Sistemas Operativos

Trabajo Práctico 1 - Threading

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe
Grupo 10

Integrante	LU	Correo electrónico
Carrasco, Andrés Nicolas	1905/21	nicocarrasco703@gmail.com
Attwell, Marcos	1663/21	marcos.attwell@gmail.com
Bailleres, Juan Manuel	130/15	jm_balle@hotmail.com
Sebastián, Iglesias Mónico	566/18	sebastianiglesias14@gmail.com

Docente	Nota

Índice

1. Introducción

1.1. Antecedentes

Hasta el momento, nos habíamos adentrado al mundo de la programación concurrente, un paradigma de programación en el que múltiples tareas o subprocesos se ejecutan simultáneamente para mejorar la eficiencia y el rendimiento de un programa. Vimos que es especialmente útil en problemas que involucran tareas independientes que pueden ejecutarse concurrentemente.

Se introdujo el concepto de *threads* (o hilos de ejecución), los cuales permiten que se lleven a cabo varias ejecuciones en el mismo entorno de un proceso. También estudiamos conceptos como los *semáforos*, *barreras*, *locks*, etc. Con el fin de evitar condiciones de carrera y lograr coherencia de los datos.

1.2. Motivación

Nos encontramos con la disyuntiva entre usar un enfoque de programación concurrente, que permita que varios *threads* puedan ejecutarse en paralelo, y posibilitando que diferentes tareas se realicen al mismo tiempo. Mejorando la eficiencia y el rendimiento, pero a costa de introducir otros problemas, como la necesidad de sincronizar *threads* para evitar problemas como condiciones de carrera. O bien, utilizar la ejecución secuencial, sacrificando la posibilidad de concurrencia.

1.3. Objetivo

En este informe, trataremos de abordar la gestión de la concurrencia en sistemas operativos, mediante el uso de *threads*. Vamos a ver de qué manera influye el uso de la concurrencia en la ejecución de programas y las decisiones tomadas para evitar condiciones de carrera.

2. Problema a resolver

Dado uno o varios textos, queremos poder contar la cantidad de apararíciones de cada palabra dentro de el(los) mismo(s), de manera rápida y eficiente.

Nuestro objetivo es implementar algún tipo de estructura de datos, de manera que permita el uso de la concurrencia para guardar y acceder a está información con velocidad, y posteriormente estudiar el impacto de su utilización para este problema.

2.1. Estructuras a utilizar

Para resolver este problema, vamos a introducir una estructura de datos llamada `HashMapConcurrente`.

Una *tabla de hash* es una estructura de datos que se utiliza para almacenar y recuperar información de manera eficiente. Funciona mediante una *función de hash* que toma una clave como entrada y la convierte en una ubicación en la tabla.

Sobre esta estructura, implementamos un *diccionario*, cuyas claves serán palabras (string) y sus significados la cantidad de apariciones de la misma (entero no negativo). Nuestra *función de hash* será la primera letra de cada palabra, y gestionaremos las colisiones utilizando listas enlazadas.

3. Implementación de la solución

Para poner en funcionamiento a el `HashMapConcurrente` necesitaremos implementar una serie de métodos (operaciones) para poder manejar la estructura, tanto para la *lista enlazada* que maneja las colisiones, como para la *tabla de hash* que almacenará la información.

3.1. Lista enlazada - `ListaAtomica.cpp`

Esta estructura debe poder ser utilizada de manera concurrente, es por eso que sus operaciones deben ser atómicas, es decir que una vez iniciadas no pueden ser interrumpidas hasta que finalicen su ejecución. Cuando una lista enlazada es atómica, las operaciones se ejecutan secuencialmente, lo que garantiza que un *thread* completo termine su operación antes de que otro pueda comenzar una operación similar en la misma estructura. Esto evita inconsistencias en los datos y garantiza la integridad de la lista.

La razón principal para requerir que la lista sea atómica es evitar condiciones de carrera. Estas ocurren cuando varios *threads* intentan modificar la misma estructura de datos al mismo tiempo, lo que puede llevar a resultados impredecibles y errores en el programa. Por ejemplo, si dos *threads* intentan agregar elementos a una lista enlazada no atómica al mismo tiempo, podrían causar una corrupción de datos, donde los nodos se agregan de manera incorrecta o se pierden debido a la interferencia mutua.

