DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

DC - UBA

Sistemas operativos

Trabajo Práctico N°2

Integrante	LU	Correo electrónico
Rodrigo Kapobel	695/12	rok_35@live.com
Esteban Luciano Rey	657/10	estebanlucianorey@gmail.com
Nicolas Hernandez	122/13	nicoh22@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) 4576-3300
http://www.exactas.uba.ar

Índice

1. Estructuras **2**

1.1. maxtarg 2

1.2. fileNMap 2

1.3. lockNFileNMap 2

2. Métodos **3**

2.1. void push_front(const T& val) (ListaAtomica) 3

2.2. void addAndInc(string key) 3

2.3. bool member(string key) 4

2.4. pair<string, unsigned int> maximum(unsigned int nt) 5

2.5. void *maxThread(void *args) 5

2.6. void findMaximums(void *args) 6

2.7. ConcurrentHashMap count_words(string arch) 6

2.8. ConcurrentHashMap count_words(list<string>archs) 7

2.8.1. void *$count\_words\_Thread$(void *args) 8

2.9. ConcurrentHashMap count_words(unsigned int n, list<string>archs) 8

2.9.1. void * process_files_Thread(void *args) 9

2.10. pair<string, unsigned int> maximum(unsigned int p_ archivos, unsigned int p_maximos, list<string> archs) 10

2.10.1. void * readFilesThread(void *args) 11

2.11. void merge(ConcurrentHashMap& cm) 12

2.12. pair<string, unsigned int> concurrent_maximum(unsigned int p archivos, unsigned int p maximos, list<string> archs) 12

3. maximums_sin_concurrencia vs. concurrent_maximum **14**

1. Estructuras

1.1. maxtarg

tupla $*$*maximums*[26]
pthread_mutex_t locks[26]
ConcurrentHashMap $*$*hashMap*

Descripción

Esta estructura es útil a la hora de calcular el máximo de un hashmap.

El arreglo *maximums* se utiliza para guardar el máximo clave-valor para cada letra del hashmap.

El arreglo *locks* permite hacer locking de la lista para que no haya más de dos hilos realizando el mismo cálculo sobre la misma lista. En el constructor de la estructura se inicializan los 26 mutexes y en el destructor se destruyen.

1.2. fileNMap

ConcurrentHashMap $*$*hashmap*
string $*filename$

Descripción

Esta estructura se utiliza cuando se desea procesar un archivo en un thread diferente.

1.3. lockNFileNMap

ConcurrentHashMap $*$*hashmap*
pthread_mutex_t $*file\_locks$
$list < string > *file\_names$

Esta estructura se utiliza para procesar archivos de forma concurrente.

$file\_locks$ es el puntero al arreglo de mutexes de cada archivo en $file\_names$. Este ultimo es el puntero al arreglo de nombres de cada archivo. El objetivo de esta estructura es que el thread creado que reciba la misma intente procesar un archivo no lockeado por otro thread, tomando el lock del mismo de ser posible.

2. Métodos

2.1. void push_front(const T& val) (ListaAtomica)

Parámetros

1. **const T& val**: Valor tipo T (Genérico) a insertar.

Pseudocódigo

$Nodo * nuevo = new\ Nodo(val)$
$nuevo \rightarrow next = head.load()$
while $!(head.compare\_and\_exchange(nuevo \rightarrow next, nuevo))$ **do**
end while

Descripción

Inserta de manera atómica el valor T pasado por parámetro. Para lograrlo se utiliza el método *compare_exchange_weak* que ofrece el tipo atómico. El método mencionado realiza una serie de operaciones de manera atómica. Chequea si el *head* sigue siendo *nuevo* $\rightarrow$ *next*, si no lo es, actualiza *nuevo* $\rightarrow$ *next* con el valor actual de *head*, si no cambió, actualiza el *head* con *nuevo*. Esto garantiza que el push se pueda realizar de manera concurrente y libre de condiciones de carrera.

Tests

```
make test-1-run
```

Para comprobar el correcto funcionamiento del método, se generan 5 archivos a partir del archivo *corpus* provisto por la catedra. Luego creamos threads que lean los archivos de manera concurrente y agreguen cada palabra del archivo a una *ListaAtomica* en común (la cual contiene *strings*) usando la función *push_front*. Se varían la cantidad de threads usados entre 1 y 5. Una vez que los threads terminan su trabajo, se imprime a stdout el contenido de la lista y se verifica que sea una permutación del contenido del archivo *corpus*.

