DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Threading en HashMap Concurrente

5 de noviembre de 2023 — Sistemas Operativos

Grupo: 2

Integrante	LU	Correo electrónico
Caire, Monserrat	407/20	monsicaire@gmail.com
Sinnona, Martín	271/20	martinsinnona@outlook.es
Lewin, Ramiro Martín	413/20	ramirolewin@hotmail.com
Collasius, Federico	164/20	fede.collasius@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Introducción

El objetivo de este trabajo será el de aplicar conocimientos sobre *threading* para la implementación de un *HashMapConcurrente*, el cual consiste en un diccionario implementado sobre una tabla de hash que soporta accesos simultáneos concurrentes.
El diccionario en cuestión almacena el número de apariciones de las palabras en uno, o varios, archivos dados. Cada entrada en la tabla de hash consiste en un *bucket* que agrupa a las palabras que comienzan con la misma letra. El mismo se implementa con una lista enlazada que consta de una operación *insertar* la cual permite añadir nuevas palabras al *bucket*.

Se espera que la implementación este libré de condiciones de carrera, *starvation* y *deadlocks*, además de cumplir con una serie de requisitos otorgados por la cátedra.

1. Inserción en la Lista Enlazada

Se procede a implementar el método ***insertar(T valor)***, siendo ésta una operación atómica que inserta un nuevo nodo al comienzo de una lista enlazada.
Se busca que esta lista sea atómica ya que de esta forma nos garantizamos que la inserción se realiza sin interrupciones, donde si un proceso quiere ejecutar la inserción mientras esta ya esta siendo ejecutada, debe esperar a tener permiso para hacerlo. De esta forma nos garantizamos de no tener condiciones de carrera.

Pata implementar la función de forma atómica se siguieron os siguientes pasos:

1. Se crea un nuevo nodo con el valor dado.
2. Se guarda el valor actual de la cabeza de la lista en cabeza_antigua.
3. Se hace que el siguiente nodo después del nuevo nodo sea la cabeza antigua.
4. Se usa *compare_exchange_weak* para intentar establecer la cabeza de la lista al nuevo nodo, esta función actúa de forma atómica. Si *_cabeza* todavía apunta a *cabeza_antigua*, devuelve true y *_cabeza* se actualiza para apuntar al nuevo nodo. Si *_cabeza* ya no apunta a *cabeza_antigua* (otro thread modifico la lista mientras estábamos entrando), entonces *compare_exchange_weak* actualiza automáticamente *cabeza_antigua* al valor actual de *_cabeza* y regresamos al comienzo del ciclo do-while para probar de nuevo.

2. Operaciones del HashMapConcurrente

A continuación se detallará cómo fueron implementadas las funciones:

void incrementar(string clave)
vector <string > claves()
unsigned int valor(string clave)

Se busca que solo haya contención en caso de colisión de hash, es decir, se debe sincronizar si dos o mas threads intentan utilizar la misma función intentando acceder al mismo bucket. En caso contrario, las operaciones deben ser concurrentes sin bloqueo. Para ello se pasa un array de mutex compartidos (shared_mutex), donde cada mutex proteje un bucket específico de la tabla de hash. La particularidad del *shared_mutex* es que protege datos compartidos de ser accedidos simultáneamente por múltiples procesos. Son especialmente útiles cuando los datos compartidos pueden ser leídos de manera segura por cualquier número de proceso simultáneamente, pero un proceso solo puede escribir los mismos datos cuando ningún otro proceso está leyendo o escribiendo al mismo

tiempo. A diferencia de otros tipos de mutex que facilitan el acceso exclusivo, un *shared_mutex* tiene dos niveles de acceso:

- compartido (shared): varios threads pueden tener el mismo mutex. Si un proceso lo ha adquirido (lock_shared()), ningún otro proceso puede adquirir el bloqueo exclusivo, pero puede adquirir el bloqueo compartido.
- exclusivo: solo un thread puede poseer el mutex. Si un proceso lo ha adquerido (lock()), ningún otro proceso puede adquirir el bloqueo (incluido el compartido).

