Sistemas Operativos

Trabajo Práctico 1

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

Threading

Informe

Integrante	LU	Correo electrónico
Szabo, Jorge Cristian	1683/21	jorgecszabo@gmail.com
Braginski Maguitman, Leonel Alan	385/21	leobraginski@gmail.com
Pannunzio, German	645/21	gepannunzio@gmail.com
Galeota, Andres	626/21	andresvgn@gmail.com

1. Introducción

En el presente trabajo, desarrollaremos dos estructuras de datos las cuales aprovecharan el uso de múltiples hilos de ejecución. Esto abre las puertas a la ejecución de procesos con multiprogramación, y para ello usaremos threads los cuales se entiende como la habilidad de tener entidades (en nuestro caso procesos) paralelos que comparten el espacio de dirección y datos. Específicamente utilizaremos la interfaz `thread` de la biblioteca estándar de `C++` que internamente contiene `pthread` del estándar POSIX. Esto traerá a consideración problemas de sincronización tales como condiciones de carrera, es decir, dado uno o más procesos concurrentes, donde ambos realizan una o más acciones en un mismo dato, la imposibilidad de conocer el resultado final pues el mismo depende de la administración del scheduler a la hora de ejecutarlos. Llamamos a estas regiones de código donde la concurrencia puede traer conflictos, secciones críticas.

Para solucionar esto, utilizaremos los semáforos, un tipo abstracto de datos que permite controlar el orden de ejecución de alguna sección crítica de nuestro código, otra solución serán los tipos de datos atómicos, los cuales garantizan que las operaciones realizadas en la variable de este tipo no permita condiciones de carrera al realizarse bajo concurrencia. Un tipo de semáforo particular será el mutex, el cual excluye todo otro proceso que intente acceder a la sección crítica hasta que termine quien este actualmente en la misma. Otro punto a tener en cuenta es la inanición (starvation) de un proceso, la misma se define como la situación donde un programa se encuentra en ejecución pero no logra realizar progreso, pues se encuentra a la espera de ingresar a alguna región critica y esto no ocurre pues otros procesos tienen prioridad. Un punto de mayor gravedad que la inanición de procesos, es la posibilidad de dealocks, el cual es el bloqueo permanente de la ejecución de uno o más hilos.

Para la realización del trabajo práctico, tomamos algunas libertades controladas sobre la implementación. Entre ellas activamos los flags `-O3 -march=native` en el Makefile con el fin de en el caso de la primera, de habilitar optimizaciones brindadas por el compilador `gcc` y obtener mejores resultados temporales, el otro flag permite utilizar instrucciones de SIMD que normalmente no están incluidas en el target de compilación para x86-64 (como las incluidas en la extensión AVX2).

También utilizamos el struct LightSwitch provisto por el libro "The Little Book of Semaphores"[1], ya que como explicaremos más adelante nos ayudaban a la perfección a resolver uno de los problemas.

A continuación desarrollamos las dos estructuras de datos en cuestión.

2. Lista Atómica

Para este trabajo estamos utilizando una lista diseñada para ser atómica. Esto significa que podemos modificarla o iterar sobre esta sin concurrir en condiciones de carrera. Esto lo logramos con una estructura con cabeza atómica y una lógica particular para la inserción de nuevos valores.

Decir que la cabeza es atómica quiere decir que siempre que estemos operando sobre la misma en caso de vernos interrumpidos por el scheduler no ocurra variabilidad en el resultado a causa de otro proceso concurrente que este operando sobre la misma lista. Y de esta forma garantizar la ausencia de condiciones de carrera y la consistencia en la estructura de datos.

Para el insertado usamos la función `compare_exchange_weak` para garantizar consistencia. De esta forma insertar desde el principio se hace como en cualquier lista normal.

1. Instanciamos un nuevo nodo con el valor a agregar en la lista.
2. Apuntamos desde este nodo a la cabeza.
3. Por último nos queda definir que la cabeza de la estructura pasa a ser este nuevo nodo, aquí usamos `compare_exchange_weak` dentro de un `while` ocasionando un busy waiting/trying. Lo que estamos haciendo es verificar que si la cabeza actual es la misma que con la que empezó el proceso, sobre-escriba esta con el nuevo nodo, sin que ningún otro thread pueda provocar una condición de carrera. Desde un punto de vista de perfomance es mejor que utilizar un mutex dentro de nuestro contexto de uso, al saber que uno se encuentra insertando en ese momento.

