Investigación

a. Principales APIs en la industria para programación multiproceso con memoria compartida

En la industria actual destacan varias APIs y librerías para la programación multiproceso con memoria compartida. OpenMP es la más utilizada en el cómputo de alto rendimiento, gracias a que permite paralelizar código mediante directivas de compilador (#pragma), facilitando la adopción de paralelismo sin cambios radicales en el código fuente (Arif & Vandierendonck, 2015). La librería estándar de C++ ofrece std::thread para el manejo directo de hilos, así como políticas de ejecución paralela desde C++17 (std::execution::par), lo que aporta mayor flexibilidad aunque puede aumentar la complejidad del desarrollo (Singanaboina, 2024).

Además, existen opciones como HPX, que implementa un modelo de ejecución asincrónica y conforme a los estándares C++, con un enfoque en escalabilidad y reducción de sobrecarga. Intel TBB se centra en un modelo basado en tareas con balanceo dinámico de carga, mientras que Taskflow introduce un enfoque moderno basado en grafos de tareas para expresar dependencias. En general, las diferencias clave radican en el nivel de abstracción: OpenMP prioriza la simplicidad, std::thread ofrece control fino, y librerías como HPX o TBB facilitan la programación de algoritmos paralelos de mayor complejidad con mejor escalabilidad (Wang, Gao, Fang, Huang & Wang, 2021).

b. Funcionalidad de OMP_NUM_THREADS y omp_get_max_threads()

En OpenMP, la variable de entorno OMP_NUM_THREADS permite especificar desde fuera del código el número de hilos a emplear en regiones paralelas, ofreciendo un control flexible sobre el grado de paralelismo. Por su parte, la función omp_get_max_threads() devuelve el número máximo de hilos que el entorno puede usar en una región paralela, lo cual depende del valor fijado por OMP_NUM_THREADS o, en su defecto, de los núcleos disponibles en el sistema (Wang et al., 2021).

c. Compilación de programas con OpenMP en GCC y Clang

Para compilar programas con OpenMP, los compiladores deben estar configurados para reconocer las directivas de compilador. En GCC y Clang esto se logra añadiendo la bandera -fopenmp al comando de compilación, lo que habilita el soporte para paralelismo. Por ejemplo: gcc -fopenmp programa.c -o programa. Además, es posible controlar opciones adicionales como la afinidad de los hilos o el número máximo de threads mediante variables de entorno como OMP_NUM_THREADS (Arif & Vandierendonck, 2015).

d. Estándares y opciones de compilador para std::thread

El uso de std::thread requiere únicamente un compilador que cumpla con el estándar C++11 o superior, ya que la interfaz de hilos se introdujo en esa versión. En cambio, las políticas de ejecución paralela de la STL (std::execution::par, std::execution::par_unseq) fueron añadidas en C++17 y C++20, por lo que se necesita compilar con estándares más recientes, empleando banderas como -std=c++17 o -std=c++20 en GCC y Clang. En algunos entornos, como Linux

con GCC, la implementación interna de estas políticas puede usar librerías como Intel TBB como backend de hilos (Singanaboina, 2024).

e. Pragma y directivas específicas de OpenMP

Un pragma es una directiva de compilador que modifica el comportamiento del programa en tiempo de compilación. En OpenMP, estas directivas se utilizan para indicar cómo paralelizar el código. Por ejemplo, #pragma omp parallel crea una región paralela en la que múltiples hilos ejecutan un bloque de código de forma simultánea. La directiva #pragma omp single asegura que un fragmento de código dentro de una región paralela sea ejecutado únicamente por un hilo, mientras que #pragma omp for distribuye las iteraciones de un bucle entre los hilos del equipo. Finalmente, la cláusula reduction permite combinar resultados parciales de varios hilos en una única variable compartida, típica en operaciones como sumas o productos acumulativos (Arif & Vandierendonck, 2015).

f. Mecanismos de sincronización en OpenMP y std::thread

La sincronización en OpenMP se logra mediante diferentes constructores. Uno de los más comunes son las barreras, que obligan a todos los hilos a esperar en un punto antes de continuar. Existen también secciones críticas con #pragma omp critical, que aseguran exclusión mutua en el acceso a recursos compartidos, y operaciones atómicas con #pragma omp atomic, que permiten modificar variables de manera segura con menor sobrecarga que una región crítica completa (Wang et al., 2021).

