Sistemas Operativos

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

Trabajo Práctico I

pthreads

Integrante	LU	Correo electrónico
Alvarez Mon, Alicia	224/15	aliciaysuerte@gmail.com
Ansaldi, Nicolas	128/14	nansaldi611@gmail.com
Suárez, Romina	182/14	romi_de_munro@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

En este trabajo se nos pide implementar una tabla de hash donde se vuelque el contenido de un texto a procesar. Las palabras irán llenando la tabla y se contará las repeticiones de las mismas. Para ello utilizaremos threads.

El objetivo de este trabajo es sincronizar los threads de modo que cada uno funcione sin pisar lo hecho por otro thread y tomando el trabajo de todos los que vinieron antes del actual. Esto es, que no se generen race conditions.

2. Desarrollo

2.1. Push front

En primer instancia se nos pide completar la función push front, del tipo lista. Esta debe poder trabajar con concurrencia, pues sino, el nodo a agregar podría no agregarse adelante de la lista, habiendo una race condition.

El problema, más detallado, consiste en: Al agregar un nodo, uno tiene que actualizar el puntero al frente de lista (_head) al nuevo nodo. En el caso de que se este corriendo concurrentemente, puede ser que dos nodos se quieran agregar a la vez; si el primero es desalojado antes que _head lo asigne como primero, el segundo no apuntara a él, y el nodo quedara inaccesible.

Algorithm 1 Void *push_front*(archivo)

```
1: atomic Nodo* nuevo
2: while (true) do
3:     aux = _head
4:     if _head.CompareAndChange(aux, nuevo) then
5:         Si paso agrego el nodo a la lista y salgo del ciclo
6:     end if
7: end while
```

Para evitar esto, utilizamos las funciones ”compare exchange strong ", store y load, todas de la librería atomic. Compare exchange se fija si el valor que creemos que esta en la cabeza de la lista sigue siendo el mismo. Si es así, entonces se cambia por el nuevo nodo a agregar. Sino, el while cicla hasta que esa condición se cumpla y una vez que eso ocurra, sale con el break. De esta manera, logramos que la lista agregue correctamente en ambientes concurrentes.

2.2. Add and inc

La función add_and_Inc añade elementos al ConcurrentHashMap. Esta función debe ser excluyente cuando agrego en una fila, pues podrían darse race conditions entre varios threads que quieran agregar la misma palabra, pero, a su vez, queremos que dos threads que quieran agregar palabras en distintas lineas puedan hacerlo, para no eliminar el paralelismo y permitir más de una thread agregando a la vez. Además debe ser mutuamente excluyente con maximum, es decir que, si una de las 2 está trabajando en el hash la otra no puede estar trabajando en el hash.

Para salvar esto, creamos una lista de semáforos con 26 posiciones, donde cada thread que este procesando una fila, entre y loquee esa sección critica y así evite que otro thread modifique el valor de esa palabra a agregar, antes o después de que él la modifique. Al mismo tiempo, esta separación permite que un thread que va a modificar otra fila, pueda hacerlo sin ser bloqueado. Para resolver el problema con maximum usamos el mismo semáforo, ya que si al menos una de las posiciones está bloqueada maximum no entra a trabajar en el hash.

Algorithm 2 void ConcurrentHashMap:: add And Inc(string key)

```
1: int pos = hash_func(key)                    ▷ hash_func hace la funcion de hash pedida por la cátedra
2:  posicion[pos].lock()
3: Iterador de lista it en la posicion pos del hash
4: bool esta = false
5: while (it.HaySiguiente() and !esta ) do
6:     if (it.Siguiente().first ==key)  then
7:         esta = true, aumentá el valor y salí de la ejecución
8:     else
9:         Seguí buscando
10:    end if
11: end while
12: if esta == false  then
13:    (*tabla[pos]).push_front(pair< string, int >(key, 1))          ▷ sino está, agregala
14: end if
15: posicion[pos].unlock()
```

Para testear las funciones anteriores nos basamos en los tests de la cátedra, además de hacer un test pequeño para probar el funcionamiento de las mismas.

2.3. Member

Esta función busca una palabra en el ConcurrentHashMap y nos dice si esa presente o no. Para programarlo, calculamos la entrada donde estaría la clave en el caso de ser agregada, y creando un iterador a la lista de la entrada correspondiente, iteramos hasta encontrarla o llegar al final de la lista. Devolvemos si la hallamos o no.

