Informe Taller de Rendimiento

David Tobar Artunduaga

Daniel Ramirez Vargas

Guillermo Aponte Cárdenas

Juan David Daza Caro

Pontificia Universidad Javeriana

Sistemas Operativos

Bogotá, 13 Noviembre 2025

Objetivos

Objetivo General

Comparar el desempeño de diferentes algoritmos con sus versiones secuenciales y en paralelo, evaluando su eficiencia en distintos sistemas de cómputo con SO Linux, determinando cuál presenta mejor aprovechamiento de recursos computacionales y por qué.

Objetivos Específicos

Diseñar y ejecutar un experimento controlado que permita evaluar la influencia del número de hilos, el tamaño de las matrices y la jerarquía de memoria en el rendimiento de los algoritmos

Interpretar los resultados obtenidos durante el desarrollo del informe, utilizando estadísticas y gráficas, para extraer recomendaciones y conclusiones que le puedan servir al lector para comprender las distintas aplicaciones de los algoritmos y los sistemas de cómputo.

Resumen

El informe presenta un análisis comparativo del rendimiento de cuatro algoritmos para la multiplicación de matrices, evaluando sus versiones secuenciales y paralelas en diferentes entornos de hardware con sistemas operativos basados en Linux. El objetivo central fue determinar cómo influyen la cantidad de hilos o procesos, el tamaño de las matrices y la jerarquía de memoria en el tiempo de ejecución, identificando qué técnic logra un mejor aprovechamiento de los recursos disponibles.

Para esto, se desarrollaron implementaciones usando fork, POSIX threads, OpenMP clásico y OpenMP optimizado por filas (usando la transpuesta). Cada algoritmo se ejecutó con distintos tamaños de matriz y varias cantidades de hilos, repitiendo cada caso 30 veces para obtener promedios estables. El experimento se automatizó mediante un Makefile, un script en Perl y un programa auxiliar para calcular los promedios, lo que permitió manejar eficientemente la gran cantidad de pruebas.

Los resultados muestran que el rendimiento mejora conforme se incrementan los hilos, pero solo hasta igualar la cantidad de núcleos lógicos del procesador; después de ese punto, el Overhead es mayor que la ganancia esperada. También se evidenció que la eficiencia depende fuertemente del hardware, especialmente del tamaño y velocidad de las memorias caché, del ancho de banda de la RAM y de la presencia de tecnologías como Hyper-threading. Entre todos los métodos evaluados, el algoritmo basado en OpenMP con recorrido por filas fue el que mejor aprovechó la localidad espacial y, por

ende, la jerarquía de memoria, obteniendo los tiempos más bajos de ejecución en la mayoría de los escenarios.

En general, el estudio confirma la importancia de elegir una estrategia de paralelismo adecuada y evidencia que no basta con aumentar la cantidad de hilos para mejorar el rendimiento: el diseño del algoritmo, la forma en que accede a memoria y las características del hardware son determinantes para obtener buenos resultados.

Marco Teórico

1. Rendimiento

El rendimiento en un sistema operativo se refiere a la eficiencia con que se utilizan los recursos del hardware (CPU, memoria, almacenamiento y dispositivos de entrada y salida) para ejecutar procesos. Este parámetro se evalúa mediante métricas como el tiempo de respuesta, la utilización del procesador y la eficiencia (Silberschatz, Galvin & Gagne, 2018).

En el contexto del taller, el objetivo principal es comparar el rendimiento de diferentes técnicas de paralelización aplicadas a un problema intensivo en diferentes maquinas: la multiplicación de matrices.

2. Procesos e Hilos

Un proceso es una instancia de un programa en ejecución que posee su propio espacio de memoria, pila, contador de programa y recursos asignados. En cambio, un hilo (thread) es una unidad más ligera dentro de un proceso que comparte su espacio de direcciones y permite ejecutar varias tareas de forma concurrente (Linux Man Pages, 2024).

La comunicación entre procesos requiere mecanismos de interproces communication (IPC), mientras que los hilos pueden comunicarse mediante variables compartidas, aunque esto puede generar condiciones de carrera si no se controlan con mecanismos de sincronización (Tanenbaum & Bos, 2015).

3. Creación de Procesos con fork()

La llamada al sistema fork() crea un nuevo proceso, denominado proceso hijo, que es una copia casi exacta del proceso padre. Ambos continúan su ejecución a partir del punto donde se invocó fork().

Cada proceso tiene su propio espacio de memoria, lo que evita interferencias, aunque incrementa el consumo de recursos. Por esta razón, el modelo basado en procesos suele ser menos eficiente que el modelo multihilo cuando se requiere compartir datos (Linux Man Pages, 2024).

4. Librería POSIX Threads (pthreads)

La librería pthreads (POSIX Threads) proporciona una interfaz estándar para la creación y gestión de hilos en sistemas Unix y Linux.

Las funciones principales son:

pthread_create(): crea un hilo y le asigna una función a ejecutar.

pthread_join(): espera a que un hilo termine.

pthread_mutex_lock() / pthread_mutex_unlock(): sincronización mediante exclusión mutua.

Cada hilo ejecuta una parte del trabajo total, accediendo a variables compartidas. Este modelo ofrece gran control sobre la concurrencia, pero requiere cuidado para evitar errores de sincronización como race conditions o deadlocks (GNU, 2024).

5. Librería OpenMP

OpenMP es una API (Application Programming Interface) diseñada para programación paralela en arquitecturas de memoria compartida. Permite añadir directivas al código (usualmente sobre bucles for) para distribuir automáticamente el trabajo entre hilos del procesador. Su directiva más usada es:

#pragma omp parallel for

Esta divide las iteraciones del bucle entre los hilos disponibles. También se pueden usar reduction, critical, barrier y sections para controlar sincronización y operaciones acumulativas.

OpenMP permite ajustar el número de hilos mediante omp_set_num_threads(n) y consultar el número de hilos activos con omp_get_num_threads() (OpenMP ARB, 2023).

6. Multiplicación de Matrices

La multiplicación de matrices cuadradas es una operación fundamental en cálculo numérico, gráficos, inteligencia artificial y simulaciones.

Matemáticamente, si A y B son matrices cuadradas de tamaño N X N, su producto C = A X B se calcula como:

$$C_{\{i,j\}} = \sum_{k=0}^{N-1} A_{\{i,k\}} \cdot B_{\{k,j\}}$$

Cada elemento $C_{\{i,j\}}$ resulta de la multiplicación fila-columna entre las matrices A y B.

Este cálculo tiene una complejidad temporal de $O(N^3)$, por lo que el paralelismo es una estrategia clave para reducir los tiempos de ejecución (Hennessy & Patterson, 2017).

7. Medición del Tiempo de Ejecución

El rendimiento de cada algoritmo se mide mediante la función gettimeofday(), que devuelve la hora actual en segundos y microsegundos desde el Epoch (Linux Man Pages, 2024).

El tiempo total de ejecución se calcula como:

$$T_{ejec} = \left(t_{fin}.tv_{sec} - t_{ini}.tv_{sec}\right) * 10^6 + \left(t_{fin}.tv_{usec} - t_{ini}.tv_{usec}\right)$$

De esta forma se obtiene el tiempo en microsegundos que tarda el programa en ejecutar el proceso de multiplicación, permitiendo comparar las distintas técnicas de paralelismo.

8. Memoria RAM y Caché

La memoria RAM (Random Access Memory) determina la velocidad de acceso a los datos durante la ejecución de los procesos.

Una memoria de mayor frecuencia (MHz) y menor latencia mejora el rendimiento, especialmente en aplicaciones que manipulan grandes volúmenes de datos, como la multiplicación de matrices (Hennessy & Patterson, 2017).

Además, la memoria caché del procesador reduce la necesidad de acceder a la RAM (memoria principal). Los algoritmos que acceden secuencialmente a la memoria, como los optimizados para recorrer matrices por filas, aprovechan mejor la localidad espacial y temporal.

9. Hyper-Threading y Multinúcleo

El Hyper-Threading es una tecnología desarrollada por Intel que permite que un solo núcleo físico ejecute dos hilos de forma simultánea. Esto se logra duplicando las unidades de ejecución lógica para mejorar el uso de recursos y reducir los tiempos de espera (Intel Corporation, 2023).

En procesadores multinúcleo, el sistema operativo distribuye los hilos entre los núcleos disponibles, permitiendo un mayor paralelismo real.

Sin embargo, el Hyper-Threading no duplica la potencia del CPU, ya que los hilos comparten recursos físicos del núcleo, como la caché L1 y las unidades aritméticas.

10. Factores que Afectan el Rendimiento

El rendimiento observado en los experimentos del taller depende de múltiples factores:

- Cantidad de hilos/procesos: Un número mayor puede mejorar el rendimiento hasta un punto, pero genera Overhead si supera la capacidad del hardware.
- Planificación del sistema operativo: Linux usa políticas como Completely Fair Scheduler (CFS) para repartir el CPU.
- Ancho de banda de la memoria principal: Limita la velocidad de transferencia entre CPU y RAM.
- Afinidad de hilos: Los hilos asignados a diferentes núcleos pueden sufrir cache misses si acceden a datos compartidos.

- E/S y latencia: Aunque mínima en este tipo de tareas, puede ser relevante si se usan archivos o salida estándar intensiva (Love, 2010).

11. Arquitectura híbrida

La arquitectura híbrida es un diseño de procesadores que combina núcleos de alto rendimiento (Performance Cores o P-cores) y núcleos de alta eficiencia energética (Efficient Cores o E-cores) en un mismo chip. Este enfoque busca optimizar simultáneamente el rendimiento y el consumo energético, asignando tareas exigentes a los núcleos potentes y procesos secundarios o en segundo plano a los núcleos eficientes (Intel, 2021).

En el contexto de los **sistemas operativos y el rendimiento**, las arquitecturas híbridas introducen nuevos desafíos en la **gestión de hilos** y la **planificación del procesador (scheduling)**. El sistema operativo debe ser capaz de identificar la naturaleza de cada proceso y decidir en qué tipo de núcleo ejecutarlo para maximizar la eficiencia global. En Linux, esta tarea es gestionada por el **scheduler del kernel**, que evalúa métricas como la carga, prioridad y afinidad de CPU para distribuir el trabajo adecuadamente (Linux Manual Page, sched(7)).

Desde el punto de vista del rendimiento paralelo, la arquitectura híbrida impacta directamente en la eficiencia del paralelismo. En entornos con OpenMP, MPI o tareas multihilo, un uso inadecuado de los E-cores puede generar cuellos de botella, ya que estos núcleos poseen menor frecuencia y caché compartida. Por ello, es esencial

comprender la arquitectura del sistema y realizar pruebas de afinidad de CPU para obtener resultados de rendimiento más precisos (Intel, 2021).

12. Ley de los Grandes Números

La Ley de los Grandes Números (LGN) es uno de los pilares de la teoría de la probabilidad. Establece que, a medida que aumenta el número de repeticiones independientes de un experimento aleatorio, el promedio de los resultados observados tiende a aproximarse al valor esperado de la variable aleatoria (Ross, 2014).

Formalmente, si $"X_1, X_2, \dots, X_n"$ son variables aleatorias independientes e idénticamente distribuidas con media μ , entonces :

$$lim_{((n\to\infty))} \left(\frac{1}{n}\right) \Sigma^{n}_{((i=1))} X_i = E[X]$$

Esta expresión indica que el promedio muestral $\overline{X}n$ converge al valor esperado μ cuando el número de observaciones n tiende al infinito. Dicho resultado se conoce como convergencia en probabilidad, y constituye la base teórica para el cálculo de promedios confiables en contextos experimentales (Wackerly, Mendenhall, & Scheaffer, 2014).

En el análisis de rendimiento computacional, la LGN tiene un papel esencial al evaluar el tiempo de ejecución de programas paralelos o secuenciales. Cada ejecución de un mismo código puede arrojar tiempos ligeramente diferentes debido a múltiples factores: variaciones en la carga del procesador, latencias de memoria, interrupciones del sistema operativo o diferencias en la asignación de hilos (Silberschatz, Galvin, & Gagne, 2018).

13. Cuello de botella en el rendimiento

El cuello de botella en sistemas computacionales se refiere a la parte del sistema que limita el rendimiento global, impidiendo que los demás componentes alcancen su máximo potencial. Este fenómeno ocurre cuando un recurso, como la CPU, la memoria, el almacenamiento o la comunicación entre procesos, se convierte en el punto de saturación que restringe la velocidad de ejecución total (Tanenbaum & Bos, 2015).