En C++, el control de sincronización con std::thread se logra a través de primitivas de la librería estándar como std::mutex, que asegura exclusión mutua, y std::lock_guard, que automatiza la gestión de locks. Además, std::condition_variable permite coordinar hilos mediante notificaciones y esperas condicionales, lo que resulta útil en patrones de productor-consumidor o colas compartidas (Singanaboina, 2024).

g. Medición del tiempo de ejecución en OpenMP y std::thread

Medir correctamente el tiempo de ejecución en entornos paralelos requiere metodologías cuidadosas. En OpenMP, se pueden utilizar micro-benchmarks que analizan el costo de cada directiva, además de funciones como omp_get_wtime(), que devuelve el tiempo de pared transcurrido y permite calcular diferencias de tiempo entre secciones de código. Para reducir ruido en la medición se recomienda ejecutar varias iteraciones y fijar la afinidad de los hilos a núcleos específicos (Wang et al., 2021).

En programas con std::thread, lo más común es emplear la librería std::chrono, midiendo el tiempo antes y después de ejecutar los hilos y calculando promedios en múltiples ejecuciones. También se evalúa la eficiencia paralela, definida como la relación entre el speedup alcanzado y el número de núcleos utilizados, lo que permite analizar qué tan bien se aprovecha el hardware (Singanaboina, 2024).

h. Generación de números seudoaleatorios thread-safe

En entornos multihilo, la generación de números pseudoaleatorios presenta el riesgo de condiciones de carrera si todos los hilos comparten un único generador global. Para evitarlo, cada hilo debe contar con su propia instancia del generador, inicializada con semillas distintas, o bien emplear generadores diseñados para ser thread-safe. En C++, esto puede lograrse asociando un motor aleatorio (std::mt19937) a cada hilo mediante variables locales de tipo thread_local. Otra alternativa es utilizar mecanismos de sincronización como std::mutex alrededor del generador, aunque esta estrategia introduce sobrecarga y limita la escalabilidad (Arif & Vandierendonck, 2015).

Descripción del Hardware empleado en los experimentos

Los experimentos de este proyecto se realizaron en un equipo con las siguientes características de hardware y sistema operativo:

- Procesador: Intel® Core™ i3-1005G1 CPU @ 1.20GHz
- Número de núcleos físicos: 2
- Número de núcleos lógicos: 4 con HyperThreading
- Arquitectura del procesador: x86_64, con soporte para ejecución multihilo
- Memoria RAM total: 3.6 GiB
- Memoria RAM disponible durante los experimentos: ~2.7 GiB
- Sistema operativo: Linux, versión del kernel 6.11.0-29-generic

Este hardware permitió evaluar el comportamiento de los algoritmos de cálculo de histogramas paralelos tanto con OpenMP como con std::thread, considerando distintos números de hilos y variantes de paralelización. A pesar de la limitación de memoria (3.6 GiB), los experimentos se ejecutaron correctamente para vectores de hasta 5 millones de elementos, lo que asegura la validez de las mediciones de tiempo, eficiencia y métricas de hardware obtenidas mediante perf.

Estrategias implementadas y análisis de los resultados obtenidos

En el desarrollo de este proyecto se implementaron tres estrategias principales para el cálculo paralelo del histograma, siguiendo las indicaciones de la sección 4. Cada estrategia se evaluó con distintos números de hilos y repeticiones, registrando métricas de tiempo, CPU y eventos de hardware mediante perf.

1. Estrategias implementadas

Histogramas privados por hilo + reducción final (private)

Cada hilo mantiene su propio histograma local y al final se realiza una reducción para combinar los resultados en el histograma global.

