DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Sistemas Operativos
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Balboa, Fernando	246/15	fbalboa95@gmail.com
Lopez Valiente, Patricio Nicolas	457/15	patricio454@gmail.com
Zdanovitch, Nikita	520/14	3hb.tch@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 siguiente Trabajo Práctico consistió en implementar la clase `ConcurrentHashMap` y algunas funciones relacionadas a la misma. Decimos que un hashMap es concurrente si soporta el uso de threads.

1.1. Threads

Un thread puede definirse como un conjunto de instrucciones independientes que pueden ser seleccionadas para correr por el scheduler del sistema operativo. En otras palabras, si una función, supongamos `main()`, tiene varios procedimientos que pueden ser corridos independientemente y/o a la vez por el sistema operativo, entonces se trata de un programa con varios threads ($multi - threaded$).

Para entender cómo funcionan los threads, es necesario conocer el funcionamiento de los procesos en el sistema operativo donde se implementan. En UNIX, los aspectos más importantes de un thread son:

- Existen dentro de un proceso padre y utilizan varios de sus recursos.
- Duplica únicamente los recursos necesarios para actuar de forma independiente (stack pointer, registros, prioridad, etc.).
- Tienen un flujo de control independiente del proceso padre, siempre y cuando el SO lo soporte.
- Es un proceso "liviano", dado que la mayor parte del overhead ocurre al crear el proceso padre.
- Puede compartir recursos con otros threads que corren tanto dependientes de él como independientes.
- En general, muere si su padre deja de existir.

Para la implementación particular de éste Trabajo, utilizamos el lenguaje `C++` y la libreria `pthreads`[1]. Vale aclarar que en estas condiciones, al crear los threads, se les asigna una función, que no puede ser método de una clase, y que devuelve y toma un puntero a void (que podría no usarse). Para pasar parámetros se suele entonces utilizar `structs` y castearlos a `void*`.

Como aclaración, diremos que en los pseudocódigos presentados, en la mayor parte de los casos, se ignoran o cambian las aridades reales de las funciones, las inicializaciones de mutex y threads, así como el manejo de memoria para el lenguaje en particular, para facilitar la comprensión del algoritmo. Ante cualquier duda, consultar el código fuente provisto.

1.2. ConcurrentHashMap

Esta clase no es más que un contenedor de pares <`string, unsigned int`>. Las claves, que son los `string`, son hasheados antes de ser almacenados. La función de hash utilizada es

[1] Para más información, consulte https://computing.llnl.gov/tutorials/pthreads/

simplemente tomar la primera letra de la palabra. Por otro lado, el entero del par representa cuántas veces se almacenó la clave que acompaña.

Para asegurar que la clase es concurrente, es necesario evitar las condiciones de carrera, es decir, que el código sólo funcione bien para ciertas ejecuciones del programa. En otras palabras, no puede depender del orden que el scheduler le asigna a los threads. Para lograrlo, es necesario entonces sincronizar los threads, utilizando estructuras que permitan la exclusión mutua de secciones críticas del código, como los mutex.

2. Ejercicio 1

2.1. Lista Atómica

Para que la lista sea atómica, debemos asegurarnos de que no se rompa cuando dos o más threads intentan agregar un elemento al mismo tiempo. Por ejemplo, supongamos que el thread 1 utiliza `push_front`. Primero crea un nuevo nodo con el valor indicado, y luego pone en el puntero al siguiente a la actual cabeza de la lista. Si ahora es desalojado por el scheduler y el thread 2 hace lo mismo, ambos nodos nuevos tendrán como siguiente a la vieja cabeza, lo cual no debería ocurrir en una lista enlazada.

Para solucionar este problema, utilizamos la función `std::atomic_compare_exchange_weak_explicit`[2] dentro de un `while`. Lo que vale la pena destacar es que ésta toma los parámetros `obj, expected` y `desired`. Primero compara los dos primeros, y si son iguales, copia `desired` a `obj`. Si no lo son, entonces copia `obj` a `expected`, todo ésto de forma atómica. Por último, devuelve `true` si `obj` era igual a `expected` y `false` en caso contrario. El pseudocódigo es entonces:

Algorithm 1 Implementación de `push_front` de Lista

`void` **push_front(val)**

1: newNode = Node(val)
2: newNode→next = head
   // Si nadie cambió la cabeza, entonces en newNode→next pongo la vieja cabeza. Si otro thread la cambió, head ya no es igual a newNode→next, por lo que la función devuelve `false`, entra en el ciclo porque está negado y guarda head (el nodo que insertó el otro thread) en newNode→next, y vuelve a probar en la siguiente iteración.
3: **while** ¬atomic_compare_exchange(head, newNode→next, newNode) **do**
   // No hago nada
4: **end while**

2.2. ConcurrentHashMap()

Las únicas variables que utiliza nuestra clase `ConcurrentHashMap` son dos contenedores `map` de la `STL`, $tabla$ y $mutexes$. La primera es una tabla de 26 posiciones, una por letra del abecedario. En cada una, hay un puntero a una lista atómica de pares <`string, unsigned int` >, donde se guardarán las claves y cuántas veces aparecen. El otro `map` también tiene 26

[2] http://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange

posiciones y guarda punteros a mutex. Éstos se utilizan para lockear una lista a la hora de utilizar la función `addAndIncN`. En conclusión, lo único que hace el constructor es reservar memoria para estas estructuras y almacenar las punteros correspondientes en los contenedores.

2.3. Función member()

Esta función sólo debe buscar la clave que recibe por parámetro en la lista que corresponda. El pseudocódigo es:

Algorithm 2 Implementación de la función `member` de `ConcurrentHashMap`

`bool` **member(**`string` **key)**

```
1: hash = primera_letra(key)
2: iterador = tabla[hash]→CrearIt()
3: Búsqueda lineal de key con el iterador en la tabla
4: if Encontró key then
5:     return true
6: else
7:     return false
8: end if
```

2.4. Función addAndInc()

La función addAndInc simplemente llama a addAndIncN con la clave que recibe y el entero 1. Esta última función busca la clave que recibe en el hashMap. Si la encuentra, suma n (parámetro) a su significado. Si no, crea el par$<$`clave, n` $>$ y lo agrega al hashMap. Nótese que es necesario lockear la lista que se modifica con un mutex para poder soportar que dos o más threads utilicen la función a la vez. El pseudocódigo es entonces:

Algorithm 3 Implementación de la función `addAndInc` de `ConcurrentHashMap`

`void` **addAndInc(**`string` **key)**

```
1: addAndIncN(key, 1)
```

Algorithm 4 Implementación de la función `addAndIncN` de `ConcurrentHashMap`

```
void addAndIncN(string key, unsigned int n)
 1: hash = primera_letra(key)
 2: mutexes[hash].lock()
        // Inicio sección crítica
 3: iterador = tabla[hash]→CrearIt()
 4: Búsqueda lineal de key con el iterador en la tabla
 5: if Encontró key then
 6:     significado(key) += n
 7: else
 8:     nuevoPar = par(key, n)
 9:     tabla[hash]→push_front(nuevoPar)
        // Fin sección crítica
10: end if
11: mutexes[hash].unlock()
```

2.5. Función maximum()

Esta función recibe como parámetro el entero positivo nt, que es la cantidad de threads a utilizar para buscar el máximo del `ConcurrentHashMap`, es decir, el par <`string, unsigned int`> con número más alto. Para lograrlo, es necesario ver todas las listas del hashMap y la idea es que cada thread recorra una o más filas enteras. En otras palabras, nunca debería ocurrir que dos threads distintos buscan el máximo en la misma lista.

Como se mencionó anteriormente, a los threads se los inicializa con una función, que es lo primero que ejecutan. Para pasarle parámetros a la misma, utilizamos `structs`. Para esta función, contamos con el siguiente:

```
struct iterador_tabla {
    mutex max_mutex
    par<string, unsigned int> max
    atomic<int> prox_lista
    map<int, Lista<par<string, unsigned int>>*>* tabla
}
```

El pseudocódigo de la función es:

Algorithm 5 Implementación de la función `maximum` de `ConcurrentHashMap`

`par<string, unsigned int >` **maximum(**`unsigned int` **nt)**

// Configuramos el struct

1: iterador_tabla iter
2: iter.max = par("", 0)
3: iter.prox_lista.store(0)
4: iter.tabla = &tabla
5: Crear vector de punteros a thread y reservarles memoria. El vector se llama threads y tiene tamaño nt.

// Inicializamos los threads como joinable para que el proceso padre pueda esperar a que terminen. Además les asignamos la función max_lista y le pasamos el struct iter como parámetro. Nótese que todos los threads reciben el puntero a la misma estructura, y no instancias distintas, por lo que cada thread ve los cambios que realizan el resto de los threads.

