncio.sleep(sleep_time)
    results[job_id] = sleep_time


async def get_results(reader, writer):
    job_id = int.from_bytes(await reader.read(4), 'little')
    data = pickle.dumps(results.get(job_id, None))
    writer.write(len(data).to_bytes(4, 'little'))
    writer.write(data)


async def accept_requests(reader, writer):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(reader, writer)
    elif op[0] == 1:
        await get_results(reader, writer)


async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1',
1936)
    async with server:
        await server.serve_forever()


asyncio.run(main())
```

Este código implementa un servidor en Python utilizando asyncio que maneja solicitudes de tareas (jobs) y devuelve resultados asociados a esas tareas. Se relaciona directamente con el código anterior, actuando como el servidor al cual se conecta el cliente para enviar y recibir datos.

```
import asyncio
import pickle
from random import randint

results = {}
```

- asyncio: Una biblioteca de Python para escribir código concurrente utilizando la sintaxis async/await.
- pickle: Utilizado para serializar y deserializar objetos en Python.
- randint: Genera números enteros aleatorios, usado aquí para simular un tiempo de procesamiento aleatorio.
- results: Un diccionario para almacenar los resultados de las tareas, donde las claves son job_id y los valores son tiempos simulados de ejecución.

```
async def submit_job(reader, writer):
    job_id = max(list(results.keys()) + [0]) + 1
    writer.write(job_id.to_bytes(4, 'little'))
    writer.close()
    sleep_time = randint(1, 4)
    await asyncio.sleep(sleep_time)
    results[job_id] = sleep_time
```

submit_job es una función asíncrona que maneja la creación de nuevos trabajos:

- Genera un job_id único incrementando el máximo job_id existente.
- Envía este job_id al cliente.
- Cierra el escritor para finalizar la transmisión.
- Simula un tiempo de procesamiento aleatorio y almacena este tiempo en el diccionario results.

```
async def get_results(reader, writer):
    job_id = int.from_bytes(await reader.read(4), 'little')
    data = pickle.dumps(results.get(job_id, None))
    writer.write(len(data).to_bytes(4, 'little'))
    writer.write(data)
```

get_results recupera y envía los resultados de un trabajo específico:

- Lee el job_id enviado por el cliente.
- Obtiene el resultado asociado desde el diccionario results, lo serializa y lo envía de vuelta al cliente.

```
async def accept_requests(reader, writer):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(reader, writer)
    elif op[0] == 1:
        await get_results(reader, writer)
```

accept_requests determina el tipo de solicitud basada en un byte inicial y llama a la función correspondiente para manejar la solicitud:

- 0 para una nueva tarea.
- 1 para recuperar los resultados de una tarea.

```
async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1',
1936)
    async with server:
        await server.serve_forever()
```

main inicia el servidor en la dirección 127.0.0.1 y puerto 1936, y lo mantiene corriendo indefinidamente para manejar solicitudes entrantes.

```
asyncio.run(main())
```

Inicia la ejecución del servidor usando el event loop de asyncio.

Este servidor es esencialmente el lado del servidor para el código del cliente presentado anteriormente. Juntos, implementan un sistema básico donde el cliente envía datos y funciones para ser ejecutadas, y el servidor maneja esas solicitudes, realiza cálculos (simulados aquí como tiempos de espera), y devuelve los resultados.

```
### sleep.py

import asyncio

async def lazy_printer(delay, message):
    await asyncio.sleep(delay)
    print(message)

asyncio.wait([lazy_printer(1, 'Lento'), lazy_printer(0, 'Full
velocidad')])
#asyncio.run()
```

El código ilustra un ejemplo simple de uso de asyncio, una biblioteca en Python diseñada para manejar la ejecución concurrente de código usando corutinas.

```
async def lazy_printer(delay, message):
    await asyncio.sleep(delay)
    print(message)
```

lazy_printer es una función asíncrona que recibe dos argumentos: delay y message. La función pausa su ejecución por el número de segundos especificado por delay (esto simula un trabajo que tarda un cierto tiempo en completarse) y luego imprime el message dado. await asyncio.sleep(delay) es crucial aquí porque le permite a otras corutinas ejecutarse mientras lazy_printer está en espera.

```
asyncio.wait([lazy_printer(1, 'Lento'), lazy_printer(0, 'Full
velocidad')])
```

Aquí se usa asyncio.wait, una función que espera a que las corutinas especificadas en el iterable (en este caso, dos llamadas a lazy_printer) terminen. asyncio.wait no inicia la ejecución de las corutinas por sí misma, solo las organiza para que se ejecuten cuando el bucle de eventos de asyncio esté corriendo.

Una aclaración importante es que asyncio.wait retorna dos conjuntos de Tasks (un tipo de objeto Future en asyncio que encapsula la ejecución de una corutina): uno para las tareas que se completaron y otro para las que no se completaron. En este contexto, el código no se está ejecutando realmente porque no hay un bucle de eventos corriendo las corutinas.

```
# asyncio.run()
```

asyncio.run() es una función que se utiliza para ejecutar la corutina principal y todas las corutinas que se lanzan desde ella. Es la forma recomendada de ejecutar código asyncio completo. asyncio.run() toma una corutina como argumento, inicia un nuevo bucle de eventos, corre la corutina, y finaliza el bucle de eventos una vez que la corutina termina.

Sin embargo, este código específicamente no está completo porque no está envuelto dentro de asyncio.run(). Un uso adecuado para ejecutar las tareas concurrentemente sería algo así:

```
async def main():
    await asyncio.wait([
        lazy_printer(1, 'Lento'),
        lazy_printer(0, 'Full velocidad')
    ])

asyncio.run(main())
```

Aquí, main() es la corutina que espera a que se complete el asyncio.wait, y asyncio.run(main()) arranca el bucle de eventos y ejecuta main() hasta su finalización.

Ejercicio 1: Extendiendo la funcionalidad del servidor

Basado en el código del servidor que utiliza asyncio para gestionar trabajos, extiende el servidor para realizar tareas más complejas. Añade una función que pueda procesar una lista de números y devolver su suma. Deberás modificar tanto el código del servidor como el del cliente para manejar esta nueva funcionalidad.

Pasos:

- Define una nueva función en el cliente para enviar una lista de números al servidor.
- En el servidor, añade una corutina que reciba esta lista, calcule la suma y almacene el resultado en el diccionario results.
- Asegúrate de que el cliente pueda solicitar y recibir el resultado de esta suma.

Ejercicio 2: Manejo de múltiples clientes

Modifica el servidor basado en asyncio para que pueda manejar múltiples clientes simultáneamente. Cada cliente debería ser capaz de enviar múltiples solicitudes sin esperar que las anteriores se completen.

Pasos:

- Modifica el código del servidor para que pueda manejar distintas solicitudes de múltiples clientes de forma asíncrona.
- Asegúrate de que el servidor pueda gestionar y mantener el estado de cada cliente por separado.
- Prueba la capacidad del servidor conectando varios clientes al mismo tiempo y realizando diferentes solicitudes.

Ejercicio 3: Simulación de tareas de larga duración

Utilizando el ejemplo de lazy_printer, crea un simulador para tareas de larga duración que afecten la respuesta del servidor basado en la complejidad de la tarea.

Pasos:

- Define varias corutinas que simulen diferentes tiempos y complejidades de tareas (por ejemplo, cálculos matemáticos complejos o procesamiento de texto).
- Utiliza asyncio.wait para manejar estas tareas de forma concurrente en el cliente y envía estas tareas al servidor.
- Añade lógica en el servidor para responder a estas tareas con diferentes tiempos de espera basados en su complejidad.

Ejercicio 4: Mejorando la eficiencia con paralelismo

Investiga cómo asyncio puede integrarse con bibliotecas de procesamiento en paralelo como concurrent.futures para mejorar la eficiencia del servidor al manejar tareas que son intensivas en CPU.

Pasos:

- Modifica el servidor para utilizar concurrent.futures.ProcessPoolExecutor para ejecutar cálculos intensivos en paralelo.
- Crea tareas que requieran intensivo uso de CPU y envíalas al servidor.
- Observa y compara el rendimiento cuando se usan corutinas simples versus la ejecución en paralelo.

```
### Respuestas
```

Parte 2

```
from collections import defaultdict
# Codigo de Thiago Rodriguez

def map_reduce_ultra_naive(my_input, mapper, reducer):
    map_results = map(mapper, my_input)

    distributor = defaultdict(list)
    for key, value in map_results:
        distributor[key].append(value)

    return map(reducer, distributor.items())


words = 'Python es lo mejor Python rocks'.split(' ')

emiter = lambda word: (word, 1)
counter = lambda emitted: (emitted[0], sum(emitted[1]))
```

```
a = list(map_reduce_ultra_naive(words, emiter, counter))

print(a)
```

El código que proporcionaste implementa una versión muy simplificada del patrón de diseño "MapReduce", comúnmente utilizado para procesar y generar grandes conjuntos de datos con un modelo distribuido de computación en paralelo. Aquí, el código no necesita un cliente ni operaciones de sleep porque es una implementación completamente sincrónica y local.