Solo cuando el exclusivo no ha sido adquirido por ningún proceso, el compartido puede ser adquirido por múltiples threads. Dentro de un mismo thread, solo se puede adquirir un bloqueo (compartido o exclusivo) al mismo tiempo. La razón principal para usar *shared_mutex* en vez de un *mutex* es para que los procesos que solamente quieran leer no tengan que esperar de mas.

incrementar(string clave)

Se busca implementar una función a la cual se le pase una clave como parámetro y busque dicha clave en el diccionario. Si la encuentra incrementa en 1 el valor de dicha clave, si no la encuentra, hace una inserción de ella con valor 1. Cuando se quiere incrementar una clave, se bloquea el mutex del bucket correspondiente usando *'lock()'*, permitiendo que otros threads modifiquen otros buckets simultáneamente.

vector <string > claves()

Se busca implementar una función que devuelva todas las claves actuales del diccionario. Esta función asegura la devolución de las claves que se encontraban en el momento en el que se ejecutó la función. Se encuentra protegida con *lock_shared()*, esto permite que varios threads puedan acceder a la vez y leer las claves. A medida que se recompilan las claves asociadas a un bucket específico se va liberando el mutex compartido asociado a dicho bucket, esto para que se puedan escribir nuevas claves en buckets ya vistos.

unsigned int valor(string clave)

Se busca implementar una función que dada una clave devuelva el valor actual. Funciona de manera similar a la función *claves*, donde se busca el valor de una clave determinada, bloqueándose la escritura para el bucket en donde esté contenida dicha clave. Como se utiliza *lock_shared()*, se permite que varios threads lean al mismo tiempo, pero se garantiza el resultado correspondiente al momento en el que se llamó a la función.

3. Búsqueda del Máximo

Se cuenta con la función *máximo(void)* que devuelve el par <clave, valor> con el valor más alto de la tabla.

Es deseable que pueda ejecutarse concurrentemente con *incrementar(string clave)* sin que esto produzca una condición de carrera. Es decir, no puede ocurrir que mientras se esté buscando el máximo se ejecute incrementar en una clave ya visitada y, por lo tanto, quizás devolviendo un máximo que en realidad no lo es. Una situación hipotética donde esto puede suceder es la siguiente:

Por un lado, un thread A está ejecutando *máximo* e identifica que la clave *'árbol'* tiene el valor 10, siendo el máximo hasta el momento. Mientras esto ocurre, un thread B esta ejecutando *incrementar('anana')* varias veces, lo cual deja a la clave *'anana'* con un valor de 11.

Sin embargo, *'anana'* se encontraba previa a *'árbol'* en el bucket correspondiente y, por lo tanto, *máximo* no pudo dar cuenta de esta modificación.
Finalmente la función *máximo* termina devolviendo, erróneamente, <'árbol', 10>cuando en realidad tenía que devolver <'anana', 11>
Con lo cual, es deseable implementar ciertos mecanismos de sincronización para evitar condiciones de carrera como la descripta en el ejemplo. Más aún, se requiere poder ejecutar *máximo* en forma paralela con una cantidad de threads determinada.

A continuación, se detallará la función **maximoParalelo(cant_threads)** que implementa lo mencionado anteriormente.

<string, unsigned int> maximoParalelo(unsigned int cant_threads)

Devuelve el par <clave, valor> cuyo valor es máximo en la tabla, de manera concurrente, haciendo uso de múltiples threads.
A cada thread se le asigna una fila de la tabla a la vez. Cuando termina de procesarla toma la siguiente disponible. Esto garantiza que todos los threads se encuentren activos en todo momento, maximizando así la concurrencia y sólo se detendrán una vez que todas las filas hayan sido procesadas.
Entre los threads se comparte el *HashMapConcurrente*, un índice atómico que indica cuál es la próxima fila a recorrer y un vector de resultados dónde se almacenan los máximos de cada *bucket*.
Con respecto a las condiciones de carrera, se utiliza un índice atómico para evitar que dos threads obtengan la misma fila. Esto evita que dos threads cualesquiera accedan a la misma fila de la tabla.
Por otro lado, el vector de resultados no presenta problemas ya que cada thread accede a una única posición que no compartirá con ningún otro.
Por último, se recorren los resultados para encontrar el máximo definitivo.

4. Carga de Archivos

Se cuenta con la función *cargarArchivo(hashMap, filePath)* la cual, como su nombre indica, carga todas las palabras del archivo indicado en el *HashMapConcurrente*.
Para la implementación de la misma no fue necesario tomar consideraciones especiales respecto de la sincronización ya que sólo utiliza la función *incrementar* y esta ya toma en cuenta todo lo necesario para su buen funcionamiento.

