Paralelización de multiplicación de matrices mediante: Hilos, Multiprocessing y MPI

Santiago García Herrera

10 de septiembre del 2025
Universidad de Caldas

Resumen

En primera instancia se analizó el código secuencial proporcionado en la actividad con el fin de comprender su funcionamiento, verificar la lógica de la multiplicación de matrices y medir el tiempo de ejecución como referencia base. Posteriormente, se revisaron los códigos trabajados en clase y se implementaron las estrategias de paralelización solicitadas: hilos, multiprocessing y MPI.

En el caso de hilos, se utilizó la librería threading de Python para dividir las filas de la matriz en dos partes, de modo que cada hilo ejecutara la multiplicación de una sección independiente. De manera similar, para multiprocessing se empleó la librería multiprocessing, creando dos procesos que trabajaron en paralelo sobre bloques distintos de filas, con el apoyo de una cola (Queue) para recolectar los resultados parciales y luego reconstruir la matriz completa.

Finalmente, con MPI se adaptó la misma lógica de partición de matrices, pero utilizando comunicación explícita entre procesos mediante mpi4py, distribuyendo filas entre los diferentes ranks y recopilando los resultados en el proceso raíz. En todos los casos, a medida que se aumentaba el número de hilos o procesos (2, 4, 6 y 8), las matrices se dividían en más bloques, lo que permitió realizar pruebas comparativas de rendimiento y posteriormente analizar los resultados obtenidos en términos de tiempo de ejecución y speedup.

Resultados

A continuación, se presentan los tiempos de ejecución obtenidos en cada una de las estrategias evaluadas (secuencial, hilos, multiprocessing y MPI), así como los valores de aceleración (*speedup*) calculados a partir del tiempo base secuencial (T1=341.9183T)

Procesos/Hilos	Secuencial	Hilo	Multiporcessing	MPI
1	341.9183	—	—	—
2	—	390.9230	157.4262	129.2700
4	—	350.8735	111.0910	77.5968
6	—	335.4182	62.9560	60.5794
8	—	336.0818	46.5342	53.5329

Tabla 1. Tiempos de ejecución por cada método de paralelización usado.

Procesos/Hilos	SpeedUp Hilos	SpeedUp Multiprocessing	SpeedUp MPI
2	0.876	2.172	2,645
4	0.974	3.078	4.406
6	1.019	5.432	5.642
8	1.017	7.347	6.390

Tabla 2. Valores de aceleración por cada método de paralelización usado.

.

La siguiente figura muestra las curvas de speedup para las tres estrategias paralelas implementadas. Se observa que multiprocessing obtuvo la aceleración más alta en la mayoría de los escenarios, seguido por MPI, mientras que los hilos no presentaron mejoras significativas debido a las limitaciones del Global Interpreter Lock (GIL) en Python.