```
from collections import defaultdict
```

Esta importación trae defaultdict desde el módulo collections. defaultdict es una subclase de dict que proporciona un valor predeterminado para la clave que no existe.

```
def map_reduce_ultra_naive(my_input, mapper, reducer):
    map_results = map(mapper, my_input)

    distributor = defaultdict(list)
    for key, value in map_results:
        distributor[key].append(value)

    return map(reducer, distributor.items())
```

La función map_reduce_ultra_naive toma tres argumentos: my_input (datos de entrada), mapper (función de mapeo) y reducer (función de reducción).

- Mapeo: map(mapper, my_input) aplica la función mapper a cada elemento en my_input, generando una lista de tuplas (clave, valor).
- Distribución: Se utiliza defaultdict para agrupar todos los valores asociados con la misma clave en una lista. Esto es necesario para la etapa de reducción.
- Reducción: map(reducer, distributor.items()) aplica la función reducer a cada par (clave, lista de valores) en el distributor.

```
words = 'Python es lo mejor Python rocks'.split(' ')
```

Esta línea divide la cadena en una lista de palabras.

```
emiter = lambda word: (word, 1)
counter = lambda emitted: (emitted[0], sum(emitted[1]))
```

- emiter: Una función lambda que toma una palabra y devuelve una tupla con la palabra y el número 1. Esta es la función de mapeo que prepara los datos para la reducción.
- counter: Una función lambda que toma una tupla (palabra, lista de unos) y devuelve una tupla con la palabra y la suma de los unos, efectivamente contando cuántas veces aparece cada palabra.

```
a = list(map_reduce_ultra_naive(words, emiter, counter))
```

Esta línea aplica la función map_reduce_ultra_naive a la lista de palabras con las funciones emiter y counter especificadas, y luego convierte el resultado (que es un iterador) en una lista.

```
print(a)
```

Imprime el resultado final, que sería una lista de tuplas, cada una mostrando una palabra y su frecuencia en la entrada original.

Ejercicio 1: Paralelización del proceso de MapReduce

Modifica el código para usar múltiples hilos o procesos que ejecuten las funciones de mapeo y reducción de manera paralela. Esto simula cómo funcionaría un entorno de MapReduce distribuido a pequeña escala.

Pasos:

- Utiliza concurrent.futures.ThreadPoolExecutor o concurrent.futures.ProcessPoolExecutor para paralelizar la función de mapeo.
- Implementa la reducción de manera que pueda manejar la entrada de múltiples hilos/procesos de manera sincronizada.
- Compara el rendimiento del enfoque paralelo con el enfoque sincrónico original.

Ejercicio 2: MapReduce distribuido

Crea una simulación de un entorno de MapReduce distribuido donde múltiples clientes pueden enviar sus datos a un servidor que coordina las tareas de mapeo y reducción.

Pasos:

- Desarrolla un servidor que pueda recibir datos de múltiples clientes y asignar tareas de mapeo a diferentes nodos (simulados por hilos o procesos).
- Los resultados del mapeo deben ser enviados de vuelta al servidor, que luego ejecutará la reducción.
- Evalúa cómo la distribución afecta la eficiencia y el tiempo de procesamiento total.

Ejercicio 3: Balanceo de carga en MapReduce

Implementa un mecanismo de balanceo de carga para optimizar la distribución de tareas de mapeo entre varios nodos de procesamiento.

Pasos:

- Diseña una función que determine cómo se distribuyen los datos entrantes entre los nodos de mapeo basados en su carga actual o capacidad.
- Implementa la lógica para que el nodo de reducción pueda esperar y combinar resultados de todos los nodos de mapeo antes de proceder.
- Analiza el impacto del balanceo de carga en la eficiencia del sistema.

Ejercicio 4: Tolerancia a fallos en MapReduce

Añade tolerancia a fallos al sistema MapReduce, permitiendo que el sistema se recupere de errores en los nodos de mapeo o reducción.

Pasos:

- Implementa una funcionalidad que detecte fallos en nodos y reasigne sus tareas a otros nodos disponibles.
- Asegúrate de que los datos necesarios para reasignar tareas estén disponibles, quizás usando replicación.
- Testea el sistema con fallos simulados y evalúa cómo afecta la robustez y el rendimiento del sistema.

Ejercicio 5: Optimización de reducción en MapReduce

Optimiza el paso de reducción para manejar grandes volúmenes de datos más eficientemente.

Pasos:

- Implementa una estrategia de "combiner" local en cada nodo de mapeo para reducir los datos antes de enviarlos al nodo de reducción.
- Asegúrate de que la función de reducción pueda manejar la entrada pre-reducida eficientemente.
- Evalúa cómo esta optimización afecta la carga de red y el tiempo de procesamiento.

```
## Tus respuestas
```

Parte 3

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor
# Codigo de Tiago Rodriguez

def map_reduce_still_naive(my_input, mapper, reducer):
    with Executor() as executor:
        map_results = executor.map(mapper, my_input)

        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)
        results = executor.map(reducer, distributor.items())
    return results


words = filter(lambda x: x!= '', map(lambda x: x.strip().rstrip(), '
'.join(open('texto.txt', 'rt', encoding='utf-8').readlines()).split('
')))

emiter = lambda word: (word, 1)
counter = lambda emitted: (emitted[0], sum(emitted[1]))
```

```
a = list(map_reduce_still_naive(words, emiter, counter))

for i in sorted(a, key=lambda x: x[1]):
    print(i)
```

Este código es una implementación más avanzada de la técnica de MapReduce, utilizando la concurrencia para paralelizar tanto la fase de mapeo como la de reducción. Hace uso del módulo concurrent.futures, específicamente ThreadPoolExecutor, para manejar la ejecución paralela de las funciones de mapeo y reducción.

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor
```

- defaultdict: Una subclase de diccionario que proporciona un valor predeterminado para claves que no están en el diccionario.
- ThreadPoolExecutor: Una clase para ejecutar llamadas de función de manera asincrónica usando un pool de hilos.

```
def map_reduce_still_naive(my_input, mapper, reducer):
    with Executor() as executor:
        map_results = executor.map(mapper, my_input)

        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)
        results = executor.map(reducer, distributor.items())
    return results
```

- map_reduce_still_naive: Una función que toma una entrada my_input, una función mapper y una función reducer. Utiliza un ThreadPoolExecutor para ejecutar las funciones mapper y reducer.
- executor.map(mapper, my_input): Paraleliza la aplicación de la función mapper a cada elemento de my_input.
- distributor: Un defaultdict que agrupa los valores asociados con cada clave producida por la función mapper.
- executor.map(reducer, distributor.items()): Aplica la función reducer a cada grupo de valores (de manera paralela), procesando los elementos que comparten la misma clave.

```
words = filter(lambda x: x!= '', map(lambda x: x.strip().rstrip(), '
'.join(open('texto.txt', 'rt', encoding='utf-8').readlines()).split('
')))
```

- Esta línea prepara words leyendo un archivo texto.txt, dividiendo el contenido en palabras, quitando los espacios en blanco innecesarios y filtrando cualquier string vacío. Este será el input para la función map_reduce_still_naive.

```
emiter = lambda word: (word, 1)
counter = lambda emitted: (emitted[0], sum(emitted[1]))
```

- emiter: Una función mapper que toma una palabra y devuelve un par (palabra, 1).
- counter: Una función reducer que toma un par (palabra, lista de unos) y devuelve (palabra, total), sumando todos los unos para contar cuántas veces aparece cada palabra.

```
a = list(map_reduce_still_naive(words, emiter, counter))
for i in sorted(a, key=lambda x: x[1]):
    print(i)
```

- list(map_reduce_still_naive(words, emiter, counter)): Ejecuta el proceso de MapReduce y convierte los resultados (un iterable de tuplas) en una lista.
- sorted(a, key=lambda x: x[1]): Ordena los resultados basado en la frecuencia de cada palabra (el segundo elemento de cada tupla).
- El bucle for imprime cada palabra junto con su frecuencia, ordenadas por frecuencia.

Los resultados dependerán del contenido del archivo texto.txt, pero en general, la salida mostrará cada palabra que aparece en el texto junto con el número de veces que aparece, ordenadas por su frecuencia. Este proceso se realiza de manera eficiente utilizando múltiples hilos para acelerar tanto el mapeo como la reducción.

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor
from time import sleep

# Codigo de Tiago Rodriguez

def report_progress(futures, tag, callback):
    not_done = 1
    done = 0
    while not_done > 0:
        not_done = 0
        done = 0
        for fut in futures:
            if fut.done():
                done +=1
            else:
                not_done += 1
        sleep(0.5)
        if not_done > 0 and callback:
            callback(tag, done, not_done)


def async_map(executor, mapper, data):
    futures = []
    for datum in data:
        futures.append(executor.submit(mapper, datum))
    return futures


def map_less_naive(executor, my_input, mapper):
    map_results = async_map(executor, mapper, my_input)
```

```
    return map_results


def map_reduce_less_naive(my_input, mapper, reducer, callback=None):
    with Executor(max_workers=2) as executor:
        futures = async_map(executor, mapper, my_input)
        report_progress(futures, 'map', callback)
        #wait(futures).done
        map_results = map(lambda f: f.result(), futures)
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)

        futures = async_map(executor, reducer, distributor.items())
        report_progress(futures, 'reduce', callback)
        #wait(futures).done
        results = map(lambda f: f.result(), futures)
    return results


words = filter(lambda x: x!= '', map(lambda x: x.strip().rstrip(), '
'.join(open('texto.txt', 'rt', encoding='utf-8').readlines()).split('
')))

def emitter(word):
    #sleep(10)
    return word, 1


counter = lambda emitted: (emitted[0], sum(emitted[1]))

def reporter(tag, done, not_done):
    print(f'Operacion {tag}: {done}/{done+not_done}')

words = 'Python es super,  Python rocks'.split(' ')
a = map_reduce_less_naive(words, emitter, counter, reporter)

for i in sorted(a, key=lambda x: x[1]):
    print(i)

#words = 'Python es super Python rocks'.split(' ')

#with Executor(max_workers=4) as executor:
#    maps = map_less_naive(executor, words, emitter)
#    print(maps[-1])
#    not_done = 1
#    while not_done > 0:
#        not_done = 0
#        for fut in maps:
#            not_done += 1 if not fut.done() else 0
```

```
#         sleep(1)
#         print(f'Aun no ha finalizado: {not_done}')
```