3. HashMap Concurrente

La implementación del HashMap es un caso de uso del problema Readers-Writers del libro de semáforos. La idea es que solo se pueda escribir o leer sobre la estructura de a una vez (como si fuese un cuarto al que solo puede acceder un escritor o un lector a la vez), esta solución no genera `Starvation`.

Para lograr esto usamos la estructura `LightSwitch` que bloquea la escritura cuando entra algún lector por primera vez y se libera cuando se van todos los lectores. Y un molinete (representado con un mutex) que bloquea la lectura mientras se este escribiendo.

- `incrementar(string clave)` entra a la sección critica solo cuando no hay lectores encolados, durante la sección critica bloquea el molinete.
- `claves()` y `valor(string clave)` esperan a que el molinete les permita pasar y al entrar en la sección critica de lectura usan el `LightSwitch` para bloquear la escritura.
- Se usa este método por cada entrada de la tabla de Hash.

El HashMap cuenta con la operación `maximo` que lee toda la tabla hasta encontrar el par (clave, valor) más grande almacenado. Notemos que implementar esto sin practicas de sincronización podría ocasionar que el máximo retornado no sea el correcto, veamos este caso:
Donde se ejecutan concurrentemente `incrementar` y `maximo`

Clave	Valor
"Mariscal"	4
"Minuto"	2
"Mundo"	5

Modificación de `incrementar` →

Clave	Valor
"Mariscal"	6
"Minuto"	2
"Mundo"	5

`maximo` termina su ejecución al haber terminado de recorrer todas las entradas en la tabla, en este caso se puede dar la condición de carrera de que se lea <"Mariscal", 4> y <"Mundo", 5>, pero antes de recorrer toda la estructura se ejecuta `incrementar` sobre "Mariscal" ⇒ el máximo retornado deja de ser correcto. Al aplicarle la estructura `LightSwitch` a cada lectura de las entradas de la tabla nos cubrimos para este problema.

Otra decisión tomado fue el hecho de utilizar un semáforo molinete y un lightswitch por lista del hashmap es decir por letra, de forma de reducir el efecto bloqueante de las operaciones y poder de esta forma obtener una mayor granularidad en relación a la concurrencia.

Por último podemos mejorar mucho el rendimiento en esta búsqueda repartiendo la tarea de búsqueda en N threads, para esto hacemos que cada entrada de la tabla sea únicamente procesada por algún thread. Los threads comparten la instancia en memoria de la estructura y usan la misma estrategia de `LightSwitch` en la lectura para mantener concurrencia.

Tenemos así la implementación de `maximoParalelo` la cual utiliza como estrategia de división de trabajo, que luego de crear los N threads, tenga una función que administre a los procesos hilos. Estos últimos solicitaran una fila de trabajo, las cuales serán disjuntas y permitirán que en al momento de finalizar almacenen el resultado parcial en una lista compartida, para nuevamente solicitar una nueva fila que todavía no se haya analizado. Para evitar condiciones de carrera, la forma de distribución de filas hacia los procesos threads se realiza a partir de una variable atómica entera compartida, que indica la fila actual que debe buscarse el máximo, luego usando `fetch_add` y tomando ciertos recaudos podemos asegurarnos que no ocurrirán race conditions. Una vez calculado los máximos parciales la función principal buscará en esta lista compartida el resultado final. Sabemos que la lista de resultados parciales es correcta pues su utilización e indexación se basa en la variable atómica de fila actual previamente descripta la cual cumple no poseer race conditions y en este caso esta propiedad es transitiva a la lista.

3.1. Cargar Archivos

Los datos de entrada son cargados en el `HashMap` con la función `cargarArchivo` o si se tienen varios archivos y se quiere hacer con multiples threads (concurrentemente), podemos usar la función `cargarMultiplesArchivos`.

`cargarMultiplesArchivos` inicializa N threads que llaman a `cargarArchivo`.

