DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 2

Pthreads

Primer Cuatrimestre de 2017 — Sistemas Operativos

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Sticco, Patricio	337/14	patosticco@gmail.com
Cadaval, Matías	345/14	matias.cadaval@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

En este TP, realizaremos la implementación de un *hashmap* preparado para realizar ciertas operaciones de manera concurrente, cuyas claves son palabras conformadas por caracteres en el rango a-z y sus valores asociados son enteros sin signo.

Este `ConcurrentHashMap<string, uint>`, es una tabla de hash abierta, lo que significa que al haber una colisión se genera una lista enlazada dentro del bucket (la cual esta preparada para ser utilizada por *threads* concurrentes). En caso de agregar una clave ya contenida en el diccionario, se incrementa su valor asociado.

La implementación fue realizada mediante una clase de `C++` y tiene la siguiente interfaz:

- `ConcurrentHashMap()`: Constructor. Crea la tabla, que tiene 26 entradas (una por cada letra del abecedario).
- `addAndInc(string key)`: Si `key` está definida, incrementa su valor, y sino, crea el par `(key, 1)`. Garantiza que sólo haya contención en caso de colisión de hash.
- `member(string key)`: retorna `true` si y sólo si `key` está definida en el diccionario. Esta operación es *wait-free*.
- `maximum(unsigned int nt)`: devuelve el par `(k,m)` tal que `k` es la clave con máxima cantidad de apariciones y `m` es su valor asociado. Su implementación cuenta con concurrencia interna, utilizando `nt` *threads*.

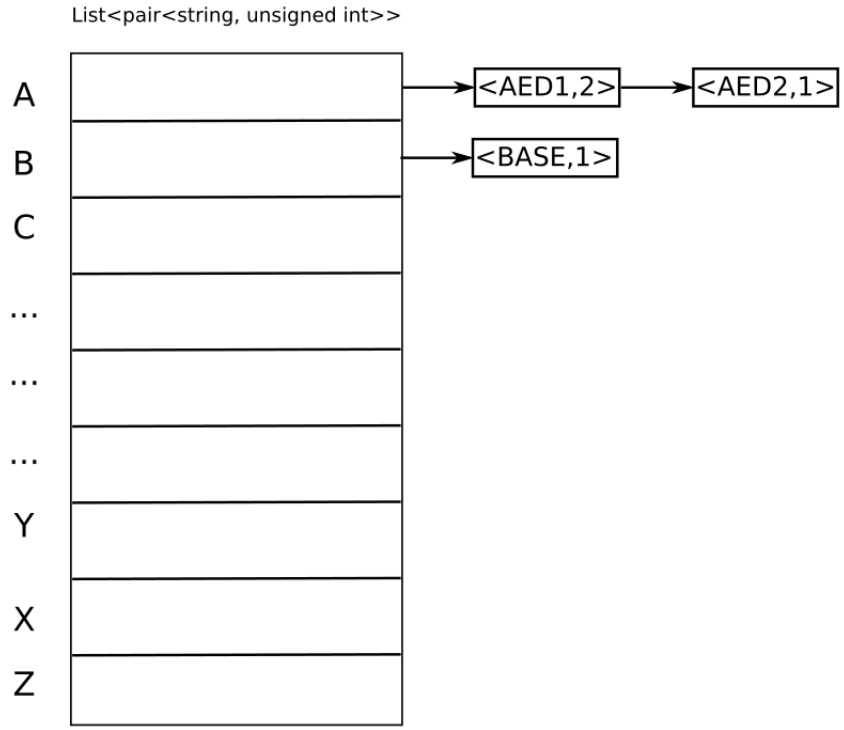

Figura 1: Estructura del diccionario

2. Ejercicios

En esta sección, explicaremos en detalle la implementación de un conjunto de funciones que utilizan un `ConcurrentHashMap`. Dado que las mismas generan instancias de `ConcurrentHashMap`, pero no son funciones propias de la clase, las declaramos como métodos estáticos dentro de la misma. Pero antes de pasar a la explicación de estas funciones, primero veamos la implementación que hicimos para los métodos públicos de la clase `ConcurrentHashMap`.

Implementación de la clase `ConcurrentHashMap`

La estructura de representación de nuestro diccionario está conformada por un vector `tabla` de 26 listas cuyos nodos son un pares del tipo `(string, uint)`.

