Taller Práctico 1: Procesamiento paralelo y el problema del viajero

Por: Jose Santiago Gonzalez y Oscar Mateo Arrubla

OBJETIVO DEL LABORATORIO

El objetivo de este laboratorio es implementar y comparar la eficiencia de un algoritmo secuencial y uno paralelo (utilizando múltiples procesos) para resolver el Problema del Viajante (Travelling Salesperson Problem, TSP) mediante el método de fuerza bruta. El resultado de este laboratorio será un informe que demuestre el speedup (aceleración) logrado con el paralelismo.

MARCO TEÓRICO

El problema del viajante (TSP) se basa en buscar una ruta donde se puedan visitar todas las ciudades solamente una vez y luego se vuelva al origen. Esto significa que es un problema NP-hard, lo cual significa que su tiempo de cómputo crece factorialmente con el número de ciudades. Por otro lado el método de fuerza bruta que vamos a usar para resolver este problema consiste en probar todas las permutaciones posibles en este caso de las rutas del viajero y elegir la más corta, esto en principio podría parecer un problema porque como ya mencionamos antes este problema crece exponencialmente mientras más ciudades halla por lo que tendremos que buscar un método que nos permita evaluar varias permutaciones en simultáneo. Ahí es donde entra en juego el procesamiento paralelo ya que nos permite evaluar cada permutación de forma independiente.

METODOLOGÍA

Representación de las ciudades:

```
ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
```

Hardware:

- CPU: 2 vCPU
- RAM 12 GB
- Entorno: Google Colab

Software:

- Lenguaje usado: Python 3.x
- Librerías usadas: itertools, math, multiprocessing, time.

Algoritmos desarrollados:

Algoritmo secuencial

Primero se representan las ciudades en una lista de coordenadas

```
ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
```

Luego se generan las rutas posibles mediante el uso de la función itertools.permutations de la librería itertools la cual nos permite ver todas las posibles permutaciones de una lista o secuencia, que en este caso es la lista de las ciudades, siendo así que podamos saber todas las posibles órdenes en las que se pueden visitar las ciudades

```
for ruta in itertools.permutations(range(n)):
```

Una vez tenemos todas las rutas posibles falta conocer las distancias de todas y elegir la más corta, primero determinamos la distancia de todas las permutaciones

```
def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])  # retorno al origen
    return total
```

Esto se hace usando la fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Además de añadir la distancia de vuelta al origen como se menciona en el problema.

Una vez se tiene esta información sobre las distancias de cada permutación se debe de elegir la distancia más corta:

```
def tsp_secuencial(ciudades):
    n = len(ciudades)
    mejor_ruta = None
    menor_distancia = float("inf")
    for ruta in itertools.permutations(range(n)):
        d = distancia_total(ruta, ciudades)
        if d < menor_distancia:
            menor_distancia = d
            mejor_ruta = ruta
    return mejor_ruta, menor_distancia
```

Esto se logra al revisar todas las rutas posibles, luego calcula la distancia total de cada una y compara cada una con la menor distancia encontrada hasta el momento hasta que acaba de revisar todas las permutaciones, devolviendo así la mejor ruta y la menor distancia.

Además se toma el tiempo que tarda el algoritmo en realizar todas estas operaciones

```
if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
    inicio = time.time()
    ruta, dist = tsp_secuencial(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

Algoritmo paralelo

En este algoritmo se usa de igual forma la lista de ciudades en coordenadas

```
ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
```

Así mismo se hace el mismo procedimiento para obtener todas las combinaciones posibles de las ciudades

```
rutas = list(itertools.permutations(range(len(ciudades))))
```

El mayor cambio viene cuando usando la librería multiprocessing se crea un Pool de procesos y cada proceso recibe una de las partes de la ruta y calcula su distancia final

