DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Threading en HashMap Concurrente

12 de diciembre 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.
Hoy en día la gestión eficiente de datos en entornos concurrentes es una necesidad creciente. La capacidad de procesar grandes volúmenes de datos de manera rápida y precisa es fundamental en muchas aplicaciones modernas, lo que hace que este trabajo práctico sea particularmente relevante.
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`.
Además de garantizar que la implementación esté libre de `race conditions, starvation y deadlocks`, cumpliendo con los requisitos establecidos por la cátedra, este trabajo también se enfoca en una serie de experimentos:

`Experimento A`: evalúa cómo los `threads` afectan el tiempo de ejecución al cargar archivos.
`Experimento B`: analiza el tiempo de carga de archivos en relación con los `threads` desaprovechados, fijando el número de `threads` en 12 y variando la cantidad de archivos hasta 16.
`Experimento C`: con una cantidad variable de `threads` y un archivo de 200.000 palabras, se compara el rendimiento al dividir el archivo en sub-archivos iguales, hasta un máximo de 15 veces, frente a la carga de un único gran archivo con un solo `thread`.
`Experimento D`: busca el valor máximo en un `hash` con 6.720.000 palabras, aumentando la cantidad de `threads` y observando el impacto en el tiempo de ejecución.
`Experimento E`: examina el punto en el que el `HashMap` comienza a ser mas eficiente que la búsqueda lineal usando "Dracula"de Bram Stocker.

1. Inserción en la Lista Enlazada

Se procede a implementar el método `insertar(T valor)`, siendo esta 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 está siendo ejecutada, debe esperar a tener permiso para hacerlo. De esta forma nos garantizamos de no tener condiciones de carrera.

Para implementar la función de forma atómica se siguieron los 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.

La atomicidad se garantiza mediante el uso de `compare_exchange_weak`. Esta función asegura que la actualización del puntero `_cabeza` al nuevo nodo sea una operación indivisible. Esto sig-

nifica que en el momento en que _cabeza se actualiza para apuntar al nuevo nodo, ningún otro thread puede modificar _cabeza. Si otro thread modifico _cabeza, compare_exchange_weak falla, reintentando la operación hasta que tenga éxito. Esto previene efectivamente las condiciones de carrera, ya que asegura que cada inserción se procese de manera completa y aislada de las demás operaciones concurrentes.

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), con un tamaño igual a la cantidad de buckets que posee el HashMap, en particular para esta implementación de tamaño 26 ya que se utiliza la primer letra de cada palabra en la función de hash. Cada mutex protege 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 con 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 con 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 threadsmodifiquen 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 necesario resolver los problemas que surgen al permitir que esta función se ejecute concurrentemente con `incrementar(string clave)`, particularmente en lo que respecta a las condiciones de carrera y la integridad de los datos. Una condición de carrera puede ocurrir si, mientras se está buscando el máximo, se ejecuta `incrementar` en una clave ya visitada. Esto puede resultar en que `máximo` devuelva un valor que ya no es el actual máximo.

Por ejemplo, consideremos un escenario donde un `thread A` ejecuta `máximo` y encuentra que la clave `árbol` tiene el valor 10, siendo el máximo hasta ese momento. Simultáneamente, un `thread B` ejecuta `incrementar('anana')` varias veces, incrementando el valor de la clave `anana` a 11. Si `anana` se encuentra antes de `árbol` en el `bucket` correspondiente, `máximo` no reconocerá esta actualización y terminará devolviendo `<árbol, 10>` en lugar del correcto `<anana, 11>`.

Es necesario entonces implementar mecanismos de sincronización para prevenir condiciones de carrera y asegurar la consistencia de los datos. Además, es importante considerar cómo la función `máximo` puede ejecutarse en paralelo con múltiples `threads` sin comprometer la integridad de los datos.

A continuación, se detallará la función `maximoParalelo(cant_threads)` que resuelve estos problemas.