En el contexto del rendimiento de sistemas operativos y la ejecución paralela, los cuellos de botella suelen presentarse cuando la capacidad de procesamiento no se distribuye eficientemente entre los hilos o núcleos disponibles. Por ejemplo, si una tarea depende en gran medida del acceso a memoria y la velocidad de la RAM es insuficiente para alimentar los núcleos de la CPU, el sistema experimentará un cuello de botella de memoria. De forma similar, en la programación paralela con OpenMP, un exceso de sincronización o regiones críticas mal definidas puede generar un cuello de botella de software, reduciendo la ganancia esperada por paralelización (Pacheco, 2011).

Procedimiento

1. Algoritmos Utilizados

Antes del diseño del experimento, se definen cuatro algoritmos diferentes para la multiplicación de matrices, con documentación intensiva y separados modularmente en distintos ficheros. Dichos algoritmos realizan la misma tarea, la multiplicación de dos matrices de un tamaño N, definido por el usuario, usando una cantidad de hilos o procesos según sea el caso también definido por el usuario. Finalmente realiza el registro e impresión del tiempo en microsegundos empleado por el sistema computacional para completar esta tarea. La diferencia en estos algoritmos se evidencia en las técnicas empleadas realizar el proceso, explicadas a continuación.

1.1 Similitudes de los Algoritmos

Los cuatro algoritmos, al realizar la misma tarea, tienen funciones en común para realizar este proceso, dichas funciones son las siguientes.

1.1.1 Función InicioMuestra().

Usa la función gettimeofday() de la librería time.h, esta función recibe una variable de tipo struct timeval, y en esta estructura guarda la hora en la que se llamó a la función, para esta función se llama a la estructura inicio y se guarda en su dirección de memoria.

1.1.2 Función FinMuestra().

Usa nuevamente la función gettimeofday(), y guarda su retorno en la dirección de memoria de la variable fin, del mismo tipo de inicio. Posteriormente resta los microsegundos iniciales y de los finales y los segundos iniciales de los finales. Guardando este resultado en la variable fin. Realiza el casteo a double y imprime este tiempo por consola.

1.1.3 Función impMatrix(double *matrix, int D).

Función que recibe un apuntador a double que corresponde a la matriz que se desea imprimir, y el tamaño de dicha matriz. Realiza un ciclo desde 0 hasta D*D, imprimiendo cada posición de la matriz y un salto de línea cuando se llega al borde de esta.

1.1.4 Función iniMatrix(double *m1, double *m2, int D)

Función que recibe dos matrices y las inicializa de la siguiente forma.

Realiza un ciclo for desde 0 hasta el cuadrado del tamaño enviado como parámetro, adentro de este ciclo usa la función rand(), que retorna valores muy grandes basados en una semilla de tiempo, y los divide entre RAND_MAX, de tal manera que este valor sea 1. Posterior a esa división multiplica el resultado por otro valor según la matriz, para darle un valor mayor que 1. El resultado lo castea a double y lo guarda en cada posición de la matriz.

1.1.5 Llamado en el main.

Todos los algoritmos realizan el llamado igual o similar de todas las funciones anteriores en el main, siguiendo este orden.

En primer lugar, verifica que los argumentos enviados por el usuario sean los suficientes, es decir, la cantidad de hilos que se usarán y el tamaño de la matriz que se va a multiplicar, guarda estas variables y según cuál sea el algoritmo, las usa para establecer el método de paralelismo, el cuál si varía según el caso y será explicado posteriormente. Realiza la declaración de 3 matrices dinámicas de tipo double y reserva la cantidad de memoria indicada según el tamaño de las matrices ingresada por el usuario, usando la función calloc, para que todos estos tamaños sean inicializados en 0. Inicializa la semilla que generará los números aleatorios usando la hora del equipo con la función time(), posteriormente usa la función inimatrix para inicializar las dos primeras matrices con valores aleatorios, las imprime y llama a la función inimatrix para tomar el tiempo antes de que se efectué la multiplicación, realiza el procedimiento necesario para multiplicar la matriz y llama a la función FinMuestra(), para calcular el tiempo que tardó en realizar el proceso, imprimiéndolo por pantalla. Finalmente, libera memoria

1.2 Algoritmo ClasicaFork

El algoritmo clásico fork, utiliza la función fork() de la librería unistd para realizar el proceso de multiplicación de matrices paralelamente utilizando diferentes procesos que se dividen en los múltiples núcleos del procesador.

1.2.1 Proceso en el Main.

Guarda la cantidad de subprocesos ingresado por el usuario en la variable num_P, luego, declara una variable llamada rows_per_process y le asigna el valor de N/num_P, de tal forma determina cuantas filas de la matriz donde se guardará el resultado va a calcular cada proceso.

Declara un ciclo for que va desde 0 hasta num_p, adentro de este ciclo declara una variable pid_t llamada pid y le asigna el retorno de la función fork(), de tal forma, el proceso padre creará los procesos hijos solicitados por el usuario.

Si la variable pid es igual a 0, es porque se encuentra ejecutando el proceso hijo, una vez acá, declara la variable start_row y le asigna el valor de i * rows_per_process, garantizando que cada proceso ejecute la función de multiplicar matriz desde una fila diferente. Declara la variable end_row y usando un operador ternario le asigna los siguientes valores, si i es el final, es decir, el último proceso, le asigna el valor de N, si no, le asigna el valor de start_row + rows_per_process. Realiza el llamado de multiMatrix y le envia como parámetro las 3 matrices, su tamaño la fila en donde inicia el proceso y la fila en donde termina. Si la matriz resultante es menor que 9, se realiza la impresión, al final realiza un ciclo for hasta el total de procesos y en él llama a la función wait(NULL), para que el proceso padre espere que todos sus hijos terminen.

1.2.3 Función multiMatrix(double *mA, double *mB, double *mC, int D, int filaI, int filaF).

Declara una variable suma de tipo double y dos apuntadores a double pA y pB. Declara un ciclo for desde filaI, que se le envia como parámetro hasta filaF, también enviado como parámetro y un ciclo desde 0 hasta el tamaño de la matriz, para recorrerla por columnas, le asigna a suma el valor de 0.0, a pA le asigna mA+i*D, es decir, el recorrido por filas de la matriz A y a pB le asigna mB+j, es decir, el recorrido por columnas de la matriz B. Declara un ciclo desde 0 hasta el tamaño de la matriz, haciendo que k se itere de uno en uno, al igual que pA, y pB se itera según D, saltando de columna en columna, adentro de este último ciclo guarda en la variable suma el valor de la sumatoria que se esté iterando en de pA, multiplicado por pB. Finalmente almacena el resultado en cada posición de mC.

1.3 Algoritmo ClasicaOpenMP

El algoritmo clásico OpemMP, realiza la multiplicación de matrices utilizando paralelismo con PRAGMAS de open MP, disponibles con la librería omp.h, paralelizando los ciclos for según la cantidad de hilos que el usuario le indique al algoritmo.

1.3.1 Proceso en el Main.

Declara una variable TH donde guarda la cantidad de hilos que se va a utilizar ingresad por el usuario. Luego, usando esta variable, se la envía como parámetro a la

función omp_set_num_threads, que le indica a la siguiente sección de código donde se emplee paralelismo que debe utilizar el número de hilos enviado a esta función.

1.3.2 Función multiMatrix(double *mA, double *mB, double *mC, int D).

Declara una variable suma de tipo double y dos apuntadores a double, pA y pB, posteriormente llama a la directiva #pragma omp for private(pA, pB, Suma), que le indica al compilador que el ciclo for lo debe realizar de manera paralela entre los hilos usados por el usuario, además indicándole que las variables pA, pB y suma serán privadas para cada hilo, de esta forma se evita la generación de condiciones de carrera.

Declara dos ciclos for desde 0, hasta D, enviado como parámetro que representa el tamaño de la matriz, para recorrerla por filas y por columnas, le asigna a suma el valor de 0.0, a pA le asigna mA+i*D, es decir, el recorrido por filas de la matriz A y a pB le asigna mB+j, es decir, el recorrido por columnas de la matriz B. Declara un ciclo desde 0 hasta el tamaño de la matriz, haciendo que k se itere de uno en uno, al igual que pA, y pB se itera según D, saltando de columna en columna, adentro de este último ciclo guarda en la variable suma el valor de la sumatoria que se esté iterando en de pA, multiplicado por pB. Finalmente almacena el resultado en cada posición de mC.

1.4 Algoritmo ClasicaPosix

El algoritmo clásico con posix utiliza las funciones de pthread de la librería pthread.h, de POSIX, para dividir el proceso de multiplicación de matrices en diferentes hilos, según como sea necesario.

1.4.1 Estructura parámetros.

Declara una estructura parámetros que en su interior contienen tres variables de tipo entero, nH, para guardar el total de hilos, idH para guardar el id del hilo y N para guardar el tamaño de las matrices.

1.4.2 Proceso en el Main.

Declara una variable n_threads y guarda en ella la cantidad de hilos ingresada por el usuario, posteriormente declara un arreglo de tipo pthread_t llamada p, y le da el tamaño de n_threads, este arreglo guarda la dirección de memoria de los diferentes hilos que se usarán para realizar el paralelismo.

Declara un ciclo for desde 0 hasta la cantidad de hilos, adentro de cada ciclo declara un apuntador de memoria a la estructura parametros, en idH guarda cada iteración de j, que equivale al id del hilo, en nH guarda n_threads y en N guarda N.

Llama a la función pthread_create de POSIX y le envia la dirección de memoria del hilo que está iterando, le envía la función multiMatrix para indicar que esta es la que debe utilizar el hilo y le envía el casteo de datos a void, para que la función lo pueda leer.

Finalmente crea un ciclo desde 0 hasta el número de hilos, en donde llama a la función pthread_join y le envia la dirección de cada hilo, para asegurar que se terminen de ejecutar todos y se unan sus resultados.

1.4.3 Función *multiMatrix(void *variable).

Función que realiza la multiplicación de matrices recibe un apuntador nulo, pues es lo necesario para que la función pueda ser ejecutada por pthread_create. Este apuntador a vacío corresponde a la estructura que contiene todos los parámetros necesarios para la ejecución del algoritmo, por tal motivo realiza el casteo de este parámetro a un apuntador de memoria a la estructura parámetros llamado data.

Declara tres variables entero, llamadas idH, nH y D y le asigna su correspondiente valor de la estructura data. Declara una variable entero llamada filaI donde guarda el valor de (D*idH)/nH, de tal forma cada hilo realiza la operación en distintas partes de la matriz, también crea una variable filaF y le asigna el valor de (D*(iDH+1))/nH, para que cada hilo sepa en donde tiene que terminar.

Declara una variable suma de tipo double y dos apuntadores a double pA y pB. Declara un ciclo for desde filaI hasta filaF, y un ciclo desde 0 hasta el tamaño de la matriz, para recorrerla por columnas, le asigna a suma el valor de 0.0, a pA le asigna mA+i*D, es decir, el recorrido por filas de la matriz A y a pB le asigna mB+j, es decir, el recorrido por columnas de la matriz B. Declara un ciclo desde 0 hasta el tamaño de la matriz, haciendo que k se itere de uno en uno, al igual que pA, y pB se itera según D, saltando de columna en columna, adentro de este último ciclo guarda en la variable suma el valor de la sumatoria que se esté iterando en de pA, multiplicado por pB. Finalmente almacena el resultado en cada posición de mC.

1.5 Algoritmo FilasOpenMP

El algoritmo clásico OpemMP, realiza la multiplicación de matrices utilizando paralelismo con PRAGMAS de open MP, disponibles con la librería omp.h, paralelizando los ciclos for según la cantidad de hilos que el usuario le indique al algoritmo, este algoritmo, además, en lugar de multiplicar las matrices recorriendo una por filas y otras por columnas, lo hace recorriendo ambas matrices por filas.

1.5.1 Proceso en el Main.

Declara una variable TH donde guarda la cantidad de hilos que se va a utilizar ingresad por el usuario. Luego, usando esta variable, se la envía como parámetro a la función omp_set_num_threads, que le indica a la siguiente sección de código donde se emplee paralelismo que debe utilizar el número de hilos enviado a esta función. Este algoritmo presenta además una pequeña variación en la función impMatrix, que recibe un parámetro entero t, que decide si una matriz se imprimirá normal o de manera transpuesta, según t sea 0 o 1.

1.5.2 Función multiMatrixTrans(double *mA, double *mB, double *mc, int D).

Declara una variable suma de tipo double y dos apuntadores pA y pB a double, posteriormente llama a una directiva #pragma omp parallel, que es una directiva de open MP utilizada para ejecutar todo el bloque de código seleccionado en los corchetes de manera paralela. Luego declara #pragma omp for, que indica que debe dividir cada iteración del primer ciclo for entre los hilos creados. Adentro declara un ciclo for que va

desde 0 hasta el tamaño de la matriz, adentro declara otro ciclo igual. Guarda en pA mA + i*D, para que recorra las filas de la matriz mA y en pB guarda mB+j*D para que recorra todas las filas de mB, inicializa la variable suma en 0.0.