El código proporcionado es una versión más sofisticada del patrón de diseño MapReduce, la cual implementa funcionalidades adicionales de manejo de concurrencia utilizando ThreadPoolExecutor de concurrent.futures

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor
from time import sleep
```

- defaultdict: Facilita el agrupamiento de resultados.
- Executor: Se utiliza para la ejecución concurrente.
- sleep: Utilizado aquí para pausas durante la monitorización del progreso.

```
def report_progress(futures, tag, callback):
    # Inicializa 'not_done' para entrar en el bucle while.
    not_done = 1

    # Bucle que se ejecuta mientras haya tareas sin finalizar.
    while not_done > 0:
        # Reinicia los contadores en cada iteración del bucle.
        not_done = 0
        done = 0

        # Itera sobre cada futuro en la lista de 'futures'.
        for fut in futures:
            # Comprueba si el futuro ha terminado y actualiza los
contadores correspondientes.
            if fut.done():
                done += 1  # Aumenta el contador de tareas
completadas.
            else:
                not_done += 1  # Aumenta el contador de tareas no
completadas.

        # Pausa de medio segundo para no saturar con demasiadas
comprobaciones.
        sleep(0.5)

        # Si hay tareas sin completar y se ha proporcionado una
función de callback,
        # llama al callback con la etiqueta actual, el número de
tareas completadas y las no completadas.
        if not_done > 0 and callback:
            callback(tag, done, not_done)
```

Esta función monitorea el progreso de un conjunto de futuros (tareas enviadas a ejecución en el Executor). Informa sobre la cantidad de tareas completadas y pendientes, permitiendo visualizar el estado de la ejecución en tiempo real.

```
def async_map(executor, mapper, data):
    # Se inicializa una lista vacía para almacenar los objetos Future.
    futures = []

    # Itera sobre cada elemento en el conjunto de datos de entrada.
    for datum in data:
        # Utiliza el executor para lanzar la función mapper de forma
asincrónica.
        # `executor.submit()` programa la función `mapper` para ser
ejecutada con el argumento `datum`.
        # Esto devuelve un objeto Future que representa la ejecución
pendiente o futura.
        futures.append(executor.submit(mapper, datum))

    # Devuelve la lista de objetos Future. Cada Future contendrá el
resultado de aplicar
    # la función `mapper` a un elemento de `data` una vez que se
complete la ejecución.
    return futures
```

Reemplaza directamente el uso de executor.map() con una lista de futuros, permitiendo un manejo más fino y personalizado de cada tarea individual, incluyendo la posibilidad de monitorización.

```
def map_reduce_less_naive(my_input, mapper, reducer, callback=None):
    # Crea un ThreadPoolExecutor con un máximo de dos trabajadores.
    # El contexto 'with' asegura que el executor se cierra
adecuadamente después de su uso.
    with Executor(max_workers=2) as executor:
        # Utiliza la función async_map para lanzar las tareas de mapeo
de forma asincrónica.
        # Retorna una lista de objetos Future que representan las
operaciones de mapeo.
        futures = async_map(executor, mapper, my_input)

        # Función para reportar el progreso de las tareas de mapeo.
        # Utiliza un callback para informar sobre el estado de la
ejecución de mapeo.
        report_progress(futures, 'map', callback)

        # Recolecta los resultados de las tareas de mapeo completadas.
        # `f.result()` bloquea hasta que el futuro se complete y
devuelve el resultado.
        map_results = map(lambda f: f.result(), futures)
```

```
        # Utiliza defaultdict para agrupar los valores por claves
generadas en el mapeo.
        distributor = defaultdict(list)
        for key, value in map_results:
            # Agrega el valor al listado correspondiente a la clave en
el diccionario.
            distributor[key].append(value)

        # Lanza las tareas de reducción de forma asincrónica sobre los
grupos de valores.
        # Cada tarea de reducción procesa los valores asociados a una
clave.
        futures = async_map(executor, reducer, distributor.items())

        # Función para reportar el progreso de las tareas de
reducción.
        report_progress(futures, 'reduce', callback)

        # Recolecta los resultados finales de las tareas de reducción.
        results = map(lambda f: f.result(), futures)

    # Devuelve los resultados de la operación de MapReduce.
    return results
```

Implementa el proceso MapReduce usando las funciones async_map y report_progress. La función ahora es capaz de reportar el progreso tanto de la fase de mapeo como de reducción.

Mejoras con respecto al código anterior

- Monitoreo de progreso: La inclusión de report_progress permite un seguimiento en tiempo real de la ejecución, lo cual es útil en entornos de producción para diagnóstico y monitoreo.
- Manejo explícito de concurrencia: Usar async_map con executor.submit() en lugar de executor.map() da un control más detallado sobre cada tarea y permite manejar excepciones, cancelaciones y otras operaciones a nivel de tarea individual.
- Configuración de parámetros del Executor: Permite ajustar el número de trabajadores (max_workers), lo que puede optimizar el rendimiento según las características del hardware y la carga de trabajo.

El programa ejecuta la función map_reduce_less_naive sobre una lista de palabras, utilizando las funciones emitter y counter como funciones de mapeo y reducción, respectivamente. Imprime los resultados, que muestran cada palabra y su frecuencia, de manera ordenada por frecuencia. Esto es idéntico en función al código anterior, pero con una ejecución más sofisticada y transparente.

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor,
as_completed
import os
```

```python
# Definimos las funciones de mapeo y reducción
def map_function(word):
    return (word, 1)

def reduce_function(item):
    word, counts = item
    return (word, sum(counts))

# Función para realizar MapReduce con seguimiento del progreso
def map_reduce_with_progress(my_input, mapper, reducer):
    # Distribuye los resultados del mapeo
    distributor = defaultdict(list)

    # Inicia un ejecutor de hilos
    with Executor() as executor:
        # Primero, ejecutamos el mapeo
        future_to_word = {executor.submit(mapper, word): word for word
in my_input}
        for future in as_completed(future_to_word):
            word, count = future.result()
            distributor[word].append(count)

        # Después, ejecutamos la reducción
        results = executor.map(reducer, distributor.items())

    # Convertimos los resultados a una lista y retornamos
    return list(results)

# Preparamos los datos de entrada leyendo desde un archivo
def prepare_data(file_path):
    with open(file_path, 'rt', encoding='utf-8') as file:
        # Limpiamos y dividimos las palabras, filtramos las cadenas
vacías
        words = filter(None, [word.strip().rstrip() for line in file
for word in line.split()])
    return words

# Ejecutamos el proceso de MapReduce
if __name__ == "__main__":
    words = prepare_data('texto.txt')
    results = map_reduce_with_progress(words, map_function,
reduce_function)

    # Ordenamos los resultados y los imprimimos
    for word, count in sorted(results, key=lambda x: x[1],
reverse=True):
        print(f"Palabra: '{word}', Frecuencia: {count}")
```

El código anterior es otra variante de una implementación del patrón MapReduce utilizando programación concurrente, específicamente a través de ThreadPoolExecutor del módulo concurrent.futures.

Similitudes:

- Uso de Concurrency: Al igual que en los códigos anteriores, este código utiliza ThreadPoolExecutor para paralelizar las tareas de mapeo y reducción, lo que permite una ejecución más eficiente al procesar múltiples elementos en paralelo.

Estructura de MapReduce:

- Mapeo: Al igual que en otros ejemplos, hay una función de mapeo (map_function) que procesa cada palabra de entrada y la mapea a un par (palabra, 1).
- Reducción: Utiliza una función de reducción (reduce_function) que suma las cuentas de cada palabra, similar a cómo se hace en ejemplos anteriores.