```
def tsp_paralelo(ciudades):
    rutas = list(itertools.permutations(range(len(ciudades))))
    with Pool(cpu_count()) as p:
        resultados = p.map(evaluar_ruta, [(ruta, ciudades) for ruta in rutas])
    mejor = min(resultados, key=lambda x: x[0])
    return mejor[1], mejor[0]
```

Mediante la función “Pool(cpu_count()” se genera un grupo de procesos igual al número de núcleos disponibles para así poder repartir el trabajo entre ellos, seguido de esto se asigna a cada proceso unos datos diferentes para que ejecute la función evaluar ruta usando una de las posibles combinaciones de ruta con la que se pueden recorrer las ciudades. Una vez hechos los cálculos en paralelo se elige el resultado con menor distancia y la mejor ruta.

RESULTADOS

Para comprobar en diferentes escenarios cada uno de los algoritmos decidimos ir aumentando el número de ciudades para ver cómo se comporta cada algoritmo empezando en 5 ciudades y aumentando hasta 7

5 ciudades:

Algoritmo secuencial:

```python
import itertools
import math
import time

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])  # retorno al origen
    return total

def tsp_secuencial(ciudades):
    n = len(ciudades)
    mejor_ruta = None
    menor_distancia = float("inf")
    for ruta in itertools.permutations(range(n)):
        d = distancia_total(ruta, ciudades)
        if d < menor_distancia:
            menor_distancia = d
            mejor_ruta = ruta
    return mejor_ruta, menor_distancia

# Ejemplo de prueba
if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
    inicio = time.time()
    ruta, dist = tsp_secuencial(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (0, 2, 3, 1, 4)
Distancia mínima: 20.448909199308726
Tiempo (s): 0.00029850006103515625
```

Algoritmo paralelo:

```python
import itertools
import math
import time
from multiprocessing import Pool, cpu_count

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])
    return total

def evaluar_ruta(ruta_ciudades):
    ruta, ciudades = ruta_ciudades
    return distancia_total(ruta, ciudades), ruta

def tsp_paralelo(ciudades):
    rutas = list(itertools.permutations(range(len(ciudades))))
    with Pool(cpu_count()) as p:
        resultados = p.map(evaluar_ruta, [(ruta, ciudades) for ruta in rutas])
    mejor = min(resultados, key=lambda x: x[0])
    return mejor[1], mejor[0]

if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3)]
    inicio = time.time()
    ruta, dist = tsp_paralelo(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (0, 2, 3, 1, 4)
Distancia mínima: 20.448909199308726
Tiempo (s): 0.018980741500854492
```

6 ciudades:

Algoritmo secuencial:

```python
import itertools
import math
import time

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])  # retorno al origen
    return total

def tsp_secuencial(ciudades):
    n = len(ciudades)
    mejor_ruta = None
    menor_distancia = float("inf")
    for ruta in itertools.permutations(range(n)):
        d = distancia_total(ruta, ciudades)
        if d < menor_distancia:
            menor_distancia = d
            mejor_ruta = ruta
    return mejor_ruta, menor_distancia

# Ejemplo de prueba
if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3), (4,2)]
    inicio = time.time()
    ruta, dist = tsp_secuencial(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (0, 5, 2, 3, 1, 4)
Distancia mínima: 20.5358803471738
Tiempo (s): 0.0030133724212646484
```

Algoritmo paralelo:

```python
import itertools
import math
import time
from multiprocessing import Pool, cpu_count

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])
    return total

def evaluar_ruta(ruta_ciudades):
    ruta, ciudades = ruta_ciudades
    return distancia_total(ruta, ciudades), ruta

def tsp_paralelo(ciudades):
    rutas = list(itertools.permutations(range(len(ciudades))))
    with Pool(cpu_count()) as p:
        resultados = p.map(evaluar_ruta, [(ruta, ciudades) for ruta in rutas])
    mejor = min(resultados, key=lambda x: x[0])
    return mejor[1], mejor[0]

if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3), (4,2)]
    inicio = time.time()
    ruta, dist = tsp_paralelo(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (0, 5, 2, 3, 1, 4)
Distancia mínima: 20.5358803471738
Tiempo (s): 0.020128726959228516
```

7 ciudades:

Algoritmo secuencial:

