DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Threading

20 de Octubre de 2024 — Sistemas Operativos

Grupo Uwuntu

Integrante	LU	Correo electrónico
Finkelberg, Zoe	1132/22	zfinkelberg@gmail.com
Rivarola, Mariana	300/19	maru.ribro@gmail.com
Ranieri, Martina	1118/22	martubranieri@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
http://www.exactas.uba.ar

Índice

1. Introducción

En este trabajo práctico vamos a profundizar sobre la gestión de concurrencia. Cuando hablamos de esto, nos referimos a las técnicas y mecanismos que utilizan los sistemas operativos para asegurar que múltiples procesos o hilos de ejecución accedan y modifiquen sobre los mismos recursos de la memoria compartida de forma controlada, sin causar comportamientos inesperados. Algunos problemas que podríamos llegar a tener cuando trabajamos con múltiples procesos son: Condiciones de carrera, Deadlock, Starvation y de Sincronización. Es fundamental poder garantizar que nuestro programa esté libre de estos problemas, ya que sino podríamos obtener una ejecución no esperada o errónea. Algunas forma de evitarlos son:

- **Mutex (Mutual exclusion):** Garantiza que solo un proceso a la vez pueda acceder a un recurso o fragmento de código en particular. Entonces si un proceso o hilo quiere acceder al recurso, primero intenta bloquear el mutex. Si el mutex está desbloqueado (1), el proceso lo bloquea con un *lock()*[1] (cambia a 0) y obtiene acceso. Si el mutex ya estaba bloqueado por otro proceso (estado 0), el proceso queda suspendido hasta que el mutex sea desbloqueado por el proceso que lo posee con un *unlock()*[2]. Con esto aseguramos la exclusividad en el acceso al recurso.
- **Variables atómicas:** Las variables atómicas cuando se ejecutan lo hacen por completo sin ser interrumpidas por otros hilos o procesos. En el contexto de programación concurrente, las operaciones atómicas garantizan que el acceso o la modificación de un recurso compartido sea consistente y libre de interferencias. Unas funciones básicas son; *load()* [3], *fetch_ add()* [4], *store()* [5], entre otras.

Para desarrollar estos temas, vamos a implementar una estructura de datos concurrente basada en una tabla hash con listas enlazadas denominada **HashMapConcurrente**, tal que nos permita procesar órdenes de compra de manera eficiente en una tienda online simulada. Para esto, abstraemos la comunicación del sistema a través de archivos de texto que contienen las órdenes, donde cada producto (clave) y su cantidad solicitada (valor) serán almacenados en un diccionario distribuido. En la estructura, debemos garantizar un acceso concurrente seguro, utilizando herramientas de sincronización provistas por la biblioteca estándar de C++.

2. Desarrollo

2.1. Lista atómica

Para este ejercicio nos fue proporcionado todo el código de **ListaAtómica**, la cual es una lista enlazada que utilizaremos más adelante para almacenar los elementos de cada bucket en la tabla de hash. Se nos pedía implementar la función **insertar(T valor)** de manera atómica y que no sea bloqueante.

El concepto de lista atómica implica que las operaciones realizadas sobre esta son indivisibles y no pueden ser interrumpidas por otros hilos. Al realizar la función *insertar()* de forma atómica, nos aseguramos de que las modificaciones a la lista sean realizadas de manera consistente, incluso cuando múltiples hilos intentan insertar elementos al mismo tiempo. Esta implementación evita condiciones de carrera al confiar en operaciones atómicas, sin el uso de bloqueos explícitos como mutex. Esto hace que la solución sea eficiente para escenarios concurrentes, pero podría presentar alta carga en sistemas con mucha contención, debido al busy waiting.

Algorithm 1 Lista atómica Insertar

```
1: Parámetros: T valor
2: nuevoNodo = new Nodo(valor)
3: nuevoNodo.siguiente = cabeza.load()
4: while ( do!cabeza.compare_exchange_weak(nuevoNodo.siguiente, nuevoNodo))
5: end while
```

Lo que hicimos fue crear un nuevo nodo para poder agregarlo a la lista enlazada con el valor recibido por parámetro. Luego le asignamos al siguiente nodo la cabeza de la lista.