<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. Esto se maneja mediante un índice atómico, `listaActual`, que garantiza que cada `thread` obtenga una fila única y evita que dos `threads` procesen la misma fila simultáneamente. Los `threads` procesan su fila asignada y, al terminar, toman la siguiente disponible usando `listaActual.fetch_add(1)`, lo que asegura que todos los `threads` se mantengan activos maximizando la concurrencia.
Para cada fila de la tabla, se utilizan `mutexes` para controlar el acceso concurrente. Antes de iniciar los `threads`, todos los `mutexes` se bloquean con `mutexes[i].lock_shared()`. Después de procesar cada fila, el `thread` correspondiente desbloquea el `mutex` usando `mutexes[idx].unlock_shared()`, asegurando así que no haya condiciones de carrera al acceder a las filas de la tabla.
Además, se utiliza un vector de resultados, `resultados`, donde cada `thread` almacena su máximo local encontrado. Como cada `thread` escribe en una posición única del vector, no hay conflictos entre ellos.
Una vez que todos los `threads` han terminado, se recorren los resultados almacenados en el vector para encontrar el máximo definitivo, comparando los valores almacenados y seleccionando el mayor.

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 cuál 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 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`.

El contenido de los archivos se generó a partir de una distribución real para los experimentos A, B, C y E y con una distribución aleatoria para el experimento D.

Para el caso de las distribuciones reales tenemos palabras que provienen del diccionario en inglés y por lo tanto, la frecuencia de la primer letra de cada palabra sigue una distribución conocida. Por ejemplo, en el inglés, es más probable que una palabra comience con **e**.
Esto resulta relevante para la eficiencia del `HashMap` ya que habrá `buckets` en dónde estadísticamente se encontrarán más palabras.
Esta elección de distribución se justifica con la necesidad de tener un escenario que se asemeje a un caso real y poner a prueba la estructura bajo dicho supuesto.
No obstante, esta distribución resulta bastante natural y no genera grandes diferencias en la cantidad de apariciones de palabras dentro de cada `bucket`. De no ser así, es decir, supongamos un escenario en el cual la mayoría de las palabras empiezan con la misma letra, se tendría un `bucket` con una gran cantidad de palabras en dónde buscar el máximo, por ejemplo, demoraría mucho más para un `thread` que hacerlo en otro con muchas menos palabras.

Es por esto que para el experimento D se optó por una distribución uniforme en dónde todos los `buckets` tengan aproximadamente la misma cantidad de palabras y así evitar lo mencionado anteriormente.

Además, una distribución uniforme permite minimizar las colisiones ya que aproximadamente todos los `threads` realizarán el mismo trabajo y no habrá tanta competencia por obtener los recursos, por ejemplo, para el caso de búsqueda del máximo. Esta competencia depende también de la cantidad de `threads` que se utilicen ya que, de haber muchos y poco trabajo, estos competirán por los recursos generando colisiones,

Experimento A

Se quiere ver cómo afecta al tiempo de ejecución al utilizar `threads` en la tarea de cargar archivos. Como el procesador que se utilizará cuenta con 12 `threads`, se fijará la cantidad de archivos en 16, ya que se quiere comprobar qué comportamiento se obtiene al utilizar mas archivos del máximo de `threads` disponibles. Se cuenta con todos los archivos idénticos que contienen 105.000 palabras, esto se debe a que se quiere tener un control mas preciso sobre las variables que pueden afectar al experimento, dando más seguridad de que cualquier diferencia en los tiempos de inserción se debe principalmente a la paralelización y no a la variabilidad de los datos.

Hipótesis: La carga de archivos mejorará sustancialmente al utilizar `threads`, aprovechando así la capacidad de procesamiento en paralelo ofrecida por la arquitectura de la computadora. Al distribuir la carga de trabajo entre múltiples `threads`, cada uno asignado a la carga de un archivo específico, se espera que la eficiencia del proceso de carga aumente.

