



# TÉCNICAS DE COMPUTACIÓN CIENTÍFICA

## OPTIMIZACIÓN PARALELO

Eduardo Marquina García

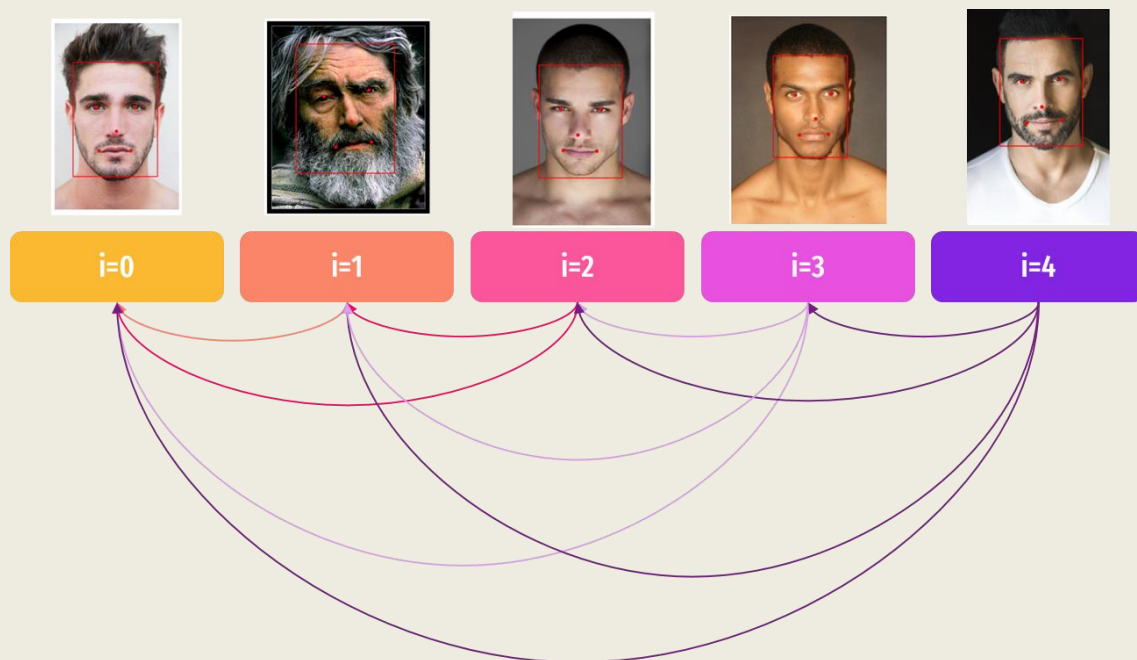
Miguel Casado Pina



## Introducción

Como se ha indicado en la práctica de optimización secuencial, en este proyecto partimos de un proceso en Python relacionado con el procesamiento de 500 imágenes sobre rostros de personas, descargadas del dataset de la plataforma Kaggle: En él se recorre recursivamente todas las imágenes de un directorio con el fin de aplicar un modelo de embbeding de imágenes mediante redes neuronales. Se trata de la red neuronal artificial MTCNN, de la librería `facenet_pytorch`.

Tras esto se obtiene el vector representativo de cada imagen para posteriormente calcular mediante `numpy`, la distancia euclidiana de cada imagen con el resto. Con esto cuantificamos la similitud de rasgos entre caras de distintas imágenes.



El objetivo de este segundo proyecto es conseguir optimizar este proceso por medio de técnicas de ejecución paralela.

## Desarrollo

Estudiando los resultados obtenidos de la primera práctica, observamos que en nuestro código existía un cuello de botella en el bucle encargado de transformar las imágenes a vectores por medio de la NN. Por ello, basándonos en la Ley de Amdahl se aplicará ahí la transformación para ejecutar de forma paralela.

Además, cabe destacar que cada vector obtenido en el primer bucle se almacena en una lista de vectores a la cual accede el segundo bucle en cargo del cálculo de la distancia euclidiana. Si escogiéramos las versiones del código de la primera práctica donde se aplican ambos procesos dentro del mismo código (versión inicial y versión tras la primera mejora), se producirían condiciones de carrera, trayendo errores al proceso y alterando la solución final.

Es por ello que partiremos de la tercera mejora aplicada, aquella que presentaba el menor tiempo de ejecución (760,707 segundos).

## Implementación

Como técnica de optimización paralela hemos empleado multiprocesamiento con la librería `ProcessPoolExecutor`. Con él generamos un proceso por cada núcleo de la CPU disponible, utilizando al máximo los recursos de la máquina.

A cada proceso se le aplica una fracción o batch del dataset de imágenes a procesar. Al tener 500 imágenes, se lanzaron paquetes de 50, obteniendo un total de 10 tareas. En el caso de nuestro ordenador, contábamos con 6 núcleos, por lo que cada núcleo disponible se ocupó de un trabajo y el resto de trabajos esperaron a tener núcleos libres para ser ejecutados.

Como técnica de evaluación y Benchmarking calculamos el tiempo de ejecución, consumo de CPU y uso de memoria en los procesos, haciendo una toma de valores por cada iteración de cada subproceso.

