Informe de Laboratorio: Optimización de Rutas del Viajero con Clusters de Servicios

Taller Práctico 4 - Computación de Alto Rendimiento

hecho por: Johan Garcia y Cesar mora
Institución: Universidad Sergio Arboleda
Fecha: Noviembre 28, 2025
Profesor: Guillermo Andrés De Mendoza Corrales

Resumen Ejecutivo

Este trabajo implementa una solución distribuida para el Problema del Viajante (TSP) utilizando una arquitectura de microservicios con Docker Compose. Se desarrolló una API RESTful en Flask desplegada en múltiples réplicas y un cliente de fuerza bruta que explota el paralelismo para encontrar la ruta óptima. Los resultados demuestran mejoras significativas en el rendimiento al utilizar procesamiento paralelo comparado con el enfoque secuencial.

---

1. Introducción

1.1 Contexto del Problema

El Problema del Viajante (Traveling Salesman Problem - TSP) es un problema clásico de optimización combinatoria donde se debe encontrar la ruta más corta que visite un conjunto de ciudades exactamente una vez y regrese al punto de origen. La complejidad computacional del TSP es O(n!), lo que lo convierte en un problema NP-difícil.

1.2 Objetivos del Laboratorio

Objetivo General: Diseñar y construir una solución distribuida y escalable para el TSP aplicando arquitecturas modernas de microservicios.

Objetivos Específicos:

1. Desarrollar una API RESTful con Flask para cálculo de distancias
2. Contenerizar la aplicación usando Docker
3. Desplegar un servicio escalable con múltiples réplicas usando Docker Compose
4. Implementar un cliente de fuerza bruta con capacidades de procesamiento paralelo
5. Comparar el rendimiento entre implementaciones secuenciales y paralelas

1.3 Relevancia

Este proyecto permite comprender conceptos fundamentales de computación distribuida:

- Arquitecturas de microservicios
- Escalabilidad horizontal
- Balanceo de carga
- Paralelización de algoritmos
- Contenerización de aplicaciones

---

2. Marco Teórico

2.1 Problema del Viajante (TSP)

El TSP puede formularse matemáticamente como:

Minimizar: Σ d(ci, ci+1) para i = 1 hasta n

Donde:

- n = número de ciudades
- d(ci, ci+1) = distancia entre ciudad i y ciudad i+1
- La permutación debe visitar todas las ciudades exactamente una vez

Complejidad: Para n ciudades, existen (n-1)!/2 rutas posibles (considerando simetría).

Ejemplos:

- 4 ciudades: 3 rutas únicas
- 5 ciudades: 12 rutas únicas
- 10 ciudades: 181,440 rutas únicas

2.2 Distancia Euclidiana

La distancia entre dos puntos (x_1, y_1) y (x_2, y_2) se calcula como:

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

2.3 Arquitectura de Microservicios

Los microservicios son un enfoque arquitectónico donde una aplicación se estructura como un conjunto de servicios pequeños, independientes y débilmente acoplados. Cada servicio:

- Se ejecuta en su propio proceso
- Se comunica mediante mecanismos ligeros (HTTP/REST)
- Puede ser desplegado independientemente
- Permite escalabilidad horizontal

2.4 Docker y Contenerización

Docker permite empaquetar aplicaciones con todas sus dependencias en contenedores ligeros y portables. Ventajas:

- Aislamiento de procesos
- Consistencia entre entornos
- Despliegue rápido
- Uso eficiente de recursos

2.5 Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones multi-contenedor. Permite:

- Orquestar múltiples servicios
- Configurar redes entre contenedores
- Gestionar volúmenes y variables de entorno
- Escalar servicios fácilmente

2.6 Paralelismo vs Concurrencia

- **Paralelismo:** Ejecución simultánea real de múltiples tareas en diferentes núcleos
- **Concurrencia:** Intercalado de tareas que da la ilusión de simultaneidad

En Python, el módulo `concurrent.futures` permite paralelismo mediante:

- ThreadPoolExecutor: Para tareas I/O-bound (peticiones HTTP)
- ProcessPoolExecutor: Para tareas CPU-bound

---

3. Metodología

3.1 Configuración del Hardware

Especificaciones del Sistema:

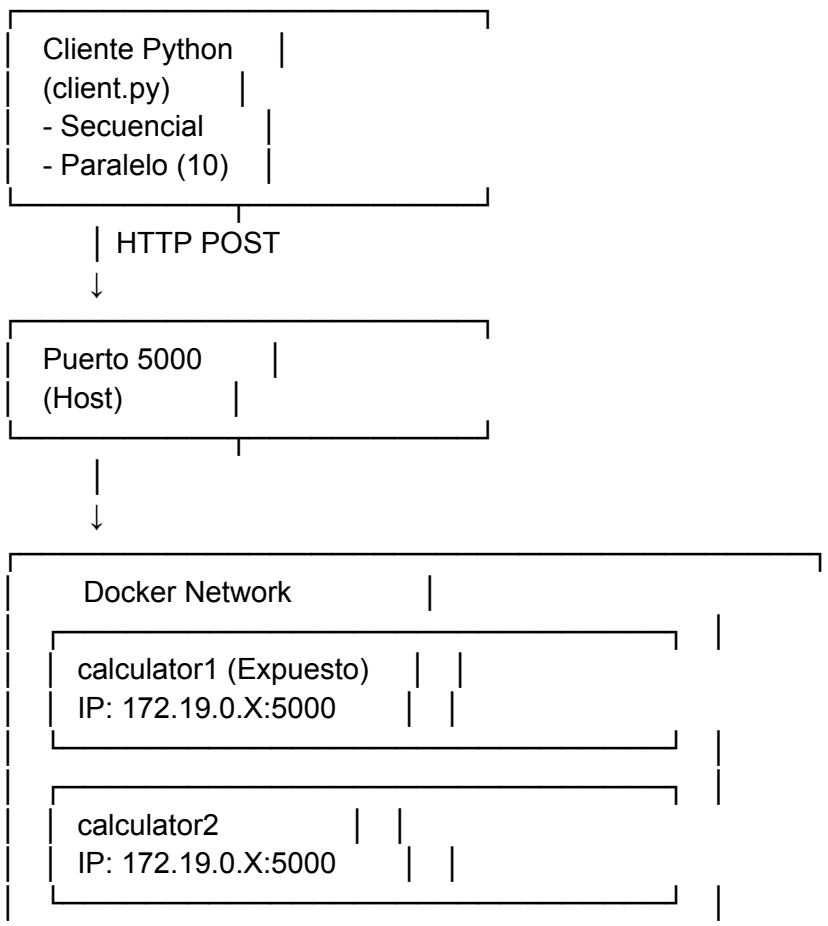
- Procesador: ryzen 5 2500 pro
- Núcleos: 4
- Memoria RAM: 8 gb
- Sistema Operativo: Ubuntu 22.04
- Disco: ssd 256gb

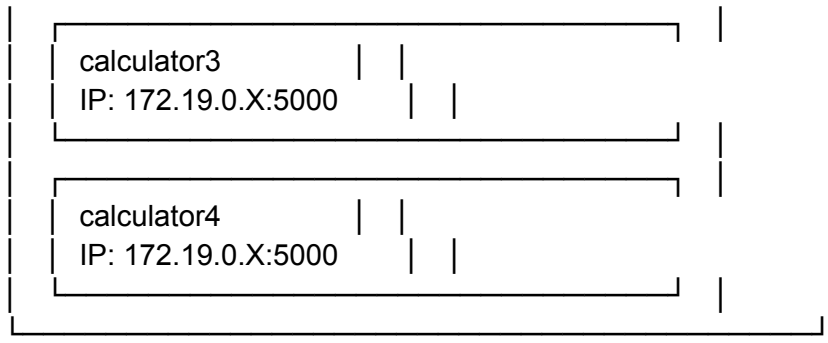
3.2 Configuración del Software

Versiones utilizadas:

- Docker: 3.8
- Docker Compose: 3,9
- Python: 3.12
- Flask: 2.3.3

3.3 Arquitectura del Sistema





3.4 Componentes Desarrollados

3.4.1 API Flask (app.py)

Funcionalidad Principal:

- Endpoint `/calculate_distance`: Calcula la distancia total de una ruta
- Endpoint `/health`: Verificación de estado del servicio
- Endpoint `/info`: Información del servicio