Uso de defaultdict: Utiliza defaultdict para colectar y agrupar resultados del mapeo antes de la reducción, de la misma manera que en los códigos anteriores.

Diferencias:

Manejo de Future:

- En este código, se utiliza future_to_word para mapear cada futuro a la palabra que fue procesada, lo que facilita la gestión y seguimiento de las tareas a medida que se completan. Esto permite un control más granular sobre las operaciones concurrentes.

Utiliza as_completed para iterar sobre los futuros a medida que se completan, lo cual es una técnica eficiente para manejar tareas que pueden terminar en tiempos diferentes.

Feedback de progreso:

- A diferencia de los códigos anteriores que implementaban funciones específicas para reportar el progreso (report_progress), este código no incluye una función explícita para reportar progreso. Sin embargo, el uso de as_completed permite actuar tan pronto como una tarea se completa, lo que puede ser útil para actualizar interfaces de usuario o logs en tiempo real.

Lectura y preparación de datos:

- Este código incluye una función prepare_data que lee directamente desde un archivo y prepara los datos para el proceso de MapReduce. Esto contrasta con algunos de los ejemplos anteriores donde los datos eran simplemente una lista de palabras o eran preparados fuera de las funciones principales.

Ejercicio 1: Implementación de MapReduce mejorada

Modifica una de las implementaciones de MapReduce presentadas para incluir un manejo de excepciones robusto que permita la aplicación continuar incluso si una de las tareas de mapeo o reducción falla. Además, implementa un sistema de logging que registre eventos como el inicio y fin de cada tarea, errores, y el progreso general de las operaciones de mapeo y reducción.

Sugerencia: Utiliza el módulo logging de Python para los registros y try-except dentro de las funciones de mapeo y reducción.

Ejercicio 2: Paralelización de la lectura de datos

Modifica la función prepare_data para que pueda leer de múltiples archivos en paralelo usando ThreadPoolExecutor, y luego combina los resultados en una sola lista de palabras antes de pasarlas al proceso de MapReduce. Sugerencia: Supón que tienes varios archivos de texto (texto1.txt, texto2.txt, etc.) y usa executor.map para leer cada uno en paralelo.

Ejercicio 3: Reducción por claves específicas

Añade una funcionalidad que permita especificar un conjunto de claves (palabras) de interés, y modifica el proceso de reducción para que solo sume las cuentas de estas claves seleccionadas.

Sugerencia: Puedes modificar la función de reducción para que primero verifique si la clave está en el conjunto de claves de interés antes de procesarla.

Ejercicio 4: Escalabilidad en el número de workers

Realiza pruebas de rendimiento variando el número de workers en ThreadPoolExecutor para observar cómo afecta al tiempo total de procesamiento del MapReduce.

Sugerencia: Utiliza un conjunto grande de datos y prueba con configuraciones de 1, 2, 4, 8 y 16 workers. Registra los tiempos de ejecución para cada configuración y analiza los resultados.

Ejercicio 5: MapReduce con Asyncio

Transforma el código MapReduce para que utilice programación asincrónica. Esto incluirá convertir las funciones de mapeo y reducción en funciones asincrónicas y usar asyncio.gather para manejar la concurrencia.

Sugerencia: Familiarízate con asyncio y cómo las coroutines pueden ser utilizadas para tareas I/O-bound como la lectura de archivos.

Ejercicio 6: Implementación Multihilo de MapReduce

Aprovecha ThreadPoolExecutor para implementar una versión de MapReduce que procese en paralelo tanto la fase de mapeo como la de reducción. Añade manejo de errores y asegúrate de que el sistema pueda recuperarse de fallos en las tareas individuales.

Sugerencia: Incorpora logging para monitorear el progreso y los posibles errores en las distintas etapas del procesamiento.

Ejercicio 7: MapReduce distribuido sobre red

Desarrolla un sistema básico donde un servidor central coordina múltiples clientes que realizan tareas de mapeo y reducción. Los clientes pueden estar en diferentes máquinas y comunicarse con el servidor para obtener tareas y devolver resultados.

Sugerencia: Utiliza sockets para la comunicación entre el servidor y los clientes. Implementa funcionalidades básicas en el servidor para asignar tareas y recolectar resultados.

Ejercicio 8: Análisis de rendimiento con diferentes niveles de concurrencia

Realiza un experimento controlado donde ejecutas la misma carga de trabajo de MapReduce con diferentes números de hilos o procesos. Documenta cómo cambia el rendimiento con respecto a diferentes configuraciones.

Sugerencia: Puedes usar ThreadPoolExecutor y ProcessPoolExecutor para variar entre multihilo y multiproceso, respectivamente. Considera la utilización de CPU y la eficiencia de I/O en tu análisis.

Ejercicio 9: MapReduce asincrónico

Reescribe una de las implementaciones de MapReduce para usar asyncio y coroutines en lugar de hilos, lo que puede ser más eficiente para operaciones I/O-bound como la lectura de archivos grandes o la espera de respuestas de red.

Sugerencia: Asegúrate de comprender cómo asyncio gestiona la concurrencia de I/O y cómo puede integrarse con operaciones de CPU-bound mediante el uso de loop.run_in_executor.

Ejercicio 10: Tolerancia a fallos en sistemas de MapReduce

Añade características de tolerancia a fallos al sistema MapReduce para manejar errores como la caída de un nodo de procesamiento. Implementa estrategias para reasignar tareas de nodos fallidos a nodos en funcionamiento sin perder datos.

Sugerencia: Considera el uso de checkpoints o la replicación de tareas para garantizar que el sistema pueda recuperarse de fallos sin necesidad de reiniciar todo el proceso desde el principio.

```
## Tus respuestas
```

Parte 4

```
from collections import defaultdict
from concurrent.futures import ProcessPoolExecutor as Executor
from time import sleep
# Codigo de Tiago Rodriguez

def report_progress(futures, tag, callback):
    # Inicializa los contadores de tareas completadas y no
completadas.
    not_done = 1
    done = 0
    # Bucle mientras haya tareas no completadas.
    while not_done > 0:
        not_done = 0
        done = 0
        # Itera sobre cada futuro para verificar si está completado.
        for fut in futures:
            if fut.done():
                done += 1  # Incrementa el contador de completadas.
            else:
```

```
                not_done += 1  # Incrementa el contador de no
completadas.
        # Pausa el bucle por medio segundo para reducir la carga de
comprobación.
        sleep(0.5)
        # Si hay una función de callback, informa el progreso actual.
        if callback:
            callback(tag, done, not_done)


def async_map(executor, mapper, data):
    # Lista para almacenar los futuros.
    futures = []
    # Envía cada elemento de los datos a la función mapper usando el
executor.
    for datum in data:
        futures.append(executor.submit(mapper, datum))
    # Devuelve la lista de futuros.
    return futures


def map_less_naive(executor, my_input, mapper):
    # Aplica la función de mapeo a cada entrada y devuelve los
resultados.
    map_results = async_map(executor, mapper, my_input)
    return map_results


def map_reduce_less_naive(my_input, mapper, reducer, callback=None):
    # Crea un ejecutor con un máximo de dos trabajadores.
    with Executor(max_workers=2) as executor:
        # Mapea las entradas y monitorea el progreso.
        futures = async_map(executor, mapper, my_input)
        report_progress(futures, 'map', callback)
        # Obtiene los resultados del mapeo.
        map_results = map(lambda f: f.result(), futures)
        # Agrupa los resultados por clave.
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)

        # Realiza el proceso de reducción y monitorea el progreso.
        futures = async_map(executor, reducer, distributor.items())
        report_progress(futures, 'reduce', callback)
        # Obtiene los resultados de la reducción.
        results = map(lambda f: f.result(), futures)
    # Devuelve los resultados finales.
    return results
```

```
def emitter(word):
    # Función de mapeo que emite cada palabra con un conteo inicial de
1.
    return word, 1


def counter(emitted):
    # Función de reducción que suma los conteos de cada palabra.
    return emitted[0], sum(emitted[1])


def reporter(tag, done, not_done):
    # Imprime el estado del progreso para la operación actual.
    print(f'Operacion {tag}: {done}/{done+not_done}')

# Prepara los datos y ejecuta el proceso MapReduce.
words = 'Python es super Python rocks'.split(' ')
a = map_reduce_less_naive(words, emitter, counter, reporter)

# Imprime los resultados ordenados por frecuencia.
for i in sorted(a, key=lambda x: x[1]):
    print(i)
```

Este código es una variante del patrón de diseño MapReduce, implementado utilizando programación concurrente con ProcessPoolExecutor de la biblioteca concurrent.futures.

```
from collections import defaultdict
from concurrent.futures import ProcessPoolExecutor as Executor
from time import sleep
```

- defaultdict: Utilizado para agrupar automáticamente valores en listas sin necesidad de inicializar manualmente cada clave.
- ProcessPoolExecutor: Una variante de Executor que utiliza procesos en lugar de hilos. Esto puede ser ventajoso para tareas computacionalmente intensivas porque evita problemas de Global Interpreter Lock (GIL) en Python.
- sleep: Utilizado para pausar la ejecución entre verificaciones del estado de las tareas.

```
def map_less_naive(executor, my_input, mapper):
    # Aplica la función de mapeo a cada entrada y devuelve los
resultados.
    map_results = async_map(executor, mapper, my_input)
    return map_results
```

Esta función se centra exclusivamente en la etapa de mapeo del proceso MapReduce:

- Aplicación de la función de mapeo: Recibe un ejecutor, datos de entrada (my_input) y una función mapper. Utiliza async_map para enviar cada elemento de entrada al ejecutor, donde la función mapper se aplica de manera concurrente.