Además, se implementó *cargarMultiplesArchivos(hashMap, cantThreads, filePaths)* la cual hace lo mismo que la anterior utilizando una cantidad de threads establecida. A continuación, se detalla brevemente su implementación.

void cargarMultiplesArchivos(HashMapConcurrente hashMap, int cantThreads, vector<string >filePaths)

Para la implementación se introdujo una variable atómica denominada *archivoActual* que determina el índice del archivo que se está procesando. Esto permite que cada thread conozca cual es el próximo archivo a procesar, evitando así condiciones de carrera.
Una vez que un thread procesa un archivo avanza al siguiente hasta que todos hayan sido cargados.
Se utilizaron variables y operaciones atómicas, como *fetch_add*, para garantizar que los threads procesen un único archivo a la vez y no salteen ninguno. Gracias a esto, se elimina la necesidad de utilizar un *mutex* para sincronizar el acceso al vector de archivos.
Luego, cada thread, hace uso de *cargarArchivo* para cumplir con su tarea.

Experimentación

Se realizaron una serie de experimentos con el fin de evaluar qué ventajas ofrece la ejecución concurrente, en términos de tiempo de ejecución, a la hora de encontrar la clave con el valor máximo y al cargar archivos de gran tamaño.

Todos los experimentos fueron realizados con un procesador Intel(R) Core(TM) i5-10600K CPU @ 4.10GHZ que cuenta con 12 threads.
Además el contenido de los archivos se generó a partir de una distribución real para los experimentos A, B y C y con una distribución aleatoria para el experimento D.

Experimento A

Se quiere ver cómo afecta al tiempo de ejecución al utilizar threads en la tarea de cargar archivos. Para esto se fija la cantidad de archivos en 16. Todos son idénticos y contienen 105.000 palabras cada uno.

Hipótesis: La carga de archivos mejora sustancialmente al utilizar threads ya que se pueden cargar en paralelo, haciendo uso de todos los hilos del procesador.