```python
import itertools
import math
import time

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])  # retorno al origen
    return total

def tsp_secuencial(ciudades):
    n = len(ciudades)
    mejor_ruta = None
    menor_distancia = float("inf")
    for ruta in itertools.permutations(range(n)):
        d = distancia_total(ruta, ciudades)
        if d < menor_distancia:
            menor_distancia = d
            mejor_ruta = ruta
    return mejor_ruta, menor_distancia

# Ejemplo de prueba
if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3), (4,2), (9,1)]
    inicio = time.time()
    ruta, dist = tsp_secuencial(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (2, 6, 3, 1, 4, 0, 5)
Distancia mínima: 26.3668322420191
Tiempo (s): 0.008276939392089844
```

Algoritmo paralelo:

```python
import itertools
import math
import time
from multiprocessing import Pool, cpu_count

def distancia(ciudad1, ciudad2):
    return math.sqrt((ciudad1[0] - ciudad2[0])**2 + (ciudad1[1] - ciudad2[1])**2)

def distancia_total(ruta, ciudades):
    total = 0
    for i in range(len(ruta) - 1):
        total += distancia(ciudades[ruta[i]], ciudades[ruta[i + 1]])
    total += distancia(ciudades[ruta[-1]], ciudades[ruta[0]])
    return total

def evaluar_ruta(ruta_ciudades):
    ruta, ciudades = ruta_ciudades
    return distancia_total(ruta, ciudades), ruta

def tsp_paralelo(ciudades):
    rutas = list(itertools.permutations(range(len(ciudades))))
    with Pool(cpu_count()) as p:
        resultados = p.map(evaluar_ruta, [(ruta, ciudades) for ruta in rutas])
    mejor = min(resultados, key=lambda x: x[0])
    return mejor[1], mejor[0]

if __name__ == "__main__":
    ciudades = [(0,0), (1,5), (5,2), (6,6), (2,3), (4,2), (9,1)]
    inicio = time.time()
    ruta, dist = tsp_paralelo(ciudades)
    fin = time.time()
    print("Ruta óptima:", ruta)
    print("Distancia mínima:", dist)
    print("Tiempo (s):", fin - inicio)
```

```
Ruta óptima: (2, 6, 3, 1, 4, 0, 5)
Distancia mínima: 26.3668322420191
Tiempo (s): 0.037323713302612305
```

ANÁLISIS DE RENDIMIENTO

Como vimos en el apartado de resultados, el ganador claramente es el algoritmo secuencial, pero ¿por qué? Primero debemos entender que el algoritmo paralelo debería de ser más eficiente que el secuencial, esto debido a que el tiempo de ejecución y el trabajo se dividen equitativamente entre P nucleos, por lo que en teoría el tiempo de este algoritmo sería:

$$T_{paralelo} = \frac{T_{secuencial}}{P}$$

y el speedup (cuanto tiempo se ahorra al usar varios núcleos) sería:

$$S = \frac{T_{secuencial}}{T_{paralelo}} = P$$

En este caso el speedup ideal sería S=P, por lo que usando los 4 núcleos que nos permite colab, se vería una velocidad 4 veces más rápida, pero claramente eso no sucede. En la práctica hay sobrecarga y limitaciones físicas que hacen que el paralelismo no escale de manera correcta, por ejemplo cada vez que se usa “Pool(cpu_count())” python crea varios procesos, luego asigna información a cada uno y recoge su resultado, todo este proceso cuando se tratan problemas pequeños hace que el costo de crear procesos y pasar datos a estos sea mayor que el tiempo que se ahorra.

CONCLUSIÓN