[1] lock()
[2] unlock()
[3] load()
[4] fetch_add()
[5] store()

Para no tener problemas de race condition en caso de que más de un nodo quiera ser insertado al mismo tiempo, utilizamos **compare_exchange_weak(T& expected, T desired)** [6], esta función lo que hace en nuestro caso es:

1. Comparar el valor actual de cabeza con nuevoNodo.siguiente.
2. Si son iguales:
   - Actualiza cabeza para que apunte a nuevoNodo, convirtiéndolo en la nueva cabeza de la lista. Retorna True.
3. Si no son iguales (es decir, otro hilo cambió la cabeza):
   - Asigna el valor actual de la cabeza a nuevoNodo.siguiente, asegurando que el nuevo nodo apunte a la cabeza actual. Retorna False.

Notemos que si obtuvimos False a la hora de comparar, significa que el valor de la cabeza fue actualizado por otro hilo, lo que impide que el hilo actual complete su operación y en este caso vamos a volver a intentar actualizar el valor de la cabeza hasta que lo podamos realizar correctamente. Esta acción es conocida como *busy waiting*. Esta implementación es *lock free*, es decir que no tiene bloqueos ya que en ningún momento utilizamos mutex ni ningún tipo de bloqueo.

Para verificar su implementación realizamos diversos casos de testing para así poder asegurar su correcto funcionamiento y capacidad de manejar operaciones concurrentes. Ya nos daban *ListaComienzaVacia* que verifica que la lista esté vacía al momento de su creación, asegurando que la longitud inicial es cero. Posteriormente, evaluamos la funcionalidad de inserción con *InsertarAgregaElemento*, donde comprobamos que al agregar un elemento, la longitud de la lista se incrementa adecuadamente.

En *InsertarAgregaElementoCorrecto* validamos que el elemento insertado sea accesible en la posición esperada. También realizamos pruebas de inserción en orden, donde al insertar múltiples elementos, confirmamos que se mantenga el orden correcto en *InsertarAgregaEnOrden*. Nosotras agregamos una versión ampliada de este último, *InsertarAgregaEnOrdenMuchos*, donde se insertan 100 elementos. Además, evaluamos el rendimiento de la lista bajo condiciones de concurrencia.

En *ConcurrentInsertar* utilizamos varios hilos para insertar elementos y verificamos que todos los elementos esperados estén presentes en la lista mientras que, en *ConcurrentInsertarMuchos*, aseguramos que múltiples hilos puedan insertar un gran número de elementos sin conflictos, validando que cada número insertado aparezca en la lista el número esperado de veces.

El código se encuentra en **/src/UnitTest1.cpp**

2.2. Hash map concurrente

2.2.1. Primera parte

Para este ejercicio se provee una implementación parcial de la clase **HashMapConcurrente**. Esta tabla de hash cuenta con 26 buckets, uno para cada letra del abecedario, cada uno de ellos representado por una *ListaAtómica*.

La primera función que implementamos fue **void incrementar(string clave)**. Si la clave pasada por parámetro existe en la tabla (en la lista del bucket correspondiente), debemos incrementar su valor en uno. Si no existe, se debe crear el par <clave, 1> y agregarlo a la tabla.

Para hacer esto debemos fijarnos en el bucket que le corresponde a esa clave, recorrer toda la lista de éste y ver si está o no. En nuestra implementación declaramos un vector de 26 mutexs en la estructura de **HashMapConcurrente**, una por cada letra *vector<mutex>mutexHash{26}*. Esto lo utilizamos a la hora de ver si existe esa clave en la tabla e incrementar su valor, o de agregarlo para proteger el acceso a cada bucket individualmente. Con esto permitimos que diferentes hilos accedan y modifiquen pares clave-valor en diferentes buckets de forma concurrente sin bloquearse entre sí. Además si dos o más hilos quieren operar sobre el mismo bucket, no colisionan y no hay contención entre ellos.

Aclaración: *en todos los pseudocódigos se omite el chequeo de intentar acceder a un valor inválido, por ejemplo el hashIndex.*

[6] compare_exchange_weak

Algorithm 2 Hash map concurrente incrementar

```
1: Parámetros: string clave
2: idx ← hashIndex(clave)                          ▷ Obtener índice hash de la clave
3: tablaActual ← tabla[idx]
4: mutexHash[idx].lock()                           ▷ Mutex para evitar condiciones de carrera
5: it ← tablaActual.begin()                        ▷ Iterador al primer elemento de la lista
6: ultimo ← tablaActual.end()                      ▷ Iterador al final de la lista
7: while it ≠ ultimo ∧ it.clave ≠ clave do
8:     it ← it.next()                              ▷ Avanzar iterador hasta el final o clave
9: end while
10: if it ≠ ultimo then
11:     it.valor ← it.valor + 1                    ▷ Incrementar valor si la clave ya existe
12: else
13:     nuevoPar ← (clave, 1)                      ▷ Crear nuevo par clave-valor
14:     tablaActual.insertar(nuevoPar)             ▷ Insertar el nuevo par en la tabla
15: end if
16: mutexHash[idx].unlock()                        ▷ Liberar el mutex
17: return
```

