DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1

Sistemas Operativos

Integrante	LU	Correo electrónico
Lucas Puterman	830/13	Lucasputerman@gmail.com
Ivan Vercinsky	141/15	ivan9074@gmail.com
Alonso Tomás	396/16	tomasalonso96@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

El motivo de este trabajo practico es implementar un diccionario que funciona sobre una tabla de hash. Esta implementacion sera de alta eficiencia y permitira accesos simultaneos manteniendo la consistencia de los datos, es decir, sera concurrente.

El diccionario solo toma strings como claves y se guarda el string con un valor entero que indica la cantidad de inserciones de dicho string, la funcion de hash utilizada consiste en simplemente tomar la primera letra del string a guardar. para las colisiones se utiliza una lista enlazada en cada letra del diccionarion(los resultados de la funcion de hash), donde las palabras nuevas se van agregando al final con su valor inicial de 1.

A continuacion se marcan las ideas iniciales y las decisiones de implementacion tomadas a lo largo del tp y finalmente se muestra una breve experimentacion para probar el funcionamiento correcto del codigo y mediciones de tiempo respecto a la funcion maximo la cual fue implementada de forma concurrente y tanbien de forma no concurrente.

2. Desarrollo

2.1. ListaAtomica

La $ListaAtomica$ es una implementación de lista usando variables atómicas. En particular, expone el método $push_front$que se encarga de lidiar con llamadas concurrentes, de esta forma, actualiza la cabeza de la lista que es atómica utilizando $compare\_exchange\_strong$.

A su vez, sus elementos se leen usando la función $load$ para así obtener el dato correcto.

2.2. ConcurrentHashMap

La estructura $ConcurrentHashMap$ cuenta con un arreglo de listas atómicas representando cada una de las letras del alfabeto. Además cuenta con un arreglo de mutex con un mutex por letra.

Se implementaron los siguientes constructores y funciones:

2.2.1. ConcurrentHashMap()

```
ConcurrentHashMap()
         -inicializar 26 listas atómicas.
         -inicializar los 26 mutex.
```

2.2.2. ConcurrentHashMap(std::string arch)

Crea el ConcurrentHashMap y carga en su estructura el archivo pasado de forma no concurrente.

```
ConcurrentHashMap(string arch)
         -Inicializar 26 listas atómicas.
         -Inicializar los 26 mutex.
         -process_file(arch)
```

2.2.3. ConcurrentHashMap(unsigned int nt, std::list<std::string>archs)

Crea un ConcurrentHashMap de forma concurrente corriendo nt threads en paralelo que procesan los archivos.

```
 ConcurrentHashMap(int nt, list<string> archs)
         -Inicializar 26 listas atómicas.
         -Inicializar los 26 mutex.
         -declarar atomico para el indice de las listas.
         -Lanzar nt threads.
         -Esperar la finalización de los threads.
```

Para esto se utiliza una variable atómica inicializada en -1 que va a representar el indice en la lista de archivos. Cada uno de los threads, pide un índice y mientras luego procesa el íesimo archivo

```
         Thread:
                 i = indice.get_and_inc
                 mientras i sea menor a la cantidad de archivos
                         process_file(archivos[i])
                         i = indice.get_and_inc
```

2.2.4. process_file(std::string arch)

Procesa las palabras de cada archivo y las agrega al ConcurrentHashMap llamando a la función add_and_inc.

2.2.5. add_and_inc(std::string key, int amount)

Esta función se fija si la palabra procesada ya se encuentra en la estructura y aumenta su valor. En caso de no estar creada, la crea. Antes de comenzar lockea el mutex de la lista.

```
-Lockea el mutex correspondiente a la letra de la palabra.
-Recorre la lista correspondiente para ver si existe la palabra.
-Si existe, incrementa su valor en amount.
-Si no existe la palabra, crea el par para la palabra y lo agrega a la lista.
-Libera el mutex correspondiente a la letra de la palabra.
```

2.2.6. ConcurrentHashMap::member(std::string key)

Devuelve un booleano indicando si la palabra indicada está presente en la estructura, recorriendo la lista correspondiente a la palabra.

