DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Tabla de Hash Concurrente

Junio 2023 Sistemas Operativos

Grupo 20

Integrante	LU	Correo electrónico
Martin Ramos	298/20	martinarielramos329@gmail.com
Mauro Azzollini	774/19	mauro.azzo3210@gmail.com
Agustina Borsato	41/21	agustinaborsato@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 general

1. Introducción

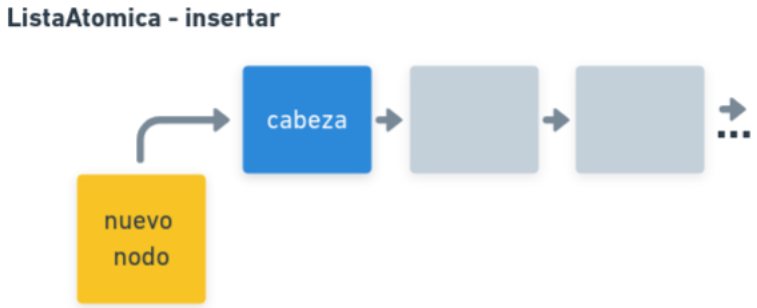
En este trabajo práctico buscamos profundizar una de las facetas más importantes que aparece al estudiar sistemas operativos, la gestión de la concurrencia, centrandonos en la utilización de threads. Se construyó una estructura de datos llamada HashMapConcurrente, que representa una tabla de hash abierta. Su interfaz de uso es la de un diccionario, cuyas claves son strings y sus valores enteros no negativos que representan la cantidad de apariciones de la palabra en el diccionario.

2. Lista atómica

2.1. Explicación

La estructura lista atómica será la que representará cada entrada de la tabla hash abierta que construímos. Lista atómica contiene una lista enlazada que une todas las claves pertenecientes a un bucket (llamamos bucket a cada posición de la tabla), cada nodo de la lista, además de contener la clave en cuestión, contiene la cantidad de veces que aparece esa clave.
Que la lista sea atómica significa que un proceso al querer ejecutar alguna operacion en la lista como insertar un nuevo nodo o modificar el valor de un nodo, no permita la operacion a otros procesos hasta que este haya terminado de realizar el propio.
Nuestra implementación mantiene la atomicidad del método implícitamente ya que cubrimos con un mutex en el momento anterior al que se llama a este y se libero posteriormente a su ejecución. (Para mas detalle véase *incrementar*).

2.2. Ejercicio 1



Se debe insertar el nuevo valor recibido por parámetro al comienzo de la lista. Sabemos que tenemos una referencia al primer elemento de la lista con la variable *cabeza*. Entonces lo que hacemos es generar un nuevo nodo con el valor que recibimos por parámetro, luego hacemos que el nuevo nodo apunte a lo que tenemos como *cabeza*, y por último, sobreescribimos la variable *cabeza* para que ahora sea el nuevo nodo que generamos. Para que el método insertar sea átomico declaramos que el nuevo nodo sea átomico, llamamos a la operación átomica *store* para guardar la dirección del nuevo nodo. Luego, obtenemos la cabeza de la lista con la operación átomica *load* para ser cargada en la variable *_siguiente* del nuevo nodo. Finalmente, cargamos en la cabeza el nuevo nodo utilizando las mismas operaciones átomicas.

Al realizar todas las operaciones de manera átomica, garantizamos que la función insertar es bloqueante.

3. Hash Map Concurrente

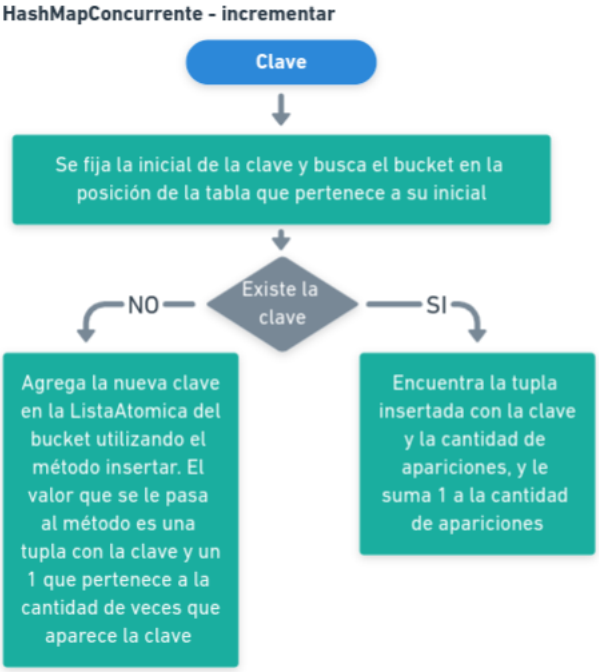
3.1. Explicación

La estructura HashMapConcurrente se encargará de mantener la tabla hash, cada posición de la tabla la llamaremos *bucket*. En total vamos a tener 26 buckets, cada uno corresponde a una letra del abecedario $a, b, c.....$ Dentro de cada bucket se van a almacenar las claves que comienzan con dicha letra.

