DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Threading

Gestión de concurrencia

Sistemas Operativos
Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Agustin Frenkel	738/18	afus.frenkel@gmail.com
Sebastian Bocaccio	287/18	sebastianbocaccio16@gmail.com
Ignacio Alonso Rehor	195/18	arehor.ignacio@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Con los resultados del paper "Optimizing power using transformations" escrito en 1995, sabemos que para ejecutar alguna tarea, el uso de muchos cores sencillos pueden tener un consumo menor (con respecto a la cantidad de watts) a un super procesador. Utilizar muchos cores sencillos trae el desafío de sincronizar distintos *threads* o procesos.

La estructura a implementar se trata una tabla de *hash* abierta que gestiona las colisiones utilizando listas enlazadas. Su interfaz de uso es la de un *map* o diccionario, cuyas claves serán *strings* y sus valores, enteros no negativos. La idea es poder aplicar esta estructura para procesar archivos de texto contabilizando la cantidad de apariciones de palabras (las claves serán las palabras y los valores, su cantidad de apariciones).

Así mismo, analizaremos las ventajas y limitaciones de implementar una estructura con estas características.

2. Implementación de la estructura

2.1. ListaAtomica

La idea era implementar una lista enlazada, con la capacidad de insertar elementos de forma concurrente. En otras palabras que el método insertar sea **atómico**.

Esto nos lleva a la siguiente pregunta ¿Qué significa que la lista o un método sea **atómico**?

Que la lista sea **atómica** significa que sus operaciones pueden ejecutarse concurrentemente sin incurrir en condiciones de carrera (*race conditions*). Esto nos dice que, para cualquier *scheduler* (mínimamente razonable), el resultado de sus operaciones se corresponderá a una sucesión de llamados de funciones de distintos *threads*.

Mientras que si un método es **atómico** no tiene que tener condiciones de carrera con múltiples llamados al mismo método. Puede suceder que un método sea atómico consigo mismo pero tenga *race conditions* con distintos métodos de la misma estructura de datos.

Usar esta lista no garantiza que el programa esté libre de problemas de concurrencia por el simple hecho de que otras partes del programa pueden estar mal implementadas. Tomemos el ejemplo de la estructura `HashMapConcurrente`, la idea de este trabajo es justamente gestionar de manera inteligente esta lista y los métodos que la utilicen para evitar tener problemas de concurrencia.

Nuestra implementación del método `insertar` utiliza un *mutex* para asegurarse que solo un thread pueda acceder y modificar la cabeza de la lista en cada momento. De esta manera nos aseguramos que no haya problemas de concurrencia y que el método sea atómico.

2.2. HashMapConcurrente

La cátedra nos brindo una implementación parcial de la clase *HashMapConcurrente*. Para la cual se nos pidió que implementemos los siguientes métodos:

- `void incrementar(string clave)`: Dada una clave, si esta existe se debe incrementar su valor por 1. Caso contrario, se debe agregar la tupla <`clave, 1`> al final de la lista enlazada determinada por la función de *hash*.
- `vector<string>claves()`: Se debe devolver un vector con todas las claves dentro del diccionario.
- `unsigned int valor(string clave)`: Dada una clave, se debe devolver el valor asociada a esta.

Así mismo, había ciertas restricciones impuestas desde el punto de vista de la concurrencia:

- Además de soportar *multithreading*, el método `incrementar` debía proveer contención únicamente en el caso de que exista una colisión de *hash*.
- El método `claves` debe ser no bloqueante y libre de espera. Así mismo, debe permitir ejecutarse concurrentemente con cualquier operación de lectura.
- Al igual que el método `claves`, el método *valor* debe ser no bloqueante y libre de espera.

Podían producirse condiciones de carrera en el caso que, mientras se estaba ejecutando un método de lectura (ya sea `claves`), el *scheduler* diera prioridad a un llamado al método `incrementar` o viceversa. Esto se debe a que, en el caso de `claves`, el método `incrementar` podría crear una nueva clave en una posición ya recorrida por el método, provocando de esta manera un resultado inconsistente.

