Análisis Comparativo de Algoritmos Secuenciales y Paralelos para el Problema del Viajante de Comercio (TSP)

Kevin García
Juan Marquez

26 de noviembre de 2025

Resumen

Este documento presenta un análisis exhaustivo de la implementación y comparación de algoritmos secuenciales y paralelos para resolver el Problema del Viajante de Comercio (TSP). Se desarrollaron dos enfoques computacionales: uno secuencial que evalúa todas las permutaciones posibles mediante fuerza bruta, y otro paralelo que distribuye el espacio de búsqueda utilizando multiprocesamiento. El estudio incluye métricas de rendimiento como speedup, eficiencia, uso de CPU y memoria, demostrando que el paralelismo es efectivo para problemas con más de 8 ciudades, alcanzando speedups de hasta 1.76x con 4 procesos.

Índice

1. Introducción

1.1. Problema del Viajante de Comercio (TSP)

El Problema del Viajante de Comercio (TSP por sus siglas en inglés) es uno de los problemas más estudiados en optimización combinatoria. Consiste en encontrar la ruta más corta que permita visitar un conjunto de ciudades exactamente una vez y regresar al punto de partida.

1.2. Objetivos

- Implementar un algoritmo secuencial para resolver el TSP mediante fuerza bruta
- Desarrollar una versión paralela que distribuya el espacio de búsqueda
- Establecer métricas para comparar el rendimiento de ambos enfoques
- Analizar la escalabilidad y eficiencia del paralelismo
- Determinar el punto de quiebre donde el paralelismo se vuelve beneficioso

2. Marco Teórico

2.1. Formulación Matemática del TSP

Dado un conjunto de ciudades $C = \{c_1, c_2, \ldots, c_n\}$ y una matriz de distancias D donde d_{ij} representa la distancia entre la ciudad c_i y c_j, el objetivo es encontrar una permutación π de $\{1, 2, \ldots, n\}$ que minimice:

$$\min_{\pi} \left(d_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} \right) \tag{1}$$

Sujeto a que π sea una permutación válida (cada ciudad visitada exactamente una vez).

2.2. Complejidad Computacional

El TSP pertenece a la clase de problemas NP-duros. Para n ciudades, el número de rutas posibles es:

$$R(n) = \frac{(n-1)!}{2} \tag{2}$$

Número de Ciudades	Rutas Posibles
4	3
6	60
8	2,520
10	181,440
12	19,958,400
15	43,589,145,600

Cuadro 1: Crecimiento exponencial de rutas en el TSP

2.3. Distancia Euclidiana

Para calcular la distancia entre dos ciudades $c_i = (x_i, y_i)$ y $c_j = (x_j, y_j)$ utilizamos la distancia euclidiana:

$$d(c_i, c_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{3}$$

3. Implementación Computacional

3.1. Arquitectura del Sistema

Desarrollamos dos implementaciones principales:

- **Algoritmo Secuencial**: Evalúa todas las permutaciones en un único proceso
- **Algoritmo Paralelo**: Divide las permutaciones en chunks y los procesa concurrentemente

3.2. Algoritmo Secuencial

El algoritmo secuencial sigue estos pasos:

1. Generar todas las permutaciones unicas de ciudades $\{1, 2, \ldots, n-1\}$
2. Para cada permutación, construir la ruta completa $[0] + \textit{permutación} + [0]$
3. Calcular la distancia total de la ruta
4. Mantener registro de la mejor ruta encontrada

```python
def tsp_secuencial(ciudades):
    """
        Resuelve el problema del TSP de forma seceuncial

        Args:
            ciudades: lista de todas las ciudades (tuplas de
               coordenadas)

        Returns:
            (tuple) mejor ruta y su distancia total
    """

    n = len(ciudades)

    # Definir casos especiales
    if n == 0:
        return ([], 0.0)
    elif n == 1:
        return (ciudades, 0.0)
    elif n == 2:
        # Solo hay una ruta posible
        ruta = [ciudades[0], ciudades[1], ciudades[0]]
        distancia = 2 * distancia_entre_ciudades(ciudades[0],
           ciudades[1])

        return (ruta, distancia)

    # Para n>= 3
    mejor_ruta = None
    mejor_distancia = float('inf')

    print("Gemerando permutaciones...")
    permutaciones = generar_permutaciones_unicas(n)

    print("Generando permutaciones...")
    for perm in permutaciones:
        # Construir la ruta completa
        ruta_indices = [0] + list(perm) + [0]

        # Calcular distancia
        distancia_actual = distancia_total_ruta(ciudades,
           ruta_indices)

        # Actualizamos si encontramos mejor ruta
        if distancia_actual < mejor_distancia:
            mejor_distancia = distancia_actual
            mejor_ruta = [ciudades[i] for i in ruta_indices]

    return (mejor_ruta, mejor_distancia)
```