2.2.7. ConcurrentHashMap::maximum(unsigned int nt)

Calcula cual es la palabra que aparece con más repeticiones en la estructura, corriendo concurrentemente en nt threads.

```
-Declara una ListaAtomica de pares y un atomico que va a ser el indice en los threads
-Lockea los mutex de todas las letras.
(para que nadie pueda agregar palabras mientras se calcula un máximo)
-Lanza nt threads, cada uno agregara sus máximos a la lista atómica.
-Espera la finalización de los nt threads.
-recorre la lista de máximos y devuelve el mayor.
```

Cada uno de los threads funciona de la siguiente manera:

```
Thread:
        i = indice.get_and_inc
        mientras i sea menor a la cantidad de letras
                bucar maximo en la lista de la letra correspondiente
                agregar maximo a la lista atómica de maximos
                i = indice.get_and_inc
```

2.2.8. ConcurrentHashMap::maximum(int p_archivos,int p_maximos, list<string>archs)

Dada una cantidad de archivos que se procesan en $p\_archivos$ threads, la función devuelve el máximo calculado en $p\_maximos$ threads. Esto es realizado creando $p\_archivos$ ConcurrentHashMaps y lanzando un thread por cada hashmap creado. Mediante un atómico que indica el indice dentro de la lista de archivos, los distintos hashmaps van procesando en su estructura los archivos. Una vez terminado, se mergean todos los ConcurrentHashMaps en un solo ConcurrentHashMap. Esto se logra lanzando una cantidad de threads (en nuestra implementación una por letra del abecedario), donde cada uno de los threads procesa la lista correspondiente a una de las letras en todos los hashmaps, indicada mediante un indice atómico compartido por todos los threads. Cuando se obtiene el ConcurrentHashMap con todos los archivos, se llama a la función maximum que devuelve el máximo.

```
-declara en vector con p_archivos hashmaps
-lanza los thread en los que se procesan los archivos
-espera la finalización de los threads.
-lanza los threads que mergean los p_archivos hashmaps en uno solo.
-espera la finalización de los threads.
-llama a la función maximum del ConcurrentHashMap resultante
```

Cada uno de los threads funciona de la siguiente manera:

```
ThreadProcesarHashmap:
        i = indice.get_and_inc
        mientras i sea menor a la cantidad de archivos
                hm->process_file(archivos(i))
                i = indice.get_and_inc
```

```
ThreadMergearHashmaps:
        i = indice.get_and_inc
        mientras i sea menor a la cantidad de letras
                por cada hashMap en HashMaps
                        por cada elemento de la lista i
                                hm.add_and_inc(elem.clave,elem.valor)
                i = indice.get_and_inc
```

2.2.9. ConcurrentHashMap::concurrent_maximum(int p_archivos, int p_maximos, list<string>archs)

Calcula el maximo de forma concurrente, crea un nuevo ConcurrentHashMap utilizando $p_archivos$ threads para leer los archivos y le calcula el máximo utilizando $p_maximos$ threads.

3. Resultados

En esta seccion se analizará con experimentacion el codigo, en especial el desempeño de las dos implementaciones de máximo, antes de evaluar se plantea como hipótesis que la implementacion concurrente de máximo va a ser mas rápida que la otra implementación por el hecho de que cada thread procesa una letra por separado, de está manera no genera contención en el add_and_inc. En el otro caso el ingreso de las palabras es aleatorio y no se puede asegurar que no se genere contención.

A continuación se pasa a experimentar para corroborar esta hipótesis.