Cuando implementamos `valor`, nos confundimos y pensamos que había *race conditions* con `incrementar`. Pero de por si el método `valor` no tiene condiciones de carrera con `incrementar`. Decidimos no cambiamos el código y lo implementamos como si hubiera *race conditions* si un llamado a `valor` tendría que leer la misma `listaAtomica` que `incrementar`.

Para evitar que se den las *race conditions*, decidimos orientar nuestra implementación al problema clásico *readers & writer*, pero permitiendo más de un escritor en la sección crítica. Podemos hacer esto ya que, dentro del método `incrementar` volvemos a exigir una exclusión mutua con respecto a la fila a escribir.

La idea principal es tener un *mutex* (`room_empty`) que notifique qué rol (*reader* o *writer*) tiene acceso a la sección crítica. Este se asigna al primero que pida acceso a la sección crítica, y solo habilita el cambio de roles cuando el último proceso con ese rol abandona la sección crítica. En otras palabras, utilizamos un `Lightswitch`:

```
mutex_readers.wait();
    // Contabilizamos at micamente la cantidad de lectores en la %
    secci n cr tica.
    readers += 1;

    // Si se trata del primero de su rol, espera a tener control de
     la secci n cr tica.
    // Si no, pasa directamente a la secci n cr tica.
    if (readers == 1):
        room_empty.wait();
mutex_readers.signal();
```

A la hora de salir de la sección crítica, el último proceso del determinado rol es quien se encarga de ceder el control:

```
mutex_readers.wait();
    // Contabilizamos at micamente la cantidad de lectores en la
    secci n cr tica.
    readers -= 1;

    // Si se trata del  ltimo  de su rol, cede el control de la
    secci n cr tica.
    // Si no, sale de la secci n cr tica sin cambios.
    if (readers == 0):
        room_empty.signal();
mutex_readers.signal();
```

En la solución original del problema, solamente un proceso con rol de escritor podía acceder a la sección crítica. Implementado de esa manera, estaríamos perdiendo la posibilidad de paralelizar el método `incrementar`, ya que la escritura a distintos índices de la tabla no genera *race conditions*. Así mismo, nuestra implementación debía permitir que los métodos de lectura se puedan ejecutar concurrentemente, y tengan prioridad por sobre las escrituras. Para esto, utilizamos *turnstiles* o «molinetes» que, bajo un *scheduler* mínimamente razonable, eventualmente le dan la prioridad a los métodos de lectura bloqueando la llegada de nuevas escrituras. Evitando así la posible inanición causada por llegadas sucesivas (e incluso infinitas) de escrituras que negarían que se ceda el control del *mutex* (`room_empty`). No obstante, esto podría causar inanición para la escritura, ya que no libera las escrituras de su molinete hasta que cede el control del *mutex* (`room_empty`). Optamos por esta decisión ya que no teníamos ninguna restricción con respecto a los métodos de escritura (`incrementar`).

2.3. maximo()

Se nos pidió que implementemos el método `maximo`, este se encargaba de buscar la clave con el valor mas alto de dentro de nuestro `HashMap`. Para esto había que recorrer toda la tabla. Recorrer todas las listas atómicas, tiene la posibilidad de no considerar un nodo ya visitado que pueda convertirse en el máximo luego de una llamada al método `incrementar`. En definitiva, `maximo` es una operación de lectura, e `incrementar` una de escritura, el resultado de una mala sincronización entre estos es que arrojen resultados no consistentes.

Por ejemplo, supongamos una lista donde hay únicamente un elemento cuyo valor es 1. Como es el primer elemento que considera, este toma el rol de máximo en la lista. Antes de considerar al próximo elemento, se ejecutan y finalizan 4 llamados al método `incrementar`: Dos para el primero, llevando su valor a 3, y dos para un nuevo elemento, llevando su valor a 2. Notemos que en ningún momento este nuevo elemento fue el máximo de la lista, pero como no se vuelve a considerar el nodo ya visitado, el máximo pasa a ser este nuevo nodo. el método `máximo` en este caso devolvería un valor incorrecto: 2.

Como el método `maximo` es una operación de lectura, la lógica de sincronización es exactamente igual a la utilizada en el método `claves`.