Gracias a la atomicidad podemos garantizar la propiedad de exclusión mutua, evitando así que dos o más *threads* intenten modificar el mismo dato a la vez.

3.1.1. Insertar elemento al inicio: `insertar(T valor)`

Debemos contar con una operación atómica para poder insertar un nuevo nodo al inicio de la lista. Para ello lo primero que se debe hacer es crear un nuevo nodo, esto se logra solicitando memoria al sistema para crear el nuevo nodo a insertar. Notar que esta operación no necesita ser atómica, pues no se accede ni modifica directamente a la estructura en este paso.

Luego, se carga la cabeza de la lista mediante la operación atómica `load()` para evitar que alguien la modifique mientras se esta cargando y luego se guarda como siguiente elemento en el nuevo nodo.

Finalmente, utilizando la operación `compare_exchange_strong()` se hace una comparación de la cabeza de la lista con la cabeza anterior, de ser iguales, se guarda en la estructura el nuevo nodo como cabeza de la lista.

3.2. Hash Map - `HashMapConcurrente.cpp`

Pasamos a la implementación de la estructura principal, el *hash map*. Su interfaz cuenta con varias operaciones, en particular nos interesan `valor`, `claves`, `incrementar`, `maximo` y `maximoParalelo`.

Para los primeros tres métodos, creamos un arreglo de 26 mutex, uno para cada clave del *hash map*, en la sección privada del mismo.

3.2.1. Obtener significado: `valor(string clave)`

Utilizamos este método para obtener el valor almacenado de una clave en el *hash map*, si esta no existe se retorna 0.

Es evidente que si bloqueamos el acceso a todo el *hash map*, o una parte grande, estamos sacrificando performance, ya que los demas *threads* no podrán realizar ninguna operación dentro de la estructura mientras se esté buscando la clave.

Por otra parte, necesitamos evitar el caso en que otro *thread* pueda modificar el valor de la clave que estamos buscando antes de que la encontremos, pues esto causará una inconsistencia.

Para evitar incurrir en condiciones de carrera, y evitar más contención de la necesaria, utilizamos un mutex que bloquea el acceso al *bucket* que corresponde a la clave ingresada, con esto logramos que mientras se este realizando la búsqueda de la clave, no se pueda acceder a esa sección del *hash map*. De esta manera podemos evitar inconsistencias en los datos, garantizando encontrar el mismo valor que estaba almacenado al momento de ejecutar la operación.

3.2.2. Incrementar clave: `incrementar(string clave)`

Si queremos incrementar el valor de una clave determinada dentro del *hash map*, usaremos este método. En caso de que la clave no exista, esta se agregará al *hash map* con valor 1.

De manera similar al método `valor()`, bloqueamos sólo el mutex correspondiente al *bucket* de la clave que nos interesa. Así evitamos un exceso en la contención, nos protegemos de posibles condiciones de carrera y logramos una mejor performance.

3.2.3. Obtener claves: `claves()`

Este método devolverá el conjunto de todas las claves almacenadas. Para esto, vamos a iterar por toda la tabla y por cada *bucket* vamos a iterar en su lista para obtener sus claves.

Notar que este retorna sólo el conjunto de claves que han sido guardadas hasta ese momento. Si luego de ejecutar la operación, se inserta una clave nueva en un *bucket* que ya fue recorrido, esta no es tomada en cuenta.

El problema surge si un *thread* quiere insertar una nueva clave a un *bucket* que todavía no fue recorrido, en cuyo caso tendremos una condición de carrera.

Para solventar esto, vamos a bloquear el acceso a toda la tabla al comienzo de la ejecución. Luego, al terminar de recorrer un *bucket*, lo desbloqueamos. De esta manera logramos suficiente contención para evitar la condición de carrera, bloqueando por un breve periodo de tiempo la totalidad de la tabla y liberando los *buckets* progresivamente.

3.2.4. Obtener máximo: `maximo()` y `maximoParalelo()`

Un problema que presentaba `maximo()` original con `incrementar()` era que al no bloquear la tabla en ningún momento, se daba lugar a condiciones de carrera.

Consideremos un ejemplo: inicialmente la primer entrada tiene un solo nodo de valor 5 y la segunda uno con 4. La ejecución en la primer iteración registra el 5 como máximo, pero antes de pasar a la segunda iteración, se ejecuta `incrementar` dos veces sobre el nodo de valor 5 y luego lo mismo ocurre en el nodo con 4, quedando 7 y 6 respectivamente. En el segundo ciclo de `maximo` se registrará como máximo el valor 6 que nunca fue el máximo.