Crea un ciclo de 0 hasta el tamaño de la matriz, que va iterando k, pA y pB, posteriormente en suma va guardando la suma de la multiplicación de cada iteración de pA y p. Finalmente guarda en cada posición de mC el resultado de suma.

2. Diseño de Experimentos

Con el fin de obtener resultados más precisos en la comparación del desempeño de cada algoritmo, se diseñó un experimento controlado orientado a evaluar cómo diferentes sistemas informáticos procesan la multiplicación de matrices, según los parámetros de ejecución definidos.

El procedimiento consiste en ejecutar repetidamente cada algoritmo con distintos tamaños de matriz y diferentes grados de paralelismo (número de procesos o hilos), registrando los tiempos de ejecución obtenidos en cada caso. Posteriormente, se calcula el promedio de los resultados para reducir el efecto de la variabilidad del sistema operativo y obtener una medida más representativa del tiempo real de cómputo.

El mismo proceso se repite en cada sistema de cómputo evaluado, con el propósito de relacionar el comportamiento del software con las características del hardware empleado, permitiendo así identificar la eficiencia y escalabilidad de los algoritmos en distintos entornos de ejecución.

2.1 Experimento Individual

Para la realización del experimento, se definieron diferentes tamaños de matriz con el propósito de evaluar cómo la carga computacional afecta el rendimiento de los algoritmos, pues entre mas grande sea la matriz, mas recursos computacionales consumirá, generando así el aumento del tiempo de ejecución, contando con mas comparaciones que permitan un mejor análisis con respecto a la eficiencia de los algoritmos.

De igual forma, se emplearon distintos números de hilos o procesos, que permiten analizar la eficiencia del paralelismo según el algoritmo empleado, esto permite determinar el punto en el cuál aumentar el número de hilos genera una reducción en el rendimiento de los algoritmos y cuál es la causa de esta afección.

Esta variación de parámetros posibilita identificar el mejor equilibrio entre el tamaño de problemas y el grado de paralelismo y comparar como distintos sistemas de cómputo aprovechan sus recursos de hardware en cada escenario.

2.1.1 Tamaños de Matriz.

Se definieron 4 tamaños diferentes para las matrices que se van a operar, empezando en 600 y duplicándolo hasta llegar a 4800, esto permite tener un aumento lineal que facilita las comparaciones del rendimiento según el tamaño de cada matriz al ser controlado y proporcional.

2.1.2 Número de Hilos.

Con el fin de evidenciar como el paralelismo mejora el rendimiento de los algoritmos, se plantearon 6 cantidades diferentes de hilos empleadas para le ejecución de los algoritmos, diseñados según la cantidad de núcleos disponibles para cada dispositivo, tema que se profundizará mas adelante. De tal forma, los valores planteados fueron los siguientes; 1, para probar la eficiencia de los algoritmos en caso de que sean ejecutados secuencialmente. 2, para probar comparar como un paralelismo mínimo puede mejorar la eficiencia en relación de la ejecución secuencial. 4, 8, 12 y 32, siento estos los tamaños máximos de núcleos disponibles para algunas de las máquinas empleadas en el experimento, esto con dos propósitos. El primero, evaluar como el overheap disminuye el rendimiento cuando una máquina crea una cantidad de hilos superior a su cantidad máxima de núcleos, pues la teoría dice que a partir de este punto es que se evidencia la afección, en caso de que sea diferente en la práctica, se presenta el análisis correspondiente de porqué sucede esto. El segundo, en la máquina que sea capaz de crear todos los hilos, evaluar como se ve dicha mejora y si se cumple según lo esperado.

2.1.3 Cantidad de Repeticiones.

Como se mencionó anteriormente, la multiplicación de matrices se realiza repetidamente siguiendo el teorema de los números grandes, en este caso, por cada valor de matriz y por cada cantidad de hilos que se usarán, se realiza el mismo proceso 30 veces y se calcula el promedio de todos los tiempos devueltos por cada algoritmo, con el

fin reducir el efecto de la variabilidad del sistema operativo y obtener una medida más representativa del tiempo real de cómputo.

2.2 Procesos de Automatización

Puesto que la ejecución de todos los procesos planteados corresponde a un proceso largo y tardado, se plantearon múltiples programas que permiten al compilación y la toma de datos automatizada, para su posterior análisis una vez se haya completado la ejecución de todos los algoritmos.

2.2.1 Makefile.

Es el utilizado para la compilación de todos los programas necesarios, en primer lugar, define las variables de compilación, sea este el compilador y las opciones para utilizar las librerías necesarias para le ejecución del código, como en openMP y en POSIX.

Define los ejecutables y los programas que se compilarán, siendo estos mmClasicaPosix, mmClasicaFork, mmClasicaOpenMp y mmFilasOpenMP, además de calculadoraPromedios que es un programa auxiliar utilizado para calcular el tiempo de los promedios.

Define en que directorio está cada módulo de los algoritmos y el nombre que recibe dicho módulo.

Define la regla principal que ejecuta todos los programas y posteriormente todas las reglas independientes, que crea el directorio de ejecutables en caso de que no exista,

compila el módulo auxiliar a objeto (.o) y compila el programa principal enlazándolo con el objeto del módulo.

Finalmente define la orden clean que elimina todo el directorio ejecutable con sus binarios y objetos.

2.2.2 lanzador.pl.

Es un script en perl que se encarga de automatizar toda la ejecución experimental de los programas de multiplicación de matrices y guardar todos los resultados en archivos .dat de la siguiente forma.

Usa ‘pwd’ para obtener la ruta del directorio actual y guarda su valor en $path, posteriormente elimina el salto de línea usando la función chomp().

Guarda el nombre de los ejecutables a correrse en variables del mismo nombre.

Declara dos listas, size_matriz, donde guarda los tamaños de las matrices a ejecutar y Num_Hilos donde guarda las cantidades de hilos a usarse, guarda además la cantidad de repeticiones en una variable.

Declara dos bucles, el primero que recorre todas las combinaciones posibles entre tamaño de matriz y el segundo que recorre las combinaciones para cantidad de hilos, adentro de estos bucles guarda en una variable por cada algoritmo la ruta en donde se encuentra el perl, el nombre del algoritmo, el tamaño, la cantidad de hilos y al final le coloca .dat, esta será la ruta donde se crearán los archivos con los resultados. Por ejemplo, /home/usuario/mmClasicaFork-4800-Hilos-8.dat, posterior a esto, los borra si ya existen.

Crea otro ciclo, que va desde 0 hasta el tamaño de repeticiones, por cada repetición ejecuta cada uno de los programas, con los diferentes tamaños de matrices y cantidad de hilos, guardando su salida en el archivo .dat.

Finalmente se calcula el promedio después de ejecutar las 30 repeticiones y se escribe al final de cada archivo .dat.

Se cierran todos los archivos.

2.2.3 calculadoraPromedios.c.

Recibe un archivo como parámetro, lee su contenido y lo va guardando en una variable acumuladora, al mismo tiempo que aumenta un contador para saber la cantidad de elementos que tenía el archivo, calcula el promedio usando los valores anteriores y los imprime por pantalla.

2.3 Sistemas de Cómputo Utilizados

Para la toma de datos desde distintos sistemas de cómputo, con el fin de evaluar el comportamiento de los algoritmos de multiplicación y el resultado de los mismo según el hardware con el que se cuenta, se utilizaron cinco sistemas de cómputo diferentes, incluyendo todas las máquinas personales de los integrantes del grupo que contaran con SO basado en Linux, una máquina virtual proporcionada por la universidad y un cluster prestado por el profesor.

Todos los procesos de ejecución del experimento fueron realizados en un entorno limpio y controlado, cerrando sesión del sistema operativo y trabajando únicamente desde la consola, esto con el fin de que tanto el procesador como la memoria se enfocaran

exclusivamente en ejecutar los algoritmos necesarios para el experimento, permitiendo que los tiempos de ejecución guardados en los archivos .dat correspondieran al trabajo totalmente dedicado del procesador en estos programas, sin presentar variaciones por la ejecución de interfaces gráficas, programas adicionales, entornos de programación, etc. En todos los sistemas de cómputo el experimento se realizó bajo las mismas reglas, garantizando una comparación justa y fiel a la realidad, en pro de obtener un análisis mucho más completo y resultados precisos. A continuación, se presenta una descripción detallada de cada máquina.

2.3.1 Lenovo Legion Pro 7i.

Máquina personal #1, con las siguientes características.

1. Uso de Hyper-Threating
2. 24 núcleos físicos
3. 32 núcleos lógicos
4. Velocidad máxima de 5.4 GHz por núcleo
5. 24 caches L1 de 2.196 MiB
6. 12 caches L2 de 32 MiB
7. 1 cache L3 de 36 MiB
8. 32 GB de RAM DDR5 a 89.6GB/S

2.3.2 Asus Gaming Tuf f15.

Máquina personal #2, con las siguientes características.

1. Uso de Hyper-Threating
2. 12 núcleos físicos
3. 16 núcleos lógicos
4. Velocidad máxima de 4.5 GHz por núcleo
5. 12 caches L1 de 0.96 MiB
6. 12 caches L2 de 9 MiB
7. 1 cache L3 de 18 MiB
8. 16 GB de RAM DDR4 a 25.6GB/S

2.3.3 Lenovo IdeaPad Gaming 3.

Máquina personal #3, con las siguientes características.

1. Uso de Hyper-Threating
2. 8 núcleos físicos
3. 12 núcleos lógicos
4. Velocidad máxima de 4.4 GHz por núcleo
5. 8 caches L1 de 0.69 MiB
6. 5 caches L2 de 7 MiB
7. 1 cache L3 de 12 MiB
8. 24 GB de RAM DDR4 a 51.2GB/S

2.3.4 Máquina Virtual.

Máquina virtual proporcionada por la universidad

1. No usa Hyper-Threating
2. 4 núcleos físicos
3. 4 núcleos lógicos
4. Velocidad máxima de 2.6 GHz por núcleo
5. 4 caches L1 de 0.32 MiB
6. 4 caches L2 de 5 MiB
7. 1 cache L3 de 42 MiB
8. 12 GB de RAM DDR5 a 76.8GB/S

2.3.5 ClusterHPC

Máquina prestada por el profesor

1. Uso de Hyper-Threating
2. 10 núcleos físicos
3. 20 núcleos lógicos
4. Velocidad máxima de 5.2 GHz por núcleo
5. 10 caches L1 de 0.64 MiB
6. 10 caches L2 de 2.5 MiB
7. 1 cache L3 de 20 MiB
8. 64 GB de RAM DDR4 a 51.2GB/S

2.3.6 Tabla comparativa.

Acceda aquí a la tabla comparativa de los equipos utilizados

Tabla 1. Dispositivos usados.

Resultados

1. Obtención de los resultados

Para la ejecución del taller y la obtención de los resultados se hizo uso de un script en Perl el cual permitió ejecutar los múltiples algoritmos de multiplicación de matrices, tanto los que usan Pthreads como OpenMP. Este script permitía ejecutar cada algoritmo y guardar de manera automática los resultados del tiempo que toma cada uno por 30 repeticiones en un archivo .dat, esto para 1,2,4,8,12 y 32 hilos.