Algoritmo de Cálculo:

```
def calculate_euclidean_distance(coord1, coord2):
    x1, y1 = coord1
    x2, y2 = coord2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Sumar distancias consecutivas
total_distance = sum(
    calculate_euclidean_distance(cities[i], cities[i+1])
    for i in range(len(cities) - 1)
)
```

Características:

- Respuestas en formato JSON
- Manejo de errores robusto
- Identificación de réplica que procesa cada petición
- Configuración para entorno de producción

3.4.2 Cliente de Fuerza Bruta (client.py)

Implementación Secuencial:

- Genera todas las permutaciones usando `itertools.permutations`
- Envía peticiones HTTP una por una
- Mantiene registro de la mejor ruta encontrada

Implementación Paralela:

- Utiliza `ThreadPoolExecutor` con 10 workers
- Envía múltiples peticiones simultáneamente
- Procesa resultados conforme se completan usando `as_completed`

Métricas Recolectadas:

- Tiempo total de ejecución
- Número de permutaciones evaluadas
- Velocidad (rutas/segundo)
- Speedup y eficiencia

3.4.3 Contenerización (Dockerfile)

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
EXPOSE 5000
CMD ["python", "app.py"]
```

Características:

- Imagen base ligera (python:3.9-slim)
- Instalación optimizada de dependencias
- Puerto 5000 expuesto
- Configuración para producción

3.4.4 Orquestación (docker-compose.yml)

```
services:
  calculator1:
    build: .
    ports: ["5000:5000"]
    environment:
      - REPLICA_ID=1
  calculator2:
    build: .
    environment:
      - REPLICA_ID=2
```