Resultado

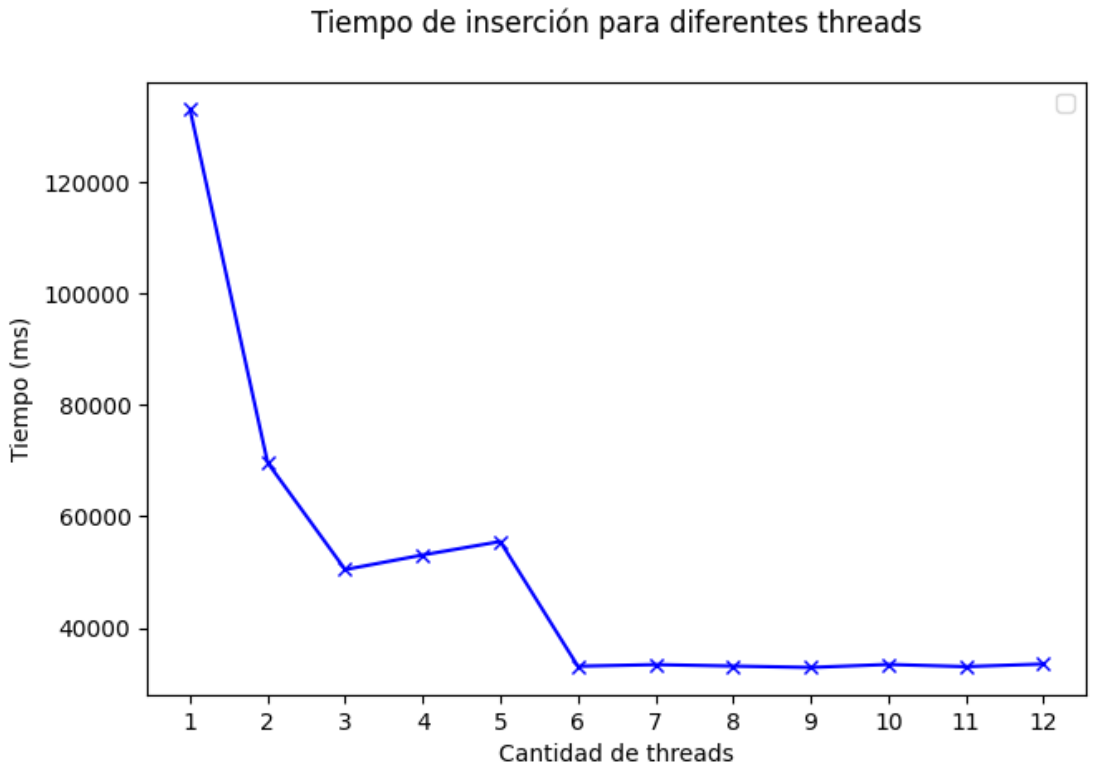

Figura 1: 6 cores - 12 threads

Conclusión

Como era de esperar, una mayor cantidad de threads supone una mejora en el tiempo de in-

serción de dichas palabras. Se alcanza un equilibrio a partir de cierto número de threads dónde ya no resulta conveniente seguir aumentando su cantidad. Se plantea que esto ocurre a partir de 6 threads ya que el procesador que se utilizó tiene 6 núcleos, y a partir de ahí se virtualizan completamente.

Experimento B

Se quiere analizar el tiempo de carga de archivos y su relación con los threads desaprovechados. Para esto se fija el número de threads en 12 y se varía la cantidad de archivos hasta un máximo de 16 archivos, los cuales son idénticos entre sí. Cada archivo contiene un total de 336.736 palabras.

Hipótesis: Cuando existen threads desaprovechados el tiempo de carga de los archivos aumenta en menor medida comparado a cuando se necesitan más threads de los que se disponen. Para los casos en que #archivos $<$ #threads, es decir, cuando hay threads sin utilizar, se espera que el tiempo de ejecución sea más similar y aumente más lentamente.

Resultado

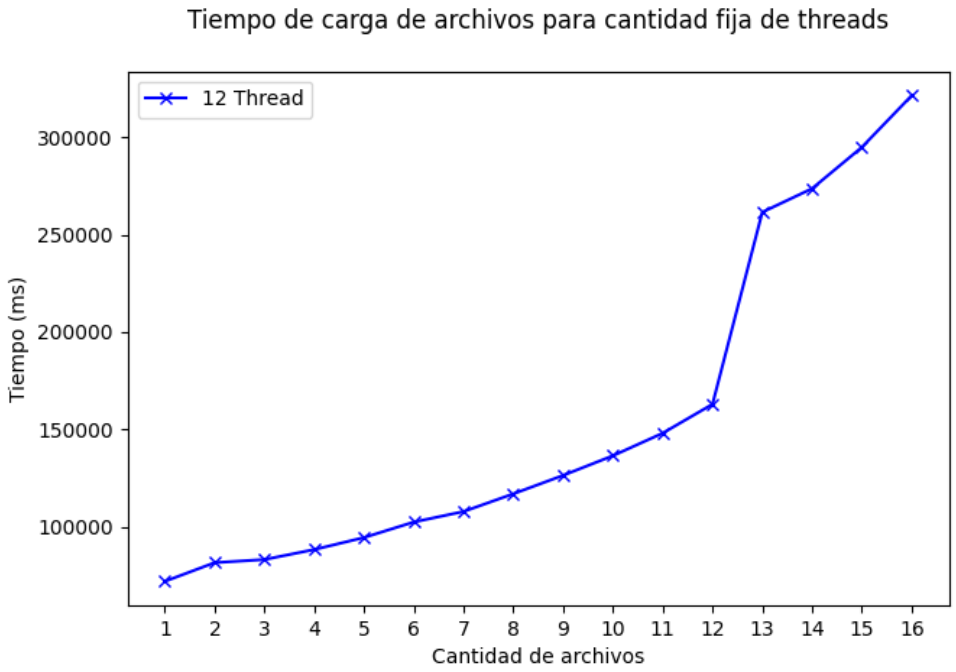

Figura 2: 6 cores - 12 threads

Conclusión

Se puede observar que el tiempo de inserción aumenta proporcionalmente a la cantidad de archivos que se procesan. Desde 1 hasta 12 archivos el crecimiento es menos pronunciado que entre 13 y 16 archivos. Debido a que cada archivo es cargado por un único thread, en cada paso estamos utilizando/aprovechando uno más. Sin embargo, el tiempo de ejecución aumenta, en menor medida, como consecuencia del bloqueo entre los threads al momento de realizar las inserciones.
Por otro lado, cuando la cantidad de archivos supera los threads disponibles, además del tiempo adicional mencionado anteriormente, se añade la espera por un thread desocupado que pueda atender la carga.