Además, para garantizar que nuestra implementación este libre de *race conditions*, utilizamos un arreglo `pthread_mutex_t mutex[SIZE_TABLE`[1]`]` y para las funciones donde buscamos un máximo, hacemos uso de un `pthread_mutex_t mutex_maximum`. El uso de estos mutex (proporcionados por la biblioteca `pthreads`), será detallado en las funciones correspondientes más adelante en este informe.

Otro detalle a tener en cuenta, es que dentro de la clase se declara la función de hash como método privado, la cual está implementada de manera que cada caracter inicial posible establezca una relación de manera unívoca con uno de los *buckets* de nuestro diccionario.

En cuanto a los métodos de la clase, los puntos que creemos convenientes a desarrollar sobre cada uno de ellos son los siguientes:

- `ConcurrentHashMap::ConcurrentHashMap()`:

  El constructor de las instancias proporcionadas por la clase, donde se inicializan tanto las *listas* como los *mutex* mencionados anteriormente.

- `bool ConcurrentHashMap::member(string key)`:

  Al no ser una operación que garantice funcionamiento concurrente, su implementación se reduce a retornar *true* si la clave pasada por parámetro pertenece a la lista contenida en el *bucket* correspondiente (el cual se calcula mediante la función de hash). Caso contrario, se retorna *false*.

- `void ConcurrentHashMap::addAndInc(string key)`:

  En este caso, al presentar un funcionamiento con posibilidad de uso concurrente, se tuvieron que tomar las precauciones necesarias para evitar *race conditions*. Cabe destacar que se garantiza que sólo haya contención en caso de colisión de hash, es decir, sólo hay locking a nivel de cada posición del vector `tabla`. Luego, si la clave no está definida, se agrega un nuevo nodo a la lista correspondiente, y en caso contrario, se incrementa su valor en el nodo ya existente.

  Para un mejor entendimiento del algoritmo, se presenta a continuación un pseudocódigo de la implementación:

[1] SIZE_TABLE = 26

Algoritmo 1 addAndInc

void ConcurrentHashMap::addAndInc(string key)

1: posicion ← fHash(key[0])
2: lock(&mutex_maximum) *//su utilización sera desarrollada más adelante*
3: lock(&mutex[posicion])
4: Agrego la nueva clave o aumento su valor
5: unlock(&mutex[posicion])
6: unlock(&mutex_maximum)

∘ `pair<string, unsigned int>ConcurrentHashMap::maximum(unsigned int nt)`:

En `maximum`, el uso de distintos hilos de ejecución concurrentes puede traer varios de los problemas vistos en la materia, por lo que debemos manejar con cuidado el procesamiento de cada *thread* creado. En concreto, el posible problema con `maximum` reside en la posibilidad de que en el medio de la búsqueda del máximo, otro proceso agregue nuevas claves en el una parte del diccionario ya recorrida. Esto se soluciona mediante el uso de `mutex_maximum`, el cual previene que se agreguen nuevas claves mientras se esté realizando la búsqueda.

Como introducción, se necesita presentar a continuación la estructura `thread_data_max`, la cual usaremos tanto para identificar a cada *thread* como para implementar la comunicación entre los mismos.

```
struct thread_data_max {
  unsigned int thread_id;
  ConcurrentHashMap* map;
  vector< pair<string, unsigned int> >* maximos;
  atomic<int>* fila;
};
```

El vector `maximos` se utiliza para que los *threads* guarden el máximo de cada fila, mientras que el entero atómico `fila` se utiliza para comunicar al resto de los *threads* cual es la próxima fila a recorrer.

En pocas palabras, nuestro algoritmo crea `nt` *threads* (acotando superiormente `nt` por SIZE_TABLE, ya que no tiene sentido crear más *threads* que las listas a recorrer), y cada uno de ellos busca la clave con valor máximo de alguna lista (por lo menos una). Mientras hayan listas por recorrer el thread seguirá tomándolas, buscando sus máximos, y guardándolos en la posición correspondiente del vector `máximos`. Podría pasar que se recorran todas las filas antes de terminar con la creación de todos los *threads*, pero esto no es un problema; el thread creado simplemente se destruirá al ver que ya no hay nada que hacer. Una vez que se termine de recorrer la tabla, el thread principal busca el máximo elemento entre los máximos de cada fila (guardados en `maximos`) y lo devuelve.

2.1. Ejercicio 1

2.1.1. Introducción

Se pide realizar una implementación para la función `push_front` de la lista atómica provista por la cátedra. Utilizando esa lista, implementamos la clase `ConcurrentHashMap` respetando las especificaciones anteriormente mencionadas.