2.3.1. maximoParalelo()

Para el método `maximoParalelo` había que instanciar múltiples threads, y que entre estos se encarguen de encontrar la palabra más ingresada. Nuestra idea fue que cada thread vaya revisando una lista atómica y que al terminar de revisarla empiece a revisar la siguiente sin terminar de revisar hasta que se hayan recorridas todas las listas.

Para lograrlo, decidimos pasarle un `struct` con 4 parámetros. El **primero** es un int atómico con el número de índice de lista que debe ser recorrido a continuación. El **segundo** es un puntero a donde está el puntero a la lista. El **tercero** es un `pair` donde se va a guardar el elemento máximo. Por último, un `mutex` que todos los threads comparten para que puedan modificar el elemento máximo de forma tal de no generar **race conditions**.

Gracias a esto, solo hace falta crear la cantidad de threads requerida, una sola vez y cada thread va a ejecutarse hasta que no le quedan *listasAtomicas* que revisar.

2.4. CargarArchivos

Por consigna, había que completar la clase `cargarArchivos` que se encargaba de insertar las palabras de un archivo dentro de una tabla de **hash**.

Para el método `cargarArchivo` no fue necesario tomar ningún recaudo desde el punto de vista de sincronización. Ya que, el mismo simplemente llamaba al método `incrementar` del `HashMap`, y este ya se encarga de eso.

Para el método `cargarMultiplesArchivos` había que instanciar múltiples threads, y que estos se encarguen de llamar al método `cargarArchivo`. Para Asia poder cargar archivos a la tabla de hash de manera concurrente.

Decidimos pasarle 2 parámetros fundamentales a cada thread para implementar este método. *El primero* es un puntero al vector que contendría los *PATH* de cada archivo. *El segundo*, un `atomic int`. El int atómico hace referencia al índice del último archivo que fue llamado a procesar. De esta manera, guardando este índice, todos los threads ciclaban hasta que este índice indicase que todos los archivos fueron procesados. Gracias a esto, solo hace falta crear la cantidad de threads requerida, una sola vez. Por otro lado, como los threads solo hacen lecturas sobre este vector, no es necesario regularlos con un **mutex**.

3. Experimentación

La idea de esta sección es evaluar qué ventajas ofrece, en términos de *performance*, la ejecución concurrente a la hora de encontrar la palabra con mayor cantidad de apariciones en un conjunto de archivos.

Con respecto a los archivos necesarios para realizar la experimentación, decidimos generar estos de manera automática. Estos archivos consisten en 5000 palabras generadas aleatoriamente a partir del conjunto `string.ascii_lowercase` de **Python**. Acordamos esta longitud de forma tal que, sin concurrencia o paralelismo, demoren aproximadamente 50 ms en ser cargados a la estructura.

Así mismo, decidimos ciertos criterios para enfocar la experimentación hacia resultados más interesantes:

- Para todo experimento centrado en encontrar la clave máxima del `HashMap`, limitaremos el número de *threads* a la cantidad de índices de la tabla del `HashMap`. Esto se debe a que, debido a como está implementado el método, estos *threads* sobrantes pueden únicamente perjudicar la performance del método y no merecen un análisis más profundo.

- Similarmente, al experimentar sobre la relación entre la cantidad de *threads* y el tiempo de ejecución del método `cargarMultiplesArchivos`, vamos a limitar estos últimos al número de archivos a cargar. Es decir, si en un experimento se plantea cargar 20 archivos, entonces la cantidad de *threads* utilizados para cargar los archivos será como máximo 20. La justificación para esto es análoga al punto anterior.

- Por último, toda medición se realizará un total de 50 veces y luego se considerará el promedio de ellas para asegurar una alto grado de fidelidad con respecto a nuestros datos.