2.2. void addAndInc(string key)

Parámetros

1. **string key**: Palabra a agregar a la estructura

Pseudocódigo

$unsigned\ int\ index = hash(key)$
$pthread\_mutex\_lock(lock\_list[index])$
$Iterador\ it = tabla[index].CrearIt()$
while it.HaySiguiente() **do**
    if key.compare(it.Siguiente().first) == 0 **then**
        $it.Siguiente().second++$

```
        pthread_mutex_unlock(lock_list[index])
        return
    end if
    it.Avanzar()
end while
tabla[index].push_front(make_pair(key, 1))
pthread_mutex_unlock(lock_list[index])
```

Descripción

Se inserta de forma atómica la palabra a la estructura. Para lograr la atomicidad de inserción, se adquiere un lock sobre la Lista correspondiente a la inicial de **key**: Se busca linealmente la clave en la Lista; de existir la entrada, se incrementa el contador de la misma, caso contrario se realiza la operación *push_front* del par <**key**, 1>.

Este locking se realiza para evitar errores de consistencia de datos, como por ejemplo, cantidad erronea de apariciones de una clave debido a condiciones de carrera producidas al realizar múltiples inserciones desde diferentes threads.

2.3. bool member(string key)

Parámetros

1. **string key**: Palabra a buscar en la estructura

Retorno

bool True si la palabra se encuentra definida

Pseudocódigo

```
unsigned int index = hash(key)
Iterador it = tabla[index].CrearIt()
while it.HaySiguiente() do
    if key.compare(it.Siguiente().first) == 0 then
        return true
    end if
    it.Avanzar()
end while
return false
```

Descripción

Dado que solo se realiza una lectura de la lista que corresponde a la clave buscada, no es necesario realizar ningún locking especial.

2.4. pair<string, unsigned int> maximum(unsigned int nt)

Parámetros

1. **int nt**: Número de threads a utilizar en el método

Retorno

pair<string, unsigned int>: Tupla que contiene la palabra de máxima cantidad de ocurrencias y la cantidad de ocurrencias respectivamente.

Pseudocódigo

for $i : 0... < 26$ **do**
  $pthread\_mutex\_lock(\&lock_list[i])$
end for
$pthread\_t\ threads[nt]$
$int\ tid$
$maxtarg\ args$
$args.hashMap = this$
for $tid : 0... < nt$ **do**
  $pthread\_create(threads[tid], NULL, maxThread, args)$
end for
for $i : 0... < nt$ **do**
  $pthread\_join(threads[i], NULL)$
end for
$tupla\ max = tupla("", 0)$
for $i : 0... < 26$ **do**
  if $args.maximums[i] != NULL\ and\ args.maximums[i] \rightarrow second > max.second$ **then**
    $max = *args.maximums[i]$
  end if
end for
for $i : 0... < 26$ **do**
  $pthread\_mutex\_unlock(lock\_list[i])$
  $delete\ args.maximums[i]$
end for
$return\ max$

Descripción

El método no es de acceso concurrente con *addAndInc*: para este fin se toman los locks de todas las listas evitando que se pueda generar una inserción durante la ejecución del método generando resultados erróneos.

2.5. void *maxThread(void *args)

1. **void *args casteable a maxtarg**

$ConcurrentHashMap\ *myMap = arg \rightarrow hashMap$

$myMap \rightarrow findMaximums(args)$

2.6. void findMaximums(void *args)

1. **void *args casteable a maxtarg**

for $i : 0... < 26$ **do**
  if $pthread\_mutex\_trylock(arg \rightarrow locks[i]) == 0$ **then**
    $Iterador\ it = tabla[i].CrearIt()$
    if $arg \rightarrow maximums[i] == NULL\ and\ it.HaySiguiente()$ **then**
      $tupla\ max = make_pair("", 0)$
      $int\ c = 0$
      while $it.HaySiguiente()$ **do**
        $tupla\ *actual = \&it.Siguiente()$
        if $actual \rightarrow second > max.second$ **then**
          $max = *actual$
        end if
        $it.Avanzar()$
      end while
      $arg \rightarrow maximums[i] = new\ tupla$
      $arg \rightarrow maximums[i] \rightarrow first = max.first$
      $arg \rightarrow maximums[i] \rightarrow second = max.second$
    end if
    $pthread\_mutex\_unlock(arg \rightarrow locks[i])$
  end if
end for