6: **for** i = 0 to nt **do**
7: crear_thread(threads[i], joinable, &max_lista, (void*) &iter)
8: **end for**

// El padre espera a que todos los threads terminen de buscar el máximo. En iter.max queda almacenado el máximo del `ConcurrentHashMap`.

9: **for** i = 0 to nt **do** // El ciclo no se ejecuta cuando i == nt.
10: join_thread(threads[i])
11: **end for**
12: return iter.max

Algorithm 6 Implementación de la función `max_lista`

```
void* max_lista(void* it_tabla)
      // Casteamos el parámetro el struct iterador_tabla
 1:   iterador_tabla* iter = (iterador_tabla*) it_tabla
      // Renombramos para facilitar la comprensión
 2:   tabla = *(iter→tabla)
      // Variables locales para almacenar el máximo encontrado hasta el momento y qué lista hay que recorrer
 3:   int actual
 4:   par<string, unsigned int > maxActual
      // El ciclo se rompe cuando ya no hay más listas para recorrer
 5:   while true do
      // Asigna la lista a recorrer. La función fetch_add(n) suma atómicamente n al valor guardado y devuelve lo que había antes de sumar.
 6:       actual = iter→prox_lista.fetch_add(1)
 7:       if actual ≥ 26 then
      // No hay más que recorrer, el thread termina su ejecución.
 8:           Exit(NULL)
 9:       else
      // dameMaximo busca linealmente con un iterador el máximo de la lista que se le pasa por parámetro.
10:           maxActual = dameMaximo(tabla[actual])
      // Lockeamos la variable iter→max para evitar condiciones de carrera.
11:           iter→max.lock()
12:           if maxActual.second >iter→max.second then
13:               iter→max = maxActual
14:           end if
15:           iter→max.unlock()
16:       end if
17:  end while
```

3. Ejercicio 2

La función `count_words` toma el nombre de un archivo y devuelve un `ConcurrentHashMap` cargado con las palabras del mismo. Para la mayoría de los ejercicios siguientes, utilizamos una función llamada `add_words`, que toma un `ConcurrentHashMap*` y un archivo y le carga las palabras al hashMap, de forma no concurrente y utilizando `addAndInc`. El pseudocódigo de `count_words` es entonces:

Algorithm 7 Implementación de la función `count_words(archivo)`

```
ConcurrentHashMap count_words(string arch)
 1:  ConcurrentHashMap h
 2:  add_words(&h, arch)
 3:  return h
```

4. Ejercicios 3 y 4

Dado que el ejercicio 3 está directamente relacionado con el 4, explicaremos primero este último. Esta versión de `count_words` toma una lista de archivos y un entero positivo n, que es la cantidad de threads a utilizar para crear el `ConcurrentHashMap` que tiene todas las palabras de la unión de dichos archivos.

Para este ejercicio, utilizamos el siguiente `struct`:

```
struct lista_archivos_hashmap{
        ConcurrentHashMap* hashMap
        list<string> archivos
        mutex arch_mutex
}
```

El pseudocódigo del ejercicio 4 es:

Algorithm 8 Implementación de la función `count_words(n, archivo)`

`ConcurrentHashMap` **count_words(**`unsigned int` **n,** `list<string>` **archs)**

```
    // hashMap a devolver
1:  ConcurrentHashMap h
    // Configuramos el struct
2:  lista_archivos_hashmap lah
3:  lah.hashMap = &h
4:  lah.archivos = archs
5:  Crear vector de punteros a thread y reservarles memoria. El vector se llama threads y tiene
    tamaño n.
6:  for i = 0 to n do
7:      crear_thread(threads[i], joinable, &load_archs, (void*) &lah)
8:  end for
    // El padre espera a que todos los threads terminen de cargar los archivos. En h queda
    almacenado el ConcurrentHashMap pedido.
9:  for i = 0 to n do
10:     join_thread(threads[i])
11: end for
12: return h
```