- Devolución de resultados de mapeo: A diferencia de map_reduce_less_naive, esta función solo devuelve los resultados de la fase de mapeo, sin proceder a la reducción.

```
def map_reduce_less_naive(my_input, mapper, reducer, callback=None):
    # Crea un ejecutor con un máximo de dos trabajadores.
    with Executor(max_workers=2) as executor:
        # Mapea las entradas y monitorea el progreso.
        futures = async_map(executor, mapper, my_input)
        report_progress(futures, 'map', callback)
        # Obtiene los resultados del mapeo.
        map_results = map(lambda f: f.result(), futures)
        # Agrupa los resultados por clave.
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)

        # Realiza el proceso de reducción y monitorea el progreso.
        futures = async_map(executor, reducer, distributor.items())
        report_progress(futures, 'reduce', callback)
        # Obtiene los resultados de la reducción.
        results = map(lambda f: f.result(), futures)
    # Devuelve los resultados finales.
    return results
```

Esta función ejecuta un proceso completo de MapReduce y se encarga de:

- Inicia un ejecutor con procesos: Utiliza ProcessPoolExecutor con un máximo de dos trabajadores. Esto permite que las tareas se ejecuten en procesos separados, lo cual es útil para evitar el bloqueo del Global Interpreter Lock (GIL) en Python, y es especialmente beneficioso para tareas computacionalmente intensivas.
- Mapeo de entradas: Utiliza la función async_map para enviar tareas de mapeo al ejecutor. Cada tarea de mapeo aplica la función mapper a un elemento de my_input. async_map devuelve una lista de objetos Future, cada uno representando una tarea de mapeo pendiente o en ejecución.
- Monitoreo del progreso de mapeo: report_progress se llama con los futures del mapeo para informar el progreso de estas tareas hasta que todas estén completadas. Esto se hace mediante una función de callback que puede actualizar interfaces de usuario o logs.
- Recolección y agrupación de resultados de mapeo: Una vez que todos los futuros de mapeo están completos, se recogen los resultados y se agrupan por clave usando defaultdict(list). Esto prepara los datos para la etapa de reducción.
- Reducción de resultados: Similar a la etapa de mapeo, se envían tareas de reducción al mismo ejecutor utilizando async_map, pero esta vez aplicando la función reducer a cada grupo de valores asociados a las mismas claves.
- Monitoreo del progreso de reducción: Al igual que con el mapeo, se monitorea el progreso de las tareas de reducción.
- Recolección de resultados finales: Finalmente, se recogen los resultados de las tareas de reducción y se devuelven.

Relación entre map_reduce_less_naive y map_less_naive

- map_less_naive puede considerarse como un subcomponente dentro de map_reduce_less_naive, específicamente encargado de la ejecución del mapeo.
- map_reduce_less_naive es una implementación más completa que no solo maneja el mapeo utilizando map_less_naive (o su funcionalidad equivalente), sino que también integra la reducción y el seguimiento del progreso en un solo flujo de trabajo.

```
from collections import defaultdict
import multiprocessing as mp
from time import sleep

def report_progress(futures, tag, callback):
    # Inicializa contadores para tareas completadas y no completadas.
    not_done = 1
    done = 0
    # Continúa mientras haya tareas no completadas.
    while not_done > 0:
        not_done = 0
        done = 0
        # Itera sobre los objetos AsyncResult para verificar su
estado.
        for fut in futures:
            if fut.ready():
                done += 1  # Incrementa contador de completados si el
futuro ha terminado.
            else:
                not_done += 1  # Incrementa contador de no completados
si el futuro aún está en progreso.
        sleep(0.5)  # Pausa para no saturar la CPU.
        # Si se proporciona una función de callback, llama a la
función con el estado actual.
        if callback:
            callback(tag, done, not_done)

def map_reduce(my_input, mapper, reducer, callback=None):
    # Utiliza un pool de procesos para ejecutar tareas de mapeo y
reducción.
    with mp.Pool(2) as pool:
        # Aplica la función mapper a cada elemento del input de manera
asincrónica.
        futures = [pool.apply_async(mapper, (word,)) for word in
my_input]
        report_progress(futures, 'map', callback)
        # Recolecta los resultados del mapeo.
        map_results = [future.get() for future in futures]

        # Agrupa los resultados de mapeo usando un defaultdict.
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)
```

```
        # Aplica la función reducer a cada grupo de elementos de
manera asincrónica.
        reduce_futures = [pool.apply_async(reducer, (item,)) for item
in distributor.items()]
        report_progress(reduce_futures, 'reduce', callback)
        # Recolecta los resultados de la reducción.
        results = [future.get() for future in reduce_futures]
    return results

def emitter(word):
    sleep(1)
    return word, 1

def counter(emitted):
    return emitted[0], sum(emitted[1])

def reporter(tag, done, not_done):
    print(f'Operacion {tag}: {done}/{not_done}')

words = 'Python es super, Python rocks'.split(' ')
a = map_reduce(words, emitter, counter, reporter)

for i in sorted(a, key=lambda x: x[1]):
    print(i)
```

El código anterior es una implementación de un proceso de MapReduce utilizando el módulo multiprocessing en Python, que maneja la concurrencia mediante procesos en lugar de hilos. Además, incorpora funciones para monitorizar el progreso de las tareas de mapeo y reducción.

```
def map_reduce(my_input, mapper, reducer, callback=None):
    # Implementa el proceso completo de MapReduce utilizando un pool
de procesos.
```

- Inicialización de un pool de procesos: Se crea un Pool de dos procesos para manejar las tareas en paralelo.
- Mapeo asincrónico: Se envían tareas de mapeo de forma asincrónica para cada elemento del input.
- Agrupación de resultados: Los resultados del mapeo se agrupan por claves en un defaultdict.
- Reducción asincrónica: Se procesan las reducciones de forma asincrónica para cada grupo de valores asociados a una clave.
- Recolección de resultados finales: Los resultados finales de la reducción se recogen y devuelven.

```
def emitter(word):
    # Simula una operación de mapeo que tarda tiempo, ideal para
procesos intensivos.
```

Función de mapeo (emitter): Toma una palabra, espera un segundo (simulando carga de procesamiento) y devuelve un par (palabra, 1).

```
def counter(emitted):
    # Reduce los resultados sumando los conteos para cada palabra
clave.
```

Función de reducción (counter): Toma una tupla de palabra y lista de conteos, y devuelve la suma de esos conteos junto con la palabra.

```
def reporter(tag, done, not_done):
    # Imprime el estado del progreso para las operaciones de mapeo o
reducción.
```

reporter: Es llamada por report_progress para imprimir el estado actual del progreso, mostrando cuántas tareas han sido completadas y cuántas están pendientes.

```
words = 'Python es super, Python rocks'.split(' ')
a = map_reduce(words, emitter, counter, reporter)
```

Proceso de MapReduce: El conjunto de palabras es procesado usando las funciones de mapeo y reducción definidas, con la capacidad de monitorizar el progreso.

```
for i in sorted(a, key=lambda x: x[1]):
    print(i)
```

Los resultados finales del proceso de MapReduce son ordenados por la frecuencia de las palabras y luego impresos.

```
from collections import defaultdict
import multiprocessing as mp
from time import sleep
#Tiago Rodriguez

def report_progress(map_returns, tag, callback):
    done = 0
    num_jobs = len(map_returns)
    while num_jobs > done:
        done = 0
        for ret in map_returns:
            if ret.ready():
                done += 1
        sleep(0.5)
        if callback:
            callback(tag, done, num_jobs - done)


def async_map(pool, mapper, data):
```

```
    async_returns = []
    for datum in data:
        async_returns.append(pool.apply_async(
            mapper, (datum, )))  #The tuple
    return async_returns


def map_reduce(pool, my_input, mapper, reducer, callback=None):
    map_returns = async_map(pool, mapper, my_input)
    report_progress(map_returns, 'map', callback)
    map_results = [ret.get() for ret in map_returns]
    distributor = defaultdict(list)
    for key, value in map_results:
        distributor[key].append(value)
    returns = async_map(pool, reducer, distributor.items())
    results = [ret.get() for ret in returns]
    return results


def emitter(word):
    sleep(1)
    return word, 1


def counter(emitted):
    return emitted[0], sum(emitted[1])


def reporter(tag, done, not_done):
    print(f'Operacion {tag}: {done}/{done+not_done}')


words = 'Python es super Python rocks'.split(' ')
pool = mp.Pool(2)
results = map_reduce(pool, words, emitter, counter, reporter)
pool.close()
pool.join()


for result in sorted(results, key=lambda x: x[1]):
    print(result)
```

El código proporcionado mejora algunos aspectos clave respecto al código anterior relacionados con la eficiencia y la estructura del manejo de procesos y tareas asincrónicas. A continuación, se destacan las mejoras y cambios realizados:

1 . Manejo mejorado de procesos del Pool

En el código anterior, el manejo del Pool de multiprocessing se realizaba dentro de la función map_reduce, lo cual limitaba su reutilización para múltiples tareas sin recrear el pool cada vez. En el nuevo código, el pool se crea fuera de la función map_reduce y se pasa como argumento.