La segunda función a implementar fue **vector<string>claves()**, la cual debía devolver todas las claves existentes en la tabla. Además no debía ser bloqueante y sí libre de inanición.

En esta implementación, no hay condiciones de carrera porque cada segmento de la tabla hash está protegido mediante un mecanismo de bloqueo individual. Antes de acceder las claves se bloquean los mutex y estos se van liberando a medida que terminamos de leer, garantizando acceso exclusivo a ese segmento durante la operación de lectura. Esto asegura que, si otro hilo está modificando el mismo segmento, dicho hilo deberá esperar a que el bloqueo sea liberado antes de realizar cambios.

Como resultado, los hilos pueden operar en paralelo sobre diferentes segmentos de la tabla hash, garantizando un alto grado de concurrencia. Aunque las claves devueltas podrían no reflejar el estado más actualizado de la tabla (es decir, pueden ser valores "viejos"), estas siempre serán consistentes y válidas en el contexto del diseño.

Algorithm 3 Hash map concurrente claves

```
1: claves ← ∅                                      ▷ Inicializamos el vector de claves vacío
2: for idx ← 0 to cantLetras − 1 do
3:     mutexHash[idx].lock()
4: end for
5: for idx ← 0 to cantLetras − 1 do
6:     for all p ∈ tabla[idx] do                   ▷ Recorremos cada par (clave, valor) en la tabla
7:         claves.push_back(p.clave)               ▷ Agregamos la clave al vector
8:     end for
9:     mutexHash[idx].unlock()
10: end for
11: return claves
```

Para la tercera tuvimos que implementar **unsigned int valor(string clave)** que devuelve el valor de clave o 0 si la clave no existe en la tabla. Para hacer esto recorremos el bucket al que pertenece la clave ingresada hasta encontrarla y retornamos su valor. En caso de recorrer todo el bucket y no encontrar la clave, devolvemos cero.

No se producen condiciones de carrera porque la función no realiza modificaciones en los datos. Únicamente se lee el contenido del bucket. Además, no es necesario usar bloqueos de escritura, ya que no estamos alterando el estado del hash map ni de las claves. Si bien pueden haber inconsistencias si otro proceso modifica un bucket que ya leímos del hash map con *incrementar()*, esto no afecta a la correcta lectura de las claves, ya que las modificaciones no interfieren directamente con el acceso a la clave en la lista.

Algorithm 4 Hash map concurrente valor

```
1: idx ← hashIndex(clave)                          ▷ Obtener índice hash de la clave
2: tablaActual ← tabla[idx]                        ▷ Obtener la lista correspondiente al índice
3: if tablaActual.longitud() = 0 then
4:     return 0                                    ▷ Si el bucket está vacío, retornar 0
5: end if
6: it ← tablaActualbegin()                         ▷ Iterador al primer elemento del bucket
7: ultimo ← tablaActual.end()                      ▷ Iterador al final del bucket
8: while it ≠ ultimo do
9:     if it.clave = clave then
10:        return it.valor                         ▷ Retornar el valor si encontramos la clave
11:    end if
12:    it ← it.next()                              ▷ Avanzar iterador
13: end while
14: return 0                                       ▷ Si no encontramos la clave, retornar 0
```

Un detalle que modificamos en las funciones ya implementadas por los profesores es que para la función **hashIndex(string clave)**, como nuestra tabla de hash, están los buckets en minúsculas, para el caso en el que nos ingresen palabras con la primera letra en mayúscula le agregamos un *tolower()* para que no nos devuelva un index inválido.

Algorithm 5 Hash map Index

```
1: Parámetros: string clave
2: return tolower(clave[0]) - 'a'
```

Para verificar tanto la correcta inserción de claves, como el conteo preciso de sus valores y claves realizamos distintos test. Ya vino implementada la validación que un HashMap vacío retorne cero al consultar el valor de una clave inexistente y que, al verificar las claves, no existan entradas y varios casos base (como agregar muchas veces la misma palabra y validar su valor en *ClavesEsCorrectoTrasVariasInsercionesMismoBucket*).