2.7. ConcurrentHashMap count_words(string arch)

Parámetros

1. **string arch**: Nombre del archivo a convertir en *ConcurrentHashMap*

Retorno

ConcurrentHashMap Copia de *ConcurrentHashMap* cargado con las palabras del archivo parametrizado

Pseudocódigo

$ifstream\ myfile(archivo)$
$ConcurrentHashMap\ hashmap$
if $myfile.is\_open()$ **then** $string\ word$
  while $myfile >> word$ **do**
    $hashmap.addAndInc(word)$
  end while
end if
$myfile.close()$
$return\ hashmap$

Descripción

Se abre el archivo parametrizado, el cual debe existir como precondición. Se parsea el contenido por espacios en blanco (considerando lo obtenido como palabras) y se lo introduce en una nueva estructura *ConcurrentHashMap*.

Para la realización del *ConcurrentHashMap* se utiliza una versión de *count_words* llamada *count_words_Thread* que recibe el *ConcurrentHashMap* por parámetro.

Tests

```
make test-2-run
```

Se genera el output deseado mediante awk y se lo guarda en el archivo *corpus.post*. El mismo contiene para cada palabra presente en *corpus*, una linea con esa palabra y la cantidad de apariciones en el archivo *corpus*. Se ejecuta la función sobre el archivo *corpus* provisto por la catedra. Luego para cada lista del *ConcurrentHashMap* generado se crea un iterador y se imprime la tupla a *stdout*. Finalmente se toma el *stdout* del test y se lo compara con el contenido del archivo *corpus.post*.

2.8. ConcurrentHashMap count_words(list<string>archs)

Parámetros

1. **list<string> archs**: Lista con los nombres de los archivos a convertir en *ConcurrentHashMap*

Retorno

ConcurrentHashMap Copia de *ConcurrentHashMap* cargado con las palabras de los archivos parametrizados.

Pseudocódigo

ConcurrentHashMap hashmap
iterator it = archivos.begin()
pthread_t threads[archivos.size()]
int tid
fileNMap args[archivos.size()]
int nt = archivos.size()
for *tid* : 0... < *nt* **do**
  args[tid].hashmap = &hashmap
  args[tid].filename = it.value
  it++
  pthread_create(threads[tid], NULL, count_words_Thread, args[tid])
end for
for *i* : 0... < *nt* **do**
  pthread_join(threads[i], NULL)
end for
return hashmap

Descripción

Se lee cada archivo parametrizado en un thread aparte. Se utiliza la estructura *fileNMap* para pasar por referencia el ConcurrentHashMap a retornar y el nombre del archivo a analizar. Cada thread inserta las palabras del archivo designado en el ConcurrentHashMap compartido.

Tests

```
make test-3-run
```

De la misma manera que en el test-2 se genera el archivo *corpus.post*. Luego se generan cinco archivos a partir del archivo *corpus*, que a su vez son la lista de archivos que se le pasan por parámetro al algoritmo. Luego de la ejecución del algoritmo se itera por las listas del *ConcurrentHashMap* generado y se imprime el contenido de ellas. Se compara con el archivo *corpus.post*.

2.8.1. void **count_words_Thread*(void *args)

Parámetros

1. **void *args casteable a fileNMap**: Contiene un *ConcurrentHashMap* y el nombre del archivo a leer.

Descripción

La inserción de cada palabra a la estructura es atómica, no así el conjunto de las mismas ya que éste método está pensado para ser ejecutado por múltiples threads a la vez.

2.9. ConcurrentHashMap count_words(unsigned int n, list<string>archs)

Parámetros

1. **unsigned int n**: Threads a utilizar en el método
2. **list<string> archs**: Lista con los nombres de los archivos a convertir en *ConcurrentHashMap*

Retorno

ConcurrentHashMap: Copia de *ConcurrentHashMap* cargado con las palabras de los archivos parametrizados

Pseudocódigo

ConcurrentHashMap hashmap
pthread_t threads[*n*]
pthread_mutex_t file_lock_list[*archivos.size*()]
int tid
lockNFileNMap args[*n*]

for $i : 0... < archivos.size()$ **do**
  $pthread\_mutex\_init(file_lock_list[i], NULL)$
end for
for $tid : 0... < n$ **do**
  $args[tid].hashmap = \&hashmap$
  $args[tid].file\_names = \&archivos$
  $args[tid].file\_locks = \&file\_lock\_list$
  $pthread\_create(threads[tid], NULL, process\_files\_Thread, args[tid])$
end for
for $i : 0... < n$ **do**
  $pthread\_join(threads[i], NULL)$
end for
for $i : 0... < archivos.size()$ **do**
  $pthread\_mutex\_destroy(file_lock_list[i])$
end for
return hashmap

Descripción

Se crea un pool de **n** threads a los que se le pasa la estructura *lockNFileNMap* con la referencia a un $ConcurrentHashMap$ compartido, la lista de nombres de archivos disponibles y un arreglo de locks (tantos como archivos en la lista).