Esto permite mayor flexibilidad y eficiencia, ya que el pool puede ser utilizado para múltiples tareas o múltiples llamadas a map_reduce sin necesidad de inicialización y terminación repetidas. El cierre (close()) y la espera de que todas las tareas se completen (join()) se manejan también fuera de la función map_reduce, lo que es una práctica recomendada para un manejo limpio y eficiente de los recursos del pool.

2 . Función async_map generalizada

La función async_map se ha generalizado y extraído fuera de la función map_reduce, haciéndola una función independiente que puede ser reutilizada para aplicar cualquier función a un conjunto de datos de manera asincrónica utilizando un pool dado. Esto no solo mejora la modularidad del código sino que también aumenta la reusabilidad de la función async_map.

3 . Reporte de progreso simplificado

La función report_progress ha sido simplificada y ahora utiliza directamente el número total de tareas (num_jobs) y las tareas completadas (done) para reportar el progreso, haciendo su implementación más clara y directa. Esto mejora la legibilidad y facilita el mantenimiento del código.

4 . Estructura de código más clara La estructura general del código ha mejorado en términos de claridad y separación de responsabilidades. Cada función tiene un propósito bien definido, y el flujo principal del programa es más claro. Esto facilita la lectura y comprensión del código, lo cual es crucial para la mantenibilidad a largo plazo y la colaboración en proyectos de software.

5 . Manejo explícito de recursos El manejo explícito del Pool fuera de la función map_reduce (creación, cierre, y join) asegura que los recursos se liberan adecuadamente, lo que es especialmente importante en aplicaciones que pueden correr por largos periodos de tiempo o que manejan muchos datos. Esto previene potenciales fugas de recursos, que podrían llevar a problemas de rendimiento o estabilidad.

```
from collections import defaultdict  # Importa la clase defaultdict
para crear diccionarios con valores predeterminados
import multiprocessing as mp  # Importa el módulo multiprocessing para
la programación concurrente
import sys  # Importa el módulo sys para manipular la configuración
del sistema
import time  # Importa el módulo time para medir el tiempo de
ejecución
from time import sleep  # Importa la función sleep del módulo time
para pausar la ejecución

def report_progress(map_returns, tag, callback):
    done = 0  # Inicializa el contador de tareas completadas
    num_jobs = len(map_returns)  # Obtiene el número total de tareas
    while num_jobs > done:  # Mientras haya tareas pendientes
        done = 0  # Reinicia el contador de tareas completadas
        for ret in map_returns:  # Itera sobre los resultados del
mapeo
            if ret.ready():  # Si la tarea ha finalizado
                done += 1  # Incrementa el contador de tareas
```

```
completadas
        sleep(0.5)  # Espera un corto periodo de tiempo antes de
volver a verificar
        if callback:  # Si se proporciona una función de callback
            callback(tag, done, num_jobs - done)  # Llama a la función
de callback con el progreso actual

def chunk0(my_list, chunk_size):
    for i in range(0, len(my_list), chunk_size):  # Itera sobre la
lista en incrementos del tamaño del fragmento
        yield my_list[i:i + chunk_size]  # Produce fragmentos de la
lista de tamaño especificado

def chunk(my_iter, chunk_size):
    chunk_list = []  # Inicializa una lista para contener los
elementos del fragmento
    for elem in my_iter:  # Itera sobre los elementos del iterable
        chunk_list.append(elem)  # Agrega el elemento al fragmento
        if len(chunk_list) == chunk_size:  # Si el fragmento alcanza
el tamaño especificado
            yield chunk_list  # Produce el fragmento
            chunk_list = []  # Reinicia el fragmento
    if len(chunk_list) > 0:  # Si quedan elementos en el fragmento
        yield chunk_list  # Produce el fragmento

def chunk_runner(fun, data):
    ret = []  # Inicializa una lista para almacenar los resultados
    for datum in data:  # Itera sobre los datos del fragmento
        ret.append(fun(datum))  # Ejecuta la función en cada dato y
agrega el resultado a la lista
    return ret  # Devuelve la lista de resultados

def chunked_async_map(pool, mapper, data, chunk_size):
    async_returns = []  # Inicializa una lista para almacenar los
resultados asincrónicos
    for data_part in chunk(data, chunk_size):  # Itera sobre los
fragmentos de datos
        async_returns.append(pool.apply_async(chunk_runner, (mapper,
data_part)))  # Aplica la función de manera asincrónica a cada
fragmento
    return async_returns  # Devuelve los resultados asincrónicos

def map_reduce(pool, my_input, mapper, reducer, chunk_size,
callback=None):
    map_returns = chunked_async_map(pool, mapper, my_input,
chunk_size)  # Realiza el mapeo en paralelo
    report_progress(map_returns, 'map', callback)  # Informa sobre el
progreso del mapeo
    map_results = []  # Inicializa una lista para almacenar los
resultados del mapeo
```

```
    for ret in map_returns:  # Itera sobre los resultados del mapeo
        map_results.extend(ret.get())  # Obtiene los resultados y los
agrega a la lista
    distributor = defaultdict(list)  # Crea un diccionario con valores
predeterminados como listas
    for key, value in map_results:  # Itera sobre los resultados del
mapeo
        distributor[key].append(value)  # Agrupa los valores por clave
    returns = chunked_async_map(pool, reducer, distributor.items(),
chunk_size)  # Realiza la reducción en paralelo
    report_progress(returns, 'reduce', callback)  # Informa sobre el
progreso de la reducción
    results = []  # Inicializa una lista para almacenar los resultados
finales
    for ret in returns:  # Itera sobre los resultados de la reducción
        results.extend(ret.get())  # Obtiene los resultados y los
agrega a la lista
    return results  # Devuelve los resultados finales

def emitter(word):
    return word, 1  # Emite cada palabra con un conteo inicial de 1

def counter(emitted):
    return emitted[0], sum(emitted[1])  # Suma los conteos de cada
palabra

def reporter(tag, done, not_done):
    print(f'Operacion {tag}: {done}/{done+not_done}')  # Imprime el
progreso de la operación

def run_map_reduce(words, chunk_size):
    pool = mp.Pool()  # Crea un grupo de procesos
    start_time = time.time()  # Obtiene el tiempo de inicio
    counts = map_reduce(pool, words, emitter, counter, chunk_size,
reporter)  # Ejecuta el proceso MapReduce
    pool.close()  # Cierra el grupo de procesos
    pool.join()  # Espera a que todos los procesos terminen
    end_time = time.time()  # Obtiene el tiempo de finalización
    duration = end_time - start_time  # Calcula la duración del
proceso
    return duration  # Devuelve la duración

if __name__ == '__main__':
    words = [word
             for word in map(lambda x: x.strip().rstrip(),
                             ' '.join(open('texto.txt', 'rt',
encoding='utf-8').readlines()).split(' '))
             if word != '' ]  # Lee las palabras de un archivo de
texto y las almacena en una lista
```

```
    chunk_sizes = [1, 10, 100, 1000, 10000]  # Tamaños de
fragmentación a probar
    results = []  # Inicializa una lista para almacenar los resultados

    for size in chunk_sizes:  # Itera sobre los tamaños de
fragmentación
        duration = run_map_reduce(words, size)  # Ejecuta el proceso
MapReduce con el tamaño de fragmentación actual
        results.append((size, duration))  # Agrega el tamaño de
fragmentación y la duración a la lista de resultados

    print("Tam fragmentacion | Duracion")  # Imprime la cabecera de la
tabla de resultados
    print("-" * 20)  # Imprime una línea divisoria
    for size, duration in results:  # Itera sobre los resultados
        print(f"{size:<10} | {duration:.2f} segundos")  # Imprime el
tamaño de fragmentación y la duración del proceso
```

Este código introduce la optimización y paralelización avanzadas en un proceso de MapReduce utilizando el módulo multiprocessing. La principal mejora radica en el manejo de los datos en trozos o chunks, lo que puede aumentar significativamente la eficiencia al procesar grandes volúmenes de datos.