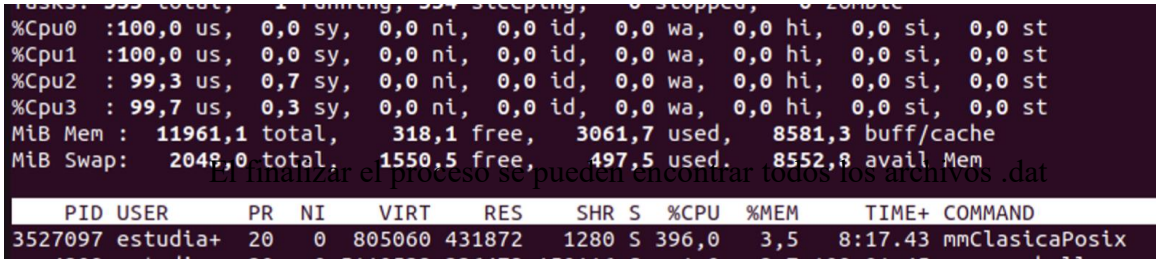
```
#crea la variable que guarda el nombre del archivo de salida .dat
$fileClasicoPosix = "$Path/$NombreClasicoPosix-".$size."-Hilos-".$hilo.".dat";
$fileClasicoFork = "$Path/$NombreClasicoFork-".$size."-Hilos-".$hilo.".dat";
$fileClasicoOpenMP = "$Path/$NombreClasicoOpenMP-".$size."-Hilos-".$hilo.".dat";
$fileFilasOpenMP = "$Path/$NombreFilasOpenMP-".$size."-Hilos-".$hilo.".dat";
```

Al momento de ejecutar el script comienza a ejecutar los programas uno a uno, un total de 30 veces, el comando de ejecución del programa se muestra en consola

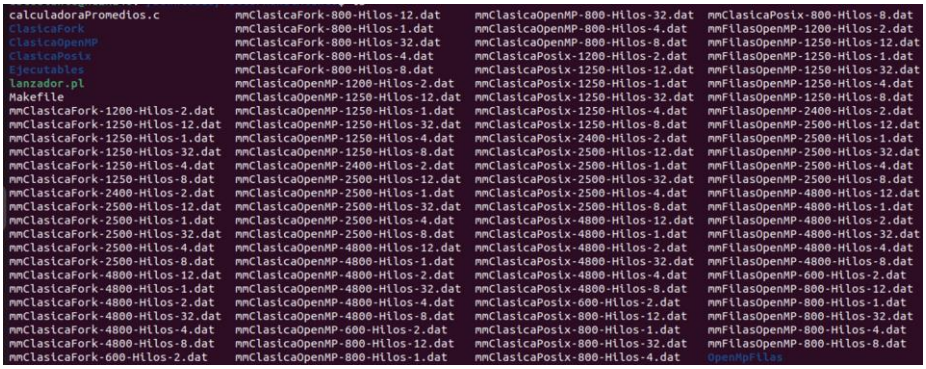
```
%Cpu0  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu1  :100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu2  : 99,3 us,  0,7 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
%Cpu3  : 99,7 us,  0,3 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  11961,1 total,    318,1 free,   3061,7 used,   8581,3 buff/cache
MiB Swap:   2048,0 total,   1550,5 free,    497,5 used.   8552,8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
3527097 estudia+  20   0  805060 431872   1280 S 396,0   3,5   8:17.43 mmClasicaPosix
```

Se puede verificar la correcta ejecución de los programas haciendo uso del comando top. Viendo el uso del CPU después de ejecutar el script de pearl y los procesos

de los programas en ejecución correspondientes a los tiempos de ejecución de los algoritmos.



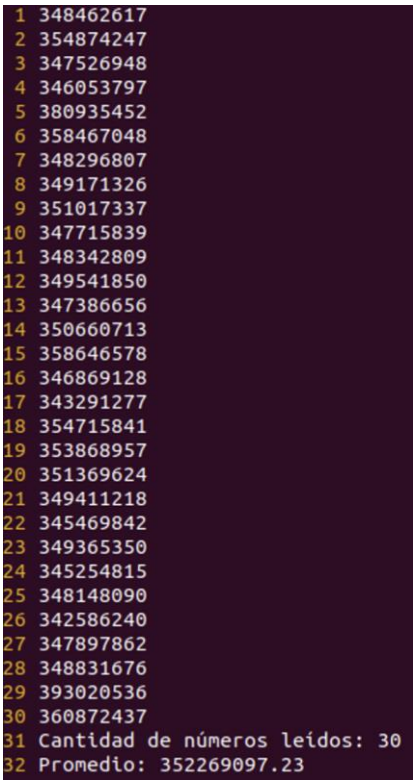
El finalizar el proceso se pueden encontrar todos los archivos .dat



El formato de cada archivo .dat consta de 30 líneas que contienen el tiempo de ejecución del programa, seguido del total de iteraciones del programa y su promedio. El promedio de los tiempos se calcula por medio del siguiente programa en C.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <nombre_archivo>\n", argv[0]);
        return 1;
    }

    FILE *archivo = fopen(argv[1], "r");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return 1;
    }

    int numero;
    long suma = 0;
    int contador = 0;

    // Leer números hasta que se acabe el archivo
    while (fscanf(archivo, "%d", &numero) == 1) {
        suma += numero;
        contador++;
    }

    fclose(archivo);

    if (contador == 0) {
        printf("El archivo no contiene números válidos.\n");
        return 0;
    }

    double promedio = (double)suma / contador;
    printf("Cantidad de números leídos: %d\n", contador);
    printf("Promedio: %.2f\n", promedio);