Cada thread intenta obtener tantos locks como le sea posible mediante intentos no bloqueantes, pero no libera los mismos luego de realizada la lectura del archivo (Es decir, un lock indica archivo procesado/-procesando). La idea es que cada thread procese todos los archivos que pueda antes de finalizar.

De obtener el i-esimo lock, comienza a insertar las palabras del i-esimo archivo en el $ConcurrentHashMap$. Al finalizar todos los threads, el programa principal retorna la copia del $ConcurrentHashMap$ compartido y destruye los locks.

Para procesar los archivos en threads separados se utiliza el método $process\_files\_Thread$.

Tests

```
make test-4-run
```

Similarmente a los tests 2 y 3 se generan los archivos *corpus.post*, *corpus-0*, *corpus-1*, *corpus-2*, *corpus-3* y *corpus-4*. Luego se corre el algoritmo variando su parametro n (cantidad de threads) entre 1 y 5. Como lista de archivos se le pasan los corpus del 0 al 4. Se comparan los contenidos del *Concurrent-HashMap* con los contenidos del *corpus.post*.

2.9.1. void * process_files_Thread(void *args)

Parámetros

1. **void *args casteable a lockNFileNMap**: Contiene la lista de nombres de archivos y un arreglo con un lock para cada uno de ellos. Además como precondición, el $ConcurrentHashMap$ no puede ser nulo.

Pseudocódigo

```
for l : 0... < arg → file_names → size() do
    if pthread_mutex_trylock(arg → file_locks[l] == 0 then
        fileNMap args2
        args2.hashmap = arg → hashmap
        iterator it = arg → file_names → begin()
        int lfin = 0
        while lfin < l do
            it++; lfin++;
        end while
        args2.filename = it.value
        count_words_Thread(args2)
    end if
end for
```

Descripción

Se busca un archivo no lockeado y el nombre en la lista de archivos y se crea una estructura *fileNMap* con el *ConcurrentHashMap* y el nombre del archivo a procesar llamando luego a *count_words_Thread()* para procesar ese archivo en un thread nuevo.

2.10. pair<string, unsigned int> maximum(unsigned int p_ archivos, unsigned int p_maximos, list<string> archs)

Parámetros

1. **unsigned int p_archivos**: Threads a utilizar en la lectura de los archivos.
2. **unsigned int p_maximos**: Threads a utilizar en el cálculo del máximo.
3. **list<string> archs**: Lista con los nombres de los archivos a analizar para calcular el máximo

Retorno

pair<string, unsigned int>: Tupla que contiene la palabra de máxima cantidad de ocurrencias y la cantidad de ocurrencias de la misma respectivamente.

Pseudocódigo

```
pthread_t threads[p_archivos]
int tid
pthread_mutex_t file_lock_list[archs.size()]
lockNFileNMap args
ConcurrentHashMap total
for i : 0... < archs.size() do
    pthread_mutex_init(file_lock_list[i], NULL)
end for
args.hashmap = &total
```

args.file_names = &*archs*
args.file_locks = *file_lock_list*
for *tid* : 0... < p_a*rchivos* **do**
    pthread_create(*threads*[*tid*], *NULL*, *readFilesThread*, *args*)
end for
for *i* : 0... < p_a*rchivos* **do**
    pthread_join(*threads*[*i*], *NULL*)
end for
return total.maximum(*p_maximos*)

Descripción

Esta versíon del método utiliza la primer versión de *count_words*. Utilizando la estructura *lockNFileNMap*, se le pasa a la misma un *ConcurrentHashMap* vacío en donde todos los threads volcarán sus datos, una estructura de locks para cada archivo (replicando la idea de pool de la versión 3 de *conunt_words*) y la lista de archivos.

