DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

6 de septiembre de 2017

Sistemas operatornameerativos
Segundo Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar
Balbachan, Alexis	994/12	alexisbalbachan@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

El propósito de este trabajo práctico es experimentar con `pthreads`, tanto por los beneficios que trae (en particular en terminos en de performance) y los desafíos que presenta (al sincronizar los varios hilos de ejecución y mantener consistencia). En este TP se implementa un ConcurrentHashMap simplificado que alberga strings y contadores de las mismas.

Para completar el trabajo práctico debimos completar una Lista Atómica provista por la cátedra, y así como implementar desde cero el ConcurrentHashMap, utilizando la Lista previamente mencionada.

2. Lista Atómica

La lista se encontraba mayormente implementada, salvo por una operación: `push_front`. Esta operación es la única que modifica la lista en sí (si ignoramos la modificación de sus elementos), por lo que esta operación es la que hace a la lista "atómica".

Nos dimos cuenta que, como modificar un puntero no es una operación atómica, no bastaría con cargar el nuevo valor para la cabeza de la lista, o podríamos perder elementos. Nuestro primer enfoque hacía un exchange atómico con la nueva cabeza, y luego cargaba la cabeza anterior como siguiente elemento. Esta operación no era puramente atómica, y pudimos hayar casos donde se daba una *race condition* y, si bien los elementos eran perdidos permanentemente, si desaparecían durante un corto lapso, durante el cual la cabeza no conocía a los elementos que estaban anteriormente.

La operación que utilizamos en la versión final es un `atomic_compare_exchange_weak`, un compare-and-swap que garantiza que, al insertar el nuevo elemento, el mismo apunta al último nodo agregado exitosamente. Como el compare-and-swap no necesariamente es exitoso al principio (alguien puede haber cargado un nuevo valor desde la última vez que fue leído), este compare-and-swap está en un loop hasta que el elemento puede ser insertado sin interrupción.

Esto significa que la inserción no necesariamente coincide con el orden en que fueron llamadas, sino que depende de los cambios de contexto/thread.

3. ConcurrentHashMap

El ConcurrentHashMap implementado toma ciertas medidas para permitir que ciertas operaciones (pero no todas) puedan ser ejecutadas en paralelo:

- Un contador de operaciones, el cual tendrá dos utilidades en la clase: vitar que se ejecute `addAndInc` cuando se está ejecutando `maximum` y vice-versa; y saber cuántas de estas operaciones se están ejecutando. Si el valor de este contador es positivo quiere decir que se están ejecutando ese número de `addAndInc`, y si es negativo se está ejecutando `maximum`.
- Un mutex que proteje dicho contador y evita que dos operaciones conflictivas se inicien accidentalmente al mismo tiempo.
- Una variable de condición que genera un broadcast cuando el contador llega a 0, permitiendo que las tareas que no podían ejecutarse lo hagan ahora y evitando el busy waiting.
- Una lista de mutexes, uno por cada fila de la tabla del hashmap, para proteger dichas filas y evitar el agregado de elementos duplicados (ya que `addAndInc` puede ser concurrente con si mismo).

Por otro lado, la operación `member` es *wait-free*, e ignora todos los locks. Como usamos la Lista Atómica, estamos seguros que la ejecución coincide a alguna ejecución secuencial: esto quiere decir, si bien es posible que el método devuelva `false` luego de que algo fue agregado, no es posible, luego de un resultado `true`, obtener un resultado `false` (que podría haber sucedido si la lista no estuviese implementada atómicamente, como en el ejemplo descrito anteriormente).

Entre otras cosas, también consideramos al implementar el HashMap:

- Utilizar un mutex en lugar de una variable de condición, pero lo consideramos ineficiente y propenso a condiciones de carrera (mientras que este es el caso de uso exacto de una variable de condición).
- Utilizar `std::atomic_uint` en lugar de `unsigned int` para las entradas del HashMap, pero resulto más sencillo usar el tipo primitivo dada la firma del método a cumplir y que un mutex por fila seguría siendo necesario en ciertos contextos.