Estos métodos no son bloqueantes, ni restrictivos en cuanto a sincronización porque como no estamos escribiendo sobre los archivos no se dan condiciones de carrera. Si se modifica el `HashMap`, pero esto se hace llamando a los propios

métodos de la clase que cuidan de la concurrencia.

Dentro de las estrategias a la hora de implementar `cargarMultiplesArchivos` usamos ideas similares a las de `maximoParalelo`, es decir definimos un proceso administrador, que se encarga de inicializar un `HashMap` y crear N subprocesos threads los cuales tendrán los file paths a modo de vector de string de todos los archivos por cargar. Estos subprocesos tendran acceso compartido al `HashMap` y además un índice compartido que indica el archivo por cargar, este ultimo se define con una variable atómica entera, que se incrementará a medida que los subprocesos lo soliciten. Los procesos threads irán finalizando una vez que no queden archivos por cargar, es decir que el índice actual de archivo sea mayor que la cantidad disponible.

3.2. Experimentación

Dentro de las experimentaciones, planteamos un mismo diccionario de 194443 palabras el cual fue utilizado tanto ordenado y desordenado y luego dividido en 26 partes. De esta forma podemos testear y tomar mediciones de nuestra función `cargarMultiplesArchivos`, para ello usamos la biblioteca chrono de `C++` en particular la función `high_resolution_clock::time_point` la cual nos permitió cronometrar los tiempos de cómputo con distintos números de threads. Lo mismo fue realizado para la función `maximoParalelo` y con ambos datos obtenidos, realizamos un archivo de `CSV` el cual fue utilizado para graficar con `Python` los resultados. Dentro de nuestras hipótesis con relación a los resultados esperados, suponíamos tiempos menores para una cantidad de threads cercana al numero lógico de threads en nuestra unidad de computo (en nuestro caso 12 threads) obtendría mejores tiempos. Por otro lado, en el otro extremo un número muy bajo de threads, tendrá tiempos mas extensos de finalización. Esperamos también que los datos ordenados posean mayor colisiones a causa de la definición de nuestra función de hash, que no es otra cosa que usar la primera letra de la palabra.

Para los tests de rendimiento se evaluaron por separado la carga de archivos y la función máximo. Para cada caso se corrió 2000 veces la función máximo y 10 veces la carga de archivos. Siempre se tomó el mejor tiempo registrado.