Cada thread, al utilizar la versión 1 de *count_words* genera un *ConcurrentHashMap* del archivo que analizó; para unirlo a los resultados globales se hace uso de la función *merge* la cual incluye toda la información del archivo parseado al *ConcurrentHashMap* global. Al finalizar todos los threads, se aplica sobre la estructura global la función *maximum* parametrizada con **p_maximos** para obtener el resultado.

Como hay un límite de *p_archivos* para procesar todos los archivos en hilos separados se utiliza *readFilesThread* para realizar esta acción.

Tests

```
make test-5-run
```

2.10.1. void * readFilesThread(void *args)

Parámetros

1. **void *args casteable a lockNFileNMap**: Estructura con el arreglo de locks, la lista d earchivos y el hashmap total donde se realiza el merge del *ConcurrentHashMap* auxiliar creado en este método.

Pseudocódigo

for *l* : 0... < *arg* → *file_names* → *size*() **do**
    if *pthread_mutex_trylock*(*arg* → *file_locks*[*l*]) == 0 **then**
        iterator it= *arg* → *file_names* → *begin*()
        int lfin = 0
        while *lfin* < *l* **do**
            it++; *lfin*++;
        end while
        ConcurrentHashMap cm = *count_words*(∗*it*)
        arg → *hashmap* → *merge*(*cm*)
    end if
end for

2.11. void merge(ConcurrentHashMap& cm)

1. **ConcurrentHashMap& cm**: Referencia al hashmap del que se obtendran los valores a mergear.

Pseudocódigo

```
for t : 0... < 26 do
    Iterador it = cm.tabla[t].CrearIt()
    while it.HaySiguiente() do
        tupla tup = it.Siguiente()
        unsigned int index = hash(tup.first)
        pthread_mutex_lock(lock_list[index])
        Iterador it2 = tabla[index].CrearIt()
        bool exists = false
        while it2.HaySiguiente() do
            if tup.first.compare(it2.Siguiente().first) == 0 then
                it2.Siguiente().second += tup.second
                exists = true
            end if
            it2.Avanzar()
        end while
        if !exists then
            tabla[index].push_front(tup)
        end if
        pthread_mutex_unlock(lock_list[index])
        it.Avanzar()
    end while
end for
```

Descripción

Se busca cada palabra del hashmap pasado por parámetro en la instancia. De existir se suma la cantidad de apariciones a la existente en la instancia. Caso contrario, se hace push de la palabra y su cantidad de apariciones utilizando *push_front()*.

2.12. pair<string, unsigned int> concurrent_maximum(unsigned int p archivos, unsigned int p maximos, list<string> archs)

Parámetros

1. **unsigned int p_archivos**: Threads a utilizar en la lectura de los archivos.
2. **unsigned int p_maximos**: Threads a utilizar en el cálculo del máximo.
3. **list<string> archs**: Lista con los nombres de los archivos a analizar para calcular el máximo

Retorno

pair<string, unsigned int>: Tupla que contiene la palabra de máxima cantidad de ocurrencias y la cantidad de ocurrencias de la misma respectivamente.

Pseudocódigo

ConcurrentHashMap hashmap = count_words(p_archivos, archs)
return hashmap.maximum(p_maximos)

Descripción

Esta versíón del método utiliza la tercer versión de *count_words*.

Se analiza de manera concurrente todos los archivos, y a diferencia de la versión no concurrente, los resultados no deben ser mergeados en una estructura compartida, sino que se insertan en ella directamente, reduciendo el uso de estructuras y el tiempo en realizar las inserciones. Finalmente se utiliza el método *maximum* para obtener el resultado.

Tests

```
make test-6-run
```

3. maximums_sin_concurrencia vs. concurrent_maximum

Para generar una idea de los tiempos de cada algoritmo, se decidió utilizar ,en una primera aproximación, los archivos *corpus* brindados por la cátedra: los mismos poseen palabras aleatorias repartidas entre 3 archivos. Para cada medición se decide ejecutar cada caso varias veces y promediar los resultados: los casos se ordenan primero por número de threads utilizados para la lectura y luego por threads utilizados para calcular la máxima ocurrencia de palabra.