Nosotras agregamos pruebas más complejas al evaluar el rendimiento y la consistencia bajo condiciones de alta concurrencia, donde varios hilos insertan elementos simultáneamente, asegurando que todos los elementos estuvieran presentes y que sus contadores fueran correctos tanto en escenarios simples como intensivos. También testeamos los casos en donde hubieran palabras con alguna mayúscula. Estos casos fueron esenciales para validar la robustez y eficiencia del HashMapConcurrente en entornos multihilo.

El código se encuentra en **/src/UnitTest2.cpp**.

2.2.2. Segunda parte

Se nos pidió completar otras dos funciones de *HashMapConcurrente*; promedio y promedioParalelo.

Para realizar el **float promedio()**, vamos recorriendo cada bucket en orden y acumulando las sumas en una variable, además de tener un contador de cuántas claves totales tenemos. Cuando finalizamos de recorrer toda la estructura, calculamos el promedio y lo devolvemos. Esta implementación no es bloqueante ya que solo es de lectura, por lo tanto como no modifica los valores no necesitamos de ningún mecanismo de sincronización.

Si las funciones *promedio()* e *incrementar()* se ejecutan concurrentemente, pueden surgir inconsistencias que hagan que el valor retornado por *promedio()* un valor no actualizado de la tabla. Esto ocurre porque los datos que está leyendo *promedio()* pueden cambiar mientras terminamos de recorrer todos los buckets. Para garantizar un valor viejo pero consistente, *promedio()* debe bloquear todos los segmentos de la tabla *hash* antes de iniciar cualquier lectura. Esto asegura que los datos leídos reflejen un estado completo y coherente de la tabla, incluso si otro hilo realiza modificaciones mientras tanto.

1. **Estado inicial:**
   - *tabla*: {(“AA”, 10), (“AB”, 10), (“B”, 20)}

   El promedio actual es:

$$\text{promedio} = \frac{10 + 10 + 20}{2} = 20$$

2. **Ejecución posible:**

   a) Antes de recorrer los segmentos, *promedio()* bloquea todos los *mutexes* asociados a los segmentos de la tabla *hash*.

   b) Luego, lee los valores de todos los segmentos:

   - De A obtiene los valores 10 y 10. Se libera mutex de A.
   - Es interrumpido por incrementar(AB), entonces ahora AB tiene como valor el 11.
   - Vuelve a ejecutar *promedio()*, de B obtiene el valor 20. Se libera mutex de B.

   c) Calcula el promedio:

$$\text{promedio} = \frac{10 + 10 + 20}{3} = 13{,}33$$

   d) Pero el promedio actual era:

$$\text{promedio} = \frac{10 + 11 + 20}{3} = 13{,}67$$

Este enfoque asegura que *promedio()* siempre devuelva un valor consistente con un estado pasado de la tabla, cumpliendo con el requisito de permitir valores viejos pero posibles, evitando inconsistencias extremas que mezclen estados intermedios.

Algorithm 6 Hash map promedio

```
1:  Sin parámetros
2:  sum ← 0.0                                          ▷ Inicializamos la suma total
3:  count ← 0                                          ▷ Inicializamos el contador de claves únicas
4:  for idx ← 0 to cantLetras − 1 do
5:      mutexHash[idx].lock()
6:  end for
7:  for idx ← 0 to cantLetras - 1 do
8:      for all p ∈ tabla[idx] do                      ▷ Recorremos las claves asociadas a la letra actual
9:          sum ← sum + p.valor                        ▷ Acumulamos el valor de la clave
10:         count ← count + 1                          ▷ Incrementamos el contador de claves
11:     end for
12:     mutexHash[idx].unlock()
13: end for
14: if count > 0 then
15:     return sum/count                               ▷ Devolvemos el promedio final
16: else
17:     return 0,0                                     ▷ No hay claves, devolvemos cero
18: end if
```

Para la función **float promedioParalelo(unsigned int cantThreads)** debemos tener en cuenta que cada hilo se encargará de procesar un bucket a la vez y no podrán haber hilos inactivos mientras haya buckets disponibles. Para cumplir con esto, implementamos tres variables compartidas entre todos los hilos; *acumulador*, *cantidadDeClaves* y *primeraLetraDisponible*, son variables atómicas.