Todos los experimentos fueron ejecutados sobre un procesador con la siguientes características:

CPU Specifications	Values
Architecture	x86_64
CPU(s)	6
Thread(s) per core	2
Core(s) per socket	3
Socket(s)	1
NUMA node(s)	1
Model name	AMD FX(tm)-6100 Six-Core Processor
CPU MHz	3229.762
CPU max MHz	3300
CPU min MHz	1400
BogoMIPS	6630.02
L1d cache	48 KiB
L1i cache	192 KiB
L2 cache	6 MiB
L3 cache	8 MiB

A continuación se presentan los experimentos planteados:

3.1. Búsqueda del elemento máximo

La idea era sacar algún tipo de conclusión acerca de las ventajas y limitaciones que tiene implementar una estructura de datos que permita concurrencia. ¿Es siempre cierto que a mayor cantidad de threads mayor la eficiencia? o Hay un punto en el que el *overhead* termina siendo un detrimento hacia la *performance* del proceso. Estas preguntas son sobre las que queríamos experimentar. Un detalle también, decidimos limitar la cantidad de threads a 26 ya que, dada nuestra implementación, es claro que no arrojaría ningún resultado útil ya que no hacen nada los threads sobrantes.

Luego, proponemos medir el tiempo que tarda en ejecutar el método `maximoParalelo` sobre una tabla con 105000 palabras cuyas primeras letras están uniformemente distribuidas, de manera tal que recorrer cualquiera de las listas atómicas demore aproximadamente lo mismo. Así mismo, proponemos registrar los cambios en estos tiempos producidos por variaciones en la cantidad de *threads* utilizados.

Esperamos que al aumentar la cantidad de *threads* utilizados, disminuya el tiempo requerido para encontrar el elemento máximo, hasta cierto punto. Suponemos que, a partir de este punto, el *overhead* requerido para gestionar todos estos *threads* será demasiado y en sí, los *threads* harán muy poco progreso, y terminarán afectando a la performance del método en general.

3.2. Cargar múltiples archivos al HashMap

La idea de este experimento será medir el efecto sobre el tiempo de ejecución del método `cargarMultiplesArchivos` que producen la ejecución concurrente. Al igual que el experimento anterior, nos interesa saber si es siempre beneficioso

incorporar más *threads* al proceso o si existe algún tipo de limitación. También, si depende de la cantidad de archivos que se pretenden cargar.

Proponemos medir el tiempo de ejecución del método primero con una cantidad fija de archivos (20), y luego variando tanto cantidad de *threads* como cantidad de archivos.

Muy similar al experimento anterior, esperamos que exista un punto de quiebre con respecto al tiempo de ejecución del método variando la cantidad de *threads* utilizados. En una primera instancia, esperamos que siempre sea beneficioso, pero cuando el *overhead* de lanzar *threads* sea más significativo, este afectará el rendimiento del método en general.

4. Resultados y discusión

4.1. Búsqueda del elemento máximo

En la Figura 1 se puede apreciar la diferencia de rendimiento para distintas cantidades de threads.

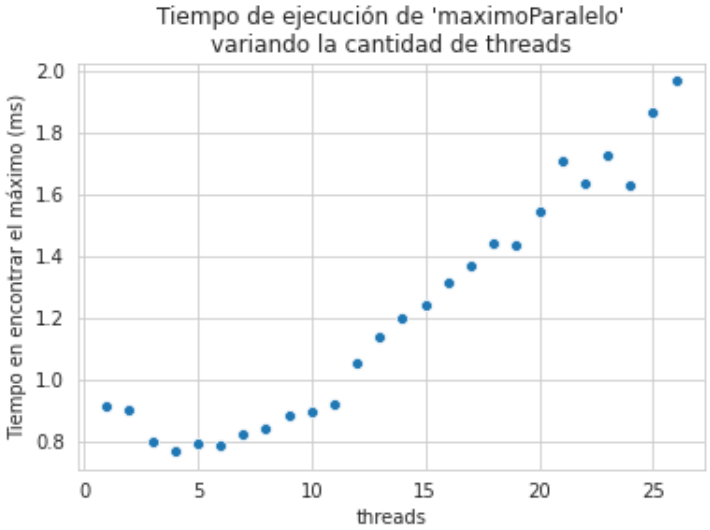

Figura 1: Tiempos de ejecución del método `maximoParalelo` para distinta cantidad de threads.

