HPC

Caso de Estudio I

Multiplicación de Matrices

Héctor Fabio Vanegas
Brayan Cataño Giraldo
Juan Miguel Aguirre

Presentado a:

Ramiro Andres Barrios

Universidad Tecnológica De Pereira
2025

Abstract

El presente documento analiza el rendimiento de la multiplicación de matrices utilizando tres enfoques distintos: implementación secuencial, multihilo (usando POSIX Threads) y multiproceso (usando POSIX Processes). Se evaluaron diversas técnicas de optimización, destacando el uso de compilador con la opción -O3 y la reordenación de matrices (transposición) para mejorar la localidad de caché.

Tabla de contenido

1. INTRODUCCIÓN

La multiplicación de matrices es una operación fundamental en muchas aplicaciones científicas e ingenieriles, donde el rendimiento computacional es crítico. En el contexto de High Performance Computing (HPC), se exploran diversas estrategias para optimizar esta operación, combinando técnicas de paralelización y optimizaciones a nivel de compilador y algoritmo.

Este reporte se centra en comparar tres métodos de implementación para la multiplicación de matrices:

- Secuencial: Una versión básica de un solo hilo.
- Multihilo: Implementación utilizando POSIX Threads para distribuir la carga de trabajo entre múltiples hilos.
- Multiproceso: Enfoque basado en POSIX Processes que utiliza la técnica fork/wait para la ejecución paralela.

Mediante una serie de benchmarks y pruebas en un entorno controlado, se evaluó el impacto de diferentes tipos de optimizaciones como -O3 y -floop-interchange en compilación, técnicas de transposición para mejor alineación de caché, además de diferentes combinaciones de estas para analizar el rendimiento de cada implementación. Además, se analiza cómo estas estrategias se comportan en función del tamaño de las matrices, evidenciando que diferentes implementaciones pueden funcionar mejor con ciertos tamaños específicos de matrices, pero una idea concluyente es que lo más importante en este caso de estudio es el manejo adecuado de acceso a la memoria.

El objetivo es proporcionar una visión integral del rendimiento y la escalabilidad de cada enfoque, identificando las mejores prácticas para la optimización de operaciones intensivas en cálculos, y ofreciendo recomendaciones basadas en la configuración de hardware y los requerimientos de la aplicación.

2. MARCO CONCEPTUAL

2.1. High Performance Computing

Cuando hablamos de HPC por sus siglas en inglés, o Computación de alto rendimiento en español, "hacemos referencia a un campo de la computación actual que da solución a problemas tecnológicos muy complejos y que involucran un gran volumen de cálculos o de coste computacional." (López, 2017) Para nuestro caso, al aumentar la dimensión de las matrices a multiplicar, esto supone un alto costo computacional, el cual, según el análisis, se ve disminuido gracias al uso de herramientas que permiten, precisamente, disminuir el tiempo de ejecución.

2.2. Multiplicación matricial

Para implementar el algoritmo en el lenguaje C es necesario comprender previamente algunos conceptos, entre ellos, como es el proceso de multiplicar matrices de forma manual, el cual podemos aprender fácilmente gracias a vídeos explicativos y a diversos sitios web, tal como Producto de matrices.

Otros sitios en internet nos dan una idea más general de la programación de la multiplicación matricial, como por ejemplo el que se expone a continuación, que ha sido extraído del blog Análisis y diseño de algoritmos.

Dadas dos matrices A y B, tales que el número de columnas de la matriz A es igual al número de filas de la matriz B, es decir:

$$A := (a_{ij})_{m \times n}$$
$$B := (b_{ij})_{n \times p}$$

La multiplicación de A por B genera una nueva matriz resultado de la forma:

$$C = AB := (c_{ij})_{m \times p}$$

donde cada elemento c_{ij} está definido por:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Esta definición permite acercarse más a una idea del código, ya que de entrada plantea las tres variables a iterar para cada una de las celdas de la matriz resultado.

2.3. Complejidad Computacional

"La Teoría de la Complejidad estudia la eficiencia de los algoritmos en función de los recursos que utiliza en un sistema computacional (usualmente espacio y tiempo)" (Vásquez, 2004). Para este estudio, la complejidad computacional permite observar que en la multiplicación de matrices se gastan bastantes recursos de procesamiento en cuanto a tiempo, es decir, para matrices de dimensiones muy grandes, el procesador estará ocupado bastante tiempo resolviendo la multiplicación, lo que motiva aún más a programar de manera paralela el algoritmo, aprovechando al máximo los recursos del procesador.

2.4. Programación paralela

"La computación paralela es el uso de múltiples recursos computacionales para resolver un problema. Se distingue de la computación secuencial en que varias operaciones pueden ocurrir simultáneamente." (Ferestrepoca., 2019) En nuestro caso de estudio se ejecutará el código de multiplicación de matrices de forma secuencial y de forma paralela, esto con el fin de comparar los resultados obtenidos por cada ejecución y de esta manera poder analizar la eficiencia de la programación paralela.

2.5. Hilos

"La diferencia fundamental entre hilos y procesos es que un proceso dispone de un espacio de memoria separado, mientras que un hilo no. De esta forma los hilos pueden trabajar directamente con las variables creadas por el proceso padre, mientras que los procesos hijos no pueden hacerlo." (Antunez, 2011) Esta información nos facilita la comprensión del principio aplicado para esta implementación.

2.6. Speedup

El speedup representa la ganancia que se obtiene en la versión paralela de un programa con respecto a su versión secuencial. Para este estudio se ha tomado el speedup como:

$$S = \frac{T^*(n)}{T_p(n)}$$

donde $T^*(n)$ hace referencia al tiempo de ejecución secuencial y $T_p(n)$ hace referencia al tiempo de ejecución paralelo.

3. MARCO CONTEXTUAL

Para evaluar el rendimiento del algoritmo, se realizaron pruebas en sus tres versiones: **secuencial, con hilos y con procesos**. Cada una de estas versiones fue sometida a diferentes optimizaciones y combinaciones de parámetros, utilizando matrices de distintos tamaños: **10, 100, 200, 400, 800, 1600 y 3200**.

En el caso de las versiones con hilos y procesos, se probaron dos configuraciones para el número de unidades de ejecución: - **6 hilos/procesos**, equivalente al número total de procesadores físicos de la máquina utilizada. - **12 hilos/procesos**, que duplica la cantidad de procesadores físicos disponibles.

En una fase preliminar, la versión secuencial del algoritmo fue evaluada con **seis pruebas** aplicando diferentes optimizaciones, las cuales se describen en la sección 4. Por su parte, las versiones con hilos y procesos fueron sometidas a **12 pruebas cada una**, permitiendo identificar las configuraciones más eficientes para la prueba principal.

La **prueba principal** consistió en ejecutar **12 iteraciones** por cada combinación de método (secuencial, hilos, procesos), configuración de optimización (**4 principales**) y tamaño de matriz (**7 tamaños en total**). En total, se llevaron a cabo **1008 ejecuciones**. El promedio de los tiempos obtenidos en estas iteraciones se presenta en la sección 5.

Los detalles sobre la implementación del algoritmo y las optimizaciones aplicadas se encuentran en la sección 4, mientras que el análisis de los resultados y la comparación entre métodos se presentan en la sección 6.

3.1. Especificaciones del equipo de prueba

Las pruebas fueron ejecutadas en un equipo con las siguientes características:

- **Procesador:** AMD Ryzen 5 4500U (6 núcleos)
- **Memoria RAM:** 7,1Gi
- **Sistema operativo:** Ubuntu 22.04.5 LTS
- **Compilador utilizado:** gcc

4. DESARROLLO

En esta sección se describe el proceso de implementación de los algoritmos de multiplicación de matrices en sus versiones secuencial, con hilos y con procesos. Se detallan las estrategias de optimización utilizadas para mejorar el rendimiento de cada versión.

Se implementaron tres versiones para la multiplicación de matrices cuadradas de tamaño $n \times n$, cada una con un enfoque diferente para mejorar el rendimiento y aprovechar los recursos computacionales disponibles:

- **Implementación secuencial:** Se desarrolla un algoritmo estándar de multiplicación de matrices sin optimización, iterando sobre los índices de las matrices de forma tradicional.
- **Implementación con hilos (multithreading):** Se distribuye la carga de trabajo entre múltiples hilos para aprovechar el paralelismo a nivel de CPU.
- **Implementación con procesos:** Se divide la ejecución entre múltiples procesos, utilizando memoria compartida para la comunicación entre ellos.

Cada implementación cuenta con dos estrategias para la multiplicación de matrices:

1. **Multiplicación estándar:** La matriz resultado se calcula con la formulación clásica $C[i][j] = \sum_k A[i][k] \times B[k][j]$.
2. **Multiplicación con transpuesta:** Se optimiza el acceso a memoria simulando la transpuesta de la segunda matriz B, lo que mejora la localización en caché de la CPU.

4.1. Flags de Ejecución

Para realizar pruebas de manera más flexible y automatizada, se implementaron las siguientes *flags* de ejecución:

- **n**: Define la dimensión de las matrices cuadradas. Si no se proporciona, el valor predeterminado es 2000.
- **–files matrixA.txt matrixB.txt**: Permite leer dos matrices desde archivos en lugar de generarlas aleatoriamente. Si se usa esta opción, también se debe especificar el valor de **n**.
- **–result outputFile**: Especifica el archivo donde se escribirá la matriz resultante. Por defecto, el resultado se almacena en `result.out`.
- **–transpose**: Activa una versión optimizada de la multiplicación de matrices que emplea la transposición de la segunda matriz para mejorar la eficiencia en el uso de caché.
- **–doublethreads**: Duplica el número de hilos utilizados en la ejecución con respecto al número de núcleos disponibles en la CPU, con el fin de analizar el impacto del sobreparalelismo en el rendimiento.

4.2. Implementación Secuencial

La versión secuencial es la más sencilla y sirve como referencia para medir mejoras de rendimiento. Se implementó en lenguaje C, realizando la multiplicación de matrices con tres bucles anidados. Además, permite leer matrices desde archivos o generarlas aleatoriamente.

4.3. Implementación con Hilos

Para mejorar el rendimiento, se desarrolló una versión con hilos (*threads*), usando la biblioteca `pthread`. Se distribuyen las filas de la matriz entre los hilos, permitiendo ejecutar la multiplicación en paralelo. Cada hilo procesa un subconjunto de filas de la matriz resultante.

El número de hilos se ajusta dinámicamente según la cantidad de núcleos disponibles en el sistema, lo que permite mejorar la eficiencia de la ejecución.

4.4. Implementación con Procesos

En esta versión, la multiplicación se divide en múltiples procesos en lugar de hilos, utilizando memoria compartida (`mmap`) para almacenar las matrices. Cada proceso es responsable de un subconjunto de filas de la matriz resultado.

El número de procesos se ajusta en función del número de núcleos del procesador. Al finalizar la ejecución, se sincronizan los procesos con `wait()` antes de escribir el resultado en el archivo de salida.

4.5. Medición de Tiempos

Para comparar el desempeño de cada implementación, se mide el tiempo de ejecución utilizando la función `clock_gettime`. La diferencia entre el tiempo de inicio y el de finalización se usa para determinar la eficiencia de cada método.

5. PRUEBAS Y ANÁLISIS

Esta sección presenta los resultados obtenidos en las pruebas realizadas para cada método, así como un análisis comparativo del rendimiento. Para mayor claridad del lector, la optimización Loop en las pruebas significa -floop-interchange.

5.1. Pruebas con la versión secuencial

Aquí se presentan los resultados obtenidos al ejecutar la versión secuencial del algoritmo, evaluando su eficiencia y las mejoras logradas con las optimizaciones aplicadas.

5.1.1 Comparación con -O3 -floop-interchange

La primera prueba realizada con la versión secuencial es una prueba para comparar si las flags de compilación -O3 y -floop-interchange hacen que la lógica del código cambie. Las 2 configuraciones utilizan un tamaño de matriz de 2000, llegando al mismo resultado para la compilación normal y la compilación con optimización, pero esta última siendo 4.13 veces

más rápida.

- **Compilación sin optimización:** 57.415518036 segundos
- **Compilación con -O3 -floop-interchange:** 13.896337045 segundos

5.1.2 Prueba preliminar secuencial

En esta prueba se evaluaron diversas combinaciones de optimizaciones en la versión secuencial. Se compararon implementaciones que aplican la optimización `-O3` junto con técnicas de -floop-interchange, Transpose o ambas, frente a la implementación estándar. Los resultados muestran que el uso combinado de optimizaciones (O3 + Loop + Transpose y O3 + Transpose) reduce considerablemente el tiempo de ejecución, alcanzando un speedup superior a 11 veces respecto a la versión estándar.

Implementation	**Time (sec)**	**Speedup**
O3 + Loop + Transpose	5.34	11.10x
O3 + Transpose	5.36	11.05x
O3 + Loop	13.45	4.41x
O3 only	13.80	4.30x
Transpose only	34.02	1.74x
Standard	59.33	1.00x

Table 1: Resultados de la prueba preliminar secuencial

5.1.3 Prueba principal secuencial

Esta prueba evalúa el rendimiento de la implementación secuencial en función del tamaño de las matrices. Para cada configuración de optimización y cada dimensión, se realizaron 12 ejecuciones, y se registraron los tiempos promedio obtenidos. Las dimensiones analizadas varían de 10 a 3200, lo que permite apreciar la escalabilidad de cada técnica optimizada en comparación con la versión estándar.

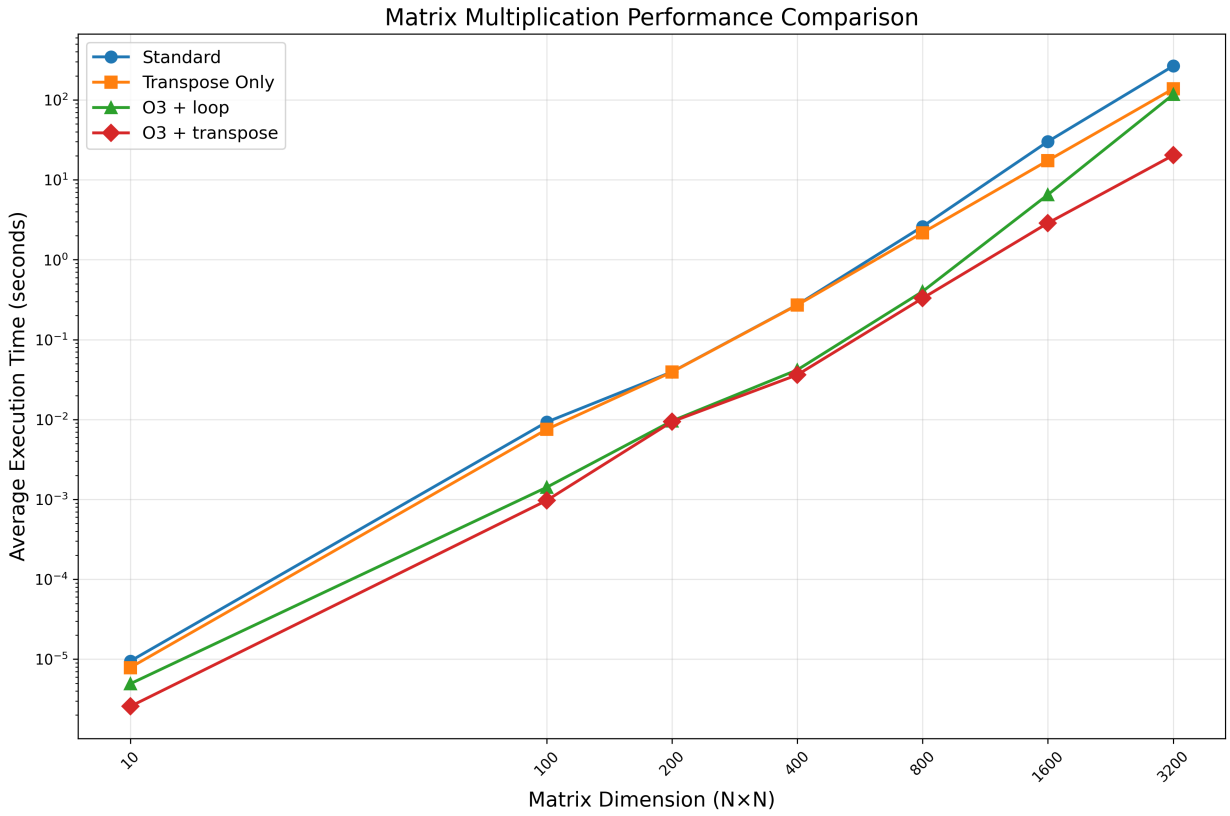

Figure 1: Tiempos de ejecución vs Dimensión de matrices (secuencial).

Dimension	O3 + loop	O3 + transpose	Standard	Transpose Only
10	0.000005	0.000003	0.000009	0.000008
100	0.001417	0.000970	0.009279	0.007521
200	0.009588	0.009387	0.039467	0.039278
400	0.041798	0.036231	0.272659	0.270871
800	0.400794	0.329565	2.606762	2.179408
1600	6.519647	2.877499	30.150426	17.425771
3200	117.031910	20.296180	265.969098	137.821635

Table 2: Resultados de la prueba principal secuencial

5.2. Pruebas con hilos

Se analiza el rendimiento de la versión paralela basada en hilos, comparando distintas configuraciones de ejecución y su impacto en la eficiencia del algoritmo.

5.2.1 Prueba preliminar con hilos

En la versión multihilo se analizaron distintas combinaciones de optimizaciones, incluyendo el uso de doble hilos (`DoubleThreads`). La prueba preliminar permite identificar que las implementaciones que combinan la optimización `-O3` con transposición ofrecen un rendimiento significativamente superior, con speedup que alcanzan hasta 8.48 veces en comparación con la versión estándar, de igual forma se analizó que el uso de dobles hilos no tiene una mejora importante con respecto a sus pares sin DoubleThreads.

Implementation	Time (sec)	Speedup
O3 + Transpose + DoubleThreads	1.07	8.48x
O3 + Transpose	1.12	8.13x
O3 + Loop + Transpose	1.13	8.03x
O3 + Loop + Transpose + DoubleThreads	1.29	7.07x
O3 + Loop	2.12	4.29x
O3 + DoubleThreads	2.37	3.84x
O3 + Loop + DoubleThreads	2.48	3.66x
O3 only	2.75	3.31x
Transpose	6.39	1.42x
Transpose + DoubleThreads	6.75	1.34x
Standard + DoubleThreads	8.99	1.01x
Standard	9.11	1.00x

Table 3: Resultados de la prueba preliminar con hilos

5.2.2 Prueba principal con hilos

La prueba principal con hilos evalúa el rendimiento en función del tamaño de las matrices, con dimensiones que varían de 10 a 3200. Los tiempos obtenidos evidencian que la paralelización mediante hilos reduce considerablemente el tiempo de ejecución, en especial para matrices grandes. Las configuraciones optimizadas (O3 + loop y O3 + transpose) muestran una escalabilidad destacada.

Dimension	O3 + loop	O3 + transpose	Standard	Transpose Only
10	0.000288	0.000281	0.000266	0.000309
100	0.000570	0.000565	0.002096	0.002217
200	0.002472	0.002341	0.012098	0.012224
400	0.015532	0.014220	0.055481	0.066472
800	0.096508	0.075750	0.481740	0.420476
1600	0.960430	0.553500	4.883142	3.728317
3200	21.823923	4.920511	50.627067	28.086868

Table 4: Resultados de la prueba principal con hilos

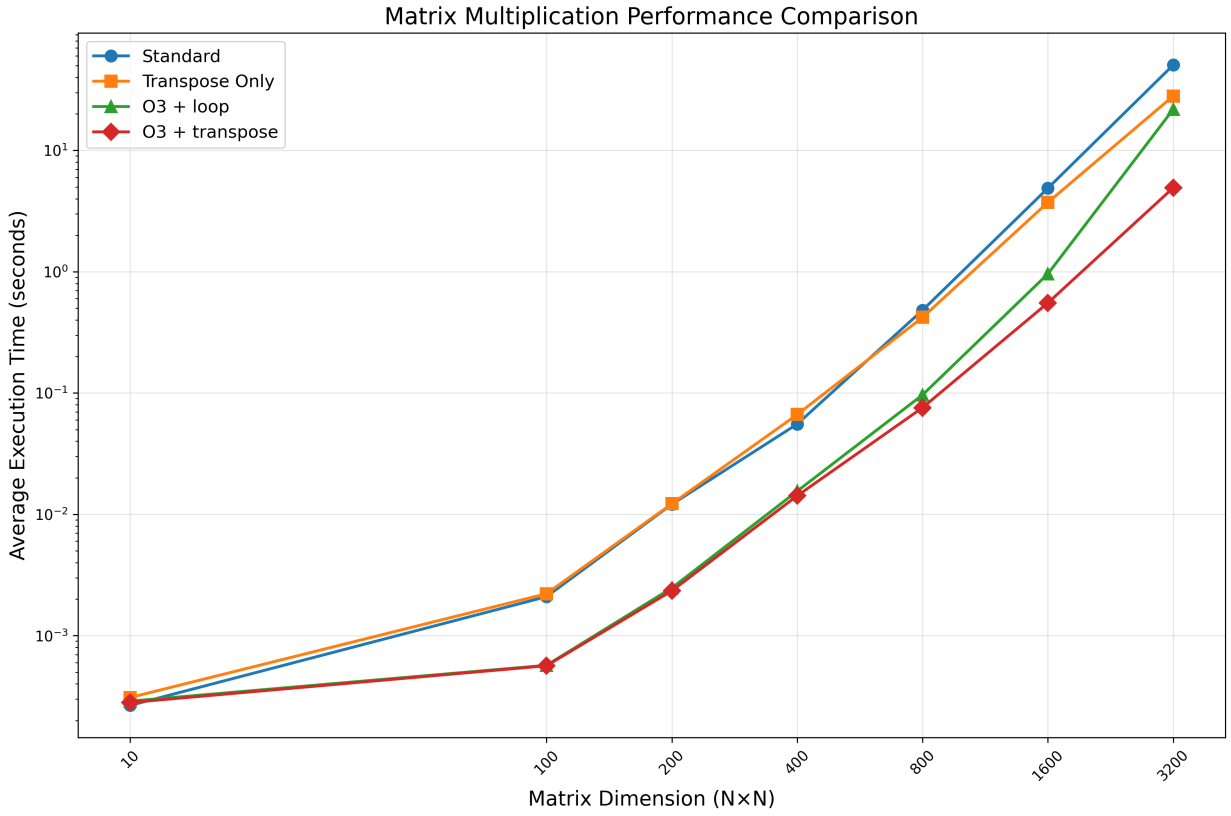

Figure 2: Tiempos de ejecución vs Dimensión de matrices (hilos).

5.3. Pruebas con procesos

En esta sección se presentan los resultados de la versión basada en procesos, destacando sus ventajas y desventajas frente a las otras metodologías.

5.3.1 Prueba preliminar con procesos

En la versión multiproceso se evaluaron diferentes configuraciones de optimización, tanto con el uso de doble hilos como sin él. Los resultados preliminares indican que la combinación de `-O3` con Loop y Transpose ofrece el mejor rendimiento (speedup de 7.57x), mientras que la versión estándar con doble hilos presenta un speedup inferior, lo que evidencia la eficacia de las optimizaciones en este modelo.

Implementation	**Time (sec)**	**Speedup**
O3 + Loop + Transpose	0.92	7.57x
O3 + Loop + Transpose + DoubleThreads	0.99	7.00x
O3 + Transpose + DoubleThreads	1.01	6.90x
O3 + Transpose	1.06	6.54x
O3 + Loop	1.94	3.58x
O3 + Loop + DoubleThreads	2.00	3.47x
O3 only	2.04	3.41x
O3 + DoubleThreads	2.05	3.39x
Transpose	6.36	1.09x
Transpose + DoubleThreads	6.56	1.06x
Standard	6.95	1.00x
Standard + DoubleThreads	10.71	0.64x

Table 5: Resultados de la prueba preliminar con procesos

5.3.2 Prueba principal con procesos

La prueba principal con procesos analiza el rendimiento de la implementación multiproceso en función del tamaño de las matrices. Se evaluaron dimensiones desde 10 hasta 3200, y los resultados muestran que las implementaciones optimizadas (O3 + loop y O3 + transpose) consiguen reducir significativamente los tiempos de ejecución en comparación con la versión estándar, demostrando una mejor escalabilidad.

Dimension	O3 + loop	O3 + transpose	Standard	Transpose Only
10	0.000651	0.000633	0.000895	0.000709
100	0.000959	0.000860	0.002359	0.002385
200	0.003016	0.002396	0.012301	0.012812
400	0.015195	0.014235	0.057606	0.057618
800	0.083484	0.112730	0.445218	0.408261
1600	1.133376	0.506965	5.635137	3.778568
3200	19.903804	4.689157	87.777612	27.808911

Table 6: Resultados de la prueba principal con procesos

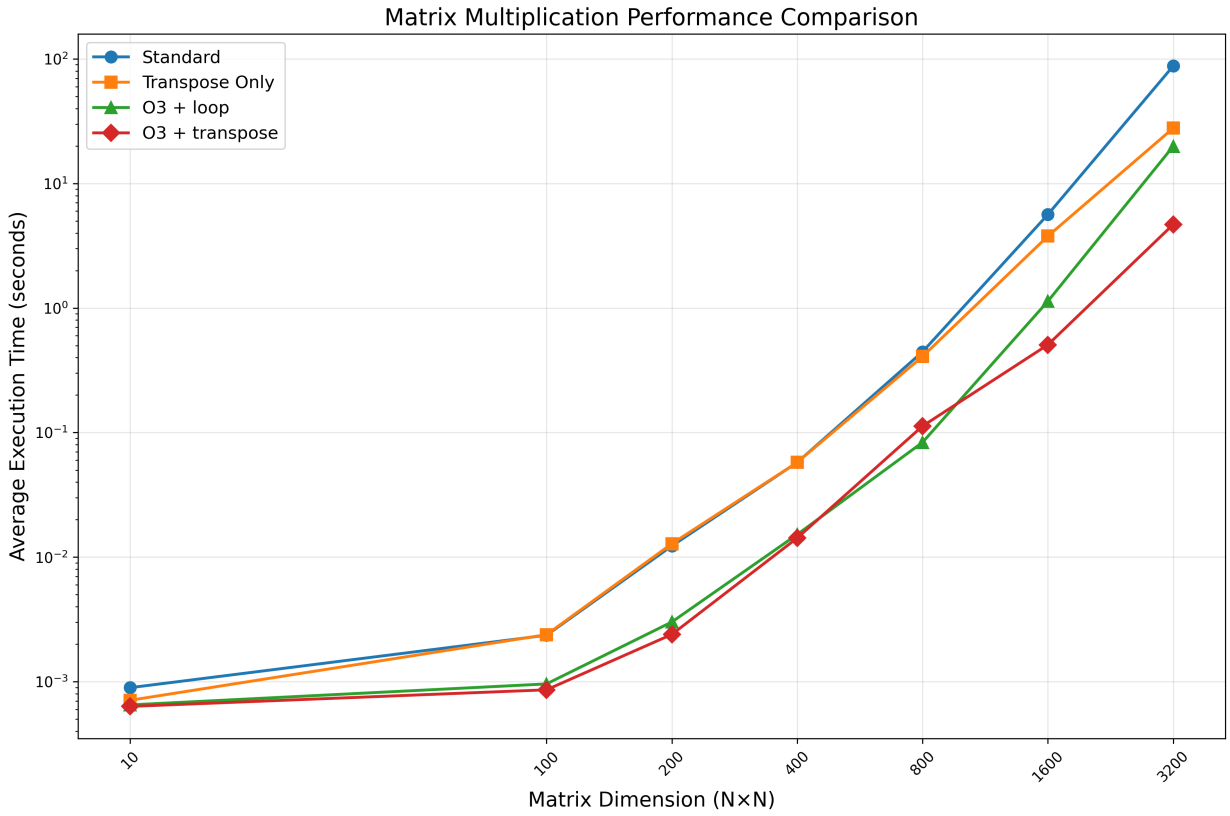

Figure 3: Tiempos de ejecución vs Dimensión de matrices (procesos).

6. CONCLUSIONES

A partir de los test preliminares se pudo evidenciar que existen configuraciones que apenas modifican el rendimiento del algoritmo, otras que lo mejoran de manera significativa e incluso algunas que pueden deteriorarlo.

La configuración -O3 destacó por estar siempre entre las mejores, demostrando un notable poder de optimización de programas. Cuando se combinó con la simulación de

la matriz transpuesta B—para mejorar la alineación en la caché—se obtuvo un speedup considerable. Este resultado fue sorprendente, ya que la implementación de la transpuesta por sí sola no mostró mejoras tan notorias, pero su sinergia con -O3 produjo un rendimiento sobresaliente.

Otro hallazgo interesante fue que aumentar el número de hilos a dos veces la cantidad de procesadores no resultó una estrategia óptima; en algunos casos se lograron tan solo segundos de mejora o incluso se empeoraron los tiempos de ejecución.

En las pruebas principales, se observó que, dentro de una configuración estándar (secuencial, hilos y procesos), la implementación basada en hilos fue la más eficiente. Sin embargo, con las optimizaciones adecuadas, el método basado en procesos pudo superar ligeramente a los hilos. Además, se constató que, para dimensiones de matrices mayores, el uso de hilos es más rentable, mientras que para dimensiones pequeñas—como en el caso de una matriz de dimensión 10—la versión secuencial resulta más adecuada.

Finalmente, se reafirma que la optimización en el uso y acceso a la memoria es crucial para mejorar el rendimiento del algoritmo. Las mejoras basadas en optimizaciones de memoria, como -O3, la transposición de la matriz y -floop-interchange, fueron las que lograron los mejores resultados, lo que sugiere que la multiplicación de matrices está limitada principalmente por el acceso a la memoria (memory bound).

Como dato interesante, en el caso de una matriz de dimensión 3200, la versión secuencial sin optimización alcanzó los 265.9 segundos, mientras que el mejor resultado obtenido (procesos con -O3 y transposición) fue de 4.68 segundos, representando un speedup de aproximadamente 56.82x.

References

Antunez, R. R. (2011). *Parallel programming: Definitions, mechanisms and trouble*. ResearchGate.

Ferestrepoca. (2019). *Programación paralela*. Ferestrepoca.

López, A. D. R. (2017). *Introducción al high performance computing (hpc)*. Encamina.

Vásquez, A. C. (2004). *Teoría de la complejidad computacional y teoría de la computabilidad*. RISI.