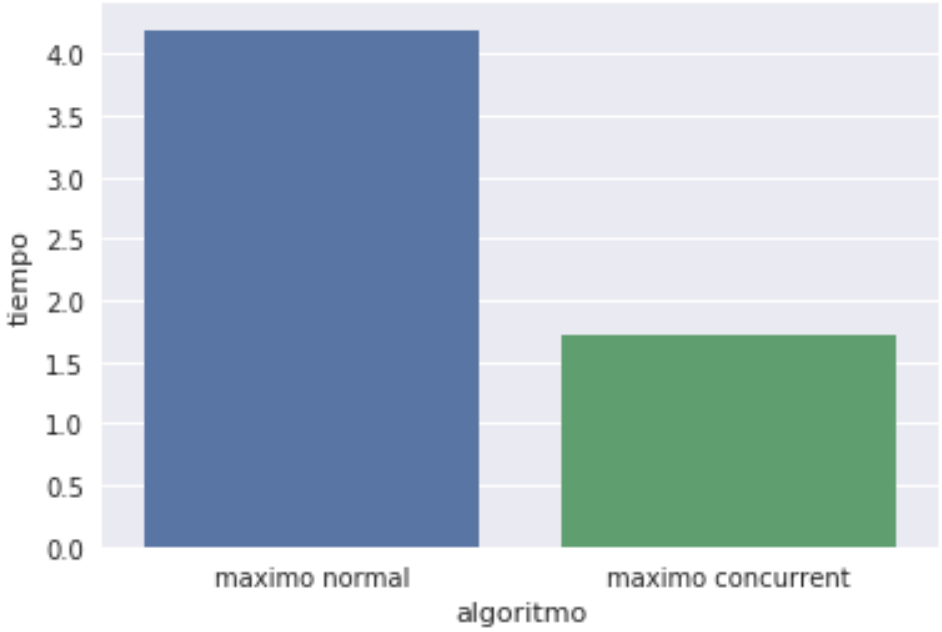
3.1. Experimentación Tiempos

Se realizó una experimentación que utiliza archivos generados aleatoriamente y compara los resultados de las dos funciones de máximos implementadas.

Las palabras se generaron utilizando el archivo $gerenador.py$ que se envía junto con el código.

Para la medición se corrieron 500 veces ambas funciones y se promediaron los tiempos.

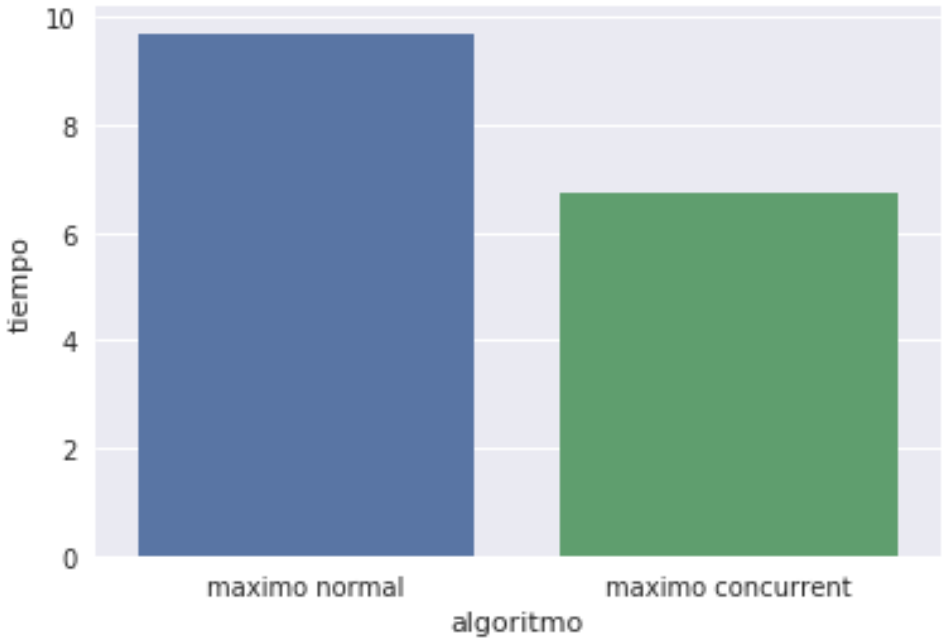
3.1.1. Tiempos de ejecucion



En este grafico se puede ver la comparación entre las dos implementaciones de maximo funcionando para 5 archivos con 5 threads. Se puede ver claramente que la implementación concurrente es más del doble de rapida.

	normal	concurrent
count	994.0	994.0
mean	4.196799688128771	1.71938472334003 94
std	2.810343303108625	1.5340264700982498
min	1.76498	0.866309
25%	2.5708175	1.08561
50%	3.1239	1.223315
75%	4.4532825	1.4992675
max	20.3669	14.2455

En esta tabla se pueden ver los valores estádisticos de la comparación. Se corrieron mil veces los algoritmos para este caso.