La estrategia utilizada distribuye dinámicamente los buckets entre los hilos de la siguiente forma:

- Cada hilo solicita un índice de bucket mediante *primeraLetraDisponible.fetch_ add(1)*. Esta operación atómica asegura que cada hilo procesa un bucket único.
- Una vez asignado un bucket, el hilo calcula la suma local y la cantidad de claves de ese bucket.
- Al terminar, los valores locales se acumulan en las variables globales (*acumulador* y *cantidadDeClaves*) mediante operaciones atómicas.

.ºs recursos compartidos entre los hilos son las variables atómicas mencionadas anteriormente, que se utilizan para acumular la suma de los valores y contar las claves de la tabla. Las operaciones atómicas garantizan la consistencia de estos recursos, permitiendo el acceso concurrente sin conflictos en la modificación de estos valores. Sin embargo, como la tabla hash está dividida en 26 buckets y cada hilo procesará una letra específica de la tabla, es necesario proteger la estructura completa de la tabla para garantizar que no haya interferencias entre los hilos al acceder a los mismos datos.

Por ello, se bloquean todos los mutexes de la tabla antes de que los hilos comiencen a procesar los valores de cada bucket. Una vez que un hilo termina de procesar un bucket, el mutex asociado se libera, permitiendo que otros hilos accedan a esos buckets sin bloqueos innecesarios. Este enfoque asegura que no existan condiciones de carrera y que cada hilo tenga acceso exclusivo a un bucket en su turno

Algorithm 7 Hash map concurrente promedio paralelo

```
primeraLetraDisponible ← 0                          ▷ Inicializamos el índice de letras disponibles (atómico)
acumulador ← 0                                      ▷ Inicializamos el acumulador de valores (atómico)
cantidadDeClaves ← 0                                ▷ Inicializamos el contador de claves únicas (atómico)
threads ← ∅                                         ▷ Vector de threads
for idx ← 0 to cantLetras − 1 do
    mutexHash[idx].lock()
end for
for i ← 0 to cantThreads − 1 do
    threads.emplace_back(thread_funcion, primeraLetraDisponible, acumulador, cantidadDeClaves)
end for
for all thread ∈ threads do
    thread.join()                                   ▷ Esperamos a que cada thread termine
end for
totalClaves ← cantidadDeClaves.load()               ▷ Obtenemos el conteo total de claves
totalSum ← acumulador.load()                        ▷ Obtenemos la suma total
if totalClaves ≠ 0 then
    return totalSum / totalClaves                   ▷ Calculamos el promedio final
else
    return 0,0
end if
```

Algorithm 8 rutinaHiloPromedio

```
Parámetros: primeraLetraDisponible, acumulador, cantidadDeClaves
while true do
    miLetra ← primeraLetraDisponible.fetch_add(1)
    if miLetra ≥ 26 then
        break                                       ▷ No hay más letras, termina el thread
    end if
    localSum ← 0                                    ▷ Inicializamos la suma local
    localCount ← 0                                  ▷ Inicializamos el conteo local
    for all p ∈ tabla[miLetra] do                   ▷ Recorremos las claves de la fila
        localSum ← localSum + p.valor
        localCount ← localCount + 1
    end for
    mutexHash[idx].unlock()
    acumulador.fetch_add(localSum)                  ▷ Actualizamos el acumulador global de forma atómica
    cantidadDeClaves.fetch_add(localCount)          ▷ Actualizamos el conteo global de forma atómica
end while
```

Para evaluar la funcionalidad del cálculo de promedios en el HashMapConcurrente, realizamos una serie de pruebas enfocadas tanto en el promedio simple como en el promedio paralelo.

Se nos dieron unos test en donde el promedio calculado para un conjunto básico de inserciones debe ser correcto. Posteriormente, exploramos la funcionalidad del promedio paralelo mediante el test *PromedioConcurrenteEsCorrecto*, donde el cálculo realizado por múltiples hilos también retornó el valor correcto.

Evaluamos la robustez del método en condiciones extremas en *PromedioConcurrenteMasHilosQueLetras*, donde se probó que el promedio se mantuviera estable incluso con más hilos que claves.

Finalmente, en el test *PromedioConcurrenteDiezHilosPidiendoYAgregando*, implementamos un escenario en el que varios hilos insertan y piden el promedio simultáneamente, garantizando que las inserciones se hayan realizado correctamente y que el promedio calculado fuera consistente. Estas pruebas aseguraron la eficacia y precisión del sistema en entornos de alta concurrencia.