- Justificación: Minimiza la contención entre hilos, ya que cada hilo escribe en su propia memoria. Esto reduce el número de fallos de caché y permite un escalado más lineal.
- Implementación en OpenMP: Cada hilo tiene un array local y se usa #pragma omp parallel con reducción manual al final.
- Implementación en std::thread: Cada thread tiene un vector local que se combina en un paso final usando sumas atómicas o adición directa, evitando locks frecuentes.

Histograma global protegido con locks/secciones críticas (mutex)

Todos los hilos escriben en un único histograma global, sincronizado mediante mutex o #pragma omp critical.

- Justificación: Garantiza consistencia de datos en todo momento, pero introduce alta sobrecarga por contención. Sirve como referencia para analizar el impacto de la sincronización explícita.
- Impacto esperado: A medida que aumentan los hilos, el tiempo de espera por locks se incrementa, lo que disminuye el speedup y la eficiencia.

Histograma global usando variables atómicas (atomic)

Se usan std::atomic<int> o #pragma omp atomic para que las actualizaciones al histograma global sean atómicas.

- Justificación: Reduce la contención respecto a mutex, permitiendo operaciones concurrentes a nivel de CPU sin bloquear hilos completos. Es un equilibrio entre consistencia y rendimiento.

2. Análisis de tiempos (t_gen, t_count, t_merge, total, real, user, sys)

Los tiempos se desglosan en:

- t_gen: tiempo en generar el vector de datos. Es independiente de la variante de paralelización, pero ligeramente más alto con más hilos por la sincronización de semilla y el reparto de trabajo.
- t_count: tiempo de conteo del histograma, que es el núcleo del algoritmo. Aquí se evidencia la contención:
    - Private: t_count más bajo y estable, creciendo ligeramente con más hilos.
    - Mutex: t_count muy alto y variable; con 4 hilos puede superar 12–20 s, reflejando la sobrecarga de sincronización.
    - Atomic: t_count intermedio, mostrando mejoras respecto a mutex pero sin alcanzar la eficiencia de private.
- t_merge: tiempo de combinación final. Solo afecta a private, siendo mínimo (<0.0001 s), lo que confirma que la reducción local es rápida.
- total: suma de todos los tiempos internos.
- real, user, sys, cpu%: medidos con /usr/bin/time. Private muestra alta eficiencia de CPU (~350% con 4 hilos) y bajo overhead de sistema; mutex tiene alta utilización de CPU y mayor tiempo en sys, indicando contención por locks.