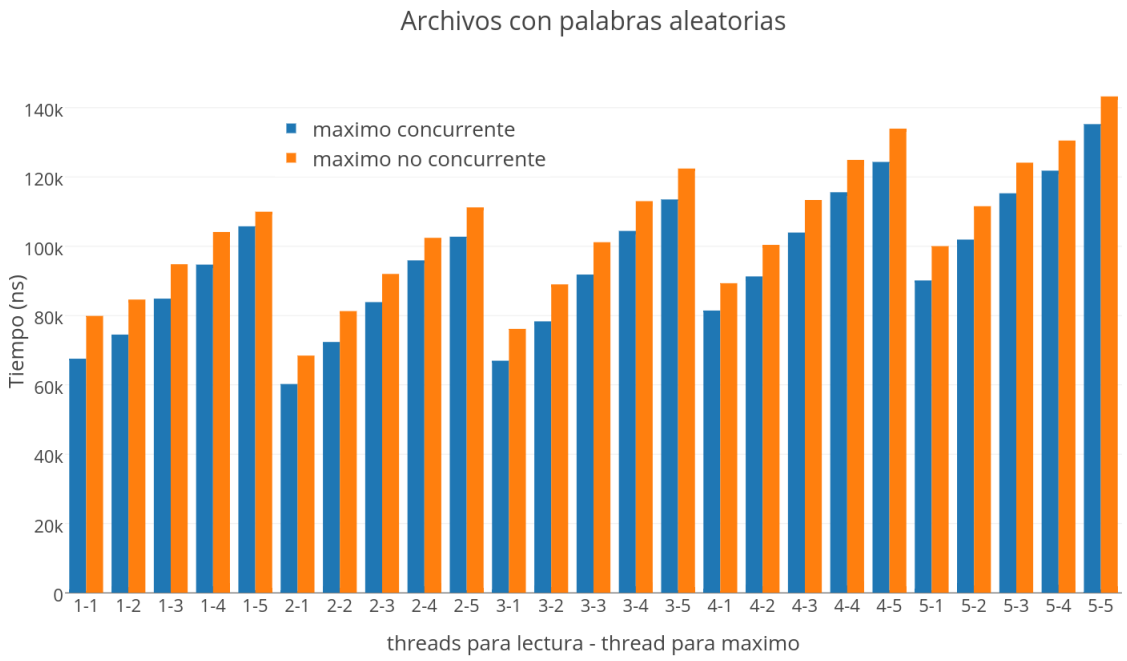

Figura 3.1: Corridas con los archivos de la cátedra

Se puede observar que, en todas las corridas, el algoritmo *concurrent_maximum* le saca una pequeña ventaja al *maximums_sin_concurrencia*. Esto podemos observarlo desde el punto de vista de creación de estructuras: el algoritmo sin concurrencia debe crear una estructura por cada archivo para luego copiar los datos a otra estructura compartida, mientras que el otro algoritmo directamente los inserta en la estructura compartida.

Al plantear un set de archivos diferentes, en donde se agruparon por archivo las palabras con una misma inicial, podemos lograr evidenciar aún más este aprovechamiento de 1 sola estructura por parte del algoritmo concurrente:

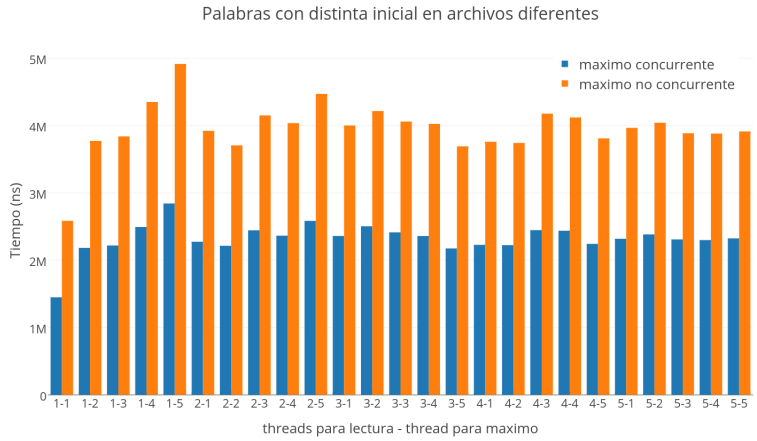

Figura 3.2: Palabras separadas por inicial en diferentes archivos

Teniendo en cuenta esta diferencia de implementación, un caso en que podemos invertir lo visto es uno en el cual la concurrencia sea un detrimento para el algoritmo: la correcta concurrencia necesita de los locks para sincronizarse, pero a la vez estos pueden generar un overhead considerable en el caso de generar un cuello de botella. Si cada archivo posee palabras con la misma inicial, como en el caso del anterior experimento: siendo que 1 solo thread analiza el archivo, entonces no se genera ninguna condición