NOMBRE DE LA ASIGNATURA:

COMPUTACIÓN PARALELA

NOMBRE DEL PROFESOR:

HEBERT SERGEI PEREZ SANTANA

NOMBRE DE LA ACTIVIDAD:

Examen Extraordinario

NOMBRE DEL ALUMNO:

LUQUIN CASTRO JESUS EDUARDO

REGISTRO:

20310467

FECHA DE ELABORACION: 01/06/2024

Objetivo del Ejercicio:

El objetivo de este ejercicio es realizar un análisis exhaustivo de las relaciones entre entidades utilizando técnicas avanzadas de computación paralela en Python. Se busca calcular estadísticas detalladas sobre las relaciones más frecuentes y la distribución de tipos de relaciones dentro del grafo de conocimiento de YAGO. Además, se pretende implementar consultas paralelizadas para optimizar el procesamiento de grandes volúmenes de datos semánticos, explorando así la eficiencia y escalabilidad de las técnicas de paralelización en el análisis de datos complejos y estructurados como los proporcionados por YAGO.

Descripción del Proyecto:

Elegí la problemática de y la computación paralela nos ayuda a resolver el problema en un tiempo razonable.

Configuración del Entorno:

Hardware Recomendado:

- CPU: Procesador multicore Intel Core i7 o superior) para ejecución en paralelo utilizando Multiprocessing, Threading u otras técnicas.
- GPU (opcional): Tarjeta gráfica compatible con CUDA u openCL

Sistema Operativo

- Windows 10

Gestión de Entorno:

- Python: Versión 3.6 o superior.

Bibliotecas y Frameworks:

plaintext

requirements.txt

Bibliotecas básicas

numpy

scipy

pandas

matplotlib

multiprocessing

Computación paralela y procesamiento distribuido

dask[complete]

joblib

tensorflow (si se va a utilizar con GPU, instalar tensorflow-gpu)

pytorch (si se va a utilizar)

Web scraping (ejemplo adicional)

beautifulsoup4

requests

Explicación de las Bibliotecas y Frameworks

- numpy: Para operaciones numéricas eficientes y manejo de arrays.
- scipy: Para herramientas y algoritmos científicos.
- pandas: Para manipulación y análisis de datos.
- matplotlib: Para visualización de datos.
- dask: Para computación paralela y procesamiento distribuido.
- joblib: Para ejecución paralela de tareas.
- tensorflow o pytorch: Frameworks de aprendizaje profundo (deep learning) con soporte para ejecución en GPU si se instala la versión tensorflow-gpu.
- beautifulsoup4 y requests: Para web scraping y manipulación de datos desde la web, como un ejemplo adicional de uso.

Instalación de Dependencias

Para instalar las dependencias desde el archivo requirements.txt, puedes ejecutar el siguiente comando en tu entorno virtual de Python:

bash

pip install -r requirements.txt

Consideraciones Adicionales

- Entorno Virtual: (venv o virtualenv) para aislar las dependencias del proyecto.

Implementación de tareas paralelas:

- **Multiprocesamiento:**

A diferencia del multiprocesamiento, los hilos comparten el mismo espacio de memoria, lo que puede ser beneficioso para tareas que requieren acceso a datos compartidos, aunque es necesario manejar adecuadamente la sincronización para evitar condiciones de carrera.

- **Multihilo:**

Primero, definimos las funciones para cada nivel del grafo que realizarán diferentes análisis sobre las ciudades. Aquí mostramos un ejemplo simplificado para dos niveles: demografía y economía.

- **GPU Computing:**

GPU Computing

Para implementar el proyecto utilizando una GPU, utilizaremos bibliotecas como TensorFlow con soporte para GPU. Aquí se mostrará cómo se puede lograr una aceleración significativa en comparación con una implementación basada solo en CPU.

- **Sincronización y Control de Concurrencia:**

Para implementar mecanismos de sincronización, podemos utilizar semáforos y bloqueos (locks) en Python para asegurarnos de que nuestra solución esté libre de condiciones de carrera (race conditions) y otros problemas de concurrencia.

- **Benchmarking y Análisis de Rendimiento:**

En este ejemplo, se utilizan locks para proteger el incremento de un contador global y semaphores para limitar el número de hilos que pueden acceder simultáneamente a un recurso. Esto ayuda a evitar condiciones de carrera y a asegurar la correcta sincronización entre los hilos.