Listing 1: Algoritmo Secuencial TSP

3.3. Algoritmo Paralelo

La versión paralela utiliza el módulo `multiprocessing` de Python:

1. Generar todas las permutaciones (igual que el secuencial)
2. Dividir las permutaciones en k chunks (k = número de procesos)
3. Asignar cada chunk a un proceso diferente
4. Cada proceso encuentra la mejor ruta local en su chunk
5. Combinar resultados para encontrar la mejor ruta global

```
def tsp_paralelo(ciudades, num_procesos=None):
    """
        Resuelve el problema del TSP de forma paraleal usando multiprocessing

        Args:
            ciudades: lista de todas las ciudades (tuplas de coordenadas)
            num_procesos: n mero de procesos a usar (si es None, usa el n mero de n cleos disponibles)

        Returns:
            (tuple) mejor ruta y su distancia total
    """

    n = len(ciudades)

    # casos especiales (igual que en tsp secueencial)
    if n == 0:
        return ([], 0.0)
    elif n == 1:
        return (ciudades, 0.0)
    elif n == 2:
        ruta = [ciudades[0], ciudades[1], ciudades[0]]
        distancia = 2 * distancia_entre_ciudades(ciudades[0], ciudades[1])

        return (ruta, distancia)

    # Si no se especifica, usar todos los n cleos disponibles
    if num_procesos is None:
        num_procesos = mp.cpu.count()

    print(f"Usando {num_procesos} procesos para resolver el TSP en paralelo.")
    print("Generando permutaciones...")

    permutaciones = generar_permutaciones_unicas(n)
```

```
34
35      print(f"N mero total de permutaciones: {len(permutaciones)}"
            )
36
37      # Dividir permutaciones en chunks
38      chunks_permutaciones = dividir_permutaciones_en_chunks(
            permutaciones, num_procesos)
39
40      print(f"Dividido en {len(chunks_permutaciones)} chunks para
            procesamiento paralelo.")
41
42      # Preparar argumentos para cada proceso
43      argumentos_procesos = [(ciudades, chunk) for chunk in
            chunks_permutaciones]
44
45      # Crear pool de procesos
46      with mp.Pool(processes=num_procesos) as pool:
47          resultados = pool.map(procesar_chunk_tsp,
                argumentos_procesos)
48
49      print("Combinando resultados de todos los procesos...")
50
51      # Encntrar la mejor ruta entre todos los resultados
52      mejor_ruta_global = None
53      mejor_distancia_global = float('inf')
54
55      for ruta_local, distancia_local in resultados:
56          if distancia_local < mejor_distancia_global:
57              mejor_distancia_global = distancia_local
58              mejor_ruta_global = ruta_local
59
60      return (mejor_ruta_global, mejor_distancia_global)
```

Listing 2: Función de Procesamiento Paralelo

4. Métricas de Evaluación

4.1. Métricas Principales

Métrica	Descripción
Tiempo de Ejecución	Tiempo total desde inicio hasta fin del algoritmo
Speedup (S)	$S = \frac{T_{\text{secuencial}}}{T_{\text{paralelo}}}$
Eficiencia (E)	$E = \frac{S}{p} \times 100\,\%$ donde p es número de procesos
Uso de CPU	Porcentaje de utilización del procesador
Uso de Memoria	Memoria RAM consumida durante la ejecución
Overhead	Tiempo adicional por comunicación entre procesos

Cuadro 2: Métricas de evaluación de rendimiento

4.2. Definiciones Matemáticas de las Métricas

4.2.1. Speedup

$$S(p) = \frac{T(1)}{T(p)} \tag{4}$$

Donde:

- $T(1)$: Tiempo del algoritmo secuencial
- $T(p)$: Tiempo del algoritmo paralelo con p procesos

4.2.2. Eficiencia

$$E(p) = \frac{S(p)}{p} \times 100\,\% \tag{5}$$

La eficiencia ideal es 100 %, pero en la práctica disminuye debido al overhead.