Algorithm 3 bool member(string key)

```
1: int pos = hash_func(key)
2: iterador de lista it en la posicion pos del hash
3: while it.HaySiguiente() do
4:     if it.Siguiente().first == key then
5:         return true
6:     end if
7:     it.Avanzar()
8: end while
9: return false
```

Nos piden que sea wait-free. Osea que en todas las posibles ejecuciones de member, siempre que se llame, sea ejecutada, y termine (con el resultado correcto).

Sabemos que ninguna otra función la traba, y puede llamarse a member durante la ejecución de otra función sin problemas. Además, no se bloquea en ningún momento. En caso de race condition, member podría no devolver la existencia de una palabra que se acaba de agregar al hash.

Sin embargo, este caso sólo podría ocurrir si el elemento buscado es agregado con add_and_inc en el medio de la ejecución de member. Si es agregado a la lista antes que el iterador busque esa posición, member lo va a encontrar, así que sólo puede haberse agregado después de que el iterador comience a iterar (como se agrega adelante de la lista, el elemento va a quedar agregado al inicio).
Pero este comportamiento no resulta un problema, porque las race conditions se generan cuando dos programas que se ejecutan a la vez llevan a un resultado que no podría lograrse en ninguna secuenciación de las funciones. En este caso, se puede ordenar a member habiendo ocurrido antes que add_and_inc, por lo cual los resultados son siempre consistentes.

Falta ver que esta función es wait-free. Para esto vemos que la función no es bloqueada por ninguna otra función y además no accede a ninguna zona crítica del hash, ya que no agregua ni modifica la información del mismo. Luego para cualquier instancia de la función sabemos que termina ya que es finita y depende de si misma, entonces podemos decir que member es wait-free.

2.4. Maximum

La función Maximum nos devuelve la tupla de la palabra que más veces fue agregada al hash, y esa cantidad de veces. El parámetro indica la cantidad de threads con la cuál se busca este elemento máximo.
Creamos los threads y las enviamos a la función maxola, que recibe un puntero al hash de la clase y un contador atómico que nos indica cuál fue la última lista revisada, para que revise la siguiente. Este último es atómico porque debemos accederlo y aumentarlo al mismo momento de manera atómica para evitar race conditions entre los threads.Pues podría ocurrir que dos threads sumen el contador a la vez, o que se aumente y pase de rango luego de acceder al ciclo.

Actualizamos un máximo global si encontramos una palabra que aparece más veces, y así cada thread va actualizándolo desde su lista. Actualizar el máximo es una sección crítica, pero que no puede resolverse con poner un máximo atómico porque depende también de la comparación con el elemento de la lista, y no podemos hacer dos comparaciones a la vez, por lo cuál utilizamos un mutex que protege esa sección, antes de la comparación entre el elemento actual y levanta la señal, luego de este(actualiza el máximo si entra al if).

Además, la función maximum debe ser mutuamente excluyente con add and inc. Para lograr esto, nos aprovechamos de la lista de semáforos del add and inc, y cuando entramos a la función lockeamos las 26 posiciones. De esta manera, los *add_and_inc* no pueden ejecutarse mientras el maximum se ejecuta, y viceversa (si un maximum quiere ejecutarse en el medio del *add_and_inc*, no va a poder por el lockeo en la entrada en la cual este está agregando). Este lockeo también impide que dos maximum se ejecuten concurrentemente.

Algorithm 4 pair< $string, int$ > maximum(int threads)

```
1: for int i = 0; i < 26; i++ do                          ▷ Bloqueamos el semáforo
2:     posicion[i].lock()
3: end for
4:  si la cantidad de nt es mayor a 26, tomar nt = 26
5: Hashcontador aux
6: aux.h = hashActual
7: aux. _ultima = 0
8: pthread_t thread[realint]
9: int tid
10: for mientras tid no sea realint do
11:     Creamos un thread usando la función maxola y pasándole la estructura aux
12: end for
13: Liberamos los threads
14: for int i = 0; i < 26; i++  do                        ▷ Desbloqueamos el semáforo
15:     posicion[i].unlock()
16: end for
17:  return _maximo                                       ▷ Variable global
```

Algorithm 5 void * maxola(void* aux)

```
1: Hashcontador* caux = (Hashcontador*)aux
2: int x
3: while ((x = (caux− > _ultima).fetch _add(1)) < 26) do
4:     Creo un iterador de lista en el bucket x
5:     while it.HaySiguiente()  do
6:         m2.lock()
7:          me quedo con el maximo del actual y el maximo global
8:         m2.unlock()
9:         it.Avanzar()
10:    end while
11: end while
```