En el gráfico, la diferencia de rendimiento entre las distintas cantidades de threads es notable. Nuestras suposiciones de que aumentar la cantidad de *threads* utilizados disminuiría el tiempo requerido para encontrar el elemento, se corroboraron. Además, se puede observar como la performance se ve deteriorada a partir de un punto. Eso significa que existe el momento en el que el *overhead* de generar threads termina siendo más denso que el beneficio que otorga generarlos.

Así mismo, una posible explicación al hecho de que emplear más *threads* perjudique a tal nivel la *perfomance*, podría ser que la distribución de las primeras letras de las palabras sea demasiado uniforme. La consecuencia de esto es que todas las listas del `HashMap` tienen una cantidad similar de palabras entre sí, y por lo tanto, el tiempo en encontrar el elemento máximo en cada una de las listas es similar para todos los *threads*. Ahora bien, debido a la existencia de una sección crítica para chequear si un determinado *thread* encontró el elemento máximo del `HashMap`, resulta indudablemente en muchos *threads* siendo bloqueados al mismo tiempo.

4.2. Cargar múltiples archivos al HashMap

En la Figura 2 se puede apreciar la diferencia de rendimiento para distintas cantidades de threads para la carga de 20 archivos.

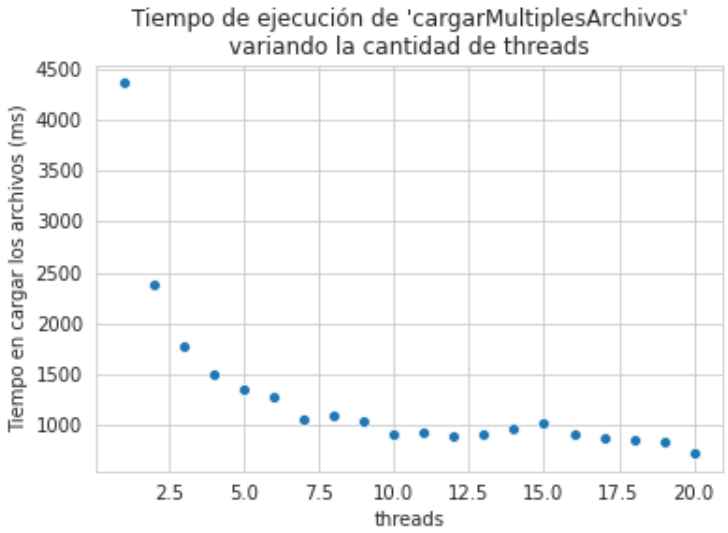

Figura 2: Tiempos de ejecución del método `cargarMultiplesArchivos`.

A diferencia del método `maximoParalelo`, podemos notar que no existe un punto de inflexión de manera que, a partir de un punto el *overhead* resultante de manejar los *threads* termine siendo perjudicial para la performance del método. Para el caso estudiado, parecería que en todo momento es preferible emplear la mayor cantidad de *threads* posibles. Ahora bien, la pregunta que surge es si sucede lo mismo para una cantidad variable de archivos.

Como se puede ver en la Figura 3, este resulta efectivamente el caso, es decir, aún variando la cantidad de archivos a cargar, siempre resulta la mejor opción maximizar la cantidad de *threads* empleados. Más aún, se puede observar que cuando la cantidad de *threads* es igual a la cantidad de archivos a cargar, el tiempo que se demora en cargar los archivos se mantiene razonablemente cerca