2.1.2. Desarrollo

Dado que la función `push_front` debe ser atómica, no nos alcanza simplemente con cambiar la cabeza de la lista por un nuevo nodo y asignarle como siguiente la antigua cabeza. Si bien la idea del algoritmo es esa, debemos ser precavidos ante la concurrencia de múltiples *threads*. Para ello, utilizamos la función `compare_exchange_weak` provista por la librería `<atomic>` de `C++`. La usamos de la siguiente manera:

Algoritmo 2 push_front

void Lista::push_front(const T& val)

```
1: Nodo* new_node ← new Nodo(val)
2: new_node→siguiente ← head.load()
3: mientras !(head.compare_exchange_weak(new_node→next, new_node))
4: fin mientras
```

Lo que estamos haciendo es, previo a salir de la función, validar atómicamente si el valor actual de `head` coincide con el que seteamos antes como `new_node→next`. Si es así sale de la función. Caso contrario reasigna `new_node→next` con el nuevo valor de `head`, y hace la validación nuevamente.

2.2. Ejercicio 2

2.2.1. Introducción

Se pide implementar una función `ConcurrentHashMap count_words(string arch)` que tome un archivo de texto y devuelva el `ConcurrentHashMap` cargado con las palabras que contiene (las mismas están separadas por espacios). En este caso, la implementación es para un funcionamiento no concurrente.

2.2.2. Desarrollo

Dado que el algoritmo implementado es bastante directo sobre el cumplimiento de la poscondición, creemos conveniente presentar un pseudocódigo del mismo a modo de explicación:

Algoritmo 3 count_words

ConcurrentHashMap count_words(string arch)

```
1: ConcurrentHashMap map
2: mientras queden palabras en arch
3:     map.addAndInc(palabra)
4: return map
```

Como mencionamos previamente, al no ser una operación que garantice un correcto funcionamiento con procesamiento paralelo, no fue necesario establecer ninguna política de sincronización.

2.3. Ejercicio 3

2.3.1. Introducción

Se pide implementar la función `ConcurrentHashMap count_words(list<string>archs)` que tome una lista de archivos de texto y devuelva el `ConcurrentHashMap` cargado con las palabras que contengan estos. En este caso, la implementación deberá ser concurrente y utilizarse un *thread* por archivo.

2.3.2. Desarrollo

Para resolver este ejercicio, además de la función pedida primero se definió una estructura adicional llamada `thread_data` que contiene los parámetros asignados a cada *thread*, de la siguiente manera:

```
struct thread_data {
    unsigned int thread_id;
    ConcurrentHashMap* map;
    string archivo;
};
```

Esta estructura, será pasada como parámetro de la función auxiliar `llenarHashMap`, la cual deberá ser ejecutada por cada *thread*.

Retomando la descripción del algoritmos, en primer lugar se crea un *ConcurrentHashMap* vacío, luego se itera cada elemento de la lista y por cada uno de estos se crea un *thread* que se encargará de recorrer todo el archivo de texto agregando las palabras al *ConcurrentHashMap* de manera similar al ejercicio anterior.

El pseudocódigo de las implementaciones recién mencionadas es el siguiente:

Algoritmo 4 llenarHashMap

Void *ConcurrentHashMap::llenarHashMap(void *thread_args)

```
1: struct thread_data* my_data;
2: my_data ← (struct thread_data*) thread_args;
3: string arch ← (my_data→archivo)
4: open arch
5: mientras queden palabras en arch
6:     my_data→map.addAndInc(palabra);
7: close arch
8: pthread_exit(NULL);
```

Algoritmo 5 count_words

ConcurrentHashMap ConcurrentHashMap::count_words(list<string> archs)

1: num_threads ← archs.size()
2: pthread_t threads[num_threads]
3: thread_data args[num_threads]
4: ConcurrentHashMap map
5: **para** cada elemento en archs
6: seteo los argumentos del futuro thread de forma que 'elemento' sea el archivo a recorrer y los guardo en args[i] donde i es el indice del elemento actual
7: creo un thread con los argumentos recién seteados y que ejecute llenarHashMap. Este lo guardo en threads[i] siendo i el ya mencionado
8: **fin para**
9: espero a que todos los threads finalicen su ejecución
10: Return map

2.4. Ejercicio 4

2.4.1. Introducción

Se pide una vez más implementar la función `count_words`. En esta ocasión el número de *threads* a ser utilizados es definido por el parámetro `nt`, pudiendo ser este menor a la cantidad de archivos. Se requiere además que no haya *threads* ociosos en caso de haber archivos aún sin procesar.