Experimento C

Para una cantidad variable de threads y un archivo con 200.000 palabras se quiere analizar el tiempo de carga.
En este caso, se quiere comparar el rendimiento al dividir el archivo en partes iguales, hasta un máximo de 15 veces, y cargarlo utilizando threads, ó cargar un único gran archivo utilizando un único thread.

Hipótesis: Es conveniente dividir un archivo en k sub-archivos de igual tamaño siendo k la cantidad de threads disponibles.

Resultado

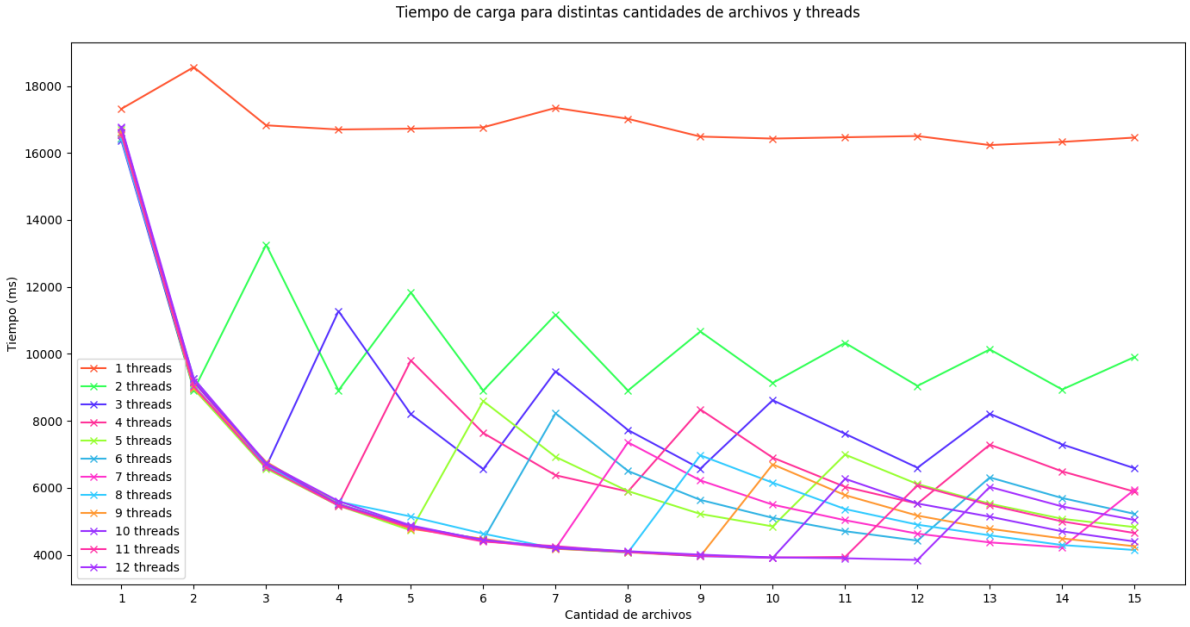

Figura 3: 6 cores - 12 threads

Conclusión

Se puede apreciar cómo a medida que se aumentan los threads el tiempo de inserción disminuye. A su vez, vale notar que el tiempo de inserción, cuando la cantidad de archivos a cargar es menor o igual a la cantidad de threads, no presenta una variabilidad significante. Es decir, la carga para 2 archivos a partir de 2 threads es muy parecida. Esto se debe a que cada thread puede atender a un archivo diferente.

Por otro lado, puede verse que para una cantidad de archivos múltiplo del número de threads el tiempo de carga resulta muy similar, mientras que para los demás se presentan picos. Esto es así debido a que al cargar una cantidad de archivos no múltiplo de la cantidad de threads que estamos utilizando, algunos threads van a tener que trabajar para atender esos archivos y otros quedarán en espera, en cambio, si se atiende una cantidad de archivos múltiplos de los threads, todos estarán atendiendo archivos al mismo tiempo, siendo estos de un tamaño mas chico que los anteriores.