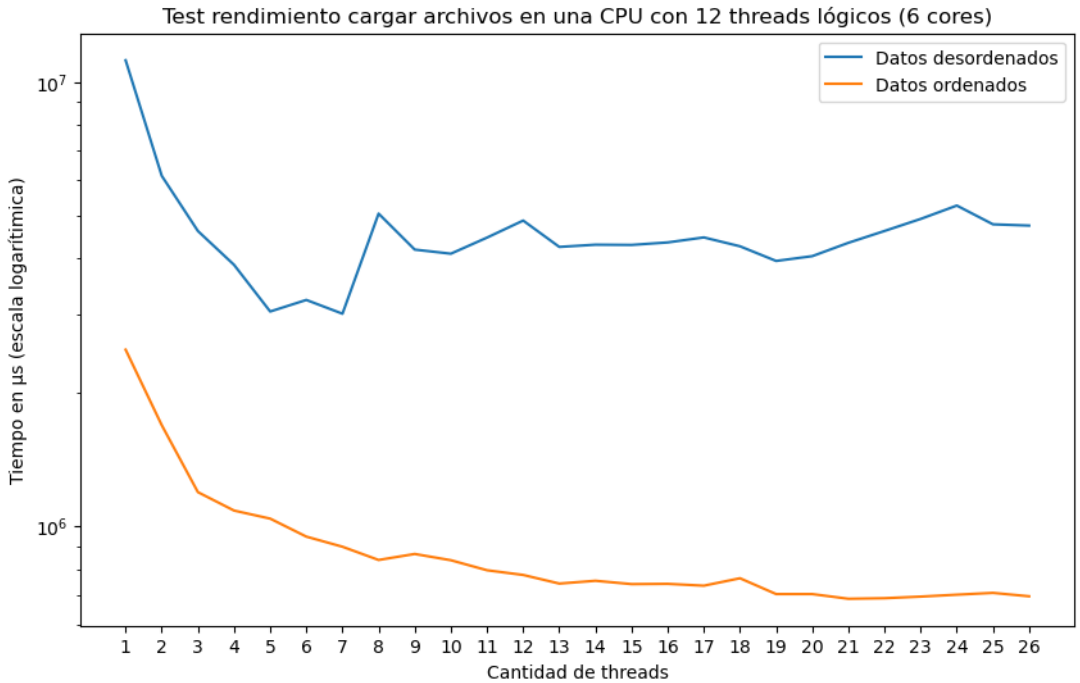


Figura 1: Performance en la carga de archivos para el caso del diccionario ordenado y desordenado.

Como se puede observar en la figura 1 el comportamiento de la carga de archivos está dentro de lo que esperábamos con la hipótesis planteada. Se observa un incremento considerable al incrementar la cantidad de threads de carga de 1 a la cantidad que posee el procesador que se está utilizando cuando se trata de un caso "óptimo"de insertar palabras en orden. Esto se debe a que se crea un escenario para que se den la menor cantidad de colisiones en la función de hash utilizada posibles. Además si siempre se accede a la misma fila de la tabla de hash, es probable que la cabeza de esta lista se encuentre válida en caché y pueda ser modificada sin necesidad de antes buscarla de una jerarquía inferior de memoria.

Cuando se analiza el caso de insertar palabras desordenadas se puede observar que incluso antes de llegar al límite de threads lógicos que posee el procesador se da un decremento considerable en performance. Esto tiene sentido pues la función de hash utilizada es muy propensa a tener colisiones y si hay muchos threads insertando al mismo tiempo es posible que haya más de un thread esperando para insertar en la cabeza de la lista. También desde un punto de vista de coherencia de caché, estar accediendo aleatoriamente a la tabla puede ocasionar que la cabeza de de fila de la tabla a la que se va a insertar se tenga que buscar en una jerarquía de memoria inferior. Más adelante se estudiará el comportamiento de la tabla con *perf* para intentar explicar esto con mayor detalle.

