

# Paralelización de un algoritmo genético para resolver el problema del viajante

Michael Alexander Fajardo, Gloria del Valle

Arquitectura de Sistemas Paralelos

11/05/2020

# Contenidos

- 1 Explicación del problema del viajante
- 2 Solución con Algoritmos Genéticos
- 3 Paralelización del GA con DEAP + SCOOP
- 4 Paralelización del GA con DEAP + MPI
- 5 Resultados
- 6 Conclusiones

# Problema del viajante



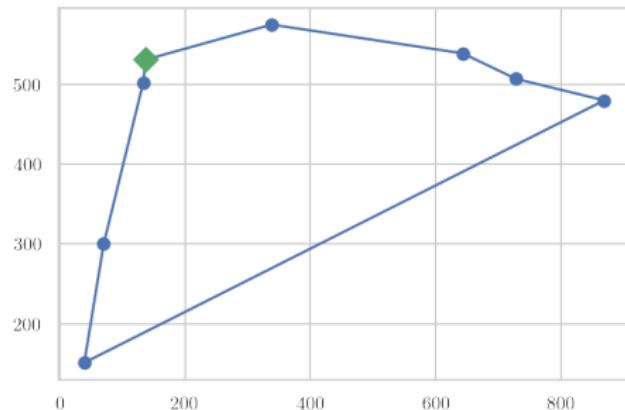
Dado un conjunto de ciudades y distancias entre cada par de ciudades, encontrar un **recorrido** con la **mínima distancia total**. Un **recorrido** comienza en una ciudad, visita todas las demás ciudades exactamente una vez y luego regresa a la ciudad inicial.

# Problema del viajante

- Se trata de un problema NP-Hard comúnmente conocido como intratable, lo que significa que no hay soluciones eficientes que funcionen para un gran número de ciudades.
- Cuanto mayor sea el número de nodos, mayor va a ser el número de rutas posibles, y por lo tanto mayor será el esfuerzo requerido para calcular todas ellas.
- Explora  $n!$  combinaciones, para  $n$  ciudades.

Código original proporcionado por [Luis Martí](#) en [Using genetic algorithms to solve the traveling salesperson problem](#)

# Estudio previo: diferentes estrategias



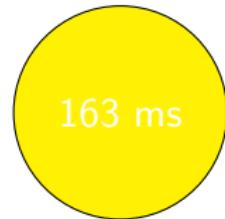
Base



No redundante

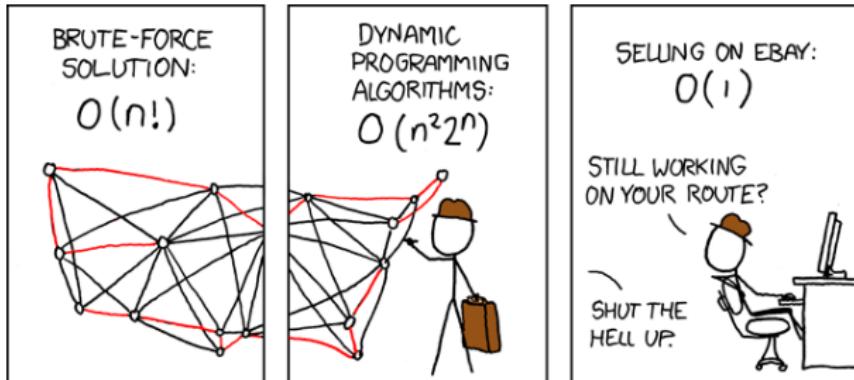


Greedy



→ Notebook del estudio

# Problemas



- Para un  $n$  mucho mayor, el problema es intratable.
- La estrategia codiciosa en general no produce una solución óptima.
- Tiene que haber mejores aproximaciones...

n cities	time
10	3 secs
12	$3 \text{ secs} \times 12 \times 11 = 6.6 \text{ mins}$
14	$6.6 \text{ mins} \times 13 \times 14 = 20 \text{ hours}$
24	$3 \text{ secs} \times 24! / 10! = 16 \text{ billion years}$

# Solución con Algoritmos Genéticos



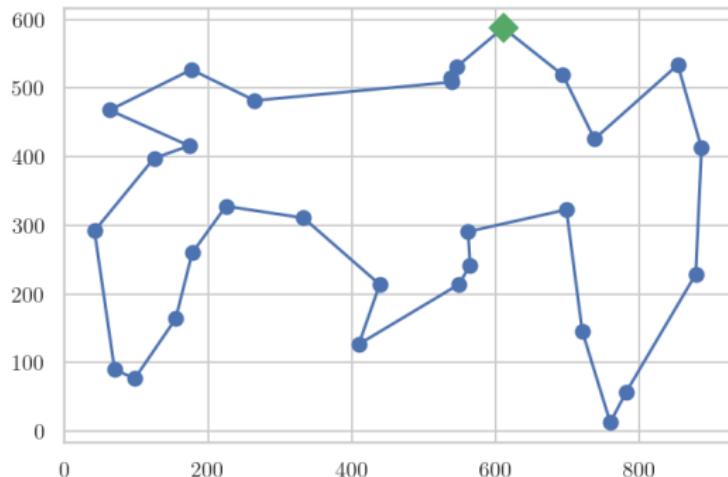
DISTRIBUTED  
EVOLUTIONARY  
ALGORITHMS IN  
PYTHON

- Las estructuras de datos son sencillas de adaptar a un problema propio.
- Permite parametrizar todo aspecto del algoritmo con la complejidad mínima.
- Facilita la paralelización del algoritmo, utilizando SCOOP.