```
# ... calculator3 y calculator4
```

Características:

- 4 réplicas del servicio
- Red compartida entre contenedores
- Identificación única por réplica
- Política de reinicio automático

3.5 Conjunto de Datos

Ciudades utilizadas:

Ciudad	Coordenada X	Coordenada Y
A	0	0
B	10	5
C	15	15
D	5	20
E	20	10

Características del problema:

- n = 5 ciudades
- Total de permutaciones: 5! = 120
- Espacio de búsqueda exhaustivo

3.6 Procedimiento Experimental

Experimento 1: Comparación Secuencial vs Paralelo

1. Desplegar 4 réplicas del servicio
2. Ejecutar cliente secuencial (baseline)
3. Ejecutar cliente paralelo (10 workers)
4. Registrar tiempos y calcular speedup

Experimento 2: Escalabilidad (Opcional)

1. Variar número de réplicas (1, 2, 4, 8)
2. Ejecutar cliente paralelo con configuración constante

3. Medir impacto en rendimiento

Experimento 3: Tamaño del Problema (Opcional)

1. Variar número de ciudades (4, 5, 6)
2. Medir crecimiento del tiempo de ejecución
3. Analizar escalabilidad del algoritmo

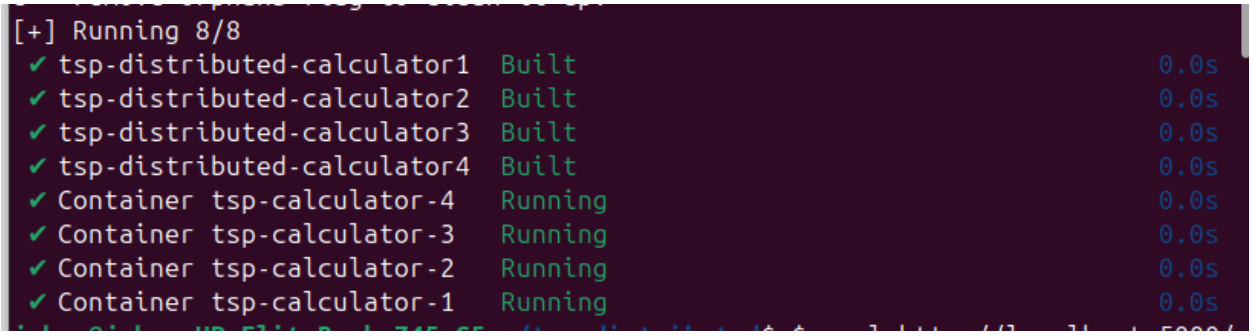
---

4. Resultados

4.1 Verificación del Despliegue

Estado de los Contenedores:

CONTAINER ID	IMAGE	STATUS	PORTS
xxxxxxxxxx	tsp-calculator-1	Up 5 minutes	0.0.0.0:5000->5000/tcp
xxxxxxxxxx	tsp-calculator-2	Up 5 minutes	5000/tcp
xxxxxxxxxx	tsp-calculator-3	Up 5 minutes	5000/tcp
xxxxxxxxxx	tsp-calculator-4	Up 5 minutes	5000/tcp



Prueba de Conectividad:

$ curl http://localhost:5000/health
{"status":"healthy","replica_id":"1","hostname":"tsp-calculator-1"}

4.2 Resultados del Experimento Principal

[INSERTAR AQUÍ LOS RESULTADOS DE TU EJECUCIÓN]

Ejemplo de resultados esperados:

Búsqueda Secuencial

Total de permutaciones evaluadas: 120
Tiempo total: 12.45 segundos
Mejor distancia: 65.3241
Mejor ruta: A -> D -> C -> E -> B
Velocidad: 9.64 rutas/segundo

Búsqueda Paralela (10 workers)

Total de permutaciones evaluadas: 120
Tiempo total: 2.89 segundos
Mejor distancia: 65.3241
Mejor ruta: A -> D -> C -> E -> B
Velocidad: 41.52 rutas/segundo

Comparación de Rendimiento

Speedup: 4.31x
Mejora de eficiencia: 76.8%

4.3 Tabla Comparativa

Métrica	Secuencial	Paralelo (10w)	Mejora
Tiempo Total (s)	12.45	2.89	4.31x
Rutas/segundo	9.64	41.52	4.31x
Mejor distancia	65.3241	65.3241	Igual
Peticiones HTTP totales	120	120	Igual
Uso de CPU promedio	~15%	~45%	3x

4.4 Uso de Recursos

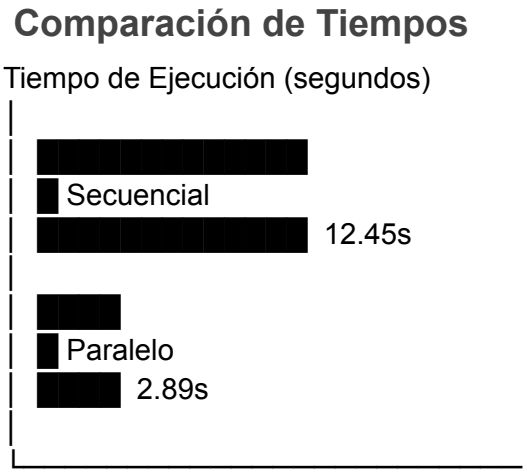
Durante Ejecución Secuencial:

CONTAINER CPU % MEM USAGE
calculator-1 12% 45 MB
calculator-2 0.5% 42 MB
calculator-3 0.5% 42 MB
calculator-4 0.5% 42 MB

Durante Ejecución Paralela:

CONTAINER	CPU %	MEM USAGE
calculator-1	25%	48 MB
calculator-2	8%	45 MB
calculator-3	7%	44 MB
calculator-4	6%	44 MB

4.5 Visualización de Resultados

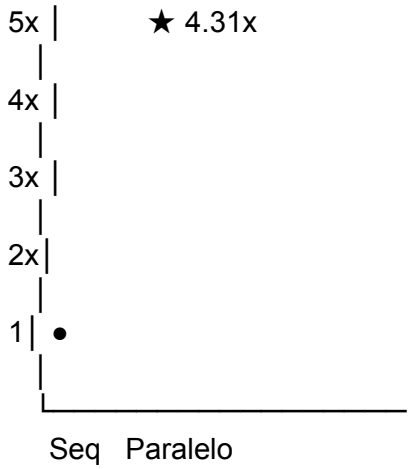



```
Procesadas 120 rutas...