Algorithm 9 Implementación de la función `load_archs`

```
void* load_archs(void* _lah)
    // Casteamos el struct recibido por parámetro
 1: lista_archivos_hashmap* lah = (lista_archivos_hashmap*) _lah
    // Variable local que almacena el siguiente archivo que debe procesar el thread
 2: string arch
    // Cada thread debería tomar un archivo distinto al resto. Hay que asegurar que nunca
    ocurre que dos threads reciben el mismo. Para lograrlo, lockeamos con un mutex la asigna-
    ción de los archivos. A cada thread se le asigna el archivo del final de la lista, y luego se lo
    elimina de la misma, para que ningún otro thread reciba el mismo.
 3: while true do
    // Lockeamos antes de checkear si hay alguno más para cargar
 4:     lah→arch_mutex.lock()
 5:     if lah→archivos.empty() then
    // No hay más archivos para cargar, el thread finaliza su ejecución.
 6:         lah→arch_mutex.unlock()
 7:         exit(NULL)
 8:     else
    // Asignamos el archivo que sigue
 9:         arch = lah→archivos.back()
10:         lah→archivos.pop_back()
    // Para no perder concurrencia, desbloqueamos el mutex antes de cargar las palabras.
    Como ningún par de threads recibe el mismo archivo, no hay condiciones de carrera durante
    add_words
11:         lah→arch_mutex.unlock()
12:         add_words(lah→hashMap, arch)
13:     end if
14: end while
```

Por último, el ejercicio 3 debe utilizar un thread por archivo. La implementación presentada simplemente llama al ejercicio 4 utilizando n = archs.size().

5. Ejercicio 5

La función `maximum` de este ejercicio toma los siguientes parámetros:

- p_archivos = cantidad de threads a utilizar para leer los archivos.
- p_maximos = cantidad de threads a utilizar para calcular el máximo de cada archivo.
- archs = lista de archivos

Como se pide no usar las versiones concurrentes de count_words, la idea es crear un `ConcurrentHashMap` por archivo utilizando p_archivos threads, luego unirlos en uno sólo, y por último buscar el máximo en este último utilizando la función `maximum` del ejercicio 1 con p_maximos como parámetro.

Presentamos a continuación los `structs` y el pseudocódigo utilizado.

```
struct hashes_archivos{
        vector<ConcurrentHashMap*> hashes
        list<string> archivos
        mutex arch_mutex
        mutex hashes_mutex
}

struct merge_struct{
        vector<ConcurrentHashMap*> hashes
        ConcurrentHashMap* h
        atomic<int> prox_hash
}
```

Algorithm 10 Implementación de la función `maximum` no concurrente

`par<string, unsigned int>` **maximum(**`unsigned int` **p_archivos,** `unsigned int` **p_maximos,** `list<string>` **archs)**

// Configuramos el struct hashes_archivos

1: `lista_archivos_hashmap` ha
2: ha.archivos = archs
3: Crear vector de punteros a thread y reservarles memoria. El vector se llama threads y tiene tamaño p_archivos.
4: **for** i = 0 to p_archivos **do**
5: crear_thread(threads[i], joinable, &hash_archs, (void*) &ha)
6: **end for**

// El padre espera a que todos los threads terminen de cargar los archivos. En ha.hashes quedan almacenados los punteros a `ConcurrentHashMap` de cada archivo.

7: **for** i = 0 to p_archivos **do**
8: join_thread(threads[i])
9: **end for**

// Ahora es necesario unir los hashMaps. Creamos la variable que va a tener la unión.

10: `ConcurrentHashMap` mergeHash

// Configuramos el struct merge_struct

11: `merge_struct` ms
12: ms.h = &mergeHash
13: ms.hashes = ha.hashes
14: ms.hash.store(0)
15: Nuevamente reservamos memoria en el vector threads para otros p_archivos cantidad de threads.
16: **for** i = 0 to p_archivos **do**
17: crear_thread(threads[i], joinable, &merge_hashes, (void*) &ms)
18: **end for**

// El padre espera a que todos los threads terminen de cargar los archivos. En mergeHash queda almacenado la unión de todos los archivos.