Implementación en serie

→ Notebook del estudio

# Solución con DEAP



- 30 ciudades.
- 100 individuos.
- 400 generaciones.

- 20% probabilidad de mutación.
- 80% probabilidad de cruce.



- Utiliza una operación de mapeo que aplica una función a cada elemento de una secuencia `futures.map` para evaluar, por ejemplo, el *fitness*.
- El código referente a esta solución se encuentra en `tsp_SCOOP.py`

## Ejecución con SCOOP

```
python -m SCOOP tsp_SCOOP.py
```

## Consideraciones del reparto del trabajo

Estimamos el *speedup* según el número de *cores* y no según el tamaño del problema.

- Dividimos la población total entre el número de *cores*.
- Creamos tantos *chunks*, como *cores* se quieran utilizar.

De esta forma el **tamaño** del problema **no cambia** y se pueden utilizar tantos *cores* como se quieran.

*Nota: Los resultados posteriores se obtienen de la ejecución en el cluster.*

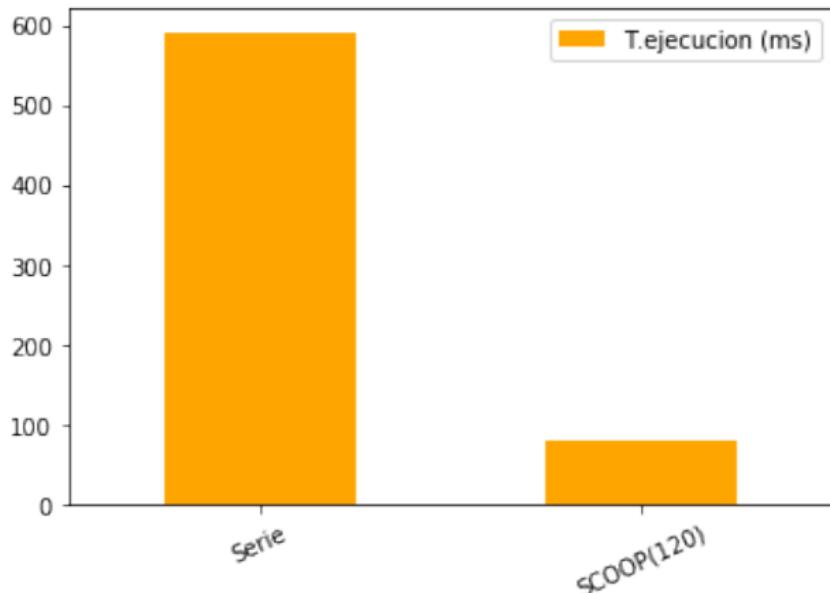
# ¿Y con MPI?

- Se basa mediante una comunicación colectiva.
  - scatter
  - gather
- En cada proceso se realiza el algoritmo de nueva generación con el *chunk* correspondiente.
- Es más eficiente cuando el problema es grande.
- Depende del tamaño del problema → puede llegar a tardar más cuando el problema es muy pequeño y el número de procesadores es elevado.

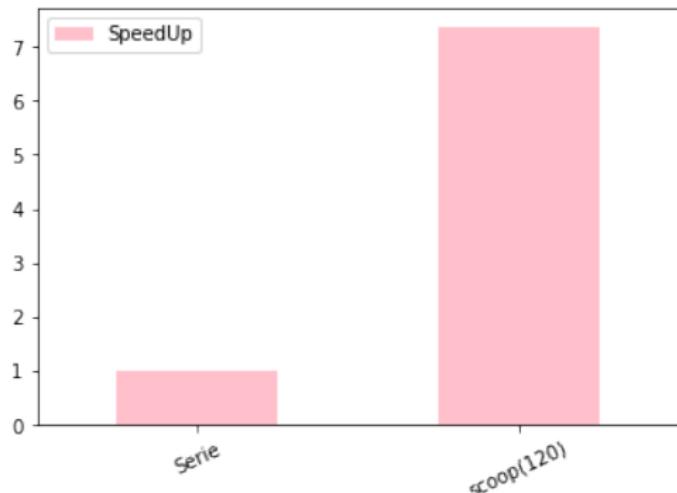
## Ejecución con MPI

```
mpiexec -n NPROC python tsp_mpi.py
```