El código se encuentra en **/src/UnitTest3.cpp**.

2.3. Tercera Parte

Para esta parte se nos pidió implementar dos funciones: **int cargarArchivo(HashMapConcurrente &hashMap, string filepath)** y **void cargarMultiplesArchivos(HashMapConcurrente &hashMap, unsigned int cantThreads, vectorstring filePaths**).

Para la función *cargarArchivo* tuvimos que leer palabras de un archivo y agregarlas al hash map, incrementando su valor si ya existía la clave o agregando un nuevo par en caso contrario. La función se realiza usando la operación *incrementar* del *hashMapConcurrente*, que maneja el acceso concurrente a la tabla de hash. Como la función *incrementar* ya tiene los mecanismos necesarios para asegurar la correcta sincronización de los datos usando los 26 mutexes para proteger los buckets individuales, no fue necesario implementar ningún manejo adicional de la sincronización en esta función. Lo único modificado en la función proveída fue:

Algorithm 9 cargarArchivo

```
1: Parámetros: hashmap, filepath
2: cant = 0
3: while file >> palabraActual do
4:     hashMap.incrementar(palabraActual)
5:     cant ++
6: end while
7: return cant
```

La función **cargarMultiplesArchivos** permite cargar de manera concurrente múltiples archivos en un HashMapConcurrente utilizando varios hilos. Para evitar que algún hilo quede inactivo mientras otros están ocupados, se utilizó un contador atómico *archivosCargados* para repartir dinámicamente los archivos entre los hilos. Cada hilo obtiene el índice del próximo archivo a procesar utilizando la operación fetch_add del contador atómico. Esto garantiza que los archivos se procesen en paralelo y que cada hilo reciba un archivo nuevo tan pronto como termina con uno.

Primero se crea la variable *archivosCargados* atómica inicializada en 0. Luego, se crean los hilos y a cada uno se le asignó la tarea de procesar los archivos utilizando la función auxiliar cargarArchivoThread. Finalmente, se espera a que todos los hilos terminen su ejecución antes de finalizar la función.

Algorithm 10 cargarArchivoThread

```
1: Parámetros: hashMap, filePaths, archivos
2: for i ← archivos.fetch_add(1) to |filePaths| − 1 do
3:     CARGARARCHIVO(hashMap, filePaths[i])        ▷ Procesamos el archivo en la posición i
4: end for
```

Para validar la funcionalidad de las operaciones de carga de archivos en el HashMapConcurrente, implementamos varios casos de prueba centrados en cargarArchivo y cargarMultiplesArchivos. Se nos dieron unos básicos para verificar que el método cargue correctamente las palabras desde un archivo específico y se ajuste correctamente. También con casos de crear uno/dos hilos para cargarlo.

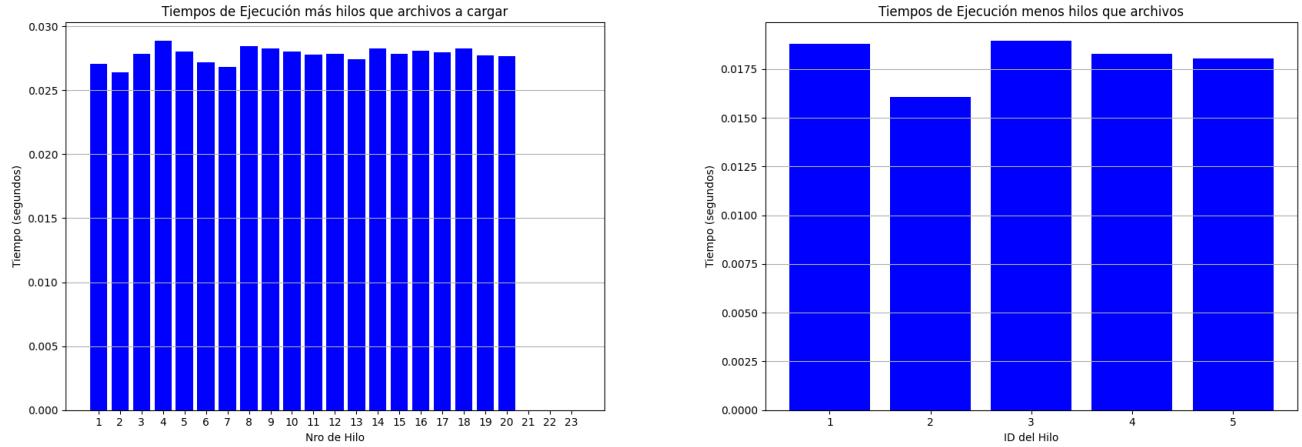
Agregamos los casos de tests *CargarMultiplesArchivosFuncionaCincoThreads* y *CargarMultiplesArchivosFuncionaCorpus* que profundizaron en la capacidad de manejar grandes volúmenes de datos. En estos, se validó que al cargar archivos repetidamente, el sistema mantuviera la integridad de los datos, verificando que los contadores se ajustaran conforme al número de inserciones.