Resultado

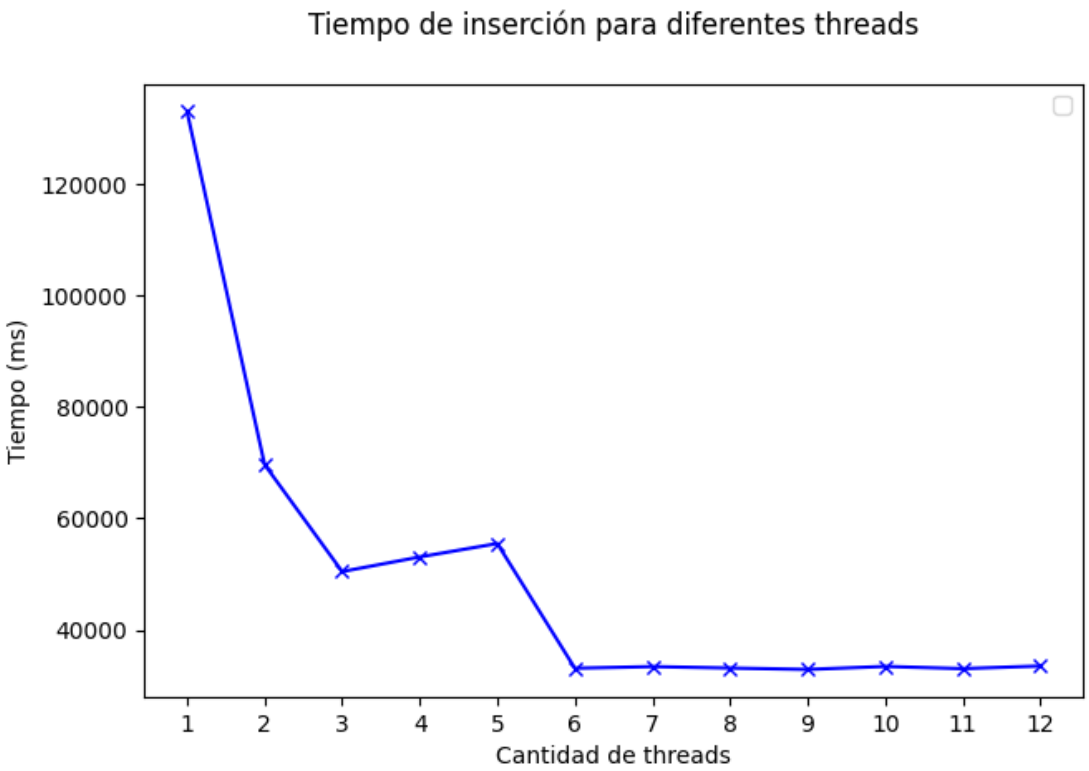


Figura 1: 6 cores - 12 `threads`

Conclusión

La observación en el tiempo de inserción indica mejoras a medida que se aumenta la cantidad de `threads` utilizados. Se puede identificar un punto de equilibrio a partir del cuál no resulta conveniente seguir aumentando la cantidad de `threads`. Este equilibrio se puede distinguir a partir de los 6 `threads`, los cuales coinciden con el número físico de núcleos con los que cuenta el procesador. Se plantea que esto puede deberse a que ya se hizo uso de todos los núcleos físicos del procesador y a partir de los 6 `threads` se virtualizan completamente.

Experimento B

Se quiere analizar el tiempo de carga de archivos y su relación con los `threads` desaprovechados.
El enfoque se encuentra en comprender la dinámica subyacente cuando algunos `threads` no se utilizan constantemente durante el proceso de carga.
Este experimento permite obtener conocimiento valioso sobre la interacción entre la capacidad del sistema y la complejidad de la carga de trabajo.

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.
Al mantener constante el número de `threads` en 12 y variar la cantidad de archivos, exploramos cómo la disponibilidad de `threads` impacta la estabilidad del proceso de carga.

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 estable y aumente más gradualmente.