Letra	Posición	Claves
a	0	palabras que comienzan con a
b	1	palabras que comienzan con b

3.2. Ejercicio 2

a) Incrementar



Tal como muestra el flujo del gráfico, cada posición de la tabla representa un bucket que contiene una ListaAtomica de las claves que comienzan con la letra del bucket en cuestión. Además, lo que vamos a guardar como valor en cada nodo de las ListasAtómicas es una tupla con la estructura $< clave, cantidadDeApariciones >$. Entonces, el proceso que se va a realizar en incrementar es, al recibir una clave, se busca el bucket que debería contener a esa clave mediante la inicial, para eso se usa la función $hashIndex$, que dada una clave, devuelve la posición de la tabla que contiene su bucket. Luego dentro del bucket (ListaAtomica), se revisa si la clave pertenece a la lista, si pertenece se incrementa en 1 (uno) la cantidad de apariciones de esa clave, de otro modo se inserta la clave en la lista utilizando el método insertar de la clase ListAtomica pasándole por parámetro $< clave, 1 >$ representando que esa clave sólo aparece una vez en el diccionario. Para el manejo de la concurrencia, se creó un vector de semáforos que llamamos $permisos\_buckets$, el vector va a contener 26 posiciones, una por cada posición de la tabla. Cada semáforo va a controlar que solo un proceso a la vez pueda escribir dentro de cada ListaAtomica, además, de esta forma garantizamos que dos procesos que quieren escribir en dos buckets diferentes, puedan hacerlo sin problema ya que lo que haremos es pedir permiso al semáforo que pertenece a la entrada de la tabla específica que queremos modificar.

b) Claves

Para la implementación de esta función, recorremos todos los buckets, y por cada uno de ellos recorremos toda la lista y almacenando cada una de las claves. Como necesitamos que no sea bloqueante, se bloquean los buckets de a uno para adquirir las claves y una vez guardadas se desbloquea, así evitamos bloquer toda la tabla.

c) Valor

Para la implementación de esta función, buscamos el bucket que deberia contener a la clave, y luego recorremos la lista atomica, si encontramos la clave devolvemos el valor que contiene en la segunda posición de la tupla almacenada. De otro modo, devolvemos 0. Como necesitamos que no sea bloqueante, se bloquean los buckets de a uno para recorrerlos y luego se desbloquea.

3.3. Ejercicio 3

a) Máximo

La función máximo lo que hace es recorrer todos los buckets buscando la clave que tenga almacenada el máximo valor. Esta búsqueda la hace en orden de los buckets, es decir, alfabéticamente, primero comienza con el bucket que contiene las claves que comienzan con A, luego el bucket de las que comienzan con B y así sucesivamente. Por otro lado, sabemos que puede ejecutarse concurrentemente con la función incrementar, lo que puede traer problemas de concurrencia. Sabemos que incrementar podría modificar los valores con los que *maximo* compara para obtener el máximo valor, entonces se podrían dar trazas como la siguiente:
Supongamos que se esta ejecutando el método *maximo* en un proceso α, y que el estado actual en el que nos encontramos es la iteración del bucket de la letra M, es decir, ya se recorrieron todos los buckets hasta el L incluído. En ese momento, se da el caso en el que el scheduler decide sacarle el control del procesador a α y se lo otorga a otro proceso llamado θ que ejecuta la función *incrementar* y lo hace con claves que comienzan con la letra M en adelante, y luego le devuelve el control a α. El problema que vamos a tener, es que α va a retomar su ejecución donde estaba, pero el máximo valor ahora podría haber cambiado dado que se ejecutó *incrementar*. Entonces, para solucionar estos problemas de concurrencia, vamos a utilizar los mismos semáforos que manejaban los permisos de los buckets que usaba *incrementar*, cada vez que querramos obtener el máximo valor, vamos a utilizar el permiso de cada bucket, imposibilitando que otros procesos puedan escribir en esos buckets mientras se esta leyendo. El bloqueo que vamos a hacer es parcial, es decir, vamos a bloquear todos los buckets al empezar a leer, pero una vez que se recorre el bucket se lo libera, esto nos asegura que la función devuelva la ”foto”del estado en el momento en el que se la llamó y a su vez no bloquear todos los buckets hasta el final de la ejecución.