Finalmente, los tests *CargarMultiplesArchivosFuncionaDosHilos* y *CargarMultiplesArchivosFuncionaDiezHilos* sometieron el sistema a una mayor carga de trabajo, usando hasta diez hilos para insertar datos de un archivo y asegurando que los resultados reflejaran la cantidad total de palabras procesadas correctamente, multiplicadas por las inserciones realizadas. Estos casos de prueba demostraron la robustez y eficiencia del sistema bajo diversas condiciones de concurrencia.

3. Experimentación

Nos resultó interesante analizar dos escenarios para la función *cargarMultiplesArchivos*, cuando la cantidad de hilos creados supera a la cantidad de archivos a procesar (Figure 1a), y cuando la cantidad de hilos es menor (Figure 1b). Cuando se crean más hilos que archivos para procesar, observamos un comportamiento interesante en la gestión de carga de trabajo. En estos casos, algunos hilos terminan su ejecución sin realizar ninguna operación. Algunos hilos podrían intentar acceder a la cola de archivos, pero como ya se han procesado todos los archivos, no encontrarán nada que hacer y simplemente finalizarán (como podemos ver en Figure 1a).

Este fenómeno resalta la importancia de equilibrar la cantidad de hilos con la cantidad de trabajo disponible, ya que una mayor cantidad de hilos no siempre se traduce en un mejor rendimiento. En cambio, un enfoque más equilibrado, donde el número de hilos estuviera alineado con la carga de trabajo, podría resultar en una ejecución más eficiente y efectiva (como podemos ver en Figure 1b).



(a) Más hilos que archivos a cargar

(b) Menos hilos que archivos a cargar

Figura 1: Comparación del tiempo de ejecución con diferentes cantidades de hilos

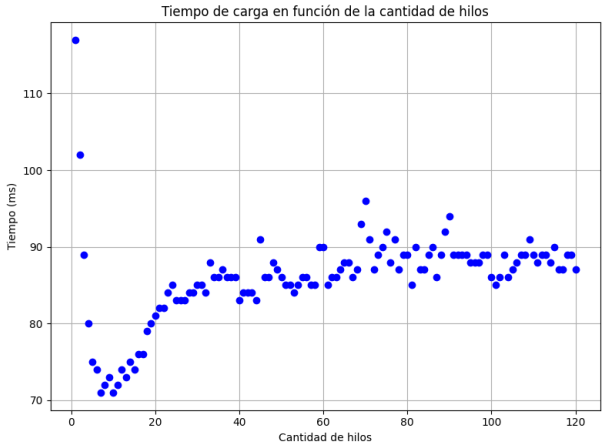


Figura 2: Tiempo de carga en función de la cantidad de hilos.

Adicionalmente, se variaron sistemáticamente la cantidad de hilos y archivos para identificar cómo esta relación afecta el tiempo total de ejecución y determinar el punto óptimo en el que se logra el mejor equilibrio entre paralelismo y eficiencia. En Figure 2 vemos cómo el rendimiento de la carga de archivos varía cuando se cambia el número de hilos involucrados en el proceso. A medida que se incrementa el número de hilos, el tiempo de carga sigue una tendencia que no es necesariamente lineal. El tiempo de carga no disminuye proporcionalmente con el aumento de hilos, lo que sugiere que hay un punto en el que añadir más hilos no resulta ser más eficiente y puede generar un sobrecosto de tiempo debido a la sobrecarga de gestión de hilos.