Experimento D

Para un *hash* determinado se busca el máximo, aumentando la cantidad de threads y evaluando como esto afecta al tiempo de ejecución.
Se utilizaron doce archivos de 560.000 palabras cada uno. Se buscó el máximo en un *hash* de 6.720.000 palabras.

Hipótesis: A medida que se incrementan los threads la búsqueda del máximo resulta más eficiente.

Resultado

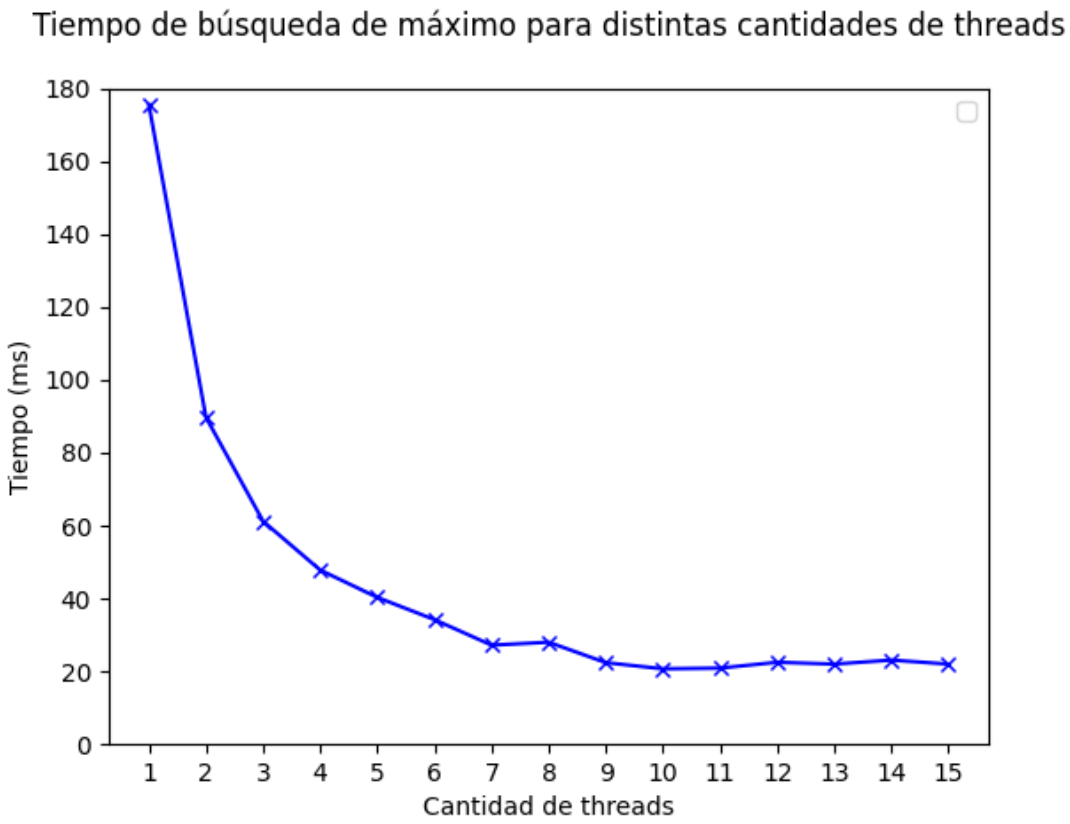

Figura 4: 4 cores - 12 threads

Conclusión

Como era de esperar, utilizar más threads resulta en una búsqueda del máximo más eficiente. Esto es así ya que se paraleliza la búsqueda entre los diferentes *buckets* del *HashMap*.

Conclusiones finales

En este trabajo, se ha logrado implementar un eficiente *HashMapConcurrente* que permite el acceso seguro y simultáneo a los datos, evitando condiciones de carrera, *starvation* y *deadlocks*.

Los experimentos respaldan la hipótesis de que el uso de múltiples threads mejora significativamente el rendimiento en la carga de archivos y la búsqueda del máximo, validando así la efectividad de la ejecución concurrente en el procesamiento de datos.

En resumen, este trabajo ha logrado implementar una solución concurrente sólida y eficiente, demostrando que la sincronización adecuada y el uso de estructuras de datos compartidas son clave para superar los desafíos de las condiciones de carrera en un entorno concurrente.