    return 0;
}
```

Ejemplo del contenido de un .dat generado:



El contenido de estos archivos .dat se transfiere a una hoja de cálculo donde se organiza la información en tablas para cada algoritmo, el tamaño de la matriz y el número de hilos utilizado en las 30 repeticiones, y calcula su tiempo medio. Quedando organizada de la siguiente manera:

CLÁSICA FORK (tiempos en microsegundos)

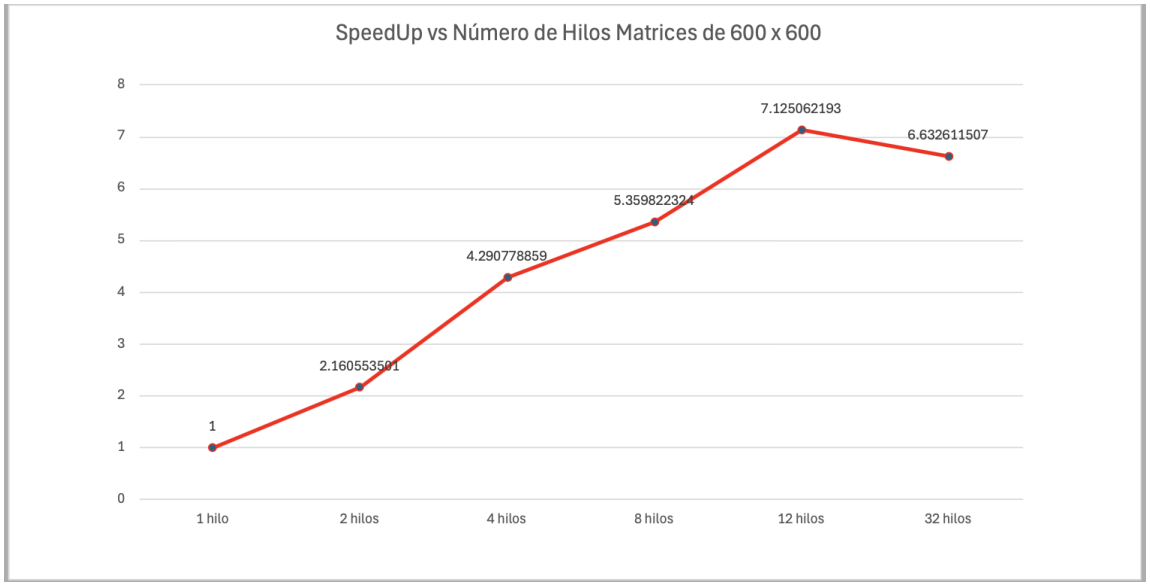
MATRIZ 600 X 600	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos	32 hilos
Repetición #1	780507	354575	174330	141634	121772	134297
Repetición #2	732207	347882	172615	164302	98188	130778
Repetición #3	740482	352830	175837	138510	105795	129361
Repetición #4	715919	1139019	171788	130556	104659	101014
Repetición #5	718793	353675	174623	125005	97192	105493
Repetición #6	725348	353254	172907	161176	126528	109813
Repetición #7	727808	348730	171022	134882	94121	123939
Repetición #8	736327	355785	176489	151404	104315	111427
Repetición #9	792565	352484	169095	138228	113045	104312
Repetición #10	747639	351549	134926	145608	98535	125214
Repetición #11	861569	347487	114162	142152	105809	103281
Repetición #12	713832	355360	191841	151830	102607	105091
Repetición #13	799163	359374	216741	137575	109409	103280
Repetición #14	754757	354781	913536	154410	116816	128070
Repetición #15	834981	345942	173294	140941	102144	106458
Repetición #16	1584517	344318	175367	131783	100849	120442
Repetición #17	793441	347936	174239	128395	104597	118811
Repetición #18	746268	348377	175418	135224	99976	126170
Repetición #19	753575	348470	174990	141500	104981	127140
Repetición #20	747521	349783	175819	124513	100725	110587
Repetición #21	789302	348347	195280	154953	107056	122770
Repetición #22	740680	349348	177330	163490	117052	120284
Repetición #23	721569	355575	176867	149042	111394	111246
Repetición #24	738643	352472	176344	127920	102658	121118
Repetición #25	822914	349136	183949	128766	93802	115514
Repetición #26	731890	344562	174174	142918	795401	99784
Repetición #27	741607	345641	175767	142738	99967	108876
Repetición #28	751120	354368	178240	129891	98064	98737
Repetición #29	814658	351108	174643	142890	164442	96243
Repetición #30	749784	358669	186786	156836	115464	115729
PROMEDIO	758983	351291	176887	141606	106523	114432

En la segunda hoja de cada Excel se encuentra el cálculo del SpeedUp teniendo en cuenta el número de hilos usados en ejecución, esto se calcula tomando el cociente entre el tiempo que se demora en ejecutar usando 1 hilo y el tiempo en X número de hilos.

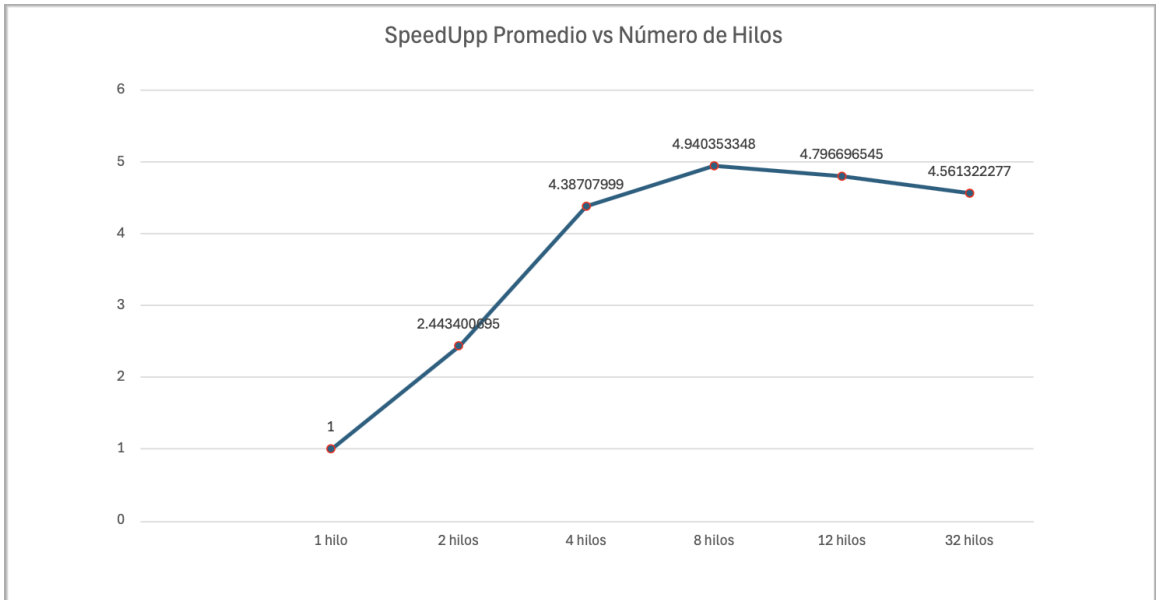
CLÁSICA FORK (tiempos en microsegundos)

# Hilos	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos	32 hilos
SpeedUp Matrices 600 x 600	1	2.1605535	4.29077886	5.35982232	7.12506219	6.63261151
Promedio Matrices 600 x 600	758983	351291	176887	141606	106523	114432
# Hilos	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos	32 hilos
SpeedUp Matrices 1200 x 1200	1	3.02582669	4.92405877	8.74194832	10.8982343	12.1885906
Promedio Matrices 1200 x 1200	7027661	2322559	1427209	803901	644844	576577
# Hilos	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos	32 hilos
SpeedUp Matrices 2400 x 2400	1	2.64924808	4.40459238	4.23808826	4.37778569	3.95013919
Promedio Matrices 2400 x 2400	79065283	29844424	17950647	18655884	18060565	20015822
# Hilos	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos	32 hilos
SpeedUp Matrices 4800 x 4800	1	2.32441766	4.35817327	2.59716444	1.86650092	1.34161919
Promedio Matrices 4800 x 4800	813397322	349935959	186637215	313186685	435787261	606280329

Para estos datos del SpeedUp sea crea una tabla para cada algoritmo y para cada tamaño de matriz utilizado, esto con el fin de tener una referencia visual de la mejora en rendimiento a medida que se utilizan más hilos.

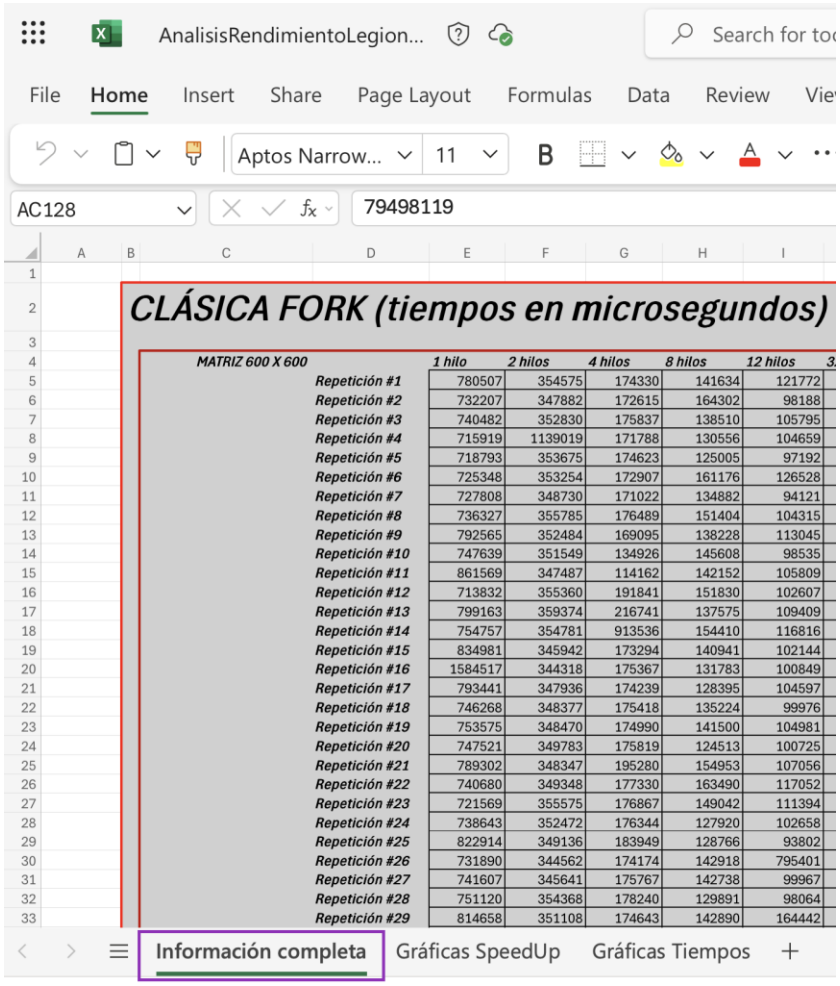


También se cuenta con una tabla que representa la media del SpeedUp en cada algoritmo, para todos los tamaños de matriz.



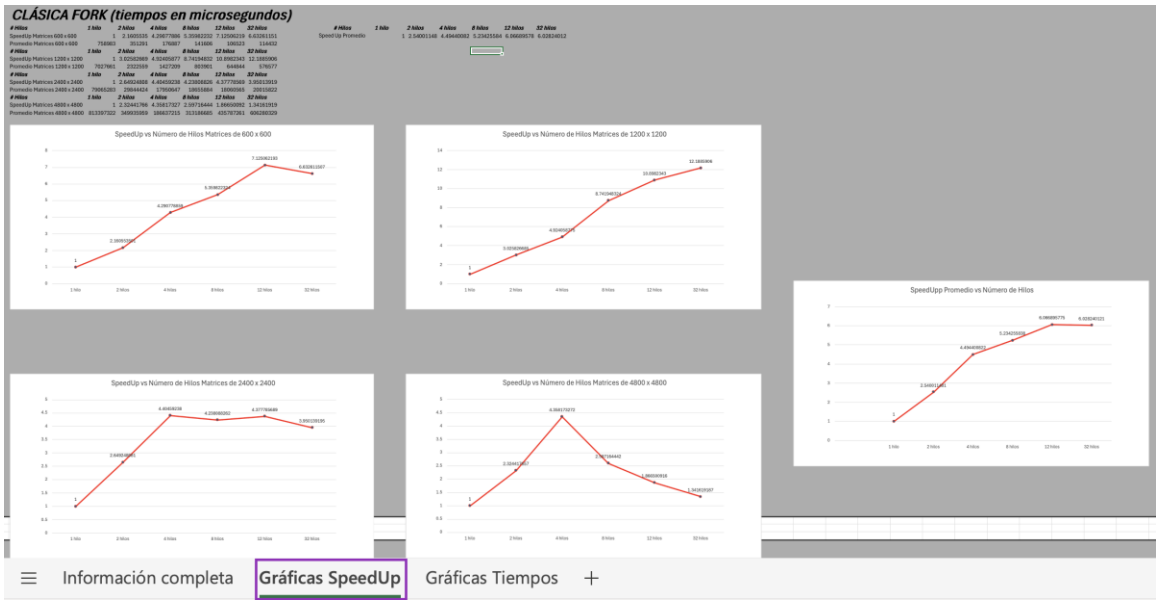
A continuación, se anexan las tablas de Excel de los resultados con la información recopilada de cada uno de los sistemas de cómputo utilizados, especificados en el punto 2.3. Y se muestra donde se encuentran los apartados con los análisis.

En la primera hoja de cada archivo Excel se encuentra la información del tiempo de ejecución para cada algoritmo, por cada número de hilos y por cada tamaño de matriz ejecutado.



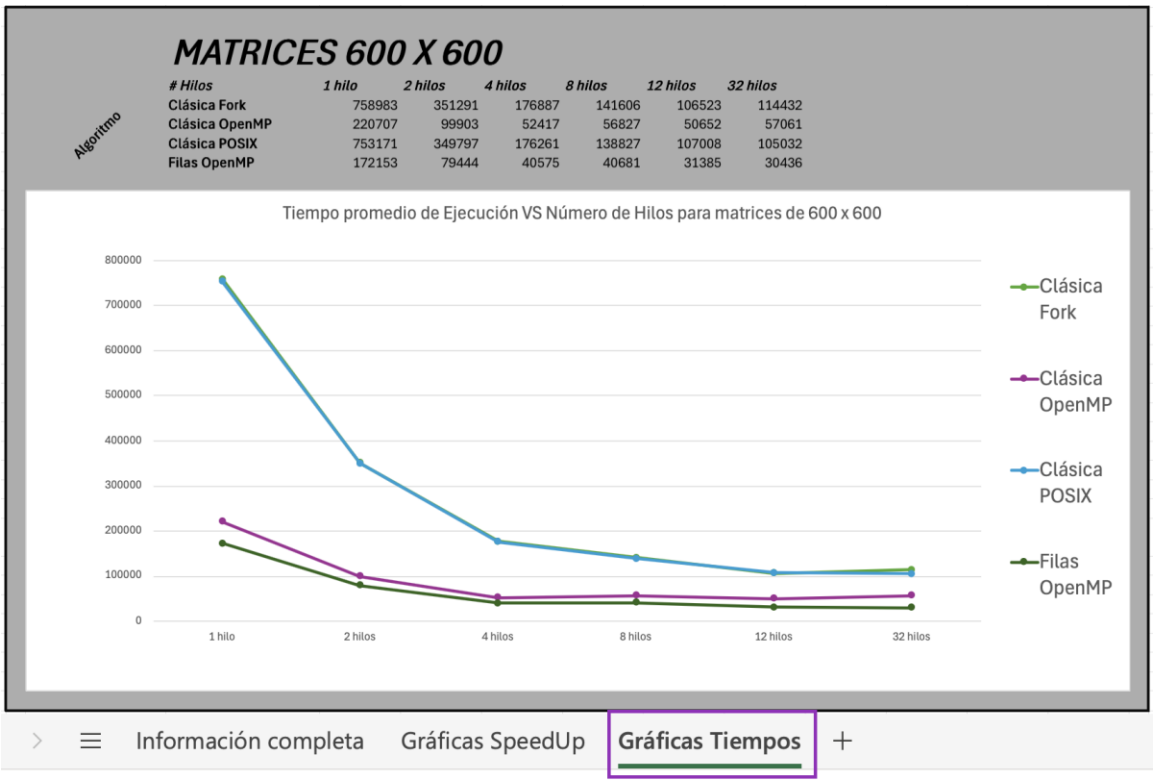
MATRIZ 600 X 600	1 hilo	2 hilos	4 hilos	8 hilos	12 hilos
Repetición #1	780507	354575	174330	141634	121772
Repetición #2	732207	347882	172615	164302	98188
Repetición #3	740482	352830	175837	138510	105795
Repetición #4	715919	1139019	171788	130556	104659
Repetición #5	718793	353675	174623	125005	97192
Repetición #6	725348	353254	172907	161176	126528
Repetición #7	727808	348730	171022	134882	94121
Repetición #8	736327	355785	176489	151404	104315
Repetición #9	792565	352484	169095	138228	113045
Repetición #10	747639	351549	134926	145608	98535
Repetición #11	861569	347487	114162	142152	105809
Repetición #12	713832	355360	191841	151830	102607
Repetición #13	799163	359374	216741	137575	109409
Repetición #14	754757	354781	913536	154410	116816
Repetición #15	834981	345942	173294	140941	102144
Repetición #16	1584517	344318	175367	131783	100849
Repetición #17	793441	347936	174239	128395	104597
Repetición #18	746268	348377	175418	135224	99976
Repetición #19	753575	348470	174990	141500	104981
Repetición #20	747521	349783	175819	124513	100725
Repetición #21	789302	348347	195280	154953	107056
Repetición #22	740680	349348	177330	163490	117052
Repetición #23	721569	355575	176867	149042	111394
Repetición #24	738643	352472	176344	127920	102658
Repetición #25	822914	349136	183949	128766	93802
Repetición #26	731890	344562	174174	142918	795401
Repetición #27	741607	345641	175767	142738	99967
Repetición #28	751120	354368	178240	129891	98064
Repetición #29	814658	351108	174643	142890	164442

En la segunda hoja se encuentran las gráficas SpeedUp con los contenidos explicados anteriormente.



Y en la tercera hoja se encuentran las gráficas de tiempo promedio por número de hilos para cada algoritmo. El cual nos permite tener una visión más general de cual

algoritmo es más eficiente y en cuantos hilos alcanza su punto ideal de rendimiento.



- Lenovo Legion Pro 7i: AnalisisPC_1
- Asus Gaming Tuf f15: AnalisisPC_2
- Lenovo IdeaPad Gaming 3: AnalisisPC_3
- Máquina Virtual: AnalisisVM
- ClusterHPC: AnalisisClusterHPC

Análisis de Resultados

1. Introducción al Análisis

Una vez evaluados los cuatro algoritmos con distintos tamaños de matrices y diferentes cantidades de hilos, se realizaron 30 repeticiones por cada caso, aplicando el principio de la ley de los grandes números. Para obtener resultados más estables, se descartaron los dos valores máximos y los dos mínimos, calculando el promedio de los tiempos restantes. Con base en esos datos, se determinó el SpeedUp obtenido al incrementar la cantidad de hilos y se analizaron los tiempos promedio de ejecución de cada algoritmo. A partir de ello, se estudiaron factores clave del hardware como la cantidad de núcleos lógicos y físicos, la jerarquía de caché, el ancho de banda de la memoria principal y el efecto del Hyper-threading junto con factores de software, como el tipo de paralelismo utilizado y los principios de localidad espacial y temporal. Finalmente, con toda esta información como base, se determinó cuál de los algoritmos ofrece el mejor desempeño en la mayoría de los escenarios para la multiplicación de matrices.