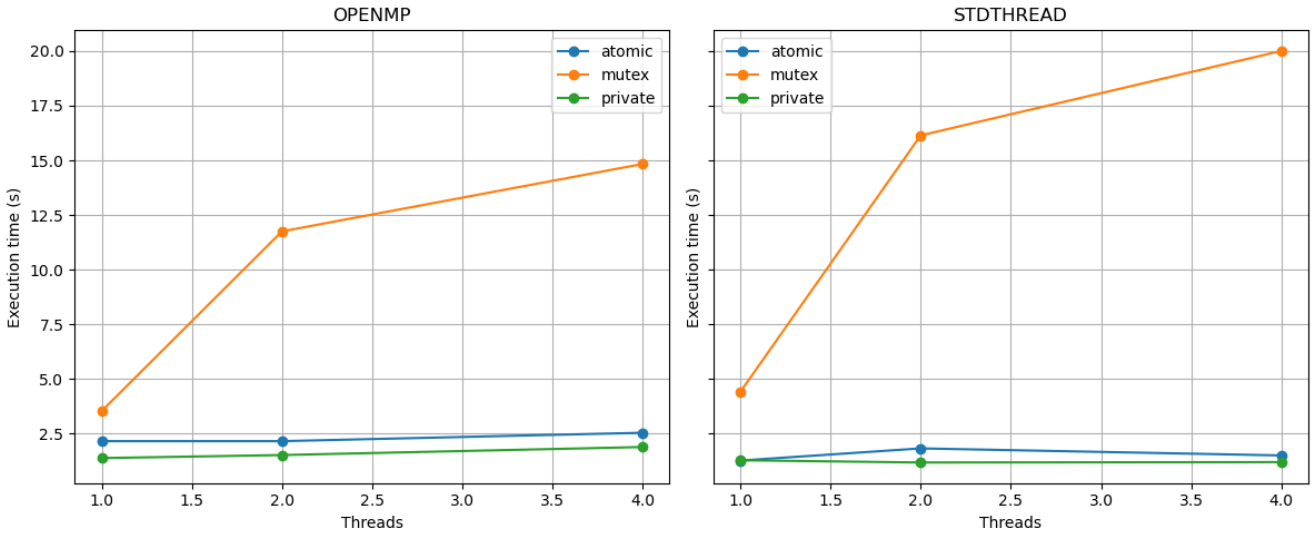
3. Tiempo de Ejecución Total

Descripción:
El tiempo de ejecución (total) muestra la duración completa de cada variante y método con diferentes cantidades de threads.

Observaciones por método y variante:

- Private (OpenMP y std::thread):
    - 1 hilo: ~1.28–1.38 s
    - 2 hilos: ~1.18–1.52 s
    - 4 hilos: ~1.20–1.52 s
    - Interpretación: La variante *private* es rápida con 1 hilo, pero no escala bien con más threads debido a que la generación de datos y conteo es ligero comparado con el overhead de paralelización.
- Mutex:
    - OpenMP: 1 hilo ~3.54 s → 2 hilos 11.75 s → 4 hilos 14.87 s
    - std::thread: 1 hilo ~4.42 s → 2 hilos 14.14 s → 4 hilos 19.34 s
    - Interpretación: El uso de *mutex* provoca degradación con más threads debido a bloqueos y espera activa.

- Atomic:
    - OpenMP: 1 hilo 2.16 s → 2 hilos 2.16 s → 4 hilos 2.31 s
    - std::thread: 1 hilo 1.27 s → 2 hilos 1.82 s → 4 hilos 1.50 s
    - Interpretación: *Atomic* mantiene estabilidad y mejora ligera con más hilos.



4. Speedup

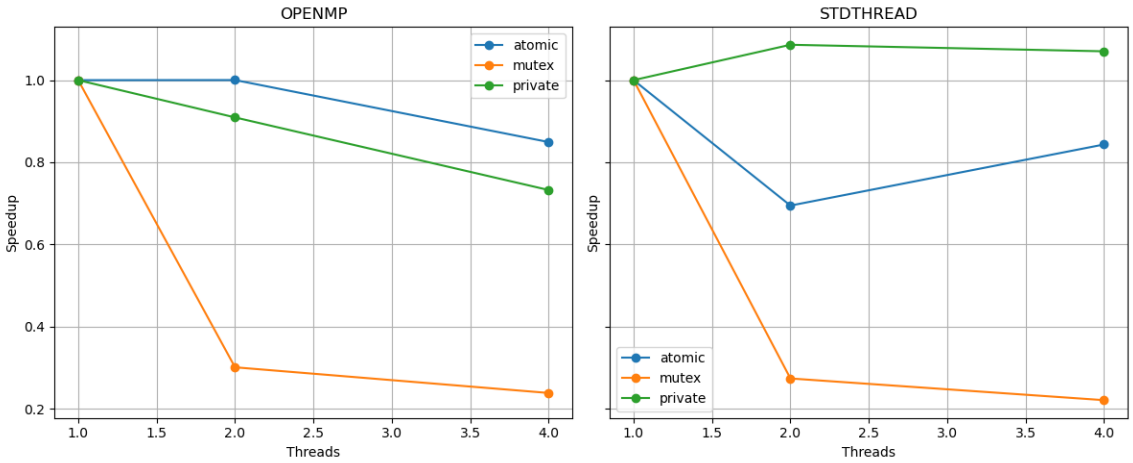
Descripción:
El *speedup* se calcula como speedup=T1/Tn , donde T1 es el tiempo con un hilo y Tn con n hilos. Indica la escalabilidad del algoritmo.