b) Máximo Paralelo

En *maximoParalelo* la idea es realizar el mismo proceso que en máximo pero poder dividir el procesamiento en *cantThreads* threads. Inicialmente vamos a utilizar el mismo método para resolver los posibles problemas de concurrencia que en *maximo*, vamos a bloquear todos los permisos de los buckets e ir liberándolos de a uno a partir que vamos procesando cada uno.

Creamos los threads y le asignamos a cada uno la función que va a ejecutar, que llamamos $buscarMaximoThreaded$. Además por referencia le pasamos dos variables ($claveMax$ y $valorMax$) en donde iremos almacenando la clave máxima encontrada con su valor. Por otra parte, tenemos una variable atómica que llamamos *buckets_procesados*, que refiere a la cantidad de buckets que ya recorrimos, como lo vamos a hacer en orden, sabemos que si $buckets\_procesados = i$, el siguiente bucket que debemos procesar es el $i + 1$. Entonces, cada thread se queda ciclando mientras la cantidad de procesados no sea mayor a la cantidad total de buckets. En cada iteración, el thread toma el bucket del valor de la variable *buckets_procesados* e incrementa esa variable en 1 para que el siguiente thread tome el próximo. Una vez que el thread sabe en qué bucket debe buscar, lo recorre y va guardando internamente la clave máxima con el valor máximo. Luego que termina el procesamiento, libera el permiso del bucket. Una vez que el thread termina de recorrer todo el bucket, vuelve a entrar en la guarda del ciclo *while*, en donde verifica si quedan buckets pendientes por recorrer, en caso positivo toma el siguiente y lo procesa nuevamente almacenandose localmente el máximo. En caso negativo, entra en una nueva sección crítica, en donde verificamos si el máximo local que tiene almacenado el thread, es el máximo que se calculó entre todos los threads, en tal caso sobreescribe las variables que le habían llegado por referencia como parámetros. Por último, volviendo al padre de los threads, esperamos que todos los threads finalicen su ejecución, y al hacerlo, tendremos dentro de las variables que pasamos por referencia a los threads el máximo valor con su clave respectiva.

3.4. Ejercicio 4

a) Cargar Archivo

Para cargar un archivo, lo que hacemos es recorrer todas las palabras del mismo, y por cada una hacemos un *incrementar* con esa palabra como argumento en la instancia hashMap recibida. No necesitamos hacer uso de ningúna herramienta de sincronización dado que la operación incrementar es atómica.

b) Cargar Múltiples Archivos

Para hacer la carga de múltiples archivos con múltiples threads, lo que hicimos fue muy similar a *maximoParalelo*, mantenemos una variable que es el contador de archivos cargados y que a su vez nos sirve de índice para acceder al archivo en cuestión, entonces cada thread queda ciclando mientras haya archivos por procesar. En cada iteración procesa un archivo llamando a la función *cargarArchivo*, y luego modifica el contador de archivos para indicar cuál es el siguiente archivo.

3.5. Ejercicio 5

Hipótesis

Se espera que mientras más grande sea la muestra (es decir, mientras más palabras haya por archivo y mientras más archivos tenga para cargar) y mayor sea la cantidad de threads utilizada, más rápida sea la ejecución, ya que el trabajo de buscar el máximo y de cargar los archivos estará divido entre los threads que se ejecutarán paralelamente. Utilizar varios threads permite distribuir el trabajo y realizar operaciones en paralelo, lo cual puede acelerar el tiempo de procesamiento total. Además, al tener múltiples threads podemos aprovechar mejor los recursos del sistema, si tenemos varios núcleos de CPU cada thread puede ejecutarse en uno diferente, de forma concurrente.

Por otro lado, podemos esperar que en muestras más pequeñas no sea conveniente el uso de muchos threads. Ya que el costo de utilizacion puede ralentizar la ejecución debido a que el tiempo requerido para la creación, sincronización de estos y cambios de contexto puede ser mayor al tiempo de ejecución real del algoritmo.