Algunas especificaciones de funciones:

chunk0(my_list, chunk_size):

- Esta función toma una lista (my_list) y un tamaño de fragmento (chunk_size).
- Divide la lista en segmentos de tamaño especificado utilizando un enfoque basado en iteración sobre índices.
- Es una alternativa a la función chunk que se presenta posteriormente en el código. Ambas hacen básicamente lo mismo, pero con enfoques ligeramente diferentes.

chunk(my_iter, chunk_size):

- Esta función toma un iterable (my_iter) y un tamaño de fragmento (chunk_size).
- Itera sobre el iterable acumulando elementos en una lista hasta que alcanza el tamaño del fragmento especificado.
- Cuando la lista alcanza el tamaño del fragmento, la función produce el fragmento y reinicia la lista.
- Si al final del iterable quedan elementos que no forman un fragmento completo, también produce ese último fragmento.
- Esta función es más genérica que chunk0 ya que puede funcionar con cualquier tipo de iterable, no solo listas.

chunk_runner(fun, data):

- Esta función toma una función (fun) y una lista de datos (data).
- Itera sobre los datos y aplica la función a cada elemento, acumulando los resultados en una lista.

- Devuelve la lista de resultados.
- Es una función de utilidad utilizada para ejecutar una función sobre un segmento de datos.

chunked_async_map(pool, mapper, data, chunk_size):

- Esta función aplica un mapper de manera asincrónica a segmentos de datos.
- Toma un pool de procesos (pool), una función de mapeo (mapper), los datos de entrada (data) y el tamaño del fragmento (chunk_size).
- Divide los datos en fragmentos utilizando la función chunk.
- Para cada fragmento de datos, utiliza el pool de procesos para aplicar asincrónicamente la función de mapeo a través de apply_async.
- Devuelve una lista de objetos de resultado asincrónicos.

Estas funciones son parte de la implementación del proceso de MapReduce. chunk y chunk0 dividen los datos en fragmentos, chunk_runner ejecuta el mapeo en cada fragmento, y chunked_async_map aplica el mapeo de manera asincrónica en paralelo sobre los fragmentos de datos. Estos fragmentos mapeados se agrupan y reducen posteriormente en la función map_reduce.

El código proporcionado utiliza el módulo multiprocessing para implementar un proceso de MapReduce que maneja la entrada de datos en fragmentos o chunks, optimizando así el rendimiento al evitar el bloqueo del Global Interpreter Lock (GIL) en Python. Este enfoque permite una verdadera ejecución paralela en múltiples núcleos de CPU.

El código ejecuta una función de MapReduce sobre una serie de palabras obtenidas de un archivo y mide el tiempo que tarda el proceso para diferentes tamaños de fragmentos. La salida del código será una serie de líneas que muestran el tiempo de ejecución para cada tamaño de fragmento probado. Por ejemplo, podrías ver algo como esto:

```
Tam fragmentacion | Duracion
--------------------
1          | 5.20 segundos
10         | 2.30 segundos
100        | 1.15 segundos
1000       | 0.85 segundos
10000      | 0.75 segundos
```

Ejercicio 1: Modificar el número de procesos

Modifica el código para cambiar dinámicamente el número de procesos en el pool (mp.Pool()). Realiza pruebas con diferentes configuraciones (p. ej., 1, 2, 4, 8 procesos) y mide cómo afecta al tiempo de ejecución del proceso de MapReduce. Resultados esperados: Documentar cómo el incremento en el número de procesos afecta la eficiencia del procesamiento, identificando el punto de saturación donde más procesos no resultan en mejoras significativas.

Ejercicio 2: Comparación con threads

Implementa una versión del proceso de MapReduce usando concurrent.futures.ThreadPoolExecutor en lugar de multiprocessing.Pool y compara el

rendimiento con la versión actual que utiliza procesos. Resultados esperados: Observar las diferencias en rendimiento y cómo el GIL afecta la versión que utiliza hilos, especialmente cuando se incrementa el número de hilos.

Ejercicio 3: Análisis de granularidad de los datos

Experimenta con una gama más amplia de tamaños de fragmentos para entender mejor cómo la granularidad de los datos influye en el rendimiento del sistema. Considera extremos más variados y tamaños intermedios. Resultados esperados: Identificar un tamaño óptimo de fragmento que maximice la eficiencia del procesamiento, y explicar por qué ciertos tamaños resultan menos eficientes.

Ejercicio 4: Implementación de funciones de MapReduce más complejas

Crea funciones de mapeo y reducción más complejas que simulen cargas de trabajo más intensivas, como procesamiento de texto o cálculos matemáticos. Resultados esperados: Observar cómo el aumento de la complejidad de las tareas afecta el rendimiento y cómo se puede ajustar el tamaño del pool y de los fragmentos para optimizar el procesamiento.

```
## Tus respuestas
```

Parte 5

```
import marshal
import pickle
import socket
from time import sleep
# Codigo de Tiago Rodriguez

def my_funs():
    def mapper(v):
        return v, 1

    def reducer(my_args):
        v, obs = my_args
        return v, sum(obs)
    return mapper, reducer


def do_request(my_funs, data):
    conn = socket.create_connection(('127.0.0.1', 1936))
    conn.send(b'\x00')
    my_code = marshal.dumps(my_funs.__code__)
    conn.send(len(my_code).to_bytes(4, 'little', signed=False))
    conn.send(my_code)
    my_data = pickle.dumps(data)
    conn.send(len(my_data).to_bytes(4, 'little'))
    conn.send(my_data)
    job_id = int.from_bytes(conn.recv(4), 'little')
    conn.close()
```

```
    print(f'Obtener data desde job_id {job_id}')
    result = None
    while result is None:
        conn = socket.create_connection(('127.0.0.1', 1936))
        conn.send(b'\x01')
        conn.send(job_id.to_bytes(4, 'little'))
        result_size = int.from_bytes(conn.recv(4), 'little')
        result = pickle.loads(conn.recv(result_size))
        conn.close()
        sleep(1)
    print(f'Resultado es {result}')


if __name__ == '__main__':
    do_request(my_funs, 'Python rocks. Python es divertido'.split(' 
'))
```

- marshal: Se utiliza para serializar el objeto del código fuente de las funciones.
- pickle: Se utiliza para serializar los datos que se envían a través de los sockets.
- socket: Se utiliza para crear conexiones de red y enviar datos.
- sleep de time: Se utiliza para pausar la ejecución durante un segundo antes de volver a intentar obtener el resultado del trabajo.

```
def my_funs():
    def mapper(v):
        return v, 1

    def reducer(my_args):
        v, obs = my_args
        return v, sum(obs)
    return mapper, reducer
```

- Esta función define dos funciones internas: mapper y reducer.
- La función mapper toma un valor v y devuelve una tupla con v como la clave y 1 como el valor.
- La función reducer toma una tupla de argumentos (v, obs) y devuelve una tupla con v como la clave y la suma de los valores en obs.
- Ambas funciones son funciones simples de mapeo y reducción que serán utilizadas en el proceso de MapReduce.

```
def do_request(my_funs, data):
    conn = socket.create_connection(('127.0.0.1', 1936))
    conn.send(b'\x00')
    my_code = marshal.dumps(my_funs.__code__)
    conn.send(len(my_code).to_bytes(4, 'little', signed=False))
    conn.send(my_code)
    my_data = pickle.dumps(data)
    conn.send(len(my_data).to_bytes(4, 'little'))
```

```
    conn.send(my_data)
    job_id = int.from_bytes(conn.recv(4), 'little')
    conn.close()

    print(f'Obtener data desde job_id {job_id}')
    result = None
    while result is None:
        conn = socket.create_connection(('127.0.0.1', 1936))
        conn.send(b'\x01')
        conn.send(job_id.to_bytes(4, 'little'))
        result_size = int.from_bytes(conn.recv(4), 'little')
        result = pickle.loads(conn.recv(result_size))
        conn.close()
        sleep(1)
    print(f'Resultado es {result}')
```