## Resultados: tiempo de ejecución en serie vs SCOOOP(120)

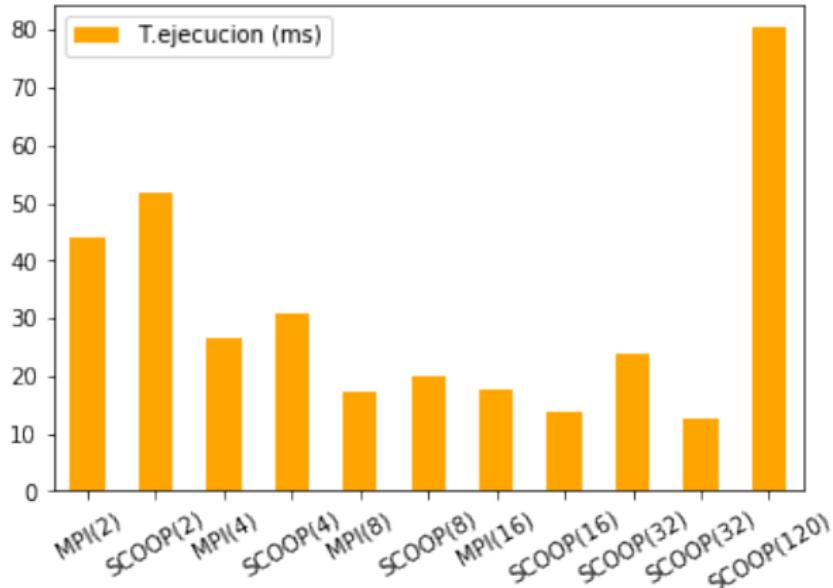


## Resultados: *speedup* en serie vs SCOOP(120)

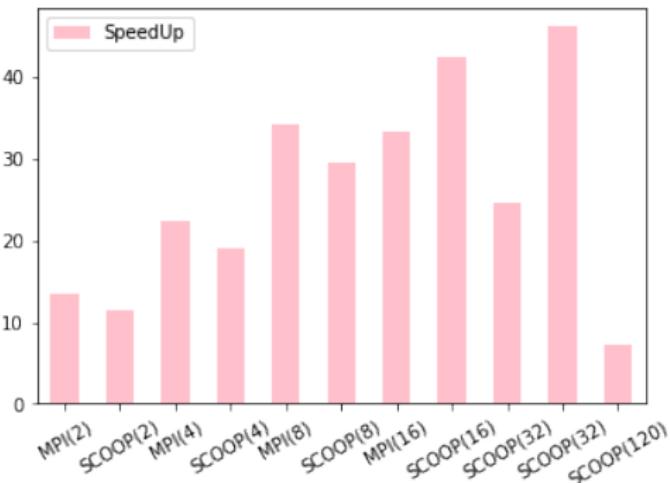


- El *speedup* que se obtiene con esta configuración es de **11.40**.
- ¿Es el mejor *speedup* que se puede obtener?

## Resultados: tiempos de ejecución en paralelo

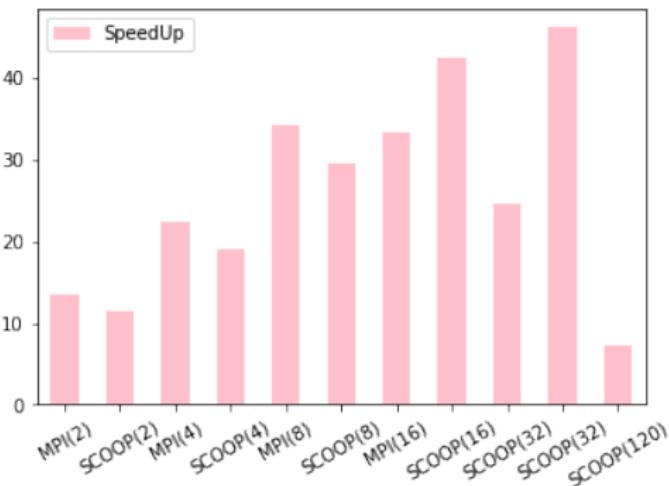


# Resultados: *speedup* en paralelo



- SCOOP con **120 cores** al final obtiene con esta configuración el **peor speedup**.
- ¿A qué se debe?
- ¿Por qué MPI deja de mejorar a partir de los 16 cores?

# Resultados: *speedup* en paralelo



El **tamaño** del problema es el **mismo** para todas las ejecuciones.

- Llega un momento en el que se tarda más en gestionar recursos que en la operación en sí.
- MPI al **utilizar** cada **procesador** para la paralelización tiene un *overhead* mayor al ser **procesos pesados**.
- SCOOP al utilizar **threads** y no procesos, puede seguir mejorando un poco más.

# Conclusiones

- MPI tiene un mejor rendimiento cuando el tamaño del problema es *adecuado*, es decir, el trabajo que corresponde a cada proceso debe tener a la par un tiempo de trabajo con el de la gestión de los recursos del mismo.
- SCOOP al utilizar pool de threads, puede utilizar más unidades de proceso, así como mejorar más al tener que administrar procesos ligeros y no pesados como MPI.
- La paralelización nos puede ayudar a correr algoritmos tan complejos en mucho menos tiempo.