Para esta función hicimos test básicos para probar su funcionamiento y un test para ver la exclusión mutua entre maximum y addAndInc. El mismo consiste en crear threads y mandarlos a ambas funciones y ver cuando

entran y salen de los semáforos. Lo que pudimos ver fue que si un maximum pasaba el semáforo no lo pasaban ningún addinc ni otro maximum por más que hubieran llegado a la función. También pudimos ver que los addinc solo se bloqueaban entre sí, si llegaban a la misma lista pero no lo hacían si iban a parar a otra lista, mientras que bloqueaban a cualquier maximum que intentara entrar.

2.5. Count words(string arch)

La función count words básica toma un archivo y lo procesa, de forma no concurrente(osea que no hay más de un thread corriendo su código al mismo tiempo), y nos devuelve un ConcurrentHashMap con todas las palabras del archivo agregadas.
Para programarlo, creamos una estructura hash, abrimos el archivo a procesar y recorremos el archivo, leyendo sus palabras y llamando *add_and_inc* con cada una, hasta llegar al final del archivo. En la lectura nos aseguramos de no leer un posible ultimo espacio dentro del file, pues si existiese, el operador >> le daría a nuestro algoritmo una palabra de mas a procesar. Como no es concurrente, no hay threads, y no hay zonas críticas o race conditions que tengamos que proteger.

Algorithm 6 ConcurrentHashMap count_words(string arch)

```
1:  ConcurrentHashMap h
2:  ifstream input
3:  input.open(archivo)
4:  string alo
5:  while ! input.eof() do
6:      input >> alo
7:      input.ignore()
8:      if input.peek() == 10 then        ▷ Esto es por si el archivo termina con un salto de linea (new line)
9:          break
10:     end if
11:     h.addAndInc(alo)
12:     return h
13: end while
```

2.6. Count words(list<string>archs)

La función count_words2, (forma en la cual diferenciamos este countwords de la función del punto anterior) toma una lista de archivos y es nuestra obligación procesarla con un thread por cada archivo.

La función no tiene sección crítica en sí misma, las posibles race conditions ocurren cuando se llama a la función add_and_inc, pero el correcto funcionamiento de la función llamada en presencia de concurrencia es manejada en su propio código.

Como debemos crear un thread y llamar a una función auxiliar por cada archivo dentro de la lista parámetro, creamos los threads pasándoles la función count_wordsaux de parámetro, y el struct hash_escritor como argumentos de esta. El hash escritor esta conformado por un puntero al hash resultado que creamos al principio de count_words2 y que luego de llenar, devolvemos, y un string que contiene el nombre del archivo a procesar en ese thread.

La función conut_wordsaux tiene el mismo funcionamiento que el count_words del ejercicio 2, pero tomando sus archivo de lectura y hash donde agrega las palabras leídas del hash_escritor.

Algorithm 7 ConcurrentHashMap count_words2(lista de archivos arch)

```
1: ConcurrentHashMap escri
2: int cantdearchivos = archs.size()
3: Hashescritor* separador[cantarchivos]
4: creamos un iterador it a la lista de archivos pasada por parámetros
5: pthread_t thread[cantdearchivos]
6: for tid= 0; tid < cantdearchivos, tid++ do
7:     separador[tid]->h = &escri
8:     separador[tid]->arch = *it
9:     createthreat(thread[tid], NULL, count_wordsaux, separador[tid])
10:    it++
11: end for
12: for tid= 0; tid < cantdearchivos, tid++ do
13:    jointhreat(thread[tid], NULL)
14: end for
15: return escri
```

Algorithm 8 void* count_wordsaux(void* separador)

```
1: Hashescritor* caux = separador
2: const char* archivo = (caux)->arch.c_str()
3: ifstream input
4: input.open(archivo)
5: string alo
6: while ! input.eof() do
7:     input >> alo
8:     input.ignore()
9:     if input.peek() == 10 then          ▷ Esto es por si el archivo termina con un salto de linea (new line)
10:        break
11:    end if
12:    h.addAndInc(alo)
13:    return h
14: end while
```

2.7. Count words(unsigned int n,list<string>archs)

La función count_words3 (como diferenciamos a esta función), toma n cantidad de threads y la lista de archivos a procesar.
Nos fijamos que n sea menor que la cantidad de archivos, pues tener más threads que archivos a procesar es innecesario. Luego creamos las estructuras necesarias y llamamos a la funcion count_words_limthreads_aux.