La aridad de la función es la siguiente:

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

2.4.2. Desarrollo

Como en el ejercicio anterior, primero fue necesario definir una estructura utilizada para guardar los datos de cada *thread* y la rutina a ser ejecutada por cada uno de estos, llamada `llenarHashMap2`.La estructura en cuestión fue la siguiente:

```
struct thread_data2 {
    unsigned int thread_id;
    ConcurrentHashMap* map;
    list<string>::iterator* it_inicio;
    list<string>::iterator it_fin;
    pthread_mutex_t* mutex_it;
};
```

En este caso, el uso de un iterador `it_inicio` para indicar cual es el próximo archivo a leer por el *thread* es necesario ya que no sabemos si hay uno por cada elemento de la lista (un *thread* podría agregar varios archivos al diccionario). El otro iterador, sirve para poder ver cuando se llegó al fin de la lista.

Con respecto al algoritmo, inicialmente se crea un `ConcurrentHashMap` vacío, el cual sera alcanzable por todos los *threads*. Luego, se crean los `nt` hilos de ejecución y se ejecuta el handler de `llenarHashMap2` en cada uno de ellos. Dado que todos ellos recorren la misma lista concurrentemente, esto podría traer problemas con el iterador, por lo que sincronizamos dicho funcionamiento mediante el uso del `pthread_mutex_t mutex_it`, el cual esta presente como parámetro en cada una de las llamadas. De esta manera, el iterador será modificado de a un *thread* por vez.

Ya en `llenarHashMap2`, mientras este iterador sea distinto al iterador que apunta al final de la lista, el *thread* que haya traspasado el *mutex* mueve el iterador del último archivo leído y trabaja sobre dicho archivo, haciendo que los demás *threads* no puedan trabajar sobre este. Una vez que el *thread* tiene un archivo asignado este libera el *mutex* de forma que otro *thread* lo pueda atravesar para así comenzar a leer un archivo distinto, generando que no haya *threads* ociosos.

Por lo tanto, una vez terminada la ejecución de todos los `threads`, el `ConcurrentHashMap` ya contiene todas las palabras de los distintos archivos, por lo tanto puede ser retornado cumpliendo la poscondición del problema.

2.5. Ejercicio 5

2.5.1. Introducción

Se pide implementar la función `maximum`, que tome como parámetro la cantidad de *threads* a utilizar para leer los archivos (`p_archivos`), la cantidad de *threads* a utilizar para calcular máximos (`p_maximos`) y la lista de archivos a procesar (`archs`). La misma no puede utilizar `count_words` para leer los archivos.

2.5.2. Desarrollo

El algoritmo propuesto se puede dividir en 3 etapas:

- Parseo de los archivos en `p_archivos` instancias distintas de `ConcurrentHashMap`: crea `p_archivos` *threads* que irán leyendo cada uno de los archivos. Esta parte de la implementación es similar a la del ejercicio 4, con la excepción de que a cada thread se le da un `ConcurrentHashMap` distinto en lugar del mismo. Utiliza la estructura `thread_data2` y la función `llenarHashMap2`.
- Merge de `ConcurrentHashMaps` en un único `ConcurrentHashMap`: por cada uno de los `p_archivos` diccionarios que se tienen, crea `p_maximos` *threads*, que van recorriendo las listas e insertando los elementos tantas veces como indique su entero asociado, utilizando `addAndInc`.
- Cálculo del máximo: invoca al método público `maximum` para el `ConcurrentHashMap` sobre el que se hizo el merge previamente.

Adjuntamos únicamente el pseudocódigo de la función `mergeHashMap`, ya que el resto del código es similar al descripto en los ejercicios previos. Esta función es la que ejecuta cada uno de los *threads* para hacer el merge de los diccionarios y recibe una estructura de tipo `thread_data_merge`, que contiene un id para identificar el diccionario, otro para el thread, el diccionario origen (del que se lee), el diccionario destino (en el que se hace el merge), y la variable atómica `fila`, compartida por todos lo threads para distribuirse las filas.

```
struct thread_data_merge {
    unsigned int map_id;
    unsigned int thread_id;
    ConcurrentHashMap* map;
    ConcurrentHashMap* merge_map;
    atomic<int>* fila;
};
```

Algoritmo 6 mergeHashMap

Void *ConcurrentHashMap::mergeHashMap(void *thread_args)

```
1: struct thread_data_merge* my_data
2: my_data ← (struct thread_data*) thread_args
3: mientras haya filas de my_data→map para mergear
4:     para cada elemento T de la lista
5:         para i → 0 .. T.second
6:             my_data→merge_map→addAndInc(T.first)
7:         fin para
8:     fin para
9: fin mientras
10: pthread_exit(NULL);
```

2.6. Ejercicio 6

2.6.1. Introducción

Se pide implementar la misma funcionalidad que en el ejercicio 5, pero utilizando las versión concurrente de `count_words`. También debemos comparar resultados entre ambas implementaciones.