Observaciones:

- Private: speedup ≈ 1 → no escala, mantener 1 hilo.
- Mutex: speedup < 1 con más hilos → degradación significativa.
- Atomic: speedup ~1–0.93 → estabilidad moderada, ligera degradación en algunos casos.

Interpretación:

- Private es eficiente solo para 1 hilo.
- Mutex pierde rendimiento rápidamente con threads adicionales.
- Atomic ofrece un buen balance entre paralelismo y control de datos.



5. Eficiencia

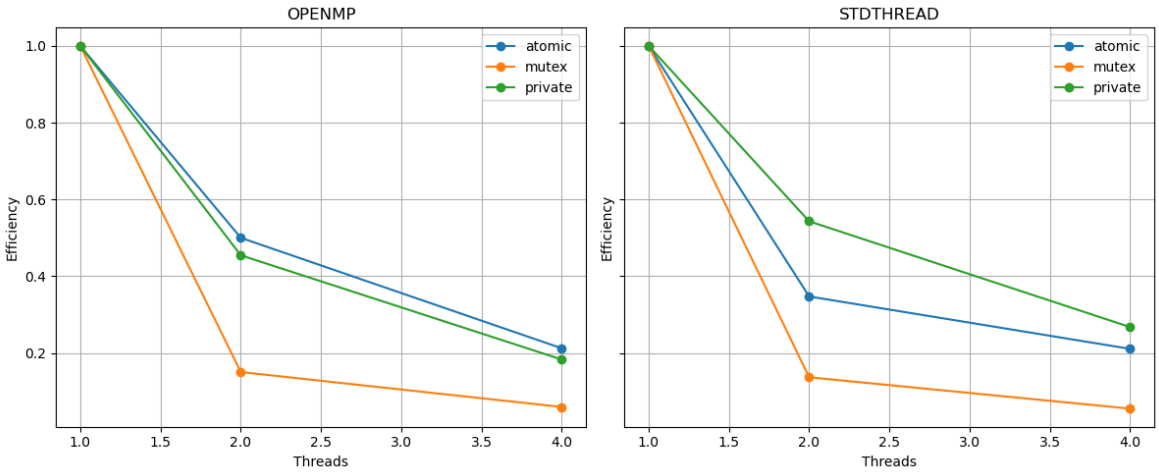
Descripción:
La eficiencia se calcula como efficiency=speedup/threads. Mide cuánto aporta cada thread al speedup total.

Observaciones:

- Private: baja eficiencia, decrece con más threads.
- Mutex: extremadamente baja eficiencia (0.06–0.15) → overhead muy alto.
- Atomic: moderada (0.23–0.25) → threads aportan parcialmente al speedup.

Interpretación:

- La eficiencia evidencia el impacto del overhead de sincronización.
- Mutex es poco eficiente con múltiples threads; Atomic mantiene un balance.