- Esta función realiza la comunicación con un servidor remoto para enviar las funciones my_funs y los datos data para procesamiento.
- Se crea una conexión a un servidor en el puerto 1936 en localhost (127.0.0.1).
- Se envía un byte b'\x00' para indicar que se enviarán las funciones.
- Se serializa el objeto de código de las funciones my_funs utilizando marshal y se envía su longitud y contenido a través del socket.
- Los datos data se serializan usando pickle y se envían también con su longitud.
- Se recibe un ID de trabajo (job_id) del servidor, que se utilizará para recuperar el resultado más tarde.
- La función espera un segundo antes de intentar obtener el resultado.
- Se crea una nueva conexión para solicitar el resultado del trabajo.
- Se envía un byte b'\x01' para indicar que se solicita el resultado.
- Se envía el job_id y se espera a recibir el tamaño del resultado.
- Se recibe el resultado serializado y se deserializa usando pickle.
- El resultado se imprime cuando está disponible.

```
import asyncio
import marshal
import multiprocessing as mp
import pickle
from queue import Empty, Queue  # PriorityQueue
import threading
import types

import chunk_mp_mapreduce as mr


# multiprocessing.Queue

work_queue = Queue()
results_queue = Queue()
results = {}
```

```
async def submit_job(job_id, reader, writer):
    writer.write(job_id.to_bytes(4, 'little'))
    writer.close()
    code_size = int.from_bytes(await reader.read(4), 'little')
    my_code = marshal.loads(await reader.read(code_size))
    data_size = int.from_bytes(await reader.read(4), 'little')
    data = pickle.loads(await reader.read(data_size))
    work_queue.put_nowait((job_id, my_code, data))


def get_results_queue():
    while results_queue.qsize() > 0:
        try:
            job_id, data = results_queue.get_nowait()
            results[job_id] = data
        except Empty:
            return


async def get_results(reader, writer):
    get_results_queue()
    job_id = int.from_bytes(await reader.read(4), 'little')
    data = pickle.dumps(None)
    if job_id in results:
        data = pickle.dumps(results[job_id])
        del results[job_id]
    writer.write(len(data).to_bytes(4, 'little'))
    writer.write(data)


async def accept_requests(reader, writer, job_id=[0]):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(job_id[0], reader, writer)  #Errors in async
        job_id[0] += 1
    elif op[0] == 1:
        await get_results(reader, writer)


def worker():  # daemon
    pool = mp.Pool()
    while True:
        job_id, code, data = work_queue.get()  # blocking
        func = types.FunctionType(code, globals(),
'mapper_and_reducer')
        mapper, reducer = func()
        counts = mr.map_reduce(pool, data, mapper, reducer, 100,
mr.reporter)
        results_queue.put((job_id, counts))
```

```
    pool.close()
    pool.join()


async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1',
1936)
    worker_thread = threading.Thread(target=worker)   # Daemon
    worker_thread.start()
    async with server:
        await server.serve_forever()


asyncio.run(main())
```

Este código implementa un sistema de MapReduce distribuido utilizando asyncio y multiprocessing en Python. Puedes explicar el funcionamiento.

```
## Tu respuesta.

from collections import defaultdict
import marshal
import multiprocessing as mp
import sys
from time import sleep
import types

# Función para monitorear el progreso de tareas asincrónicas en el
pool de procesos
def report_progress(map_returns, tag, callback):
    done = 0
    num_jobs = len(map_returns)  # Cantidad total de tareas
    while num_jobs > done:
        done = 0  # Reinicia el contador de tareas completadas en cada
iteración
        for ret in map_returns:  # Revisa cada tarea en el conjunto de
resultados
            if ret.ready():  # Verifica si la tarea está completa
                done += 1
        sleep(0.5)  # Espera antes de la próxima revisión para reducir
carga del CPU
        if callback:  # Si existe un callback, reporta el progreso
            callback(tag, done, num_jobs - done)

# Función para dividir una lista en fragmentos de tamaño especificado
def chunk0(my_list, chunk_size):
    for i in range(0, len(my_list), chunk_size):
        yield my_list[i:i + chunk_size]

# Función para dividir cualquier iterable en fragmentos de tamaño
```

```
especificado
def chunk(my_iter, chunk_size):
    chunk_list = []
    for elem in my_iter:
        chunk_list.append(elem)
        if len(chunk_list) == chunk_size:
            yield chunk_list
            chunk_list = []
    if len(chunk_list) > 0:
        yield chunk_list

# Función que ejecuta una función sobre un conjunto de datos; la
función es pasada como código serializado
def chunk_runner(fun_marshal, data):
    # Deserializa el código de función y lo convierte en una función
ejecutable
    fun = types.FunctionType(marshal.loads(fun_marshal), globals(),
'fun')
    ret = []
    for datum in data:
        print(fun(datum))  # Opcional: imprimir cada resultado para
depuración
        ret.append(fun(datum))  # Aplica la función y almacena el
resultado
    return ret

# Función para aplicar map de manera asincrónica utilizando un pool de
procesos
def chunked_async_map(pool, mapper, data, chunk_size):
    async_returns = []
    for data_part in chunk(data, chunk_size):
        # Empaqueta cada función y datos y los envía al pool
        async_returns.append(pool.apply_async(chunk_runner,
(marshal.dumps(mapper.__code__), data_part)))
    return async_returns

# Función principal de MapReduce
def map_reduce(pool, my_input, mapper, reducer, chunk_size,
callback=None):
    # Mapeo: Aplica la función mapper a los datos de entrada
    map_returns = chunked_async_map(pool, mapper, my_input,
chunk_size)
    report_progress(map_returns, 'map', callback)
    map_results = []
    for ret in map_returns:
        map_results.extend(ret.get())  # Recopila todos los resultados
del mapeo

    # Agrupa los resultados del mapeo por clave
    distributor = defaultdict(list)
```

```
    for key, value in map_results:
        distributor[key].append(value)

    # Reducción: Aplica la función reducer a los resultados agrupados
    returns = chunked_async_map(pool, reducer, distributor.items(),
chunk_size)
    report_progress(returns, 'reduce', callback)
    results = []
    for ret in returns:
        results.extend(ret.get())  # Recopila todos los resultados de
la reducción
    return results

# Función para emitir cada palabra con un conteo inicial de 1
def emitter(word):
    return word, 1

# Función para sumar conteos de cada palabra
def counter(emitted):
    return emitted[0], sum(emitted[1])

# Función callback para imprimir el progreso de la operación
def reporter(tag, done, not_done):
    print(f'Operacion {tag}: {done}/{done+not_done}')

# Ejecución del código
words = 'Python es super, Python rocks'.split(' ')
pool = mp.Pool()  # Crea un pool de procesos
results = map_reduce(pool, words, emitter, counter, 10, reporter)
pool.close()  # Cierra el pool
pool.join()  # Espera que todos los procesos terminen

# Imprime resultados finales
for result in sorted(results, key=lambda x: x[1]):
    print(result)
```

El código proporcionado implementa una versión avanzada del patrón MapReduce utilizando multiprocessing, manejo de datos en segmentos (chunks), y serialización de funciones para permitir su ejecución en un entorno multiproceso.

Ejercicio 1: Sistema de análisis de datos distribuido

Implementa un sistema distribuido para analizar datos enviados a un servidor centralizado utilizando técnicas de MapReduce y serialización de funciones.

Descripción:

- Modifica el código para soportar la ejecución de diferentes análisis estadísticos (media, mediana, moda) que los clientes pueden solicitar dinámicamente.
- Implementa una funcionalidad en el servidor para recibir funciones de análisis como entrada, serializadas desde el cliente, ejecutarlas y devolver los resultados.

- Asegúrate de que el sistema pueda manejar múltiples solicitudes de análisis simultáneamente utilizando multiprocessing.

Ejercicio 2: Servicio de procesamiento de consultas en tiempo real

Crea un servicio que procese consultas en tiempo real sobre un conjunto de datos compartido, utilizando un modelo distribuido y asyncio para la gestión de consultas.

Descripción:

- Utiliza sockets para permitir que múltiples clientes envíen consultas que incluyan funciones de procesamiento de datos serializadas.
- Desarrolla un sistema de cola de trabajo donde las consultas se distribuyan entre varios procesadores utilizando multiprocessing.
- Implementa lógica para manejar consultas concurrentes y asegurar que los resultados se devuelvan de manera asincrónica a los clientes apropiados.

Ejercicio 3: Sistema de ejecución de tareas asincrónicas programables

Construye un sistema que permita a los usuarios definir tareas programables que se ejecuten en el servidor, utilizando una combinación de asyncio y multiprocessing.

Descripción:

- Configura un servidor que pueda recibir definiciones de tareas (funciones y datos) serializadas de los clientes.
- Las tareas deben ser planificadas y ejecutadas en paralelo utilizando un pool de procesos.
- Utiliza asyncio para gestionar la comunicación entre el cliente y el servidor, asegurando que las respuestas se envíen de vuelta a los clientes tan pronto como las tareas se completen.

Ejercicio 4: Sistema de monitoreo de red distribuido

Desarrolla un sistema para monitorear el estado de múltiples servidores en una red, recopilando datos de manera asincrónica y procesándolos en paralelo.

Descripción:

- Implementa un servidor central que recibe datos serializados de estado desde múltiples nodos cliente.
- Los datos pueden incluir métricas como CPU, memoria, uso de disco, etc., y deben ser procesados utilizando funciones MapReduce para generar informes de salud del sistema.
- Utiliza asyncio para manejar las solicitudes de los nodos de manera asincrónica y multiprocessing para el procesamiento en paralelo de los datos recogidos.

Ejercicio 5: Plataforma de pruebas A/B distribuida

Crea una plataforma que realice pruebas A/B en tiempo real sobre datos de usuario recogidos de diferentes ubicaciones, utilizando técnicas de procesamiento distribuido.

Descripción:

- Configura un sistema que reciba datos de experimentos A/B de clientes a través de la red, utilizando socket.
- Serializa lógica de análisis y envíala a un servidor que utilice multiprocessing para evaluar los resultados de las pruebas A/B en paralelo.
- Emplea asyncio para responder a las solicitudes de análisis, devolviendo resultados basados en el análisis de datos.

```
### Tus respuestas
```

Entregable

Presente este cuaderno desarrollado en tu repositorio personal.