19: **for** i = 0 to p_archivos **do**
20: join_thread(threads[i])
21: **end for**
22: return mergeHash.maximum(p_maximos)

Algorithm 11 Implementación de la función `hash_archs`

```
void* hash_archs(void* _ha)
    // Casteamos el struct recibido por parámetro
 1: hashes_archivos* ha = (hashes_archivos*) _ha
    // Variable local que almacena el siguiente archivo que debe procesar el thread
 2: string arch
    // Igual al ejercicio anterior, aseguramos que dos threads nunca reciben el mismo archivo.
 3: while true do
    // Lockeamos antes de checkear si hay alguno más para cargar
 4:     ha→arch_mutex.lock()
 5:     if ha→archivos.empty() then
    // No hay más archivos para cargar, el thread finaliza su ejecución.
 6:         ha→arch_mutex.unlock()
 7:         exit(NULL)
 8:     else
    // Asignamos el archivo que sigue
 9:         arch = ha→archivos.back()
10:         ha→archivos.pop_back()
11:         ha→arch_mutex.unlock()
    // Reservamos memoria para un nuevo ConcurrentHashMap y le cargamos las palabras del archivo
12:         ConcurrentHashMap* h = new ConcurrentHashMap()
13:         add_words(h, arch)
    // Guardamos el puntero en hashes. Aseguramos que no hay condiciones de carrera con un mutex
14:         ha→hashes_mutex.lock()
15:         ha→hashes.push_back(h)
16:         ha→hashes_mutex.unlock()
17:     end if
18: end while
```

Algorithm 12 Implementación de la función `merge_hashes`

`void*`**merge_hashes(**`void*` **_ms)**

// Casteamos el struct recibido por parámetro

1: `merge_struct*` ms = (`merge_struct*`) _ms

// Variable local que almacena el siguiente hashMap que debe procesar el thread

2: `int` actual

3: **while** true **do**

4: actual = ms→prox_hash.fetch_add(1)

5: **if** actual ≥ ms→hashes.size() **then**

// No hay más hashMaps para unir, el thread finaliza su ejecución.

6: exit(NULL)

7: **else**

// Asignamos el hashMap que sigue

8: `ConcurrentHashMap*` hashMap = ms→hashes[actual]

// Para mejorar la concurrencia, hacemos que cada thread recorra sus listas en orden distinto. Si recorrieran todos la lista de la misma letra, habría mucha contención en el `addAndIncN` del mergeHash. Llamamos orden a un vector que tiene los enteros de 0 a 25 en un orden random.

9: **for** i = 0 to 26 **do**

// Renombramos para mejorar comprensión

10: `Lista<par<string, unsigned int> >*` lista = hashMap→tabla[orden[i]]

11: Agregar todos los elementos de lista a ms→h utilizando addAndIncN(elem.first, elem.second).

12: **end for**

13: **end if**

14: **end while**

6. Ejercicio 6

Como en esta versión de maximum se permite usar la función count_words concurrente, nuestro algoritmo se reduce a crear un único `ConcurrentHashMap` utilizándola y devolver su máximo con el método `maximum`. Por lo explicado anteriormente, este procedimiento no tiene condiciones de carrera. El pseudocódigo:

Algorithm 13 Implementación de la función `maximum_concurrente`

`par<string, unsigned int>` **maximum_concurrente(**`unsigned int` **p_arch,** `unsigned int` **p_max** `list<string>` **archs)**

1: `ConcurrentHashMap` h = count_words(p_archs, archs)

2: `par<string, unsigned int>` max = h.maximum(p_max)

3: return max

6.1. Performance y Concurrencia

El tema central de este Trabajo Práctico fue la concurrencia con threads. Una de las ventajas deseables de procesar concurrentemente es reducir los tiempos de cómputo. Motivados por ésto,

desarrollamos el siguiente experimento:

Se usaron dos tipos de sets de pruebas:

- Los sets IG_γ: IG_{10}, IG_{100}, IG_{1000}. Cada archivo IG_γ está compuesto por la misma palabra α repetida γ veces. Dicha palabra es la misma para todos los archivos.
- Los sets RND_λ: RND_{10}, RND_{100}, RND_{1000}. Cada archivo RND_λ está compuesto por λ palabras aleatorias β_i, $i \in (1, \lambda)$ respectivamente. Además, para que las palabras en particular de cada archivo no influyan en la medición, por cada tamaño, se utilizaron diez archivos con palabras distintas. Dicho de otra forma, los archivos rndX_Y tienen tamaño X, con $X \in 10, 100, 1000$, y las palabras de rndX_Y son distintas a las de rndX_Z para todo Y $\neq$ Z.

Sobre estas entradas se utilizaron las funciones `maximum` en sus versiones concurrente(C) y no concurrente(NC), variando la concurrencia permitida para la carga de archivos de 1 a 10 threads. Cada una de estas mediciones fue repetida 50 veces y se tomo el promedio con el fin de evitar posibles outliers y reducir el error de la medición. Como una vez ya procesado el hashMap global las dos funciones calculan el máximo de igual manera, se dejó fija la variable p_max en 10.