6. Uso de CPU (%CPU)

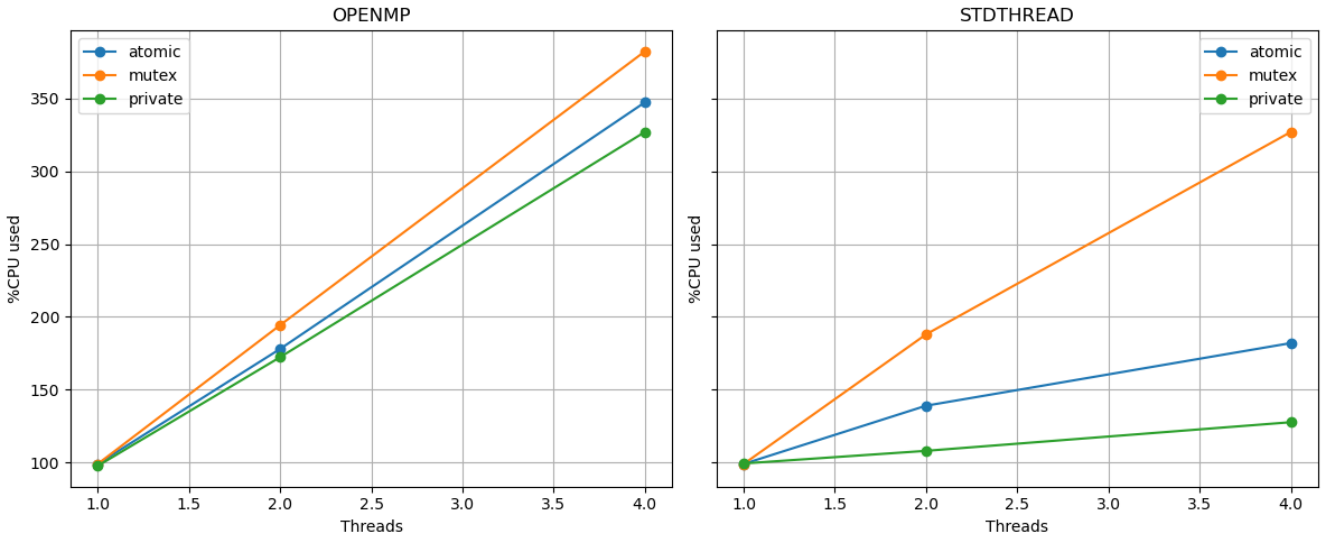
Descripción:
Muestra el porcentaje de CPU utilizado por cada variante y número de threads.

Observaciones:

- Private/OpenMP: 313–350% con 4 hilos → CPU parcialmente ocupado.
- Mutex/OpenMP: 379–386% → threads ocupados pero rendimiento bajo.
- Atomic/OpenMP: 335–362% → CPU utilizada de forma equilibrada.

Interpretación:

- Private es eficiente y con bajo overhead.
- Mutex consume CPU sin mejorar rendimiento debido a espera en locks.
- Atomic logra buen balance entre uso de CPU y rendimiento.



7. Fallos de Cache (Cache Misses)

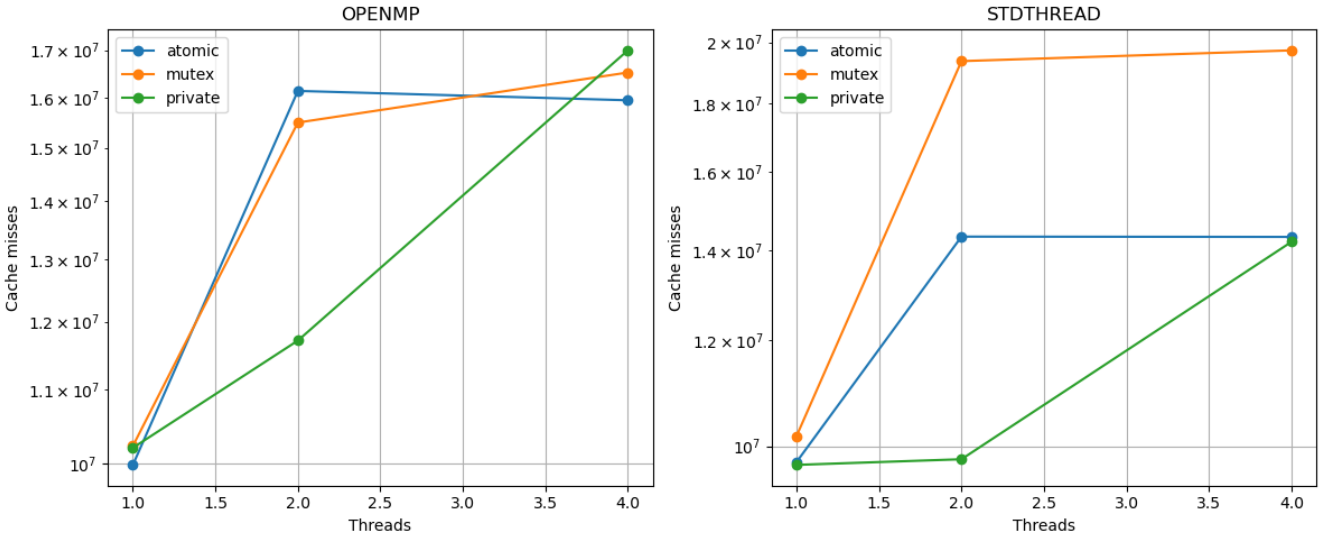
Descripción:
El número de *cache misses* indica cuántas referencias a memoria principal no se pudieron resolver en cache, afectando rendimiento.