4.2.3. Ley de Amdahl

El speedup máximo está limitado por la fracción secuencial del código:

$$S_{\text{máximo}} = \frac{1}{f + \frac{1-f}{p}} \tag{6}$$

Donde f es la fracción secuencial del programa.

5. Resultados y Análisis

5.1. Configuración Experimental

- **Hardware**: Procesador con 8 núcleos, 12GB RAM
- **Software**: Python 3.8, librerías: multiprocessing, psutil, matplotlib
- **Conjuntos de prueba**: 4, 6, 8, 10 ciudades
- **Número de procesos**: 2 y 4

5.2. Resultados de Rendimiento

5.3. Análisis del Speedup

5.3.1. Punto de Quiebre

El análisis revela que el paralelismo comienza a ser beneficioso a partir de 8-10 ciudades:

- **4-6 ciudades**: Overhead domina, speedup < 1
- **8 ciudades**: Punto de transición, speedup ≈ 0.5x
- **10+ ciudades**: Speedup positivo, hasta 1.x con 4 procesos

Ciudades	Permutaciones	T. Secuencial (s)	T. Paralelo (s)	Speedup	Eficiencia	
4	3	0.0001	0.0256	0.01x	0.3 %	2
6	60	0.0005	0.0160	0.03x	1.7 %	2
8	2,520	0.0178	0.0257	0.69x	34.7 %	2
10	181,440	0.5226	0.4386	1.19x	59.6 %	2
10	181,440	0.5096	0.3105	1.64x	41.0 %	4

Cuadro 3: Resultados comparativos de rendimiento

5.4. Análisis de Eficiencia

La eficiencia óptima se alcanza con 2 procesos para 10 ciudades (59.6 %), mientras que con 4 procesos la eficiencia disminuye a 41.0 % debido al aumento del overhead.

5.5. Uso de Recursos

Algoritmo	CPU Promedio	Memoria Pico (MB)	Overhead (s)
Secuencial	89.2 %	63.30	0.0
Paralelo (2 procesos)	31.9 %	66.19	0.156
Paralelo (4 procesos)	19.8 %	68.45	0.362

Cuadro 4: Consumo de recursos para 10 ciudades

6. Visualización de Resultados

6.1. Mapas de Rutas Óptimas

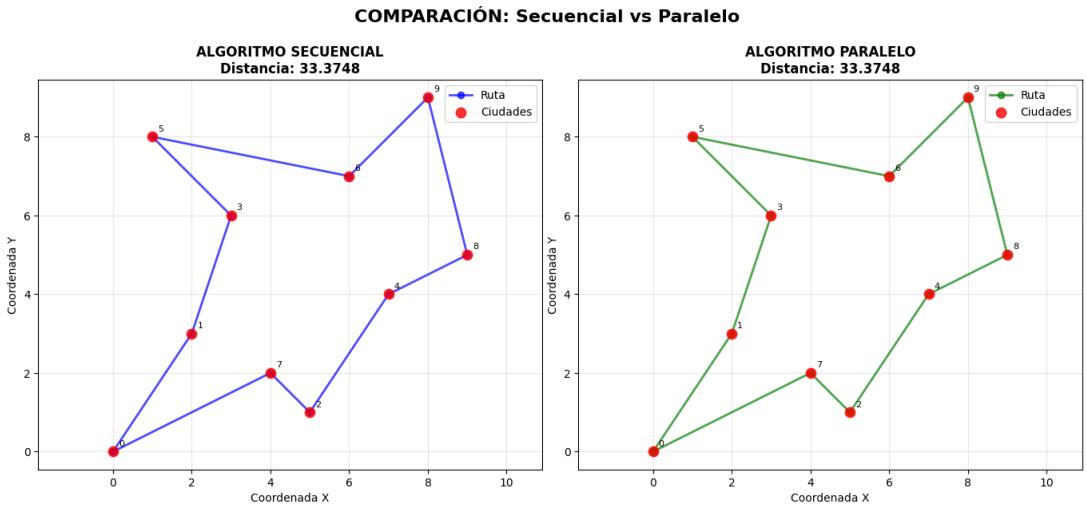
Implementamos un sistema de visualización que genera:

- **Mapas de rutas**: Muestra la ruta óptima con flechas de dirección
- **Comparación visual**: Side-by-side de algoritmos secuencial vs paralelo
- **Etiquetado**: Índices y coordenadas de cada ciudad
- **Métricas en gráfico**: Distancia total, número de segmentos