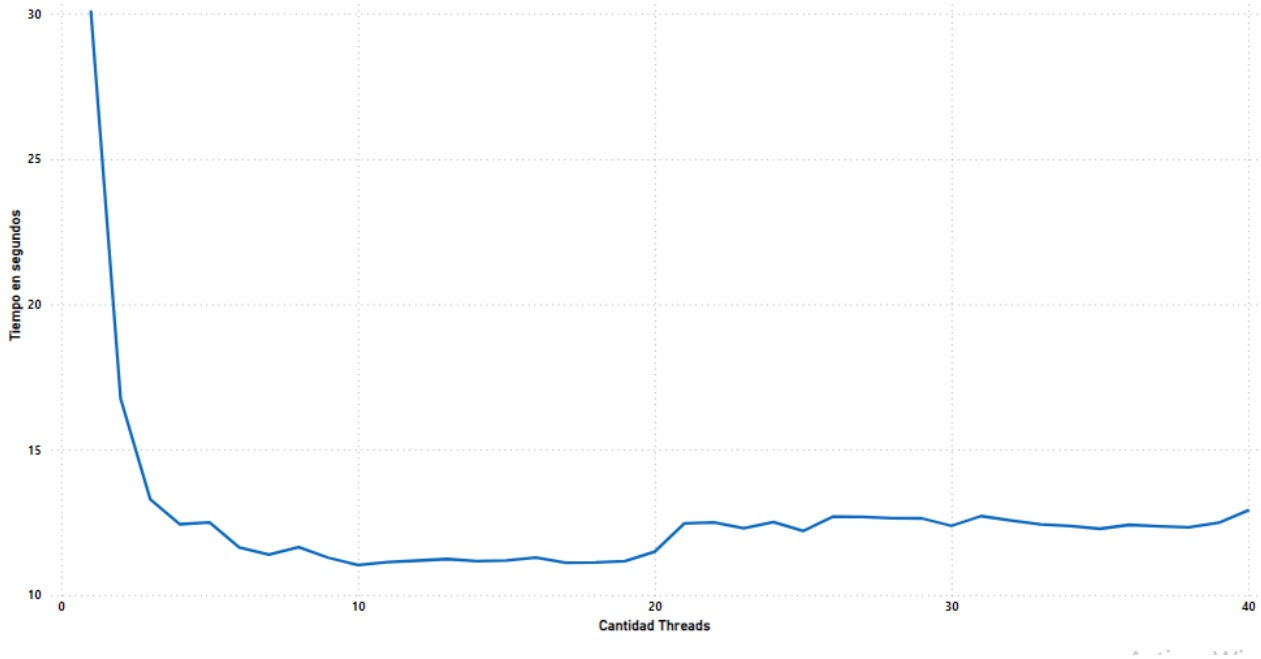
Experimentación

Para evaluar la performance de la ejecución concurrente a la hora de buscar la palabra que aparece más veces en la lista y de cargar los archivos realizaremos una serie de pruebas, midiendo el tiempo de ejecución del programa utilizando muestras de distintos tamaños y variando la cantidad de threads utilizados en las operaciones.