Observaciones:

- Private → 17M–42M → bajo tráfico de memoria.
- Mutex → 327M–683M → overhead de memoria muy alto.
- Atomic → 156M–220M → moderado, mayor que private pero mucho menor que mutex.

Interpretación:

- Mutex genera muchas más referencias de memoria, explicando la caída de rendimiento.
- Atomic es intermedio, mantiene consistencia sin saturar la cache.



8. Conclusiones Generales

1. Private:
    - Ideal para baja concurrencia (1 hilo).
    - Escalabilidad limitada, pero muy eficiente en CPU y memoria.
2. Mutex:
    - Ineficiente con múltiples threads.
    - Altos tiempos de ejecución y cache misses.
3. Atomic:
    - Buena opción intermedia.
    - Mantiene estabilidad, eficiencia razonable y control de memoria.

Recomendación:

- Para problemas de generación de histogramas grandes (~100M elementos), usar Atomic si se requiere paralelismo.
- Private sirve para ejecución rápida de un solo thread.
- Evitar Mutex salvo que la sincronización sea imprescindible.

Análisis Cualitativo del Código Implementado con OpenMP y std::thread

El código implementa un histograma paralelo usando OpenMP y std::thread, con tres variantes de sincronización: *private*, *mutex* y *atomic*. Ambas versiones siguen la misma estructura: generación de datos, conteo de histogramas y combinación de resultados cuando es necesario (*private*).

En la generación de datos, OpenMP aprovecha #pragma omp parallel for para paralelizar automáticamente la inicialización del vector, mientras que std::thread realiza esta operación de forma secuencial en el hilo principal, lo que puede afectar el rendimiento con grandes volúmenes de datos.

Durante el conteo del histograma, las variantes muestran diferencias notables:

- Private: cada hilo tiene su propio histograma local, evitando conflictos y contención. Luego se realiza un merge final. Es la variante más eficiente y escalable.
- Mutex: todos los hilos acceden al histograma global mediante locks. Garantiza seguridad, pero provoca alta contención y menor speedup.
- Atomic: permite incrementos concurrentes sobre el histograma global sin bloquear completamente, logrando un buen equilibrio entre seguridad y paralelismo.

OpenMP facilita la distribución de trabajo y el balance de carga automáticamente, mientras que std::thread requiere dividir los datos en chunks manualmente, lo que puede generar desbalance si no se hace cuidadosamente.

En conclusión:

- La variante *private* es la más rápida y escalable.
- *Atomic* ofrece un balance entre velocidad y seguridad.
- *Mutex* es seguro pero penaliza el rendimiento.
- OpenMP simplifica la paralelización, mientras que std::thread da más control pero requiere manejo manual.

Referencias

Arif, M., & Vandierendonck, H. (2015). *A case study of OpenMP applied to Map/Reduce-style computations*. Recuperado de https://pureadmin.qub.ac.uk/ws/portalfiles/portal/16748759/A_case_study_of_OpenMP_Applied_to_MapReduce_style_Computations.pdf

Singanaboina, S. Y. (2024). *Performance analysis of C++ parallel algorithms in HPX* (Master's thesis). Louisiana State University. Recuperado de https://repository.lsu.edu/cgi/viewcontent.cgi?article=7152&context=gradschool_theses

Wang, P., Gao, W., Fang, J., Huang, C., & Wang, Z. (2021). *Characterizing OpenMP synchronization implementations on ARMv8 multi-cores*. Recuperado de https://jianbinfang.github.io/files/2021-10-24-hpcc.pdf