Para evitar esto y además permitir un grado de concurrencia, al comenzar la ejecución, se bloquea toda la tabla. Luego se van desbloqueando progresivamente todas las entradas que ya fueron revisadas. Con esto logramos evitar condiciones de carrera, ya que el resultado es el correcto respecto del momento que se llamo a `maximo`. Sin embargo, puede que este desactualizado con respecto al máximo real cuando llega el resultado.

Para implementar `maximoParalelo` tuvimos que realizar algunas modificaciones.

Los *threads* hacen uso compartido de la memoria del *hash map*. Tienen acceso a priori, a todos los *buckets*, por lo que es necesario que compartan variables para enterarse que sección está siendo revisada por otro *thread* y cual es el máximo actualmente.

En primer lugar creamos tres variables: una variable atómica compartida por los *threads* para guardar el índice del *bucket* que estamos, la inicializamos en 0. Un puntero a un par `string` × `int` donde guardamos el elemento más grande encontrado hasta el momento, también compartido por los *threads*. Un mutex que utilizaremos al momento de actualizar el máximo.

Ahora, inicializamos la cantidad de *threads* pasada por parámetro con la función `buscarMaximoThread`, en esta se entra en un loop (mientras el índice sea menor que la cantidad de letras) donde se incrementa atómicamente el índice actual y comprueba si es menor a la cantidad de letras: si es mayor o igual sale del ciclo, pues indica que ya se han recorrido todos los *buckets*. Caso contrario, procede a buscar el máximo en la lista del *bucket* actual. Finalmente, bloquea el mutex para actualizar el máximo (sección crítica), si el valor que encontró es mayor que el valor que se tenía hasta el momento, se sobreescribe. Luego se desbloquean tanto el mutex para actualizar el máximo como el mutex que bloquea ese *bucket*

Con esta implementación logramos sortear posibles condiciones de carrera que podrían haber surgido por la compartición de variables, haciendo que estas sean atómicas ó utilizando mutex para lograr exclusión mutua. Y además utilizando la misma estrategia que en `maximo`, eludimos inconsistencias en los datos.

3.3. Carga de archivos - `cargarArchivo.cpp`

El método `cargarArchivo` se encarga de leer un archivo pasado por parámetro y cargar todas sus palabras en el *hash map* también pasado por parámetro. Se implemento utilizando el método incrementar del *hash map*, quien es el encargado del manejo de las colisiones de *hash*, solucionando los posibles problemas de concurrencia.

En el caso del método `cargarMultiplesArchivos`, toma por parámetros el vector de archivos a leer, la cantidad de threads entre los que se va a dividir el trabajo y un hashmap donde se van cargar las palabras. Para este, utilizamos una idea similar a la de `maximoParalelo`, o sea para cada thread se utiliza `cargarArchivo` y se actualiza un índice atómico que indica cual es siguiente archivo a cargar.

4. Experimentación y análisis

Ya vimos en detalle los desafíos que presentaban implementar un estructura que permita concurrencia y como solventamos los obstáculos que fueron apareciendo a lo largo de la implementación. Ahora, resta ver si realmente el uso de la concurrencia nos ayuda a resolver nuestro problema con mayor eficiencia y velocidad.

4.1. Introducción

Se hicieron 4 experimentos que consideramos importantes para analizar las funciones y como dividir las tareas afecta su performance.

4.1.1. Consideraciones

Se utilizo la biblioteca `std::chrono` para contabilizar el tiempo de ejecución de las distintas funciones.

Para obtener datos mas fiables se corrió cada prueba 5 veces y luego se promedio para obtener el resultado final. Obviamente se desprecio también cualquier resultado muy alejado de la norma (outliers) producido por situaciones ajenas al programa en si y más relacionadas con el ambiente de ejecución.

Se trato en lo posible de correr todos los experimentos en el mismo sistema para obtener resultados consistentes entre si.

También se debe tener en cuenta que no tiene sentido correr los experimentos sobre mas de 26 threads, esto se debe a como esta implementado el *hash map* que contiene 26 listas.