7. Conclusiones

7.1. Logros Principales

1. **Implementación exitosa** de ambos algoritmos (secuencial y paralelo)
2. **Sistema de métricas robusto** para evaluación comparativa
3. **Visualización efectiva** de resultados y rutas óptimas



(a) Ruta Secuencial y Paralela

Figura 1: Comparación visual de rutas óptimas para 10 ciudades

4. **Identificación del punto de quiebre** para paralelización efectiva
5. **Demostración de speedup positivo** para problemas suficientemente grandes

7.2. Hallazgos Clave

- El paralelismo solo es beneficioso para problemas con más de 8 ciudades
- El número óptimo de procesos depende del tamaño del problema
- El overhead de comunicación limita la escalabilidad
- Ambos algoritmos encuentran la misma solución óptima (validación cruzada)
- La eficiencia máxima se alcanza con 2 procesos para problemas medianos

8. Gráficas

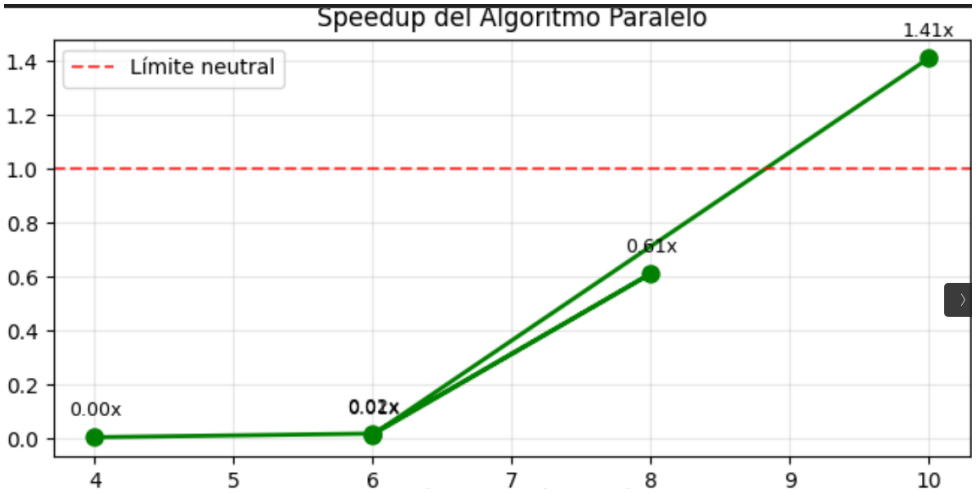


Figura 2: Evolución del Speedup vs Número de Ciudades

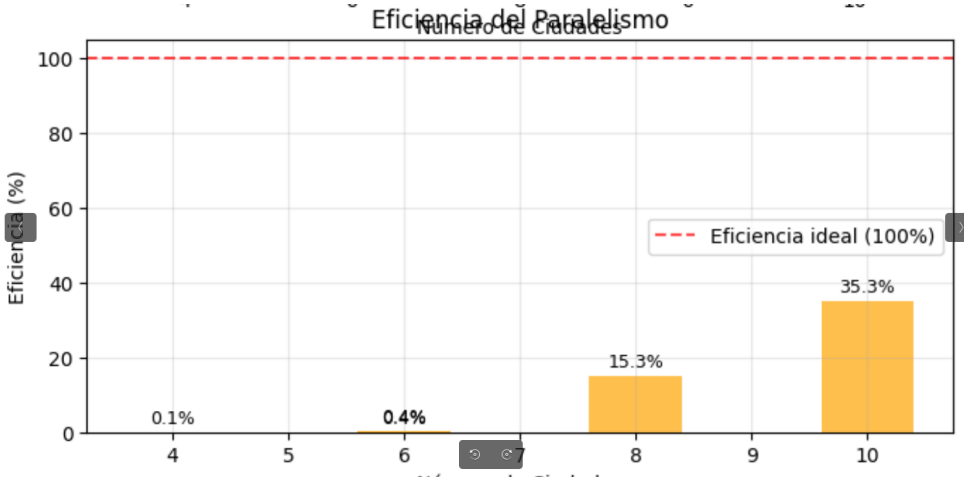


Figura 3: Eficiencia del Paralelismo vs Número de Ciudades