Algorithm 9 ConcurrentHashMap count_words3(unsigned int n,lista de archivos arch)

```
1: int realnt = min(n, archs.size())
2: ConcurrentHashMap escri
3: vector<string> archivador
4: creamos el iterador iteraux a la lista de archivos arch
5: while iteaux ! = archs.end() do
6:     archivador.pushback(*iteaux)
7:     iteaux++
8: end while
9: HashescritorConc * separador
10: separador− >h = &escri
11: separador− >vecti = &archivador
12: separador− >ult_pos.store(0)
13: pthread_t thread[realnt]
14: for tid = 0; tid < realnt; tid++ do
15:     crearthread(thread[tid],NULL, count_words_limthreads_aux, separador )
16: end for
17: for tid = 0; tid < realnt; tid++ do
18:     jointhread(thread[tid], NULL )
19: end for
20: return escri
```

count_words_limthreads_aux se diferencia a las anteriores de la forma en que ésta, debe asegurarse de manejar a los threads inactivos, y no permitir que mas de 1 thread trabaje en un mismo archivo.
Esto lo hacemos con el while

while (**do**$(x = (caux- > ult).fetch\_add(1)) < ((caux- > vecti)- > size()))$

end while
Que chequea que la variable ultima no sea mayor a la cantidad de archivos dentro del vector de archivos a leer, ult es un contador atómico para evitar race conditions. Por último, realizamos el agregado de palabras como en el item anterior con el archivo pasado por ult.

2.8. Maximum(unsigned int p archivos, unsigned int p maximos, list<string>archs) No Concurrente

Esta función toma la lista de archivos, la cantidad de threads para leer archivos, y la cantidad de threads para procesar el máximo.
Comenzamos procesando todos los archivos, con un thread por archivo. La consigna nos dice que no podemos usar versiones concurrentes de count_words.Entonces, creamos un vector de hash(uno por cada archivo), y las threads escribirán cada archivo en un hash diferente, llamando a count_words (no concurrente) para ello.Aquí, el contador atómico nos sirve para diferenciar indexar tanto el archivo como el hashmap a utilizar.

Luego de procesados todos los archivos, para realizar el máximo debemos fusionar los hashmap para conseguir el verdadero máximo, ya que la palabra más usada podría estar dispersa en distintos hash parciales. Con un while leeremos cada una de las entradas de todos los hashmaps y las agregaremos a un único hashmap final y luego con este finalmente, llamaremos a la función maximum previamente implementada con la cantidad de threads que el parámetro nos pide.

Algorithm 10 ConcurrentHashMap maximum(unsigned int p_archivos, unsigned int p_maximos, lista<string>archs)

```
1: int realtn = min(p_archivos, archs.size())
2: vector<ConcurrentHashMap> escri(archs.size(), new ConcurrentHashMap)
3: vector<string> archivador
4: lista<strings>::iterador iteaux = archs.CrearIt()
5: while iteaux ! = archs.end() do
6:     archivador.pushback(∗iteaux)
7:     iteaux++
8: end while
9: HashescritorConc2 * separador
10: separador− >h = &escri
11: separador− >vecti = &archivador
12: separador− >ult_pos.store(0)
13: pthread_t thread[realnt]
14: for tid = 0; tid < realnt; tid++ do
15:     crearthread(thread[tid],NULL, count_words_limthreads_aux2, separador )
16: end for
17: for tid = 0; tid < realnt; tid++ do
18:     jointhread(thread[tid], NULL )
19: end for                                          ▷ aqui cada posicion de escri tiene un hash
20: ConcurrentHashMap hash_recolector
21: for i por cada posicion del vector de hash (escri) do
22:     for j por cada una de las 26 entradas do
23:         Lista<string, int> ::iterador it = (escri[i]− >entrada(j))− >CrearIt
24:
25:         while it.Haysiguiente() do
26:             agrego a hash_recolector el la clave it.Siguiente() tantas veces como estaba
27:             it.Avanzar()
28:         end while
29:     end for
30: end for                                  ▷ Aca hash_recolector es la fusion de todos los hash parciales
31: <string, int> supermax = hash_recolector(p_maximos)
32: return supermax
33:
```

Algorithm 11 void* count_words_limthreads_aux2(void* separador)

```
1: HashescritorConc2* caux = separador
2: int x
3: while (x = (caux− >ult_pos).fetch_add(1)) < ((caux->vecti)->size()) do
4:     string archivo =(*(caux->vecti))[x]
5:     ConcurrentHashMap* hash_actual = (*(caux->h))[x]
6:     hash_actual = count_words(archivo)
7: end while
```