Resultado

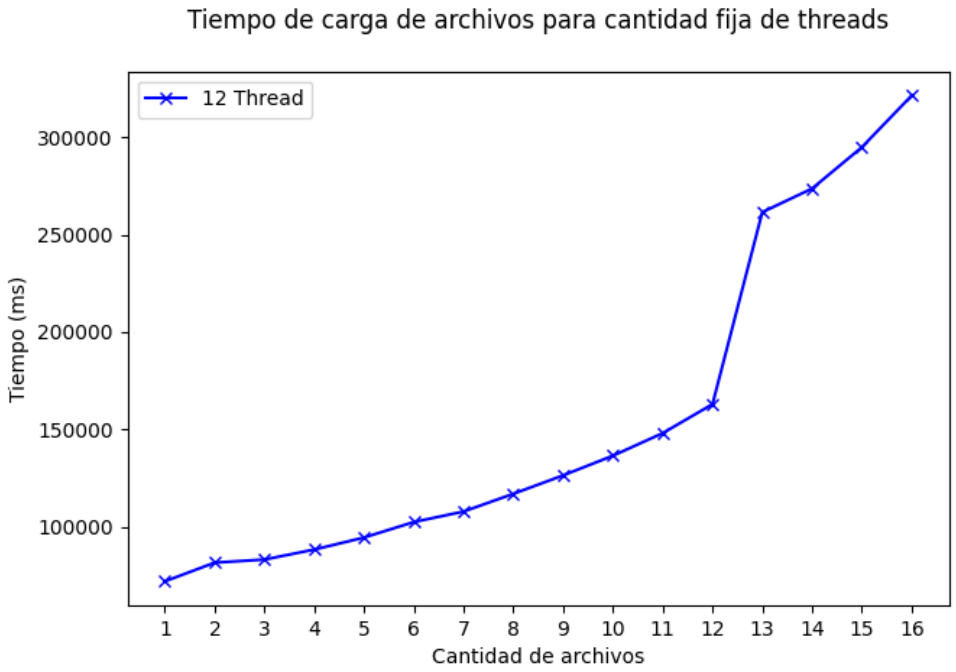


Figura 2: 6 cores - 12 `threads`

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 se está 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. Esto se debe a que, al tener una cantidad de archivos mayor a la cantidad de `threads` disponibles se debe esperar a la liberación de un `thread` para que se haga cargo de los archivos que están en espera. Esto puede verse como un caso en el que la demanda supera la oferta de `threads` disponibles, contribuyendo al aumento del tiempo de ejecución.

Experimento C

El siguiente experimento tiene como objetivo analizar el tiempo de carga en la inserción de archivos para una cantidad variable de `threads` al procesar un archivo que contiene 200.000 palabras.
Se propone una comparación de rendimiento al dividir el archivo en partes iguales hasta un máximo de 15 archivos y cargarlos utilizando `threads`. Esto se contrastará con la carga de un único archivo de gran tamaño (el archivo inicial) mediante un único `thread`.

Hipótesis: Se espera que la eficiencia de la inserción de palabras mejore con el uso de `threads` y la división del archivo en `k` sub-archivos de igual tamaño, siendo `k` la cantidad de `threads` disponibles.

Se anticipa que ocurra esto debido a que un único archivo de gran tamaño podrá ser insertado por un único `thread`, en cambio, a medida que el archivo es dividido en partes mas chicas, éstas podrán paralelizarse, haciendo un mejor uso de la características del procesador.