Los resultados obtenidos fueron los siguientes:

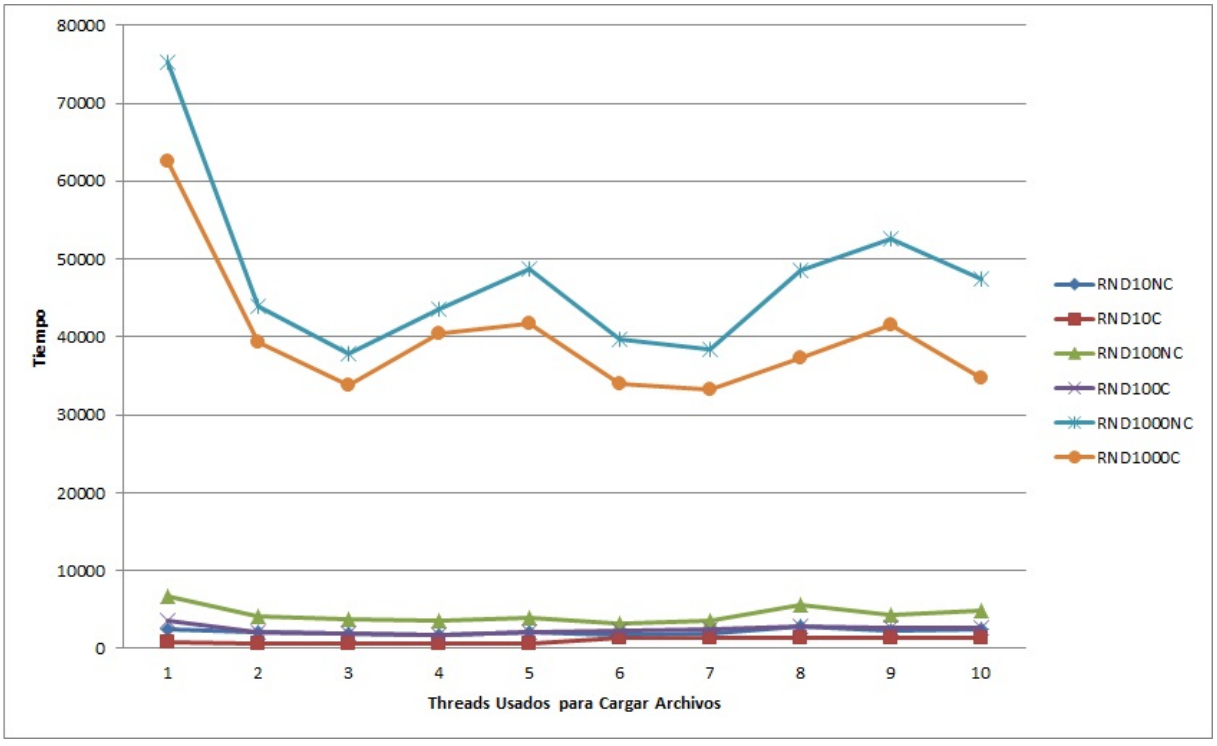

Figura 1: Complejidad Temporal del lote RND.