4. Conclusiones

La solución actual es adecuada para manejar cargas concurrentes moderadas debido al uso de estructuras como HashMapConcurrente y sincronización por medio de mutexes. Sin embargo, considerando un escenario como el Black

Friday, donde millones de compras ocurren simultáneamente, podrían surgir los siguientes inconvenientes y oportunidades de mejora:

Inconvenientes posibles:

- **Contención en los mutexs:** El uso de 26 mutexes para proteger los buckets del hash map puede generar contención si muchas compras simultáneas intentan acceder a la misma clave o bucket. Esto podría llevar a una disminución en el rendimiento debido al tiempo que los hilos pasan esperando por el acceso al recurso.
- **Ineficiencia en la distribución del trabajo:** En la función *cargarMultiplesArchivos*, si la cantidad de hilos es desproporcionada respecto a los archivos disponibles, algunos hilos estarán inactivos, desperdiciando recursos. Además, un enfoque de asignación más dinámico podría ser necesario para grandes volúmenes de datos.

Oportunidades de mejora

- **Mejorar la estrategia de hash y partición:** Implementar una tabla de hash más grande y escalable, donde el número de buckets crezca dinámicamente según la carga. Usar un algoritmo de hashing más uniforme para distribuir mejor las claves entre los buckets, minimizando la contención.
- **Monitoreo y balance dinámico de hilos:** Implementar un sistema de monitoreo que ajuste dinámicamente el número de hilos según la carga real, evitando la creación de hilos innecesarios o inactivos.

Si en la tienda tengo solo dos elementos, probablemente los dos estén en dos buckets de la tabla hash. Si hay concurrencia, muchos hilos van a intentar acceder al mismo tiempo a estos dos buckets, lo que va a generar alta contención en esos buckets. Muchos hilos van a estar bloqueados esperando a que otros terminen su operación de incrementar. Cuando se trabaja con pocos elementos, el uso de concurrencia no es útil porque no hay suficiente carga de trabajo distribuido. Los hilos gastarían tiempo esperando a acceder a los buckets, lo que llevaría a insuficiencias. Para este caso, una implementación secuencial o con un máximo de dos hilos (uno para cada bucket) resulta más eficiente. Esta alternativa elimina las esperas innecesarias asociadas a la sincronización, reduce la complejidad de implementación y garantiza un uso más eficiente de los recursos del sistema.

Para implementar la función $calcularMedianaParalela$ de manera concurrente, primero debemos recolectar todos los valores almacenados en la tabla de hash, luego ordenarlos y, finalmente, calcular la mediana. Un posible problema es que varios hilos podrían intentar acceder a los datos (leerlos o modificarlos) en los buckets al mismo tiempo, lo que podría generar *race conditions*. Para evitar esto, podemos hacer que cada hilo que acceda a un bucket tenga su propia sección crítica.

La sección crítica estará protegida mediante un mutex para cada bucket, lo que evitará que otros hilos accedan simultáneamente a los mismos datos sin necesidad de bloquear toda la tabla hash. Esto también ayuda a evitar cuellos de botella al permitir mayor paralelismo, ya que diferentes hilos pueden trabajar en distintos buckets de manera simultánea. Una vez que los hilos tienen acceso concurrente a los buckets, es necesario juntar los valores. Podemos usar un vector local en cada hilo donde se contengan los valores de los buckets que le han sido asignados. Esto reduce la necesidad de sincronización entre los hilos ya que cada hilo trabaja de forma independiente con sus valores.

Para calcular la mediana, necesitamos que los valores estén ordenados. Lo ideal es implementar algoritmos de ordenamiento paralelo. Cada hilo puede ordenar su subconjunto de valores de manera local. Luego, los resultados de los diferentes hilos pueden ser juntados eficientemente, usando un algoritmo de unión paralela.

Después de que cada hilo ha ordenado su subconjunto de valores, y tras juntar los subconjuntos, podemos proceder a calcular la mediana:

- Si la lista tiene longitud impar, la mediana es el valor en la posición $n/2$ de la lista ordenada (donde n es el número total de elementos).
- Si la lista tiene longitud par, la mediana es el promedio de los valores en las posiciones $n/2 - 1$ y $n/2$ de la lista ordenada.

Como cierre, es importante remarcar que la implementación concurrente mejora el rendimiento al aprovechar múltiples hilos, reduciendo el tiempo de ejecución. El uso de mutexs y variables atómicas garantiza la consistencia de los datos y evita condiciones de carrera. Sin embargo, es crucial manejar correctamente los bloqueos para evitar contención y asegurar que los mutexs se liberen lo antes posible. Aunque la concurrencia mejora el rendimiento, un manejo incorrecto de la sincronización puede generar problemas de eficiencia, por lo que se debe equilibrar cuidadosamente el uso de recursos y la gestión de hilos.