Resultado

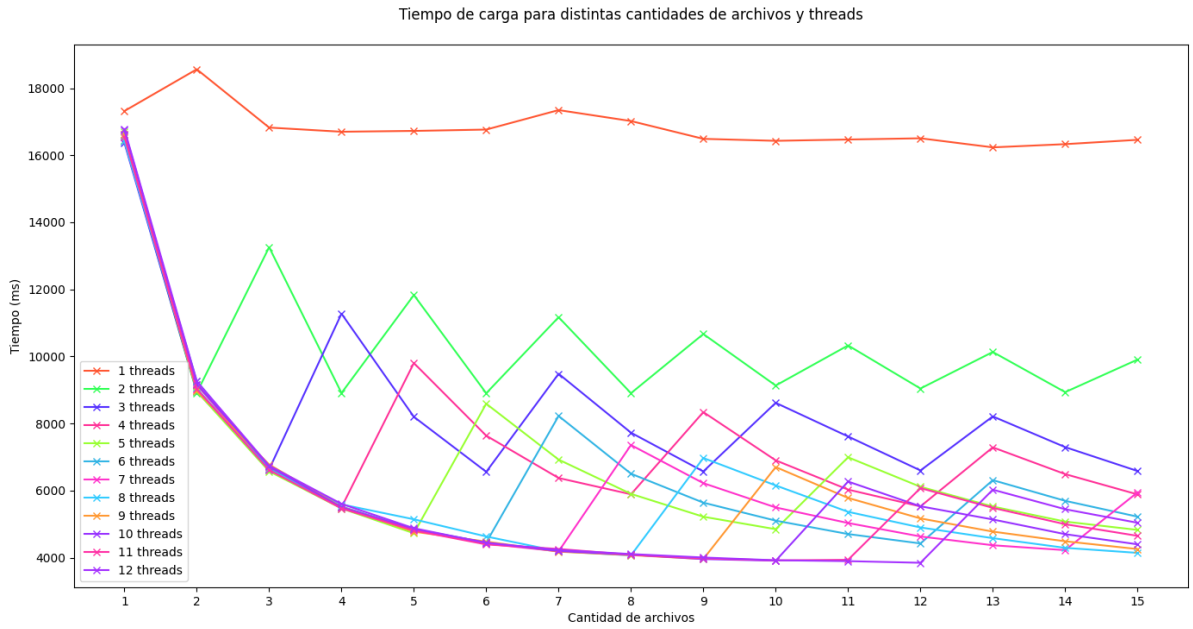


Figura 3: 6 cores - 12 `threads`

Conclusión

Se puede ver 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 cómo esto afecta al tiempo de ejecución.

Para llevar a cabo este experimento se seleccionó un conjunto de datos compuesto por 6.720.000 palabras distribuidas en 12 archivos de 560.000 palabras cada uno. Esta elección se fundamenta en la observación de que la función de búsqueda del máximo es intrínsecamente rápida.
La inclusión de una cantidad considerable de palabras tiene como objetivo evaluar el rendimiento de la paralelización en un escenario donde la función ejecuta su tarea de manera eficiente, permitiendo así obtener una comprensión más completa del impacto de los `threads` en este contexto específico.
A su vez se cuentan con 12 archivos debido a que se cuentan con 12 `threads` en el procesador utilizado para la experimentación.

Hipótesis: Conforme se incrementa la cantidad de `threads` utilizados, se experimenta una mejora en la eficiencia de la búsqueda del máximo. Esto de debería a la capacidad de los `threads` para trabajar en paralelo, permitiendo una distribución más efectiva de la carga de trabajo y, por ende, una reducción en el tiempo total de ejecución.