2. Análisis de escalabilidad con el número de hilos

A pesar de que los equipos utilizados contaban con procesadores de distintas especificaciones incluyendo variaciones en el número de núcleos físicos y lógicos, determinadas en parte por la presencia o ausencia de Hyper-threading se decidió probar

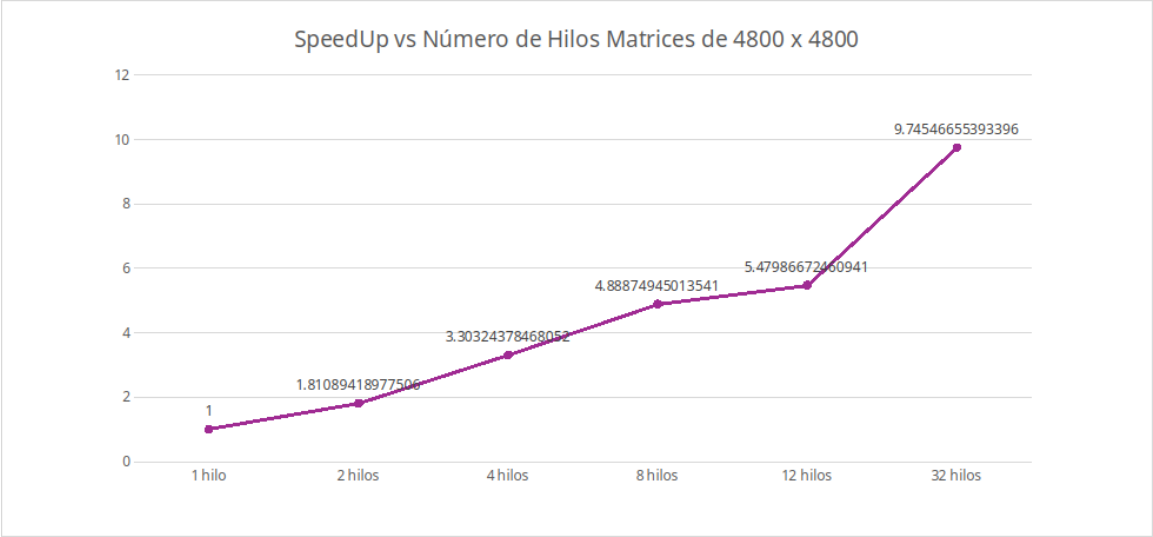
las mismas cantidades de hilos en todas las máquinas para mantener la congruencia de los resultados.

Tras construir y analizar las gráficas, se identificaron comportamientos relevantes, especialmente en la evolución del SpeedUp, el cual se vio limitado tanto por el overhead como por el punto máximo teórico dado por la cantidad de núcleos lógicos disponibles en cada equipo.

2.1 Incremento de rendimiento hasta el límite de cores lógicos

En la teoría, bajo un escenario ideal, contar con N núcleos trabajando en paralelo para resolver una misma tarea en lugar de un solo núcleo implicaría una reducción del tiempo de ejecución a 1/N del tiempo secuencial. Sin embargo, esta aproximación suele verse afectada por diversos factores prácticos como, por ejemplo, la creación de hilos que conlleva un costo adicional, pues cada hilo requiere reservar memoria y registrarse en el sistema operativo. De igual forma, el cambio de contexto entre hilos dentro de un mismo núcleo depende del planificador del sistema, sobre el cual no se tiene control directo. Además, un acceso excesivo a las memorias caché más cercanas al procesador puede provocar retrasos, ya que obliga a recurrir con mayor frecuencia a niveles de memoria más lejanos y lentos. Todos estos aspectos son los que comprenden el concepto del Overhead (entendido como el costo adicional necesario para que los hilos puedan operar en paralelo). Debido a este fenómeno, incluso al aumentar la cantidad de hilos en un factor M, el tiempo de ejecución no se reduce exactamente a 1/M del valor original, pues

el Overhead limita la eficiencia paralela. Tal como se observa en la *Gráfica 1,* el incremento del SpeedUp no crece con el mismo factor respecto al número de hilos utilizados. Por ejemplo, pasando de 1 hilo a 32 hilos, el SpeedUp es únicamente de 9.75

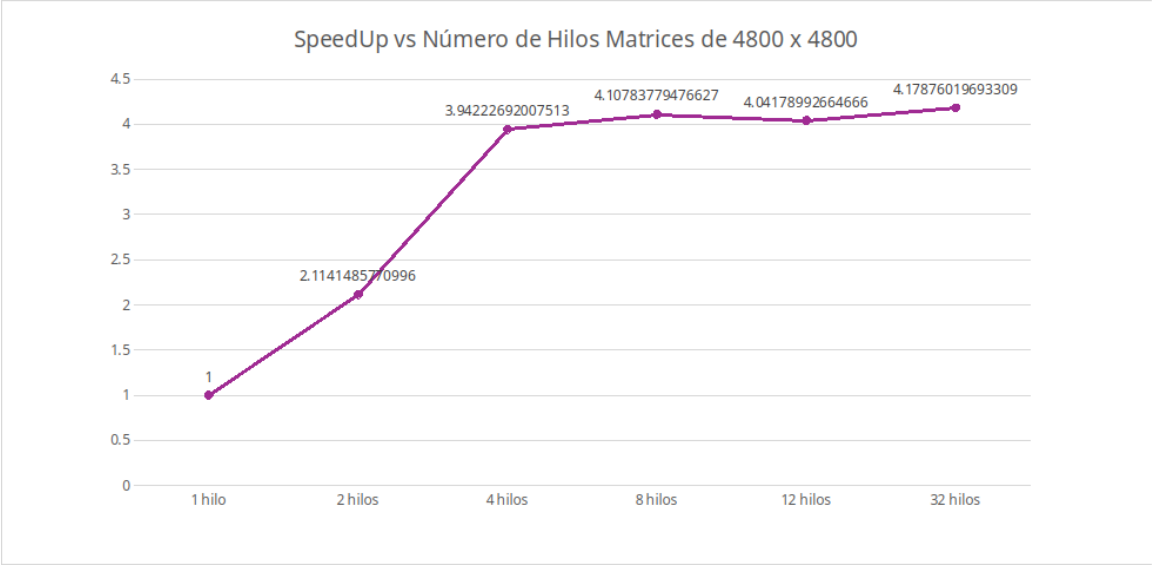


Gráfica 1. SpeedUP vs Número de hilos utilizando el algoritmo de multiplicación Filas OpenMP con matrices cuadradas de 4800 x 4800 en un equipo con 32 Cores Lógicos de procesador

2.2 Comportamiento al superar la cantidad de cores lógicos

A diferencia del inciso 2.1 del análisis, en este apartado se consideran los casos en los que la cantidad de hilos creados supera el número de núcleos lógicos disponibles en la máquina utilizada. El punto máximo eficiente ocurre cuando el número de hilos es igual al número de núcleos lógicos, es decir, en esa situación, cada hilo puede asignarse a un

núcleo sin generar conflictos, sin embargo, al superar ese límite, los hilos comienzan a competir por los mismos recursos de procesamiento, lo que provoca multiplexación entre ellos. Este fenómeno es una de las principales causas del Overhead, ya que los hilos excedentes dejan de trabajar de forma verdaderamente paralela y pasan a ejecutarse de manera concurrente no paralela. En términos simples, esto equivale a una secuencialidad disfrazada de paralelismo, con el agravante de que la multiplexación implica un trabajo adicional (Overhead). En consecuencia, crear más hilos que núcleos lógicos tiene, en el mejor de los casos, un efecto neutro sobre el rendimiento, y en otros escenarios puede incluso reducir la eficiencia del sistema. Tal como se muestra en la *Gráfica 2*, a partir de los cuatro hilos, el incremento del SpeedUp se vuelve mínimo, pudiendo justificarse como pequeñas desviaciones de medición, sin embargo, es evidente la tendencia hacia un efecto neutro o, en algunos casos, negativo.



Gráfica 2. SpeedUP vs Número de hilos utilizando el algoritmo de multiplicación

Filas OpenMP con matrices cuadradas de 4800 x 4800 en un equipo con 4 Cores Lógicos de procesador

3. Impacto del hardware

El hardware constituye un factor determinante al comparar el desempeño entre diferentes máquinas, ya que componentes como las memorias caché, la memoria RAM, el procesador y sus características internas influyen de manera directa y significativa en el comportamiento de los algoritmos. Estas variaciones afectan los resultados obtenidos bajo las distintas configuraciones utilizadas durante el experimento, haciendo que cada equipo responda de forma particular frente a las mismas condiciones de prueba.

3.1 Cachés y principio de localidad espacial

Las memorias caché desempeñan un papel fundamental en el análisis del comportamiento del sistema desde la perspectiva del hardware. En la jerarquía de memoria se distinguen los niveles L1, L2 y L3, los cuales se caracterizan por su proximidad al procesador. Estas memorias permiten un acceso extremadamente rápido a los datos, aunque presentan una capacidad limitada. En general, cuanto mayor es la velocidad de la memoria caché, menor es su capacidad de almacenamiento y menor el número de núcleos que la comparten. En ciertos casos, el tamaño de las matrices utilizadas en el experimento supera la capacidad total de la caché, obligando al sistema a acceder a la memoria RAM, lo que significa ralentizar el proceso de acceso a los datos y

por ende, el tiempo total de ejecución. Dado que las matrices se componen de datos del tipo double, cada uno con un tamaño de 8 bytes, es posible estimar la carga de trabajo total mediante la siguiente expresión:

$$Tamaño\ Total\ (bytes) = n \cdot n \cdot 8 \cdot 3$$

donde n representa la dimensión de la matriz, el factor 8 corresponde al tamaño en bytes de cada elemento y el factor 3 considera las dos matrices multiplicadas más la matriz resultante. El resultado obtenido puede convertirse a KiB dividiendo entre 1000, y a MiB dividiendo entre 1,000,000, lo que permite dimensionar el peso aproximado de la carga de trabajo empleada durante el experimento. Lo anterior se hace aún más evidente en las matrices de tamaño 2400 y 4800, ya que, al aplicar la fórmula descrita previamente, se determinó que su peso en memoria supera la capacidad de caché de cualquiera de las cinco máquinas utilizadas en el experimento.

3.1.1 Diferencias en el SpeedUp entre matrices grandes y pequeñas.

Para lo que se va a analizar a continuación, se consideró, en primer lugar, un algoritmo caché-friendly (multiplicación por filas con OpenMP) ejecutado en la máquina Lenovo Legion Pro 7i, la cual cuenta con la mayor capacidad de caché entre los equipos evaluados.

En teoría, podría esperarse que el speedup obtenido al incrementar el número de hilos fuera más notable en matrices pequeñas como las de 600 o 1200 que en matrices de mayor tamaño como 2400 o 4800, debido a que las primeras pueden mantenerse