Sistema utilizado:

- CPU: Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz
- Nº de Núcleos físicos: 4
- N º de Threads: 4
- Memoria: 24GB

4.2. Experimento 1 - Distribución uniforme - Máximo Paralelo

Nuestra hipótesis inicial es que a mayor cantidad de threads, mas eficientes van a ser los resultados obtenidos. Obviamente considerando las limitaciones de la implementación y el procesador.

Para este experimento, se utilizó el archivo `igualCantidadDePalabrasXLetra.txt` el cual contiene 4000 palabras para cada letra de la "a" hasta la "l". Se utilizo este dataset para tener un trabajo uniforme entre las listas.

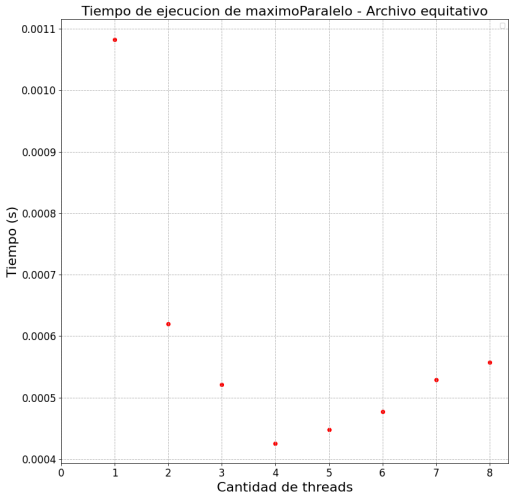

Figura 1: Tiempo de ejecución de la función maximoParalelo hasta 8 threads

Como se puede ver en la figura, los mejores resultados se obtuvieron para 4 threads, esto intuimos sucede por que el procesador esta limitado por sus 4 threads, por lo que no tendria sentido que a mas threads obtenga mejores resultados. También se puede ver que a partir de 4 threads comienza a aumentar de nuevo el tiempo de ejecución, esto puede deberse al trabajo adicional que realiza el procesador en la creación y organización de threads.

4.3. Experimento 2 - Distribución No Uniforme - Máximo Paralelo

La idea del experimento es analizar que sucede cuando no hay una distribución equitativa en la letra inicial de las palabras o sea, un caso contrario al del experimento anterior.

Nuestra hipótesis fue que si se concentra la mayor cantidad de palabras en una cantidad de reducida de filas se iba a ver limitada la performance por como esta implementado el hashMap, o sea, que a partir de cierto punto no va a mejorar la performance por que los threads leyendo las pocas filas con muchas palabras van a ser el limitante.

Se utilizo el dataset `wordset-experimento2.txt` con alrededor de 80000 palabras en el cual las primeras 3 letras concentran el 90 % del total de palabras.

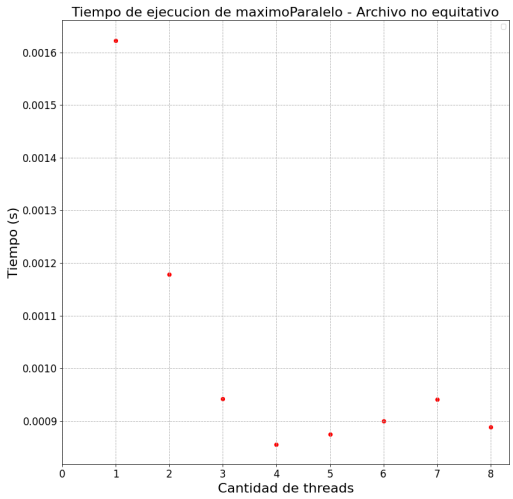

Figura 2: Tiempo de ejecución de la función maximoParalelo hasta 8 threads

Como se puede ver la peor performance se produce con una cantidad de threads menor a 3, esto sucede porque no permite paralelizar el trabajo de calcular las 3 filas. Luego, a partir de 3 threads los tiempos mejoran considerablemente, hasta alcanzar alcanzar el mejor resultado para 4 threads. Esto se debe a que el tiempo de ejecución que le tomara a los threads que calculan la fila a, b y c es mayor al tiempo que tarda un solo thread en calcular el máximo de las otras filas. Esto nos indica que para un dataset con las características del que se esta usando, no importaría si se aumenta la cantidad de threads por que el limitante son esas tres listas.

Finalmente como conclusión vemos que la performance de *hash map* esta ligado a la distribución de las palabras en el dataset.

4.4. Experimento 3 - Carga de archivos - Archivos grandes

La idea en este experimento fue analizar como varia la carga de archivos con distinta cantidad de threads.