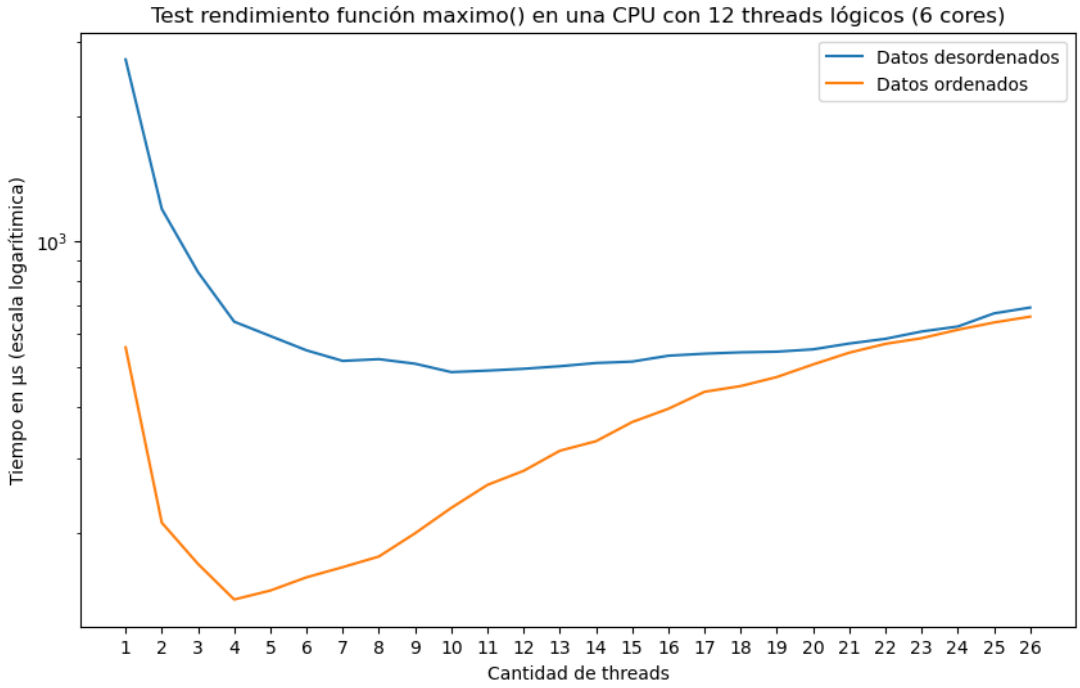
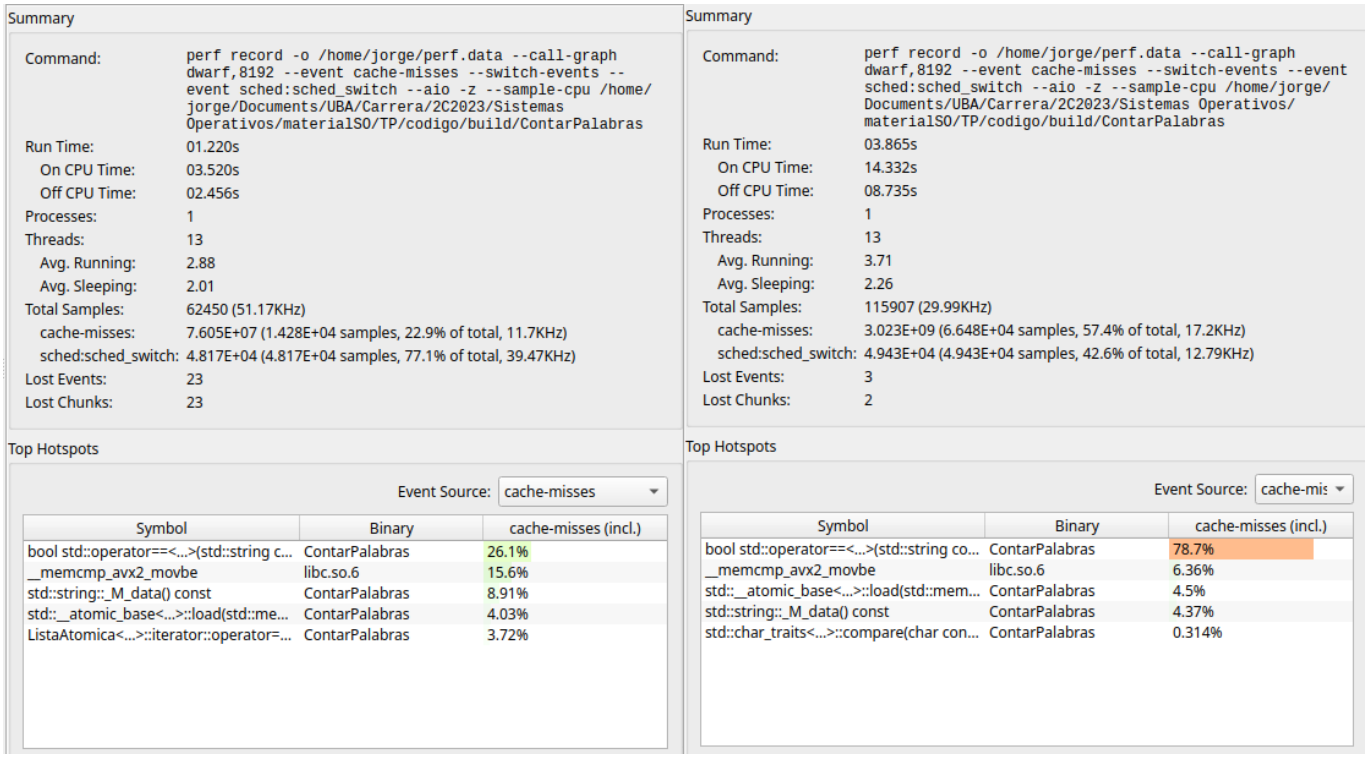


Figura 2: Performance en encontrar la palabra repetida más veces para el caso del diccionario ordenado y desordenado.