Aclaración: Lo llamamos No Concurrente para diferenciarlo con el maximum donde podemos usar el count_word concurrente, pero de hecho hay concurrencia en ambas funciones(por eso el contador atómico, al que llamamos ult_pos, y la concurrencia del maximum del ej1 por sí misma. Además, entrada es una funcion de la clase ConcurrentHashmap que me devuelve puntero a la lista del hash en la posicion pedida

2.9. Maximum(unsigned int p archivos, unsigned int p maximos, list<string>archs) Concurrente

Esta función tiene el mismo funcionamiento que el maximum del ejercicio anterior, pero podemos utilizar versiones concurrentes de count_words. Debido a que nos pasan la cantidad de threads para leer archivos por

parámetro, usamos el count_words3 con el p_archivos como parámetro, y llamamos a maximum del hash resultante con p_maximos. Devolvemos la tupla resultado, que es el máximo que queremos.

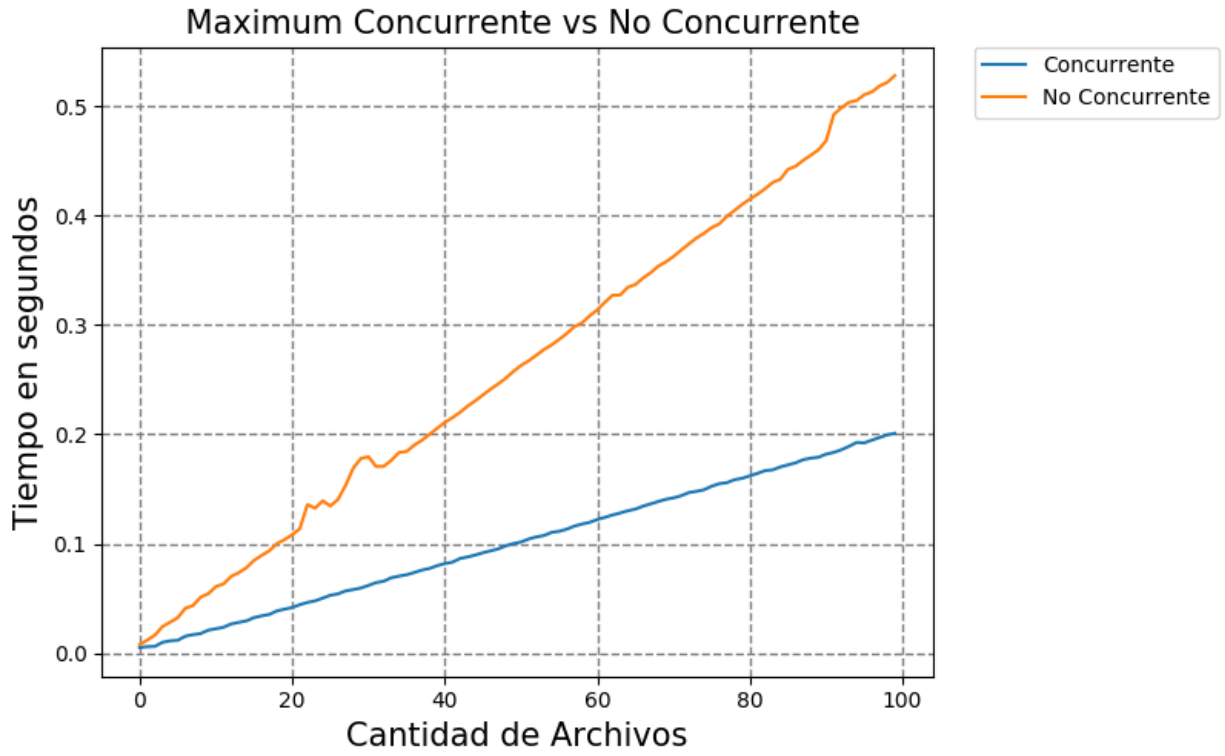
Algorithm 12 ConcurrentHashMap maximum(unsigned int p_archivos, unsigned int p_maximos, lista<string>archs)

```
1: ConcurrentHashMap escri = count_words3(p_archivos, archs )
2: pair<string, int> supermax = escri.maximum(p_maximos)
3: return supermax
```