2.6.2. Desarrollo

Identificamos a esta función con el nombre `maximum_cw`. La implementación utilizada para esta versión es bastante sencilla. Simplemente invocamos a `count_words` con `p_archivos` *threads*, y luego llamamos a `maximum` para el `ConcurrentHashMap` generado.

Algoritmo 7 maximum_cw

ConcurrentHashMap ConcurrentHashMap::maximum_cw
(uint p_archivos, uint p_maximos, list<string> archs)

```
1: ConcurrentHashMap map ← count_words(p_archivos, archs);
2: pair <string, uint> max ←map.maximum(p_maximos);
3: return max
```

2.6.3. Experimentación

En la siguiente sección, realizaremos una serie de comparaciones entre la implementaciones presentadas en los últimos dos ejercicios sobre un mismo problema. En principio, nuestra hipótesis es que la del ejercicio cinco debe ser más rápida, ya que al realizar la búsqueda en varios `ConcurrentHashMap` independientes, deberían haber menos *locks* entre los distintos *threads*, los cuales son eventos costosos en términos de tiempo de ejecución como ya vimos en la materia.

En cuanto a los experimentos realizados, la cantidad de archivos va a ser fija (5 archivos de 200 palabras cada uno) mientras que con respecto a la medición de los tiempos, los mismos se toman en nanosegundos mediante la biblioteca `time.h` provista por `C++`.

. En primera instancia realizamos mediciones sobre el tiempo de ejecución variando p_max entre uno y cincuenta, mientras que p_arch se mantiene constante en 5. Mientras que en una segunda serie de pruebas, fijamos p_max en cinco mientras que p_arch oscila entre uno y cincuenta.

A continuación, presentamos los resultados:

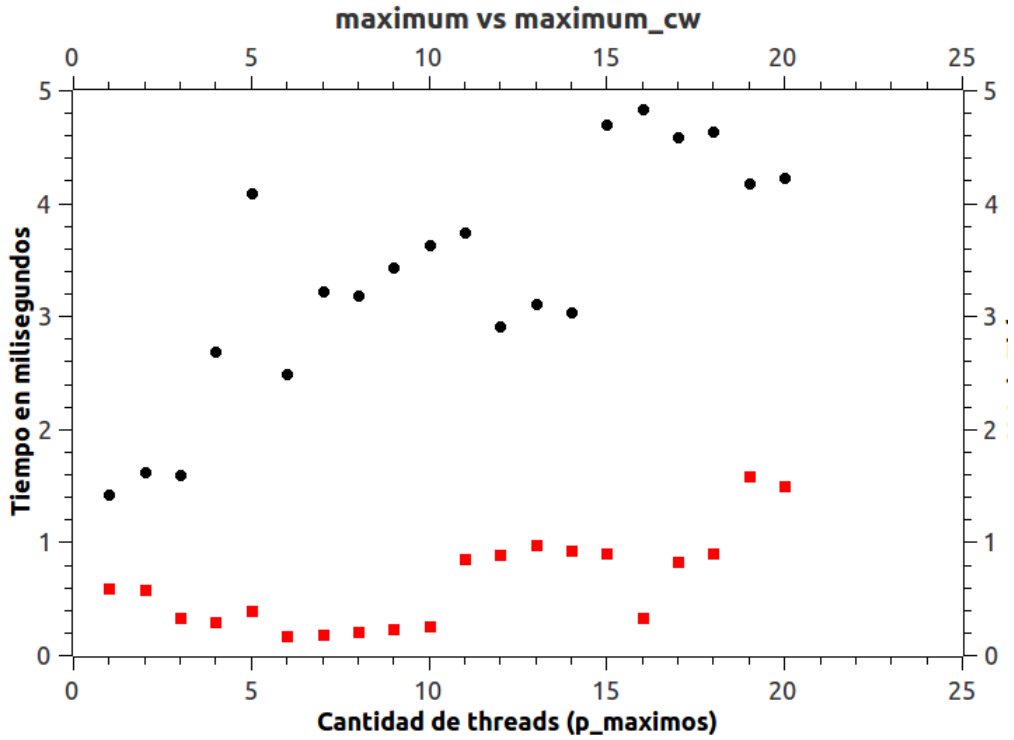

Figura 2: Resultados experimento 1

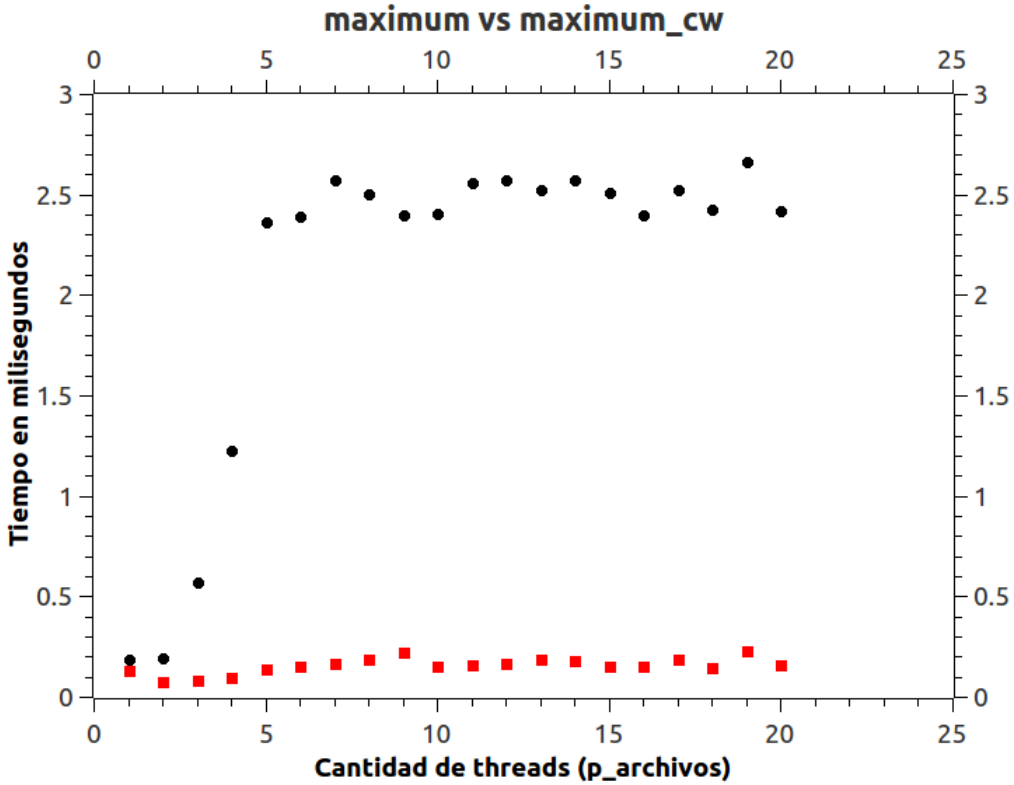

Figura 3: Resultados experimento 2

Cómo se puede ver, nuestra hipótesis fue refutada a raíz de los resultados de la experimentación, donde se puede ver claramente una mejor performance en términos de tiempo de ejecución para el algoritmo detallado en este ejercicio. Creemos que esto es debido a como realizamos el *merge* en la función del ejercicio 5. Queda para futuras experimentaciones evaluar si esta era la causa y buscar una manera más eficiente de *mergear*.

3. Tests

Agregamos algunos tests a los provistos por la cátedra, con el objetivo de probar las funciones implementadas. Hay un test por cada función:

- *test-1a*: Valida el correcto funcionamiento del método `push_front` de la clase `Lista` con múltiples threads. Utiliza 10 threads que llaman a la función de manera concurrente.
- *test-1b*: Valida el correcto funcionamiento del método `maximum` de la clase `ConcurrentHashMap`.
- *test-2*: Valida el correcto funcionamiento de la versión de `count_words` que recibe el nombre del archivo como parámetro.
- *test-3*: Valida el correcto funcionamiento de la versión de `count_words` que recibe el número de *threads* y la lista de archivos como parámetros.
- *test-4*: Valida el correcto funcionamiento de la versión de `count_words` que recibe la lista de archivos como parámetro.
- *test-5*: Valida el correcto funcionamiento de la función `maximum`.
- *test-6*: Valida el correcto funcionamiento de la función `maximum_cw`.
- *test-7*: Valida el correcto funcionamiento de la función `maximum_cw`.
- *test-9*: Valida el correcto funcionamiento de la función `maximum`.