Resultado

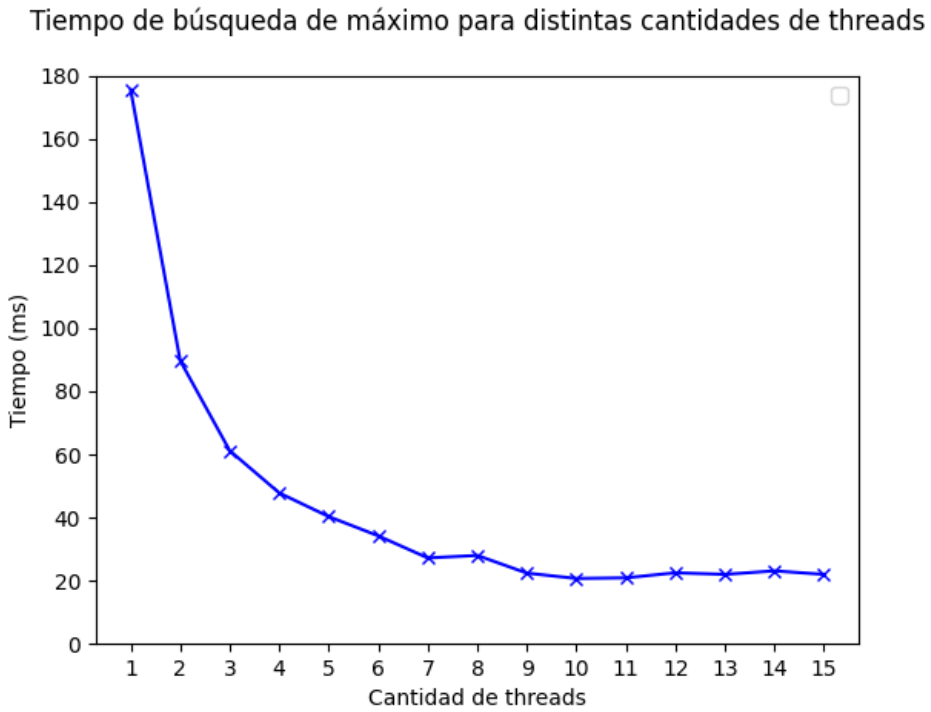


Figura 4: 4 cores - 12 `threads`

Conclusión

Se puede observar que al 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`.

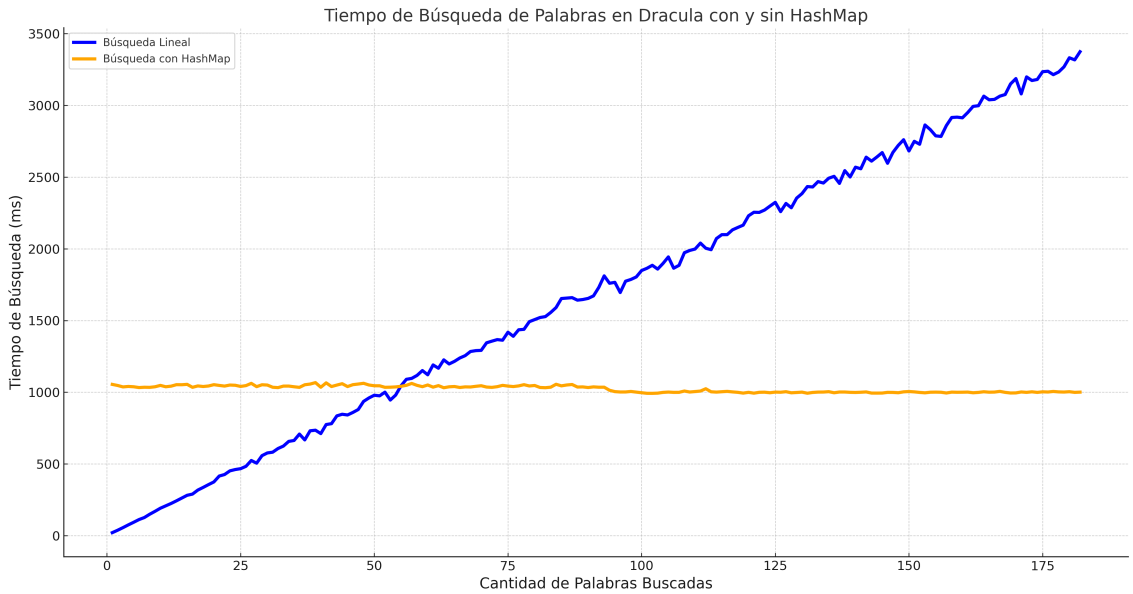
En cuanto a las colisiones entre `threads`, estas se manejan de manera eficiente. Cuando un `thread` completa la búsqueda en su `bucket` asignado, procede a buscar en el siguiente. En este proceso, se asegura de seleccionar un `bucket` disponible, es decir, uno en el que no esté trabajando actualmente otro `thread`. Esto minimiza el impacto de las colisiones en el rendimiento, ya que cada `thread` puede avanzar de manera independiente y evitar conflictos con otros `threads` que buscan en la misma región del `HashMap`.

Experimento E

El propósito de este experimento es comparar dos métodos de búsqueda de palabras repetidas en el texto del libro "Drácula"de Bram Stoker, que contiene un total de 162,417 palabras. El experimento se centra en dos enfoques: la búsqueda lineal y el uso del `HashMap`. En la búsqueda lineal, recorrer el texto palabra por palabra implica leer el libro entero tantas veces como indica el eje x; por ejemplo, un valor de 5 en el eje x significa que el libro se leyó completamente 5 veces para contar las repeticiones de cada palabra buscada.
Por otro lado, el enfoque con `HashMap` implica crear esta estructura de datos una sola vez, cargarla con la totalidad del texto y luego realizar consultas para encontrar la frecuencia de hasta 5 palabras diferentes, correspondiendo al mismo valor en el eje x. Este método, a pesar de tener un costo inicial de configuración, podría ser más eficiente a medida que el número de consultas de palabras aumenta.
Las palabras buscadas se seleccionaron de manera aleatoria del libro.

Hipótesis: Para un número bajo de palabras a buscar, la búsqueda lineal será más eficiente debido al coste inicial de crear y cargar un `HashMap`. Sin embargo, a medida que el número de palabras buscadas aumente, se espera que el `HashMap` se torne más eficiente. A partir de cierto punto el `HashMap` va a compensar su costo inicial para un numero de palabras lo suficientemente alto.

Resultado



Conclusión

El gráfico muestra que el método de búsqueda lineal es inicialmente más rápido que el uso de un `HashMap` para el conteo de palabras repetidas `Drácula`. Sin embargo, esta ventaja disminuye y se invierte a medida que la cantidad de palabras buscadas se incrementa. El punto de quiebre específico se encuentra alrededor de las 53 palabras: ahí es donde el tiempo de ejecución de la búsqueda lineal supera al del `HashMap`.

El propósito de este experimento no es solo identificar cuál método es más eficiente, sino también establecer un entendimiento de la escalabilidad de ambos enfoques a partir de múltiples búsquedas en un conjunto de datos grande y estático, como lo es una novela.

La importancia de este experimento radica en su aplicación en sistemas donde se realizan búsquedas frecuentes y múltiples. Es importante conocer el punto en el cual la concurrencia se vuelve más eficiente. En un entorno de muchos datos donde las búsquedas pueden ser una operación costosa, un enfoque concurrente a partir de cierto volumen de consultas puede ofrecer ventajas considerables en términos de rendimiento.

Estos resultados apoyan la transición de una búsqueda secuencial a una estructura de datos más compleja cuando la frecuencia de las consultas justifican el costo inicial de su creación.

Conclusiones finales

En este trabajo, hemos implementado con éxito un `HashMapConcurrente` eficiente, que facilita el acceso seguro y simultáneo a los datos, mitigando `race conditions`, `starvation` y `deadlocks`.

Los experimentos realizados confirman nuestra hipótesis inicial: el uso de múltiples `threads` incrementa el rendimiento tanto en la carga de archivos como en la búsqueda del valor máximo. Esto confirma la eficacia de la ejecución concurrente en el procesamiento de grandes volúmenes de datos. Sin embargo, es importante reconocer que la creación y gestión de `threads` implica un `overhead` significativo. En determinadas situaciones, los costos asociados con la inicialización de estructuras internas, liberación de recursos, y la coordinación y comunicación entre `threads`, pueden sobrepasar los beneficios obtenidos de la concurrencia.

El uso extensivo de `threads`, además, puede consumir excesivamente los recursos del sistema, como la memoria y el poder de procesamiento, llevando a una potencial disminución en el rendimiento global. Por tanto, es necesario encontrar un equilibrio adecuado en la cantidad de `threads` utilizados, para maximizar la eficiencia sin caer en una sobrecarga de recursos.

En conclusión, este trabajo logró desarrollar una solución concurrente robusta y eficaz. Los resultados demuestran que una sincronización cuidadosa y la utilización de estructuras de datos compartidas son importantes para superar los retos inherentes a las condiciones de carrera en entornos de ejecución concurrente.