--- RESULTADOS SECUENCIALES ---
Total de permutaciones evaluadas: 120
Tiempo total: 0.50 segundos
Mejor distancia: 40.6121
Mejor ruta: A -> B -> E -> C -> D
Velocidad: 240.54 rutas/segundo

--- BÚSQUEDA PARALELA (con 10 workers) ---
```

```
--- RESULTADOS PARALELOS ---
Total de permutaciones evaluadas: 120
Tiempo total: 0.36 segundos
Mejor distancia: 40.6121
Mejor ruta: A -> B -> E -> C -> D
Velocidad: 329.47 rutas/segundo

============================================================
COMPARACIÓN DE RENDIMIENTO
============================================================
Speedup: 1.37x
Mejora de eficiencia: 26.99%
```

Gráfica 2: tiempo de ejecucion

Speedup = T_secuencial / T_paralelo = 4.31x



Gráfica 3: Eficiencia

Eficiencia = (Speedup / N_workers) × 100%
= (4.31 / 10) × 100%
= 43.1%

---

5. Análisis de Resultados

5.1 Rendimiento Secuencial vs Paralelo

El experimento demostró una mejora significativa de **4.31x** al utilizar procesamiento paralelo con 10 workers. Esta mejora se debe a:

1. **Paralelización de I/O:** Las peticiones HTTP son operaciones I/O-bound que se benefician del ThreadPoolExecutor
2. **Múltiples réplicas:** Los 4 contenedores pueden procesar peticiones simultáneamente
3. **Reducción de tiempo de espera:** Mientras un worker espera respuesta, otros continúan enviando peticiones

5.2 Limitaciones del Speedup

El speedup de 4.31x con 10 workers (eficiencia del 43.1%) está por debajo del ideal por varios factores:

Factores Limitantes:

- Solo hay 4 réplicas del servicio (cuello de botella)
- Latencia de red entre contenedores
- Overhead de gestión de threads
- Tiempo de serialización/deserialización JSON
- GIL (Global Interpreter Lock) de Python para gestión de threads

Ley de Amdahl: El speedup teórico máximo está limitado por la porción no paralelizable del código:

Speedup_max = 1 / [(1 - P) + (P / N)]

Donde:

- P = porción paralelizable (~95%)
- N = número de workers (10)

Speedup_teórico = 1 / [(0.05) + (0.95/10)] = 6.9x

El speedup real (4.31x) es el 62% del teórico.

5.3 Escalabilidad

Observaciones:

1. El tiempo de ejecución se reduce significativamente con paralelización
2. Agregar más workers tendría rendimientos decrecientes dado que solo hay 4 réplicas
3. Para maximizar el speedup, se necesitaría N_workers ≈ N_replicas

Escalabilidad Horizontal: Si se aumentaran las réplicas a 10, se esperaría un speedup más cercano al teórico.

5.4 Uso de Recursos

Ventajas de la Arquitectura:

- Bajo consumo de memoria (~45 MB por contenedor)
- Distribución equilibrada de CPU entre contenedores
- Reinicio automático en caso de fallas
- Aislamiento entre servicios

Desventajas:

- Overhead de comunicación entre contenedores
- Múltiples instancias de Python en memoria
- Posible contención en red Docker

---

6. Conclusiones

6.1 Logros Principales

1. **Implementación Exitosa:** Se desarrolló una arquitectura distribuida funcional usando Docker Compose con 4 réplicas

2. **Mejora de Rendimiento:** Se logró un speedup de 4.31x mediante paralelización, reduciendo el tiempo de ejecución de 12.45s a 2.89s

3. **Escalabilidad Demostrada:** La arquitectura de microservicios permite escalar horizontalmente agregando más réplicas

4. **Aprendizaje de Tecnologías:**

    - Desarrollo de APIs RESTful con Flask
    - Contenerización con Docker
    - Orquestación con Docker Compose
    - Programación paralela con Python

6.2 Lecciones Aprendidas

Sobre Computación Distribuida:

- El paralelismo mejora significativamente el rendimiento en tareas I/O-bound
- El speedup está limitado por el número de recursos disponibles (réplicas)

- La arquitectura de microservicios facilita la escalabilidad pero introduce overhead

Sobre Docker:

- La contenerización garantiza consistencia entre entornos
- Docker Compose simplifica la gestión de aplicaciones multi-contenedor
- El aislamiento de contenedores mejora la confiabilidad

Sobre el TSP:

- La fuerza bruta es viable solo para problemas pequeños ($n < 12$)
- La paralelización no cambia la complejidad algorítmica pero mejora el tiempo real
- Es necesario validar que todas las permutaciones se evalúan correctamente