3. Experimentos

3.1. Ejercicio 5 vs Ejercicio 6

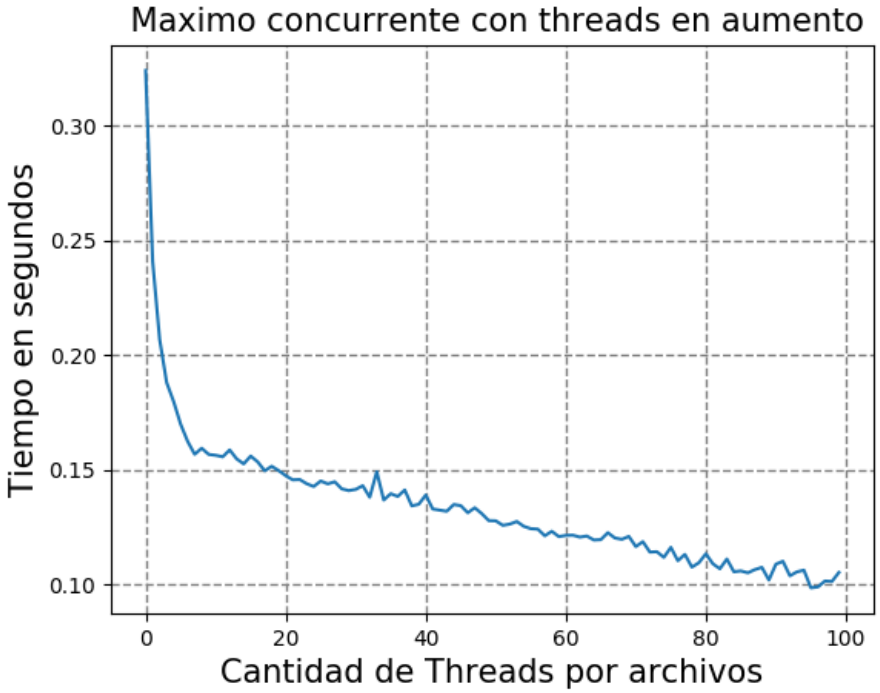
En este experimento aumentamos la cantidad de archivos, dejando constante la cantidad de threads, tanto para archivos como para máximos, hasta llegar a una relación de 25 a 1 (archivos contra threads). Y tomamos el tiempo de la función maximum tanto la versión concurrente como no concurrente.



Podemos ver que el algoritmo con concurrencia es más rápido que el que no usa concurrencia. Esto se debe a que el algoritmo sin concurrencia crea tantos hash como archivos y luego los junta en un único hash para calcular el máximo, mientras que el segundo tiene un solo hash que lo llena usando concurrencia. En ambos casos vemos que la cantidad de archivos pasados influye en el algoritmo. Esto, como después vemos, se debe a la relación de threads y archivos.

3.2. Threads en Aumento

Acá queremos ver que pasa cuando disminuye la diferencia entre threads y archivos del experimento anterior. Para esto empezamos con un solo threads, o sea, realizamos el algoritmo sin concurrencia hasta llegar a una relación 1 a 1 con los archivos (esta relación comienza 100 a 1).



Lo que podemos ver es que a medida que aumentamos la cantidad de threads, de archivos, el tiempo de ejecución del algoritmo disminuye. Notar que la relación llega 1 a 1 y no pasa a haber más threads que archivos porque la función toma como máximo de threads la cantidad de archivos.Suponemos que esto se debe a que el algoritmo se aprovecha de la paralelización que le proveen los threads para leer el archivo, logrando una mejor performance.

4. Conclusiones

4.1. Modificación de Test de la cátedra

Se hicieron leves modificaciones a los test de la cátedra para que compilasen con nuestros .cpp y .h y llamaran adecuadamente de las funciones de la estructura hash. Además incluimos un test más (igual que el $test - 5$) para chequear la función maximum no concurrente (ej5).

4.2. Como correr los test

Para compilar el tp, usar "make compilar".
Para ejecutar todos los tests nuestros, usar "make ejecutar"después de usar "make compilar".
No modificamos la forma de compilar y ejecutar los test de la cátedra que estaban en el makefile por defecto.

4.3. Conclusiones finales

En este trabajo pudimos apreciar la mejora en eficiencia al usar varios threads por función, pero evitando las posibles race conditons que pudieran surgir.
A simple vista, nos parece muy benéfico colocar más cantidad de threads para procesar, pero esto conlleva mas obligaciones que asumir. Si no evitáramos que ensucien sus variables compartidas, el resultado obtenido no seria el correcto.