Nuestra hipótesis fue, nuevamente, que a mayor cantidad de threads más eficiente iba a ser la carga de archivos.

Se utilizaron 12 datasets con 4000 palabras que inician con la misma letra cada uno.

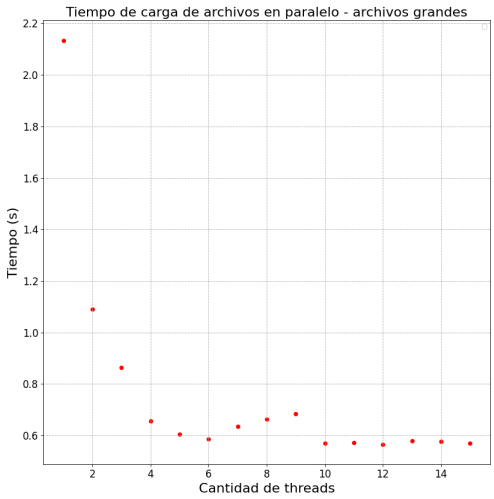

Figura 3: Tiempo de carga para archivos grandes en paralelo

Como podemos ver en los resultados se obtiene una gran mejoría con el aumento de threads. Otra vez limitado por la cantidad de threads del sistema. Lo que podemos observar es que a partir de 12 threads se obtiene una mayor constancia en los tiempos, esto se debe a que se utilizaron 12 archivos.

Para agregar otra comparación tenemos que si combinamos los 12 archivos en un solo archivo, la carga demora 2.56 segundos, o sea que dividiendo la carga en varios archivos y con varios threads se obtiene una amplia ($\sim 400\,\%$) mejoría.

4.5. Experimento 4 - Carga de archivos - Archivos pequeños

La idea de este experimento es intentar ver como afectan otros costos, como por ejemplo la creación y organización de threads, la carga de archivos.

Nuestra hipótesis es que con archivos más pequeños, vamos a poder ver los resultados más afectados por estos costos.

Para este experimento utilizamos los archivos `test-1`, `test-2` y `test-3` 10 veces cada uno. Se corrió hasta en 60 threads.

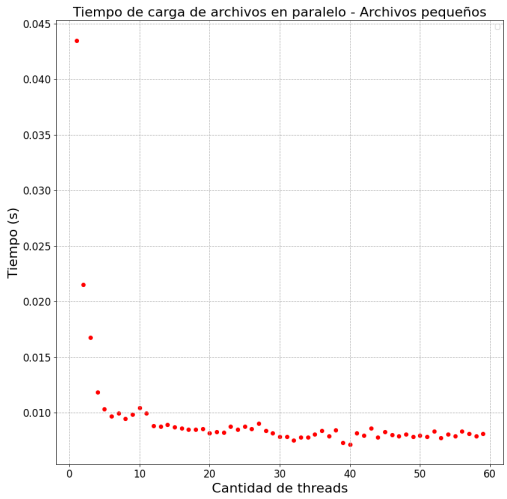

Figura 4: Tiempo de carga para archivos pequeños en paralelo

Como se puede ver en la figura no se noto ningún cambio general en la tendencia a partir de cierta cantidad de threads, si algunos altos y bajos locales pero estos lo atribuimos a procesos que corren de fondo por el sistema operativo y afectan algunas corridas puntuales del programa.

4.6. Replicación

Se agregó al Makefile ya provisto por la cátedra un nuevo target `make experiment` para compilar los cuatro experimentos vistos. Luego para ejecutarlos, se debe ingresar al directorio `src/Experimentacion/` y ejecutar el archivo `.py` que se desea replicar.

En el archivo `README.md` se encuentran las instrucciones.

5. Conclusiones

Con los resultados obtenidos en la experimentación pudimos observar que la concurrencia aumenta considerablemente la performance en tareas que tienen alta carga de procesamiento. Pero también vimos que aumenta la dificultad de producir algoritmos que sean correctos y no tengan condiciones de carrera, *deadlocks* o *starvation*. Ya que testearlos se vuelve muy dificultoso.

También pudimos ver que en otros casos puede dar resultados no tan favorables, como por ejemplo en datasets con distribuciones muy concentradas en algunos puntos, como el del experimento 2. O en el experimento 1, que a partir de cierta cantidad de threads también se puede ver degradado el rendimiento.