Como nota al margen, nos encontramos con una dificultades relacionadas con como C y C++ manejan la memoria y los constructores por copia/operadores de asignación:

- las clases `std::mutex`, `std::atomic` y otras primitivas de sincronización de la biblioteca STD de C++ eliminan explicitamente sus constructores y operadores de asignación por copia, eliminando a su vez los del ConcurrentHashMap y la Lista Atómica. Si bien estas clases contienen utilidades para evitar errores (como lock guards), optamos por utilizar los structs provistos en `pthread.h` (como `pthread_mutex_t`) para simplificar nuestra implementación.
- Al asignar un ConcurrentHashMap, C debe copiarlo a su destino. Sin embargo, como nuestra tabla es una lista de punteros a Listas Atómicas, y las mismas no cuentan con constructor por copia, utilizamos `std::shared_ptr` en lugar de punteros puros. De esta forma, cuando el HashMap es copiado y luego destruído, la tabla no es destruída, y las listas que fueron reemplazadas sí. Esto viene con la desventaja que la çopia"no es enteramente tal, y si se realiza una copia y se modifica, la tabla original también es modificada. Sin embargo, esto no debería afectar el caso de uso de este TP.

4. Uso de threads

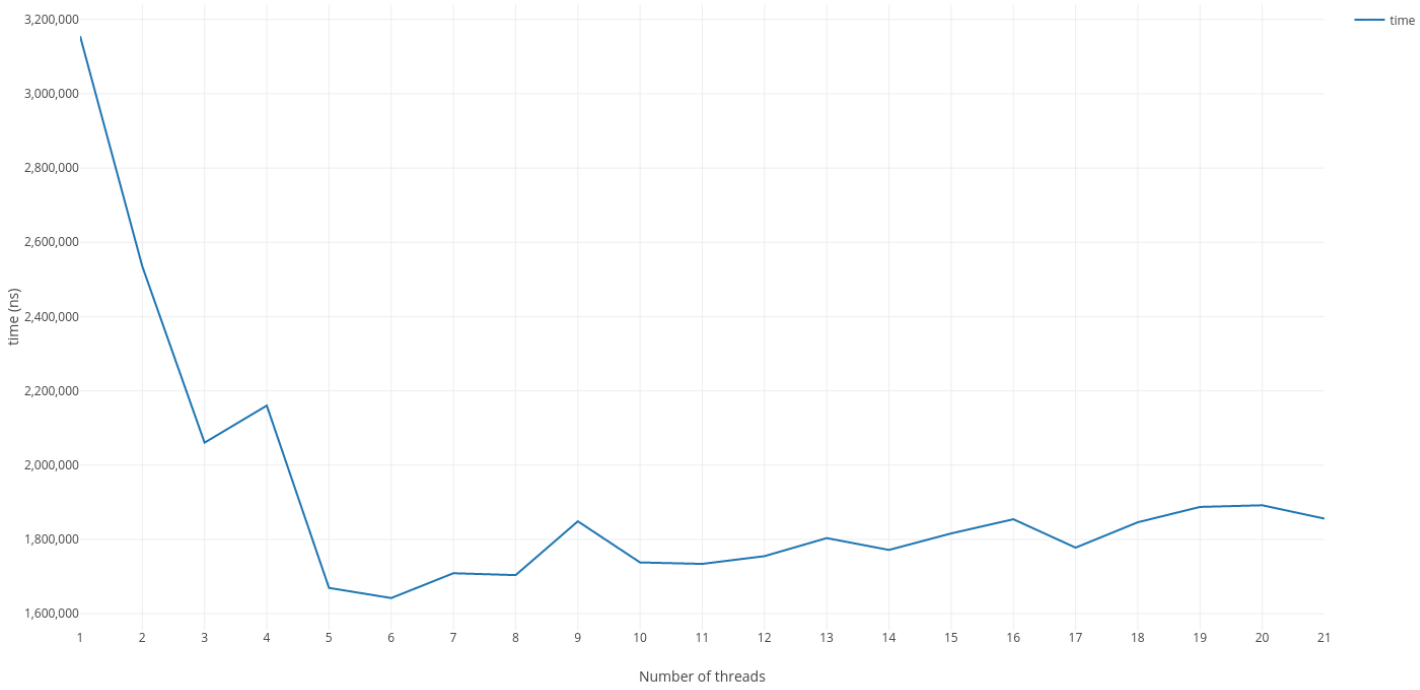
Para la implementación de las funciones concurrentes tuvimos que tener en cuenta los siguientes detalles:

- Para usar `pthreads`, consideramos conveniente definir structs relacionados a cada función a ser llamada desde los threads auxiliares. Estos structs contienen casi exclusivamente punteros/referencias.
- Para la mayoría de los casos, para evitar el cálculo duplicado de resultados utilizamos contadores atómicos. De esta forma, cada thread estaba informado de si había más trabajo para hacer, o si estaba finalizado.
- En el caso de `maximum`, como el mismo utiliza concurrencia interna y debe publicar un único resultado, también utilizamos una variable atómica para el máximo encontrado, con publicación de resultado via compare-and-swap (al igual que la Lista Atómica). Originalmente utilizamos un mutex para este mismo propósito, pero consideramos que un CAS era probablemente más eficiente que un lock.
- Originalmente habíamos implementado esto con `std::thread`, que tiene una implementación más sencilla (se pueden pasar parámetros arbitrarios en lugar de crear un struct). Sin embargo, entendemos que está fuera del scope de este TP, por lo que reimplementamos todo con `pthreads`.

5. Experimentación

5.1. Impacto del uso de threads

Con el objetivo de verificar si la utilización de múltiples threads mejora el rendimiento de nuestro proceeso, se decidió medir el tiempo de ejecución de la funcion `count_words` concurrente dejando fijo el número de archivos a cargar en 5 y variando el número de threads. Se realizaron 500 mediciones por thread y se calculó el promedio entre estas:



Como puede observarse, el tiempo de procesamiento se reduce drásticamente hasta que la cantidad de threads iguala a la cantidad de archivos siendo procesados, siendo 5 y 6 threads la cantidad que arroja valores mínimos de tiempo. La diferencia de tiempo entre estos dos casos puede ser explicada por la ejecución de otros procesos ajenos al experimento (pertenecientes al sistema operativo), los cuales pueden hacer variar las mediciones.

También puede apreciarse que, si bien los primeros 2 o 3 threads generan una gran diferencia, la mejora de performance es cada vez menor por thread adicional hasta llegar al mínimo, cuando el tiempo de ejecución aumenta en lugar de decrecer por cada thread agregado. Esto se puede ser causado por múltiples motivos:

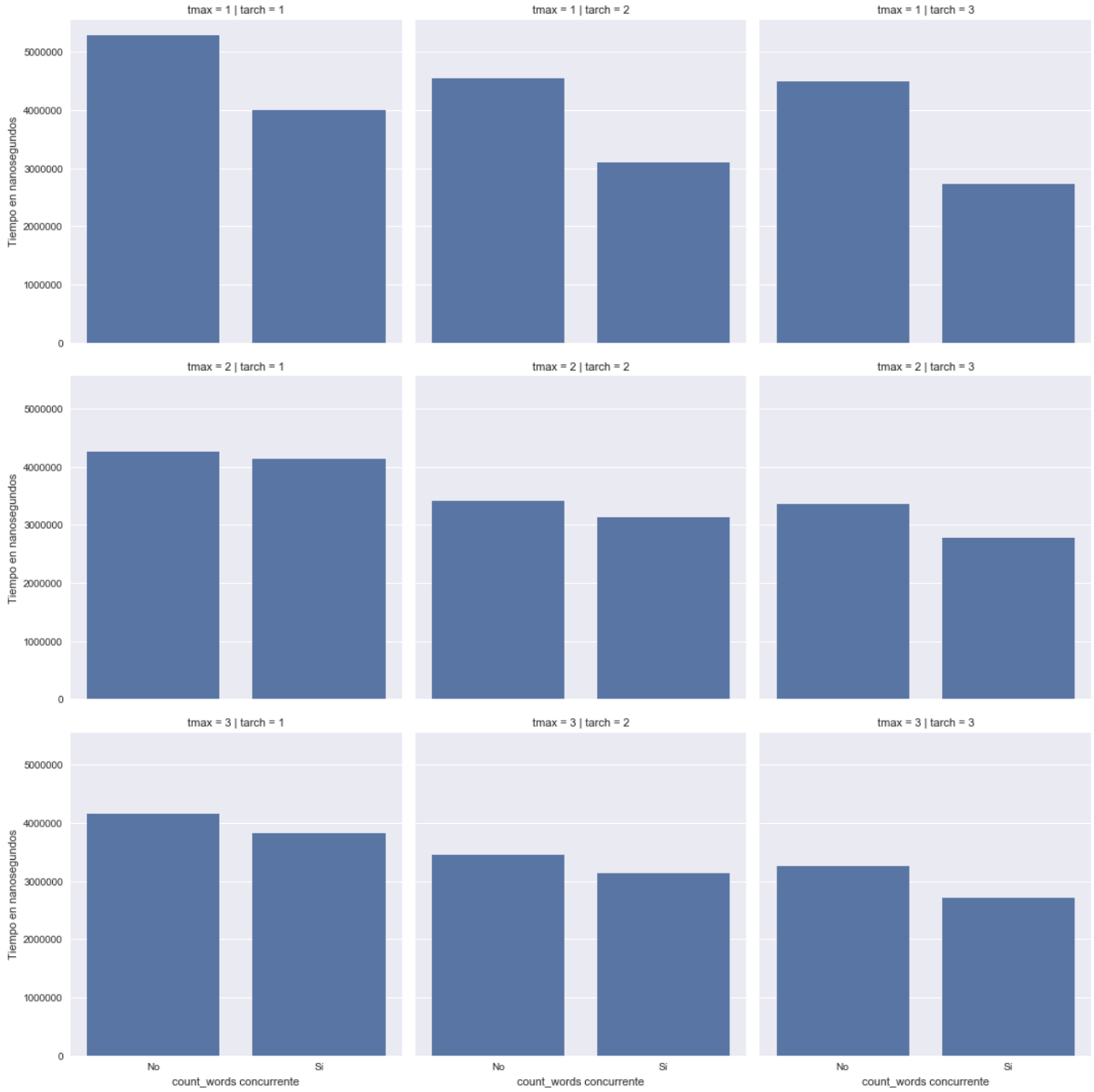
- Hay un overhead causado por switchear entre threads, al aumentar su número habrá más cambios, lo que puede reducir la eficiencia del proceso.
- A medida que aumentan los threads, aumentan las llamadas al sistema (syscalls) para su sincronización (mutexes), estas son operaciones costosas, por lo que muchas llamadas pueden enlentecer el proceso.
- Por último, en nuestra implementación particular cada thread toma un archivo, y dos threads no pueden procesar el mismo archivo a la vez (no podíamos garantizar que esto fuese verdaderamente atómico), por lo que los threads adicionales al 5to generan overhead sin traer beneficio alguno (simplemente finalizan de inmediato al verificar que no hay archivos disponibles).

5.2. Distintas implementaciones de `ConcurrentHashMap::maximum`

Hicimos unas pruebas del tiempo de ejecución de las dos versiones de

```
static pair<string, uint>ConcurrentHashMap::maximum
(uint p_archivos, uint p_maximos, list<string>archs);
```

correspondientes a los ejercicios 5 y 6 del trabajo práctico. La versión del ejercicio 5 no utiliza la versión concurrente de `ConcurrentHashMap::count_words`, y la versión del ejercicio 6 (llamada `maximum_6` para desambiguar) sí.



En este gráfico se pueden apreciar 2 cosas:

- como ya mencionamos, la mejora de performance al utilizar threads es muy notoria, pero la misma es cada vez menor a medida que se van agregando threads; y
- la función que utiliza la versión con concurrencia de `ConcurrentHashMap::count_words` puede hacer mejor uso de los threads que tiene disponibles, en particular durante la carga de los archivos, lo que era esperable ya que en la versión no concurrente se generan varios HashMaps y se deben copiar esos resultados a un único HashMap. También suponemos que, dado que todos los HashMaps almacenan sus datos en orden y tienen mutexes por fila de la tabla, es posible que haya muchas esperas durante la copia al HashMap unificado.

6. Conclusiones

Por lo que pudimos observar, el uso de threads puede incrementar enormemente la performance de las aplicaciones más sencillas. Sin embargo, el manejo de threads no es sencillo y debe realizarse con cuidado: por un lado, sin las estructuras de sincronización correctas, es muy facil generar inconsistencias, perder datos, etc.; por el otro, un uso excesivo de estas estructuras también puede tener un impacto pesado en el rendimiento, deshaciendo el beneficio obtenido al paralelizar; por úlitmo, el abuso de los mismos threads puede resultar en trabajo innecesario y perdida de ciclos de procesamiento.