completamente dentro del caché, mientras que las segundas requieren acceder en gran medida a la memoria RAM. No obstante, los resultados experimentales no confirman plenamente esta hipótesis. Esto podría deberse a distintos factores, desde errores de medición hasta la posibilidad de que el efecto se vuelva más evidente con matrices de dimensiones aún mayores. Con los datos que tomamos, las diferencias no son evidentes al comparar ejecuciones del mismo algoritmo, sino más bien al contrastar algoritmos con diferentes patrones de acceso a memoria, como ocurre entre las multiplicaciones clásicas con sus distintos métodos de paralelización y la multiplicación por filas con OpenMP, donde la segunda presenta una mayor capacidad para aprovechar la jerarquía de caché.

3.1.2 Diferencias de tiempo entre diferentes algoritmos.

En este apartado se analiza la ventaja significativa del algoritmo de multiplicación de matrices por filas utilizando la transpuesta paralelizando con OpenMP.

La multiplicación clásica de matrices recorre una de las matrices por columnas, lo que implica realizar saltos amplios en memoria. Según el principio de localidad espacial, este comportamiento resulta ineficiente, ya que se cargan en la caché múltiples datos contiguos que no serán utilizados de inmediato, ocupando espacio innecesario. Con el tiempo, esta falta de aprovechamiento provoca un mayor número de accesos a la memoria principal (RAM), cuyo tiempo de acceso es considerablemente más alto que el de las memorias caché. Este fenómeno se ilustra en la *Foto 1*, donde el recorrido por columnas (representado por las flechas rojas) provoca que las cachés se llenen de información que

no será reutilizada de forma inmediata (indicada en casillas azules), forzando así a los hilos o procesos a acceder repetidamente a la memoria RAM.

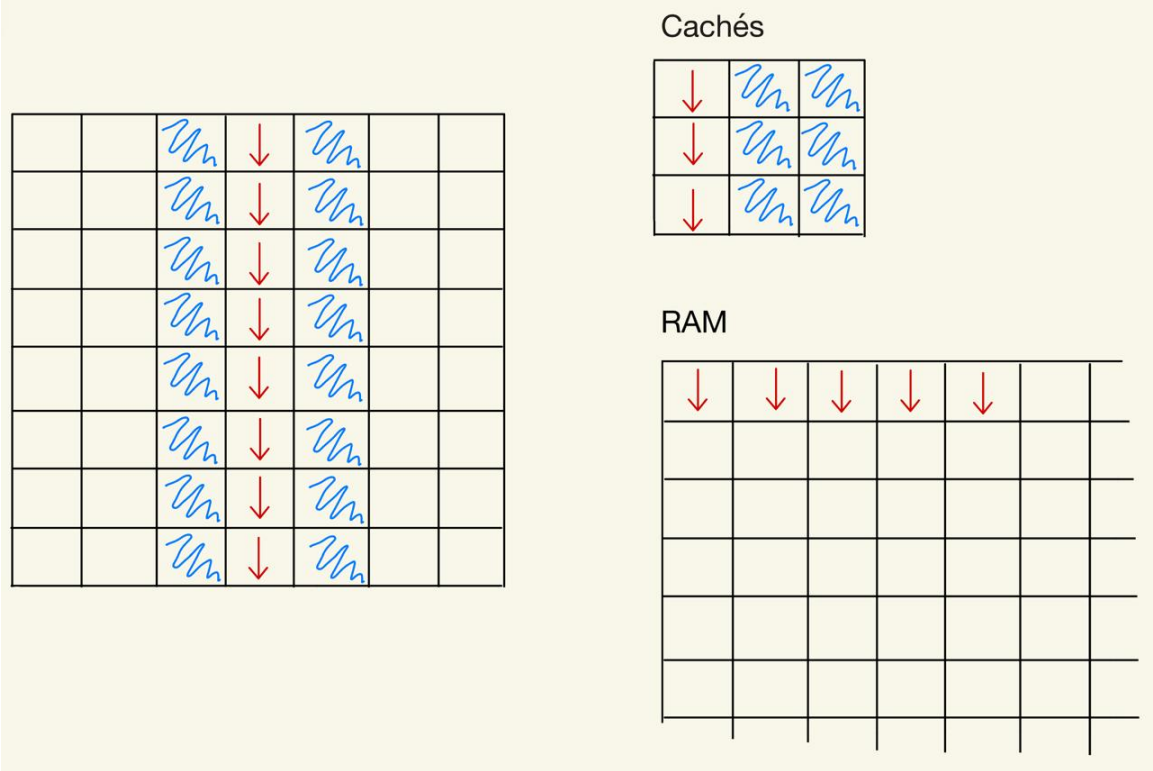


Foto 1. Ejemplo de uso de jerarquía de memoria recorriendo una matriz por columnas

Por otro lado, el algoritmo de multiplicación por filas utilizando la matriz transpuesta permite un aprovechamiento óptimo de las memorias caché, ya que los datos cargados

son utilizados casi de inmediato. Este patrón de acceso favorece el principio de localidad espacial, evitando que se desperdicien recursos de memoria y reduciendo la necesidad de recurrir a la RAM. En la *Foto 2* se aprecia claramente este comportamiento, las flechas rojas representan el recorrido por filas, donde cada dato accedido es procesado y almacenado de forma contigua, garantizando un flujo de ejecución mucho más eficiente.

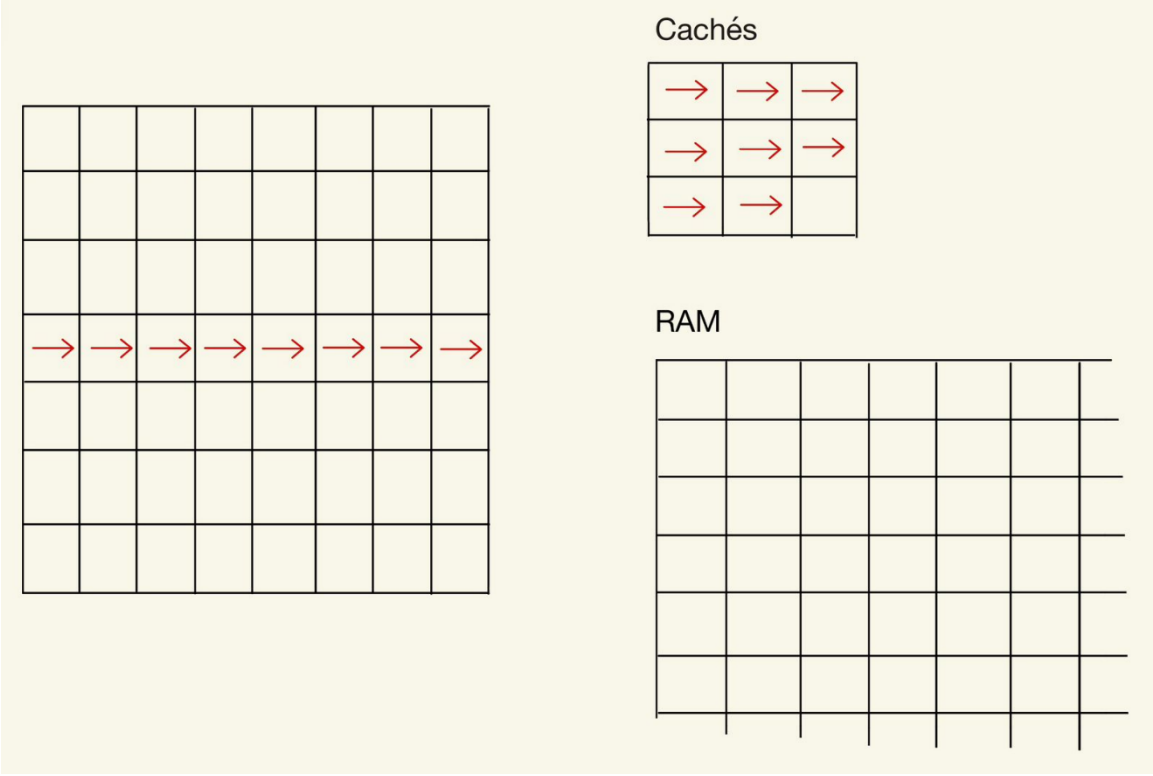


Foto 2. Ejemplo de uso de jerarquía de memoria recorriendo una matriz por filas

3.2 Ancho de banda de la memoria RAM

Cabe aclarar que, aunque existen matrices cuyo tamaño en bytes supera la capacidad total de las memorias caché, y por tanto requieren acceder a la memoria RAM, los algoritmos que aprovechan el caché (caché-friendly) logran minimizar dicho acceso y maximizar el uso de los niveles L1, L2 y L3. En contraste, los algoritmos que no aprovechan eficientemente la jerarquía de memoria se ven obligados a acceder con mayor frecuencia a la RAM, la cual, a pesar de contar con un amplio ancho de banda y ser capaz de transferir decenas de millones de bytes por segundo, no puede igualar la alta eficiencia de las memorias caché.

3.3 Velocidad del procesador y Hyper-Threading

Considerar las características del procesador como un factor determinante en los tiempos de ejecución resulta esencial para esta investigación. La frecuencia del procesador, expresada en GHz, indica la cantidad de ciclos de reloj que el procesador puede ejecutar por segundo. Este parámetro es especialmente relevante si se tiene en cuenta que muchas operaciones como la multiplicación o el acceso a memoria RAM requieren más de un ciclo de reloj para completarse. En contraste, operaciones más simples, como la suma, pueden realizarse en un solo ciclo. Por lo tanto, un procesador con una frecuencia de 2.6 GHz tendrá una menor capacidad de procesamiento en

términos tiempo que otro que opere a 5.4 GHz, lo que se traduce en diferencias significativas en la eficiencia y el rendimiento de los algoritmos ejecutados.

Otro factor relevante es la tecnología Hyper-Threading. Tal como se explicó en el marco teórico, esta tecnología permite que un núcleo físico del procesador simule la existencia de dos núcleos lógicos, aprovechando de forma más eficiente los recursos del núcleo físico. En esencia, el Hyper-Threading permite que, mientras un hilo lógico se encuentra esperando la respuesta de la memoria caché o de la memoria RAM, el otro hilo pueda utilizar los recursos disponibles para continuar la ejecución. Sin embargo, esto no implica que el rendimiento se duplique, ya que ambos hilos comparten los mismos recursos físicos del núcleo. En la práctica, esta tecnología puede incrementar el rendimiento del sistema, especialmente en programas donde los accesos a memoria representan un cuello de botella. No obstante, si los accesos a memoria son muy rápidos o poco frecuentes, los dos hilos lógicos tenderán a competir por los mismos recursos, lo que puede provocar incluso una disminución en el rendimiento general. Estas características son abordadas en el inciso 5 del análisis donde se estudian los comportamientos anómalos de las gráficas.

4. Comparativa entre algoritmos

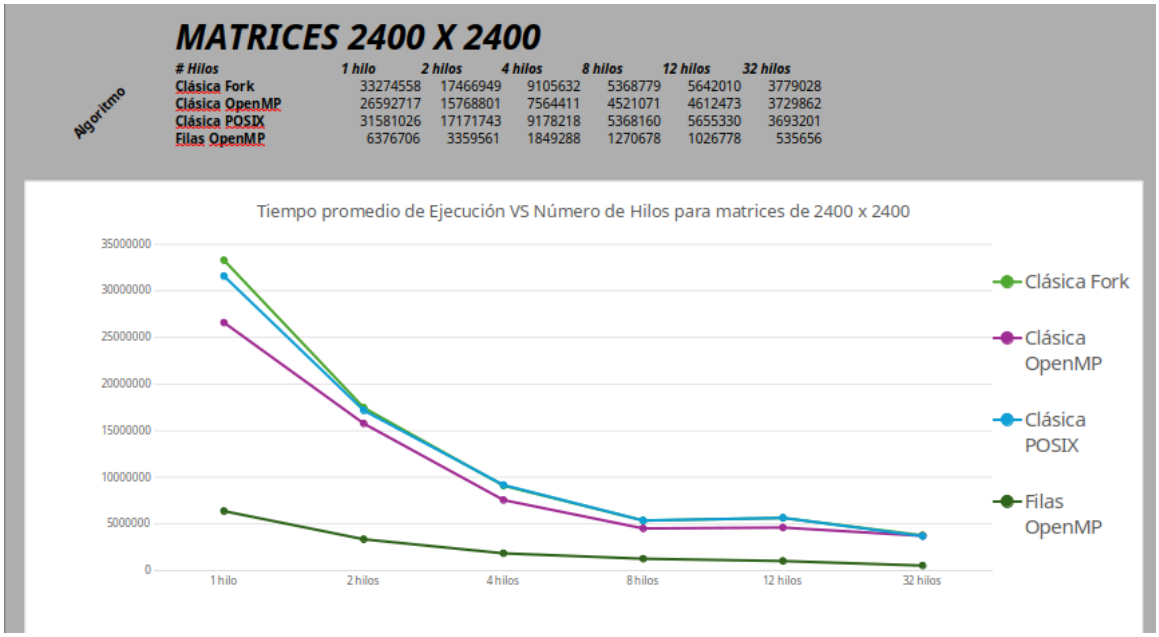
Las estrategias de multiplicación de matrices varían significativamente según el grado de aprovechamiento de los principios de localidad espacial y temporal, así como de las técnicas empleadas para paralelizar el proceso de multiplicación.

4.1 Multiplicación clásica usando Fork

En general, el algoritmo de multiplicación clásica implementado con Fork es el más lento entre todos los evaluados. Aunque su rendimiento suele ser similar al del algoritmo de multiplicación clásica basado en pthreads POSIX, se evidencia una diferencia suficiente en los tiempos de ejecución para concluir esto. Fork requiere más tiempo para completar la operación. Esto se debe a que los procesos creados mediante Fork son más pesados en términos de uso de memoria y generan un mayor costo en los cambios de contexto.

4.2 Multiplicación clásica usando POSIX (pthreads)

El algoritmo basado en los pthreads , aunque más rápido que el implementado con Fork debido a la menor carga y mayor facilidad de gestión de los hilos, tiende a presentar tiempos de ejecución muy similares. Esta cercanía en el rendimiento se refleja en la *Gráfica 3*, donde las curvas de tiempo de ambos algoritmos llegan a solaparse.

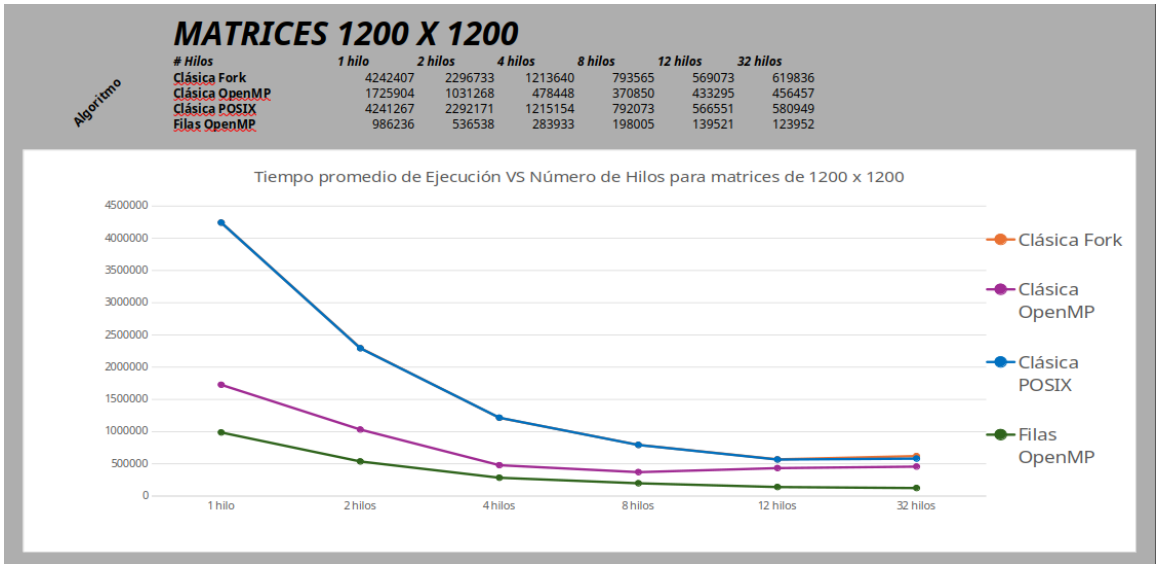


Gráfica 3. Ejemplo de solapamiento entre las curvas de tiempo del algoritmo Clásico Fork y del algoritmo clásico POSIX con matrices cuadradas de 2400 x 2400 en un equipo con 32 cores lógicos de procesamiento

4.3 Multiplicación clásica usando OpenMP

El algoritmo de multiplicación clásica implementado con OpenMP demuestra ser considerablemente más eficiente que las versiones clásicas basadas en Fork y pthreads en la mayoría de los casos. Esto se debe a que las secciones paralelas gestionadas por OpenMP son automatizadas, lo que permite una administración más eficiente de los hilos. Además, OpenMP presenta un mayor nivel de optimización interna en comparación con

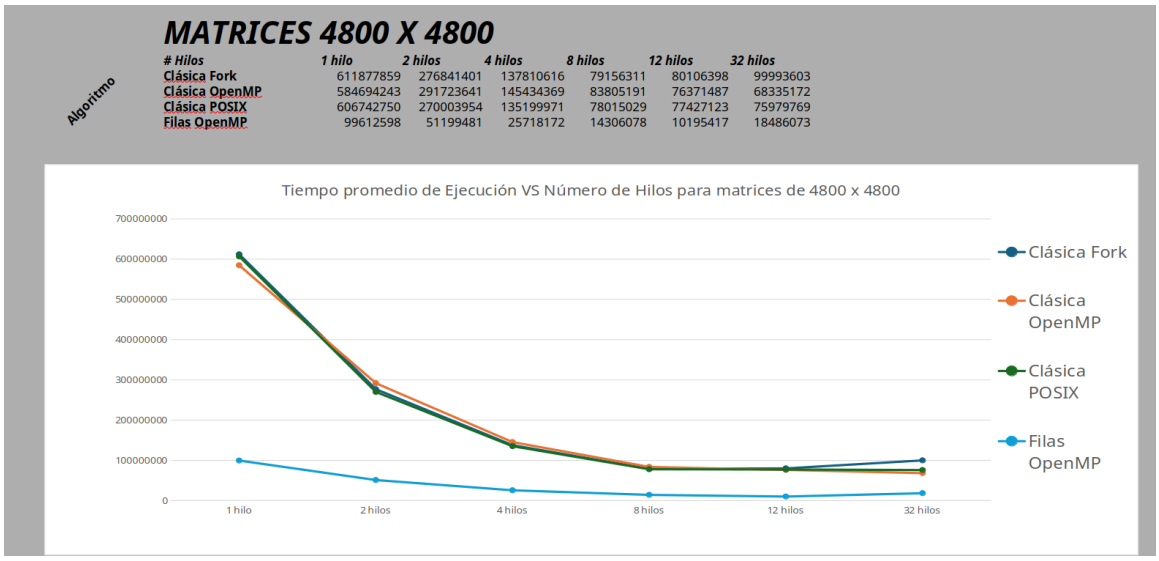