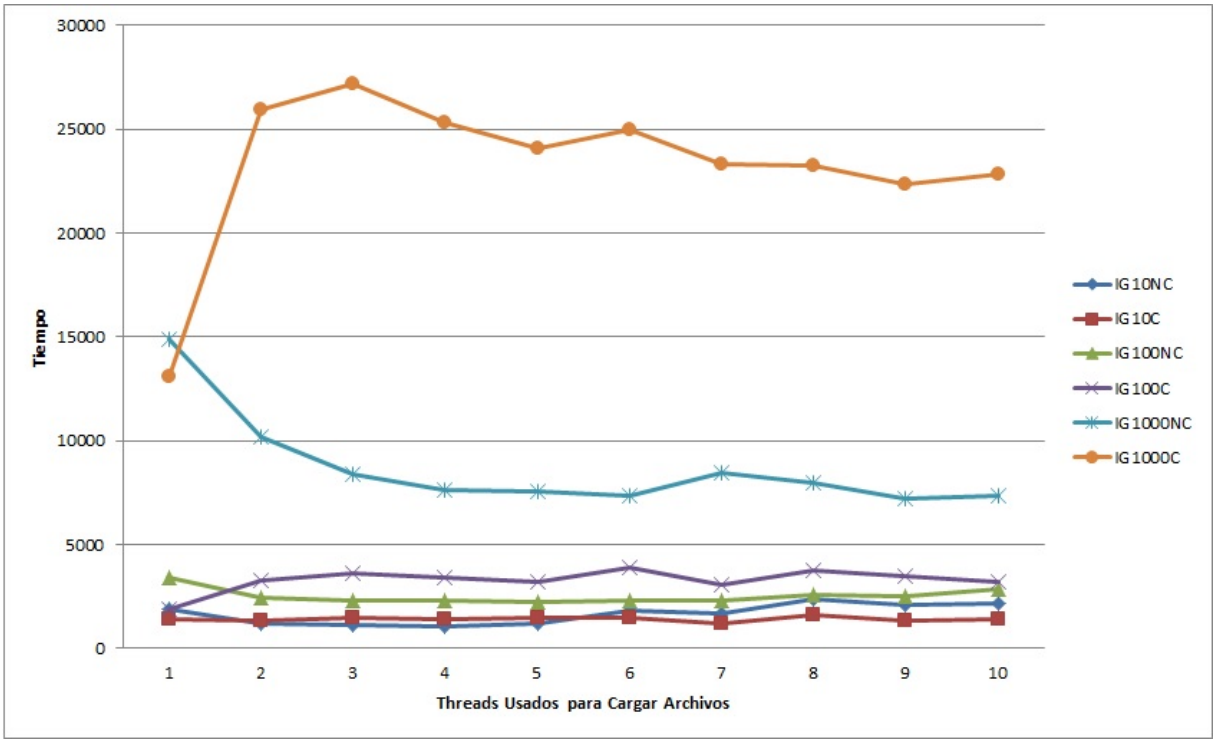

Figura 2: Complejidad Temporal del lote IG.

En la Figura 1 vemos como para $\lambda = 10, 100$, la versión concurrente es ampliamente superior. En cambio, para $\lambda = 1000$, a pesar de ser palabras aleatorias, la probabilidad de tener gran cantidad de colisiones es considerablemente más grande dado el tamaño del set, por lo que vemos que no hay tanta diferencia en los tiempos.

Este fenómeno se puede apreciar en mayor medida en la Figura 2. Para $\gamma = 10, 100$, la performance de los dos es similar, pero para $\gamma = 1000$ vemos como la versión C es ampliamente inferior a su análoga NC. Ésto se debe al gran numero de colisiones que genera agregar siempre la misma palabra. En la NC ocurre internamente en cada hashMap y luego en el merge. Dado que estos hashMap tienen la información concentrada, colisiona muchas menos veces. Como se puede apreciar, ésto tiene un gran impacto en la performance.

El experimento nos muestra que, a pesar de que en muchos casos procesar concurrentemente es beneficioso, depende en gran medida de la entrada al problema y de nuestra habilidad para resolver los problemas que se nos presentan de forma tal que se maximice la concurrencia, y a su vez se minimice el tiempo de bloqueo.

7. Tests

Para comprobar la correctitud de nuestros algoritmos, desarrollamos los siguientes test:

- `test_push_front`: Este test lanza diversos threads que modifican la misma lista y comprueban que el elemento se haya agregado. Una vez que los threads terminan, el proceso padre comprueba que la lista posea los elementos que cada thread agregó.
- `test_hashMap`: Este test lanza diversos threads, los cuales modifican el mismo `ConcurrentHashMap` utilizando, `addAndInc` para agregar palabras tanto nuevas como repetidas. Luego com-

prueba que éste posea las palabras que cada thread agregó, y que encuentra correctamente el máximo. Por último, agrega una misma palabra hasta modificar ese máximo y comprueba que efectivamente la función `maximum` devuelve el nuevo resultado correcto.

- `test_count_words`: Este test comprueba que para el mismo set de datos divido en $\alpha = 1, 3, 5$ partes, las distintas versiones de `count_words` construyan objetos observacionalmente iguales para distintos grados de concurrencia.
- `test_maximum`: Este test comprueba que para el mismo set de datos divido en $\beta = 1, 3, 5$ partes, tanto las versiones concurrente como no concurrente de `maximum` funcionen correctamente para distintos grados de concurrencia. Para asegurar su correcto funcionamiento, se las compara con el método `maximum`, que ya fue testeado.