- **Documentación:**

Documentación

Código Documentado

python

import time

import threading

import multiprocessing

```
import psutil

# Función para calcular la demografía de una ciudad
def calculate_demographics(city):
    time.sleep(1)
    return f"Demografía de {city}"

# Función para analizar la economía de una ciudad
def analyze_economics(city):
    time.sleep(2)
    return f"Análisis económico de {city}"

# Ejecución secuencial
def sequential_execution(cities):
    results_demographics = [calculate_demographics(city) for city in cities]
    results_economics = [analyze_economics(city) for city in cities]
    return results_demographics, results_economics

# Ejecución paralela con hilos
def parallel_execution_threads(cities):
    results_demographics = [None] * len(cities)
    results_economics = [None] * len(cities)
```

```
    threads_demographics = []
    for i, city in enumerate(cities):
        thread = threading.Thread(target=calculate_demographics, args=(city, results_demographics, i))
        threads_demographics.append(thread)
        thread.start()

    threads_economics = []
    for i, city in enumerate(cities):
        thread = threading.Thread(target=analyze_economics, args=(city, results_economics, i))
        threads_economics.append(thread)
        thread.start()

    for thread in threads_demographics:
        thread.join()
    for thread in threads_economics:
        thread.join()

    return results_demographics, results_economics

# Ejecución paralela con multiprocesamiento
def parallel_execution_multiprocessing(cities):
```

```
    with multiprocessing.Pool() as pool:
        results_demographics = pool.map(calculate_demographics, cities)
        results_economics = pool.map(analyze_economics, cities)
    return results_demographics, results_economics


# Medir tiempo de ejecución
def measure_execution_time(func, *args):
    start_time = time.time()
    results = func(*args)
    end_time = time.time()
    execution_time = end_time - start_time
    return execution_time, results


# Benchmark de tiempo de ejecución
def benchmark():
    cities = ['Guadalajara', 'Ciudad de México', 'Monterrey', 'Cancún', 'Mérida']

    seq_time, seq_results = measure_execution_time(sequential_execution, cities)
    print(f"Tiempo de ejecución secuencial: {seq_time} segundos")

    thr_time, thr_results = measure_execution_time(parallel_execution_threads, cities)
    print(f"Tiempo de ejecución con hilos: {thr_time} segundos")
```

```
    mp_time, mp_results = measure_execution_time(parallel_execution_multiprocessing,
cities)
    print(f"Tiempo de ejecución con multiprocesamiento: {mp_time} segundos")


# Medir uso de recursos
def measure_resource_usage(func, *args):
    process = psutil.Process()
    start_time = time.time()
    results = func(*args)
    end_time = time.time()
    execution_time = end_time - start_time
    cpu_usage = process.cpu_percent(interval=1)
    memory_usage = process.memory_info().rss / (1024 * 1024)  # Convertir a MB
    return execution_time, cpu_usage, memory_usage, results


# Benchmark de uso de recursos
def benchmark_resources():
    cities = ['Guadalajara', 'Ciudad de México', 'Monterrey', 'Cancún', 'Mérida']


    print("Benchmark de Recursos para Ejecución Secuencial:")
    seq_time, seq_cpu, seq_memory, seq_results =
measure_resource_usage(sequential_execution, cities)
    print(f"Tiempo de ejecución: {seq_time} segundos")
```

```
    print(f"Uso de CPU: {seq_cpu}%")
    print(f"Uso de Memoria: {seq_memory} MB\n")

    print("Benchmark de Recursos para Ejecución con Hilos:")
    thr_time, thr_cpu, thr_memory, thr_results = measure_resource_usage(parallel_execution_threads, cities)
    print(f"Tiempo de ejecución: {thr_time} segundos")
    print(f"Uso de CPU: {thr_cpu}%")
    print(f"Uso de Memoria: {thr_memory} MB\n")

    print("Benchmark de Recursos para Ejecución con Multiprocesamiento:")
    mp_time, mp_cpu, mp_memory, mp_results = measure_resource_usage(parallel_execution_multiprocessing, cities)
    print(f"Tiempo de ejecución: {mp_time} segundos")
    print(f"Uso de CPU: {mp_cpu}%")
    print(f"Uso de Memoria: {mp_memory} MB\n")

if _name_ == "_main_":
    benchmark()
    benchmark_resources()
```

Archivo README.md

markdown

```
# Análisis Paralelo del Grafo de Ciudades Mexicanas
```

Este proyecto implementa y compara soluciones secuenciales y paralelas (usando hilos y multiprocesamiento) para analizar un grafo de 5 ciudades mexicanas en 7 niveles.

- Bibliotecas: numpy, scipy, pandas, matplotlib, dask[complete], joblib, tensorflow (tensorflow-gpu opcional), pytorch, beautifulsoup4, requests, psutil

Instala las dependencias con:

```bash
pip install -r requirements.txt
```

Ejecución

Para ejecutar el benchmark de tiempo de ejecución:

```bash
python benchmark.py
```

Para ejecutar el benchmark de uso de recursos:

```bash
python benchmark_resources.py
```

Estructura del Proyecto

benchmark.py: Contiene la implementación del benchmark de tiempo de ejecución.

benchmark_resources.py: Contiene la implementación del benchmark de uso de recursos.

requirements.txt: Lista de bibliotecas necesarias para el proyecto.

Resultados

Los resultados del análisis incluyen el tiempo de ejecución y el uso de recursos (CPU y memoria) para cada enfoque (secuencial, hilos y multiprocesamiento).