Al graficar los datos obtenidos con la función `maximoParalelo` obtenemos la figura 2 en la cual observamos mejores tiempos al utilizar 4 threads en el caso del diccionario ordenado y 10 threads en el caso desordenado. Observamos también que un mayor número de threads utilizados no implica un mejor tiempo, e incluso puede entorpecer a causa de un mayor ratio de context switchs. Esto es especialmente notable cuando se supera la cantidad de threads lógicos del procesador, donde bajo ningún caso de test se puede observar un beneficio. Notamos así, que luego de 20 threads los tiempos coinciden seguramente pues el beneficio esperado por una mayor concurrencia se ve eclipsado por la limitación física de unidades lógicas disponibles, y por el mayor rate de context switchs.

Sospechamos que esta disparidad en tiempo para encontrar el máximo para los mismos datos insertados en un orden distinto en la tabla de hash se debe a que los datos en el caso de insertarse ordenados se encuentran contiguos en memoria. Esto es altamente favorable para la óptima utilización de caché de una CPU.

También se puede observar que usar 4 threads para el caso de palabras ordenadas es lo más rápido para esta computadora específica. Esto es razonable pues la comparación de strings es una tarea muy intensiva en operaciones de memoria. Es posible que este punto óptimo varíe dependiendo de la configuración de caché y velocidad de memoria de cada computadora.



(a) Diccionario ordenado (b) Diccionario desordenado

Figura 3: Estadísticas del hit-rate en caché de contar palabras en un el mismo diccionario ordenado y desordenado. La cantidad de threads usados para ambos casos en la misma. 6 threads para la carga y 6 para el máximo.

Para intentar aclarar la segunda hipótesis a la que llegamos sobre el comportamiento de la función máximo para palabras ordenadas y desordenadas se utilizó *perf*, el profiler standard de Linux, para tomar estadísticas del hit-rate en todos los niveles de caché del programa. Se ejecutó el mismo programa en los dos casos, que consistía en cargar todas las palabras a la tabla de hash y luego encontrar el máximo. La única variable en los dos casos fue el órden del diccionario que se cargó del disco. Las imágenes son una captura de pantalla de *hotspot*, una interfaz gráfica para analizar los datos de *perf*.

Como se puede observar en ambos casos las funciónes que más cache-misses encontraron durante la ejecución del programa es el operador "==" y *memcmp*. Esto es razonable pues comparar strings es casi lo único que hace este programa.

También se puede observar que el hit-rate en el caso de un diccionario ordenado es mucho mayor que el desordenado. Esto confirma nuestra segunda hipótesis sobre la disparidad de la función máximo en los dos casos (mismas palabras, distinto orden) que la caché jugó un papel importante en los resultados de rendimiento.

4. Conclusiones

Como nota final al trabajo, podemos citar la importancia de analizar a la hora de desarrollar una estructura de datos o funciones concurrentes o paralelizables, el efecto que posee el mayor o menor número de threads utilizados y como esto interactúa con la estructuras internas propias. Vemos así que en proyectos de mayores dimensiones por ejemplo la biblioteca matemática `Numpy` de `Python` asigna como uso óptimo de computo concurrente usando una cantidad de threads igual al número de cores, otras programas permiten al usuario la elección de cores utilizados, de forma que el mismo configure la ejecución.

También es sumamente importante tener en cuenta el funcionamiento de la caché cuando se diseñan estructuras de datos. Como se pudo observar en estos test, resulta más ventajoso ordenar los datos de forma contigua en memoria que paralelizar excesivamente algunas operaciones. En la práctica es deseable conocer lo más que se pueda sobre el tipo de dato que se va a estar almacenando y el contexto de uso se la estructura. Así se puede llegar a un balance óptimo a la hora de paralelizar operaciones pudiendo aprovechar la caché de CPUs actuales.

A su vez, la experimentación y la búsqueda de respuestas a los resultados nos llevó a la utilización del profiler `perf`

de Linux en conjunto de la GUI hotspot que nos permitió recompilar datos de ejecución en particular, el hit-rate de la caché.

5. Referencias

[1] Downey Allen B. *The Little Book of Semaphores (2nd Edition)*. CreateSpace, v2.1.5 edition, 0.