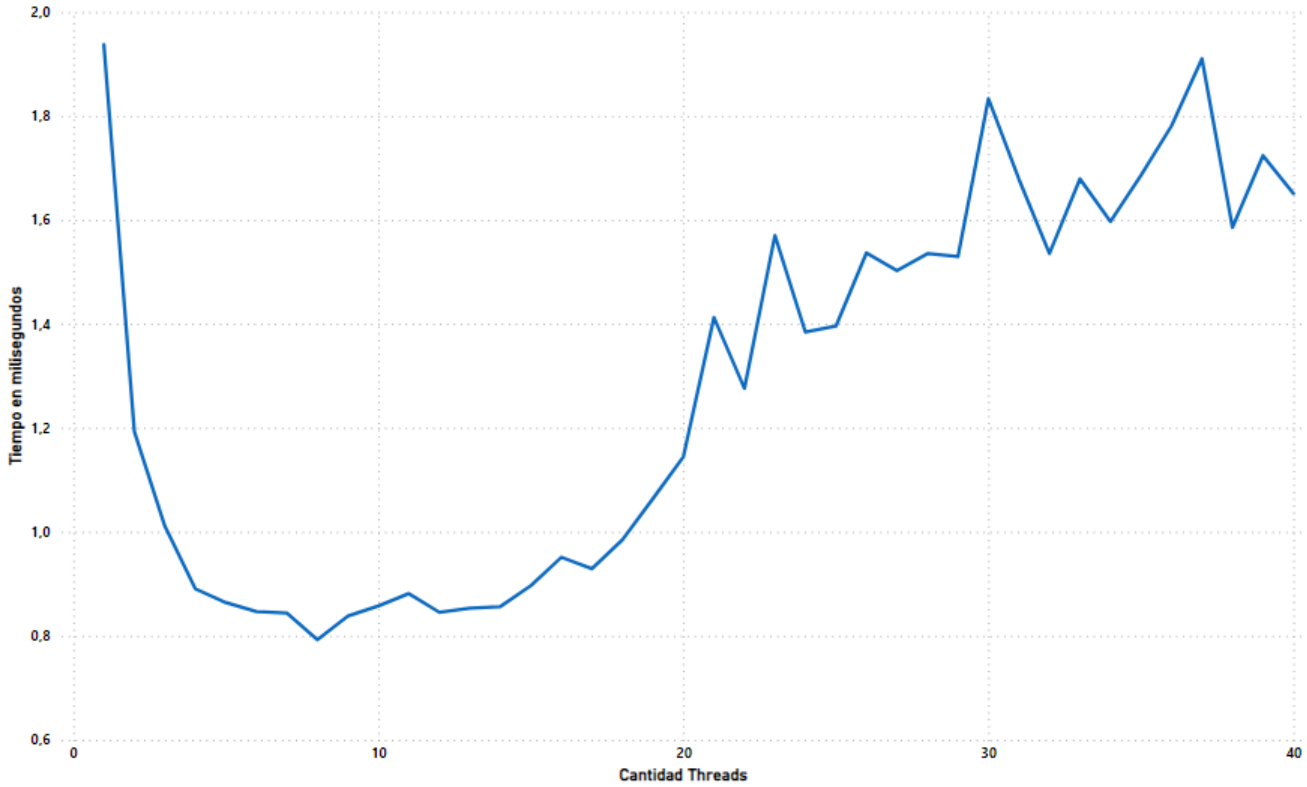
Las pruebas fueran realizadas cargando veintiún archivos de distintas longitudes, con aproximadamente 244.000 palabras en total. En los gráficos podemos ver el promedio de los tiempos de ejecución de los archivos utilizando entre un y veinte threads, con veinte intentos de cada uno. Luego se realizaron pruebas con hasta cuarenta threads para el cálculo del promedio de la ejecucción de máximo paralelo y cargar archivos. Para reproducir nuevamente las pruebas leer archivo *README.txt*. Adicionalmente, se adjunta un archivo *resultados.csv* en el cuál se muestran los resultados de la experimentación.
La computadora en la cual se realizaron los experimentos cuenta con las siguientes caracteristicas:

Especificacion	**Valor**
Architecture	x86_64
Address sizes	39 bits physical, 48 bits virtual
Byte Order	Little Endian
CPUs	12
Core(s) per socket	6
Thread(s) per core	2
CPU max MHz	4600,0000
CPU min MHz	800,0000
L1d cache	192Kib (6 instances)
L1i cache	192Kib (6 instances)
L2 cache	1,5Mib (6 instances)
L3 cache	12Mib

Gráficos



1) Promedio de carga archivos en función de la cantidad threads



2) Promedio de búsqueda de Máximo en función de la cantidad de threads

Conclusiones

Como conclusiones, y en base a las pruebas y experimentaciones presentadas, vemos que los tiempos de ejecución disminuyen cuando la cantidad de threads utilizados en el procesamiento se acerca a la cantidad de Cores que la máquina posee. Cuando esto pasa, tenemos la cantidad óptima de threads para el procesamiento, ya que un thread puede tomar un núcleo. Por otra parte, no tenemos threads adicionales, lo que genera que no haya overheads de cambios de contexto ni creación de threads innecesarios que consumen recursos. Además, la implementación que tenemos fuerza a que todos los threads ejecuten al menos una vez, lo que además hace que si tenemos más threads que lo que la máquina puede soportar en un momento específico, los cambios de contexto

sean inevitables. Debemos tener en cuenta, que las muestras en las que pudimos presenciar la disminución de tiempos son aquellas en las que la cantidad de palabras/archivos eran grandes. Y esto se da por lo planteado en la hipótesis, donde la utilización de múltiples threads es útil siempre en cuando la creación/utilización de los mismos no sea mayor en tiempos que la ejecución propia del algoritmo, en tal caso, las pruebas son ineficaces.