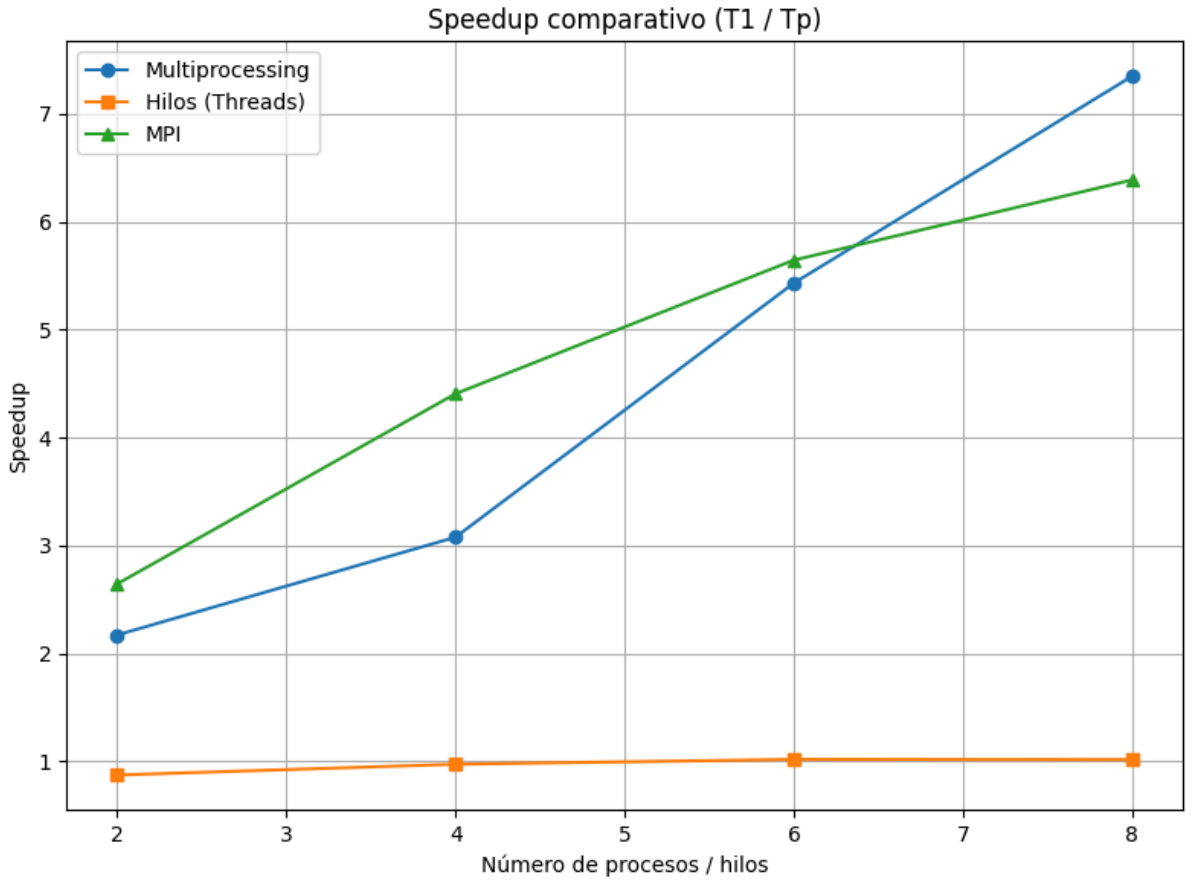

Imagen 1. Gráfica comparativa de aceleración.

Análisis de rendimiento

1. **Estrategia con mejor rendimiento**

La estrategia que mostró mejor rendimiento fue multiprocessing, alcanzando un speedup cercano a 7.35× con 8 procesos, lo cual se acerca bastante al ideal de 8×. La estrategia MPI también tuvo un buen comportamiento, con un speedup de 6.39× en 8 procesos, superando incluso a multiprocessing en los escenarios de 2 y 4 procesos. En contraste, la estrategia de hilos prácticamente no mejoró respecto al tiempo secuencial, con un speedup cercano a 1×. Esto confirma que en Python los hilos no permiten un verdadero paralelismo para tareas computacionales intensivas, debido al Global Interpreter Lock (GIL).

2. **¿Se logró una aceleración lineal?**

No se logró una aceleración completamente lineal. En multiprocessing, el speedup fue bueno pero siempre por debajo de la línea ideal. Esto puede explicarse por la Ley de Amdahl: aunque gran parte de la tarea es paralelizable, siempre queda una fracción secuencial que limita el rendimiento. Además, existe un sobrecosto de creación y comunicación entre procesos. En MPI, el comportamiento fue eficiente hasta 6 procesos, pero al llegar a 8 procesos la ganancia fue menor de lo esperado. La causa probable es la sobrecarga de comunicación entre procesos, ya que MPI debe transmitir y recolectar grandes porciones de datos (matrices de 1500×1500). En hilos, no se logró aceleración lineal en absoluto, debido a que el GIL no permite que múltiples hilos ejecuten bytecode de Python en paralelo en distintos núcleos. El resultado es que, aunque haya varios hilos, el cómputo sigue siendo prácticamente secuencial.

3. **Hipótesis sobre el comportamiento de cada estrategia**

Hilos: El bajo rendimiento se debe al GIL, que impide el uso real de varios núcleos en operaciones intensivas de CPU. En este caso, los hilos solo introducen sobrecostos de administración sin ofrecer beneficios significativos.
Multiprocessing: Escaló bien porque cada proceso se ejecuta en un núcleo independiente, evitando la limitación del GIL. Sin embargo, el rendimiento no es perfectamente lineal debido a la comunicación y sincronización entre procesos, así como al tiempo necesario para dividir y unir los datos.
MPI: Mostró buen rendimiento en un rango bajo-medio de procesos (2–6), pero a medida que se incrementan los procesos, la sobrecarga de comunicación entre nodos (broadcast, scatter, gather) se vuelve dominante. Esto hace que la ganancia se reduzca en comparación con multiprocessing cuando se llega a 8 procesos.

Conclusiones

En conclusión, la paralelización de la multiplicación de matrices permitió evidenciar que la estrategia de multiprocessing alcanzó el mejor rendimiento global con un speedup cercano a 7.35 en 8 procesos, mientras que MPI mostró un buen desempeño aunque limitado por los costos de comunicación en configuraciones mayores. Por el contrario, la estrategia de hilos no presentó mejoras debido al Global Interpreter Lock (GIL) de Python. Estos resultados confirman que, aunque se logra una aceleración significativa, esta no es lineal, en concordancia con la Ley de Amdahl y los sobrecostos inherentes a la coordinación entre procesos.