Intentamos compartir datos entre procesos con el uso de `Manager` de la librería `multiprocessing`, al igual que sincronización con el uso de cerrojos. Sin embargo, no fuimos capaces de comunicar correctamente los procesos obteniendo el dato final de memoria y uso de CPU.

Como alternativa de evaluación, usamos la herramienta `htop` de sistemas de distribución `Linux Debian`, el cual se encarga de evaluar el rendimiento de los núcleos de la CPU relacionado con los procesos activos. Además, monitorizamos el rendimiento de forma más visual por medio del administrador de tareas, accediendo al Pool de procesos de Python en el momento de su ejecución.

## Resultados y conclusiones

Mediante este proceso conseguimos reducir el tiempo de ejecución de 760,707 segundos a 224,647 segundos, lo cual supone una mejora notable.

El uso de memoria durante la ejecución ha sido de 10,6 GB de media sobre los 16 GB de RAM con lo que contamos. Es decir, un 66,25% de la memoria.

El uso de CPU consumido por el proceso ha sido de alrededor del 80%.

Como es lógico, se usa la gran mayoría de la capacidad de cómputo de la CPU, al aprovechar todos sus núcleos. Además, cada núcleo ocupado utilizaba entre el 97 y el 100% de su capacidad.

A continuación, se dejan capturas de pantalla de las métricas tomadas:










### Uso de la capacidad del núcleo de cada proceso por cada iteración (solo algunos)

```
Recopilado: CPU: 100.0%, Memoria: 8906.734375MB
Recopilado: CPU: 100.0%, Memoria: 10429.0703125MB
Recopilado: CPU: 100.0%, Memoria: 10461.15625MB
Recopilado: CPU: 100.0%, Memoria: 10930.171875MB
Recopilado: CPU: 100.0%, Memoria: 10589.09375MB
Recopilado: CPU: 100.0%, Memoria: 10396.94921875MB
Recopilado: CPU: 100.0%, Memoria: 10392.77734375MB
Recopilado: CPU: 99.7%, Memoria: 10343.90234375MB
Recopilado: CPU: 100.0%, Memoria: 10718.25MB
Recopilado: CPU: 100.0%, Memoria: 10500.74609375MB
Recopilado: CPU: 100.0%, Memoria: 10970.45703125MB
Recopilado: CPU: 99.7%, Memoria: 10863.953125MB
Recopilado: CPU: 100.0%, Memoria: 10738.6484375MB
Recopilado: CPU: 100.0%, Memoria: 10873.03125MB
Recopilado: CPU: 100.0%, Memoria: 10912.546875MB
Recopilado: CPU: 100.0%, Memoria: 10866.71875MB
Recopilado: CPU: 100.0%, Memoria: 11435.84375MB
Recopilado: CPU: 100.0%, Memoria: 11209.734375MB
Recopilado: CPU: 100.0%, Memoria: 11288.05078125MB
```

## Htop

```
0[|||||98.7%] 3[|||||100.0%]
1[|||||100.0%] 4[|||||100.0%]
2[|||||100.0%] 5[|||||100.0%]
Mem[|||||10.6G/15.9G] Tasks: 4, 1 thr; 1 running
Mem[|||||10.6G/15.9G] Load average: 0.52 0.58 0.59
Swp[|||||607M/29.0G] Uptime: 02:48:18
3[1] 4[1] 5[1]
```

**Administrador de tareas con los procesos Python. (El porcentaje de cada proceso está relacionado con el porcentaje total de la CPU, no del núcleo en sí).**

 Python 3.11 (9)	81,5%	5.559,2 MB	0,1 MB/s
 Python	0%	203,2 MB	0 MB/s
 Python	0%	201,9 MB	0 MB/s
 Python	5,8%	961,0 MB	0 MB/s
 Python	0%	553,2 MB	0 MB/s
 Python	32,2%	963,6 MB	0,1 MB/s
 Python	23,3%	1.245,7 MB	0,1 MB/s
 Python	9,9%	617,4 MB	0,1 MB/s
 Python	10,2%	611,6 MB	0,1 MB/s

## Conclusiones

Aunque las optimizaciones secuenciales son útiles, si se cuenta con la capacidad de cómputo y con un proceso que no genere condiciones de carrera, la optimización paralela mejora notablemente el tiempo de ejecución de un proceso, aprovechando al máximo las capacidades de la CPU.

Cabe destacar que este tipo de implementación tiene asociado un tiempo de ejecución para poner a disposición los núcleos de la CPU y crear los procesos hijos. Es por ello que este tipo de técnicas son útiles cuando se trata de un proceso de cálculo exhaustivo, dado que sino, el tiempo de creación de procesos sería superior al tiempo de mejora en sí.

Se puede encontrar este trabajo en el repositorio de GitHub [https://github.com/mcasadop/TCC\\_faces](https://github.com/mcasadop/TCC_faces), en la carpeta /Segunda. Existe un README.md con la descripción de los pasos para ejecutar correctamente el código.