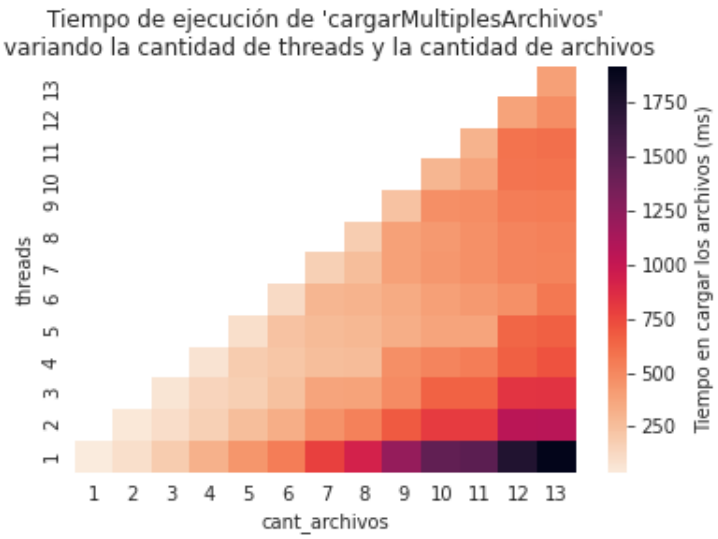

Figura 3: Tiempos de ejecución del método `cargarMultiplesArchivos` para distinta cantidad de threads y archivos.

con el tiempo que toma cargar un archivo utilizando un *thread*, tal como se puede ver en las Figuras 4 y 5. Es decir, incrementar la cantidad de *threads* empleados es realmente paralelizar.

5. Conclusión

Notamos como conclusiones:

- Las mejoras en rendimiento obtenidas al paralelizar son notables y deben ser un factor a tener en cuenta para resolver todo tipo de problemas. Sin embargo, también se concluye que no es óptimo aumentar la cantidad de threads indefinidamente ya que existe un punto en el cual esto se vuelve contraproducente.
- En cuanto a la cantidad de threads que se deben emplear, no podemos aseverarlo ya que esto depende de cada implementación y las características del hardware.

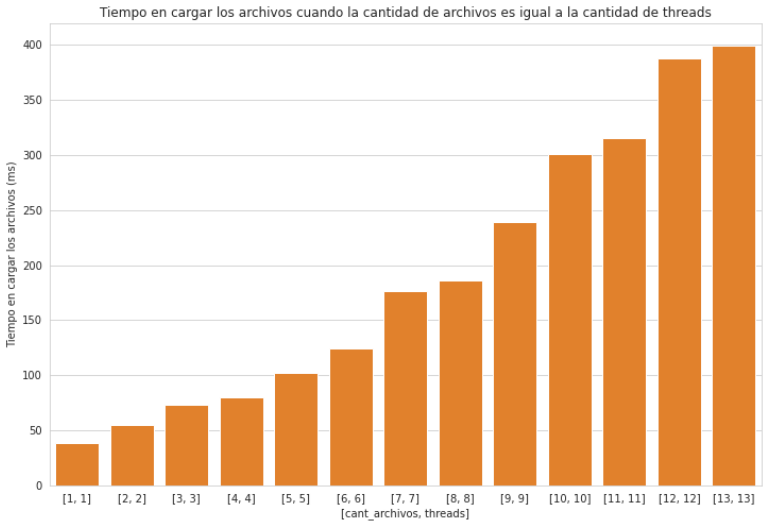

Figura 4: Tiempos de carga cuando la cantidad de archivos es igual a la cantidad de *threads*

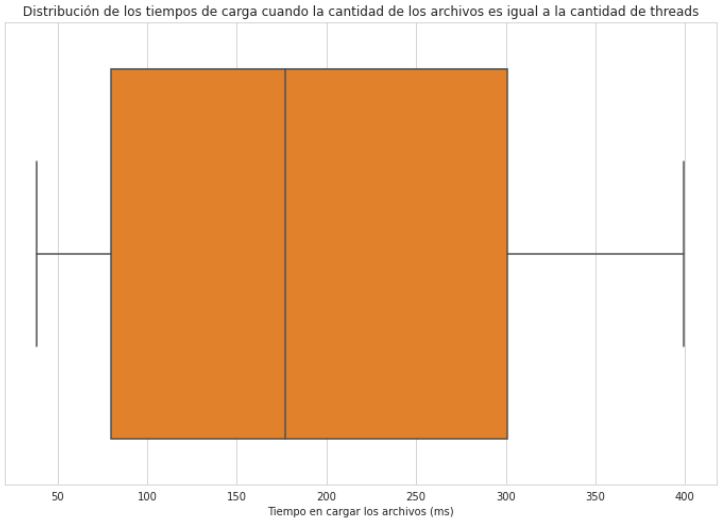

Figura 5: Distribución de los tiempos de carga cuando coinciden a cantidad de archivos y *threads*