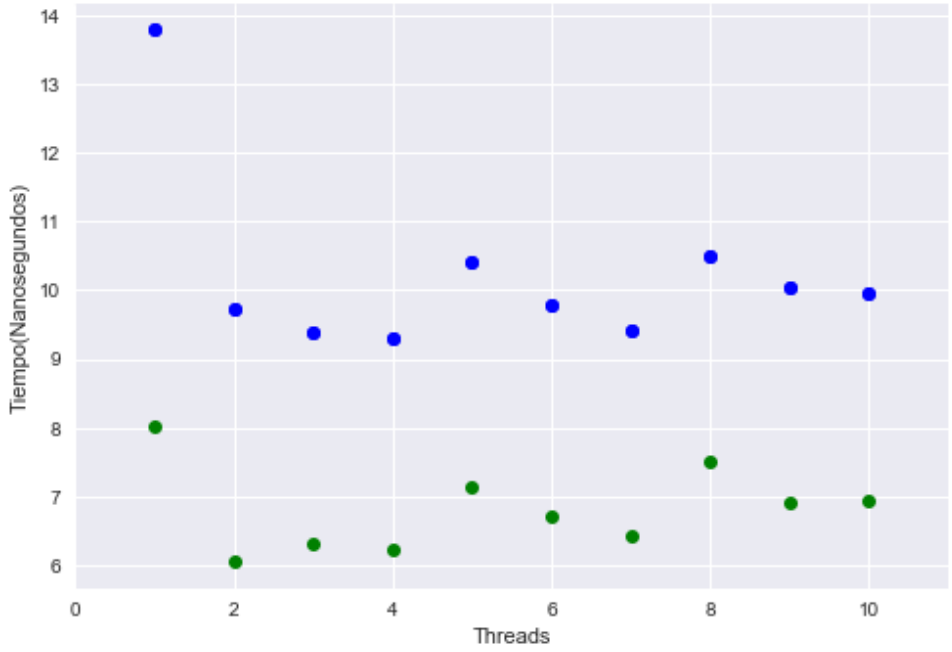
Aquí se puede ver la comparación de ambos para 11 archivos con 1 solo thread.
Este resultado a priori es extraño ya que ambas se ejecutan de forma no concurrente. Sin embargo, al tener en cuenta que la versión no concurrente crea un hashmap para luego pasarlo a otro hashmap y ahí calcular el máximo, tiene sentido pensar que pueda haber demorado mayor cantidad de tiempo que su equivalente concurrente.

	normal	concurrent
count	491.0	491.0
mean	9.699297087576372	6.7319692871690 46
std	4.632436986573542	3.4697132874875187
min	5.53769	4.58423
25%	7.19579	5.25685
50%	8.17784	5.73923
75%	9.847895000000001	6.5640350000000005
max	33.5729	27.7283

En esta tabla se pueden ver los valores estadísticos de la comparación, en este caso los algorimos se corrieron 500 veces.

3.2. Experimentación Threads

Además, se realizó una experimentación corriendo ambas instancias variando la cantidad de threads con $1 \leq n \geq 10$. Ambas implementaciones se corrieron 500 veces para cada n y se promediaron los tiempos arrojando los siguientes resultados:



Donde el verde representa a la versión concurrente y el azul a la no concurrente.

Como esperabamos, encontramos que la versión concurrente arroja tiempos menores que su contraparte no concurrente. Sin embargo, se puede observar que al aumentar los threads no mejora los tiempos significativamente, de hecho en algunos casos los empeora. Esto podría tratarse a que al agregar palabras en la estructura se bloquean con un mutex las listas, haciendo mayores los tiempos de espera al haber muchos threads.

Por otro lado, a simple vista parece haber una correlación entre los tiempos.

4. Conclusión

En este trabajo práctico se implementó la estructura ConcurrentHashMap de forma concurrente y no concurrente. Además de obtener una noción práctica de conceptos y estructuras de datos para concurrencia vistas en la materia como Atómicos y Mutex, se pudo observar experimentalmente como afecta la performance de un programa al ser ejecutado de forma concurrente.

Tal como suponíamos, las distintas funciones en su versión concurrente se desempeñaron mejor que sus equivalentes no concurrentes comprobando asi que nuestra hipótesis inicial era correcta.