las implementaciones de Fork y pthreads desarrolladas en este trabajo. Esta ventaja explica la superioridad observada respecto a los otros dos algoritmos clásicos de multiplicación. Un ejemplo muy claro se muestra en la *Gráfica 4*, donde las curvas correspondientes a Fork y pthreads se solapan, mientras que el algoritmo clásico con OpenMP evidencia un rendimiento notablemente superior.



Gráfica 4. Ejemplo de superación notable por parte del algoritmo clásico utilizando OpenMP respecto a los otros dos algoritmos clásicos con matrices 1200 x 1200 en un equipo con 12 cores lógicos de procesamiento

4.4 Multiplicación por Filas OpenMP (con transpuesta)

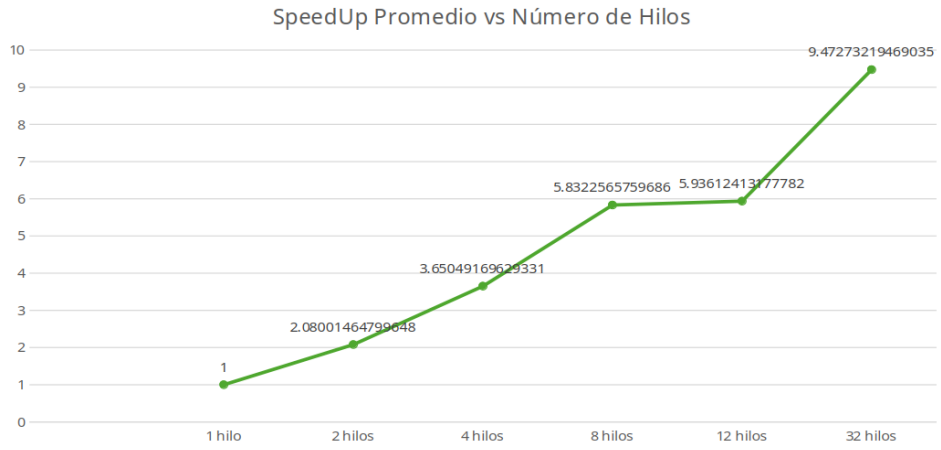
El algoritmo de multiplicación por filas utilizando la matriz transpuesta se consolida, sin lugar a duda, como el más eficiente de todos. Esta conclusión se sustenta no solo en los resultados experimentales donde consistentemente registró los tiempos de ejecución más bajos con una diferencia considerable, sino también en el análisis teórico, que evidencia su excelente aprovechamiento de los cachés gracias a la aplicación efectiva del principio de localidad espacial. Un ejemplo particularmente bueno se observa en la *Gráfica 5*, donde el algoritmo basado en la transpuesta mantiene un rendimiento significativamente superior, independientemente de la cantidad de hilos utilizados.



Gráfica 5. Ejemplo de superación absoluta respecto a los otros 3 algoritmos en matrices de 4800 x 4800 usando el Cluster HPC (máquina con 20 cores lógicos de procesamiento)

5. Interpretación de comportamientos anómalos

En algunos casos, se observa una disminución en la eficiencia de los algoritmos al incrementar la cantidad de hilos, incluso cuando aún existen núcleos lógicos disponibles en el sistema. Un ejemplo general de este fenómeno se muestra en la *Gráfica 6*.



Gráfica 6. SpeedUp general Promedio vs Número de hilos utilizando el algoritmo de multiplicación Clásico POSIX en un equipo con 32 Cores Lógicos de procesador

Identificamos que este fenómeno es recurrente en equipos que incorporan arquitecturas híbridas y tecnología Hyper-Threading. En nuestro experimento, el Hyper-Threading mostró un efecto neutro o incluso ligeramente perjudicial en algunos casos. Esto se debe a que, como se discutió en el inciso 3.3 del análisis, los accesos a memoria que requieren mucha espera no son frecuentes en nuestro caso, por tanto, los hilos lógicos terminan compitiendo por los mismos recursos físicos dentro de un mismo núcleo físico, lo que reduce la eficiencia. Por otra parte, en sistemas con arquitectura híbrida, la

distinción entre los núcleos Performance (más rápidos pero de mayor consumo energético) y los núcleos Efficiency (más lentos y de menor consumo) resulta fundamental. Durante el procesamiento, los núcleos Performance se saturan primero, y al incorporarse los núcleos Efficiency, estos ralentizan la ejecución global, es decir, mientras los Performance finalizan sus tareas, deben esperar a que los Efficiency concluyan las suyas, lo que genera un aporte nulo o incluso negativo al rendimiento total. Este comportamiento se evidencia claramente en la *Gráfica 6*, donde el SpeedUp muestra una ganancia prácticamente nula al pasar de 8 a 12 hilos. Es importante resaltar que dicha gráfica corresponde a la ejecución en un equipo con 24 núcleos físicos 8 Performance y 16 Efficiency, confirmando así la saturación inicial de los núcleos Performance.

Conclusiones

El estudio permitió evidenciar que la eficiencia de los algoritmos de multiplicación de matrices depende directamente de la manera en que se aprovechan los principios de localidad espacial y temporal, así como de la estrategia de paralelización utilizada. Entre las implementaciones clásicas, el uso de Fork presentó los mayores tiempos de ejecución debido al costo de creación de procesos y al alto consumo de memoria, mientras que pthreads mostró un rendimiento ligeramente mejor al aprovechar hilos más livianos y de gestión más eficiente. Sin embargo, ambos algoritmos demostraron limitaciones frente a entornos de ejecución intensivos.

La implementación con OpenMP (multiplicación clásica) logró una mejora significativa en el rendimiento respecto a los otros algoritmos de multiplicación clásica. Esto se debe a la automatización del manejo de hilos y a las optimizaciones internas que ofrece la API, reduciendo el overhead y mejorando la distribución de la carga de trabajo. No obstante, fue el algoritmo OpenMP por filas el que alcanzó los mejores resultados generales, evidenciando una optimización superior del uso de la jerarquía de memoria al aprovechar de forma efectiva el principio de localidad espacial de los datos.

En cuanto a la escalabilidad, se observó que el incremento del número de hilos mejora el rendimiento solo hasta alcanzar el límite de núcleos lógicos disponibles. Superar este umbral genera sobrecarga y competencia por recursos, disminuyendo la eficiencia. Además, en sistemas con arquitecturas híbridas y tecnología Hyper-Threading, el

rendimiento no siempre aumenta con la cantidad de hilos, ya que los núcleos lógicos pueden competir por los mismos recursos físicos o ralentizarse al sincronizar núcleos de distinto tipo (Performance y Efficiency).

Finalmente, los resultados confirman que el rendimiento óptimo no se alcanza únicamente incrementando la cantidad de hilos, sino mediante un equilibrio entre el diseño del algoritmo, el patrón de acceso a memoria y las características del hardware. El algoritmo de multiplicación por filas con OpenMP se consolida como la mejor alternativa, al ofrecer el mayor aprovechamiento de los recursos computacionales y el menor tiempo de ejecución en todos los escenarios evaluados.

Bibliografía

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Linux Man Pages. (2024). pthreads(7). Recuperado de https://man7.org/linux/man-pages/man7/pthreads.7.html

  Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems (4th ed.). Pearson.
- Linux Man Pages. (2024). fork(2). Recuperado de https://man7.org/linux/man-pages/man2/fork.2.html
- GNU Project. (2024). The GNU C Library – POSIX Threads. Recuperado de https://www.gnu.org/software/libc/manual
- OpenMP Architecture Review Board. (2023). OpenMP Application Programming Interface v5.2. Recuperado de https://www.openmp.org
- Hennessy, J. L., & Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.
- Linux Man Pages. (2024). gettimeofday(2). Recuperado de https://man7.org/linux/man-pages/man2/gettimeofday.2.html
- Hennessy, J. L., & Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.
- Intel Corporation. (2023). Intel® Hyper-Threading Technology. Recuperado de https://www.intel.com
- Love, R. (2010). Linux Kernel Development (3rd ed.). Addison-Wesley.

- Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities.* AFIPS Conference Proceedings, 30, 483–485. https://doi.org/10.1145/1465482.1465560
- Wackerly, D., Mendenhall, W., & Scheaffer, R. (2014). *Mathematical statistics with applications* (7th ed.). Cengage Learning.
- Pacheco, P. (2011). *An introduction to parallel programming*. Morgan Kaufmann.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems* (4th ed.). Pearson.
- Intel. (2021). *12th Gen Intel Core processor architecture overview (Alder Lake)*. Intel Developer Zone. https://www.intel.com
- Linux Manual Page. (n.d.). *sched(7) — Linux manual page*. https://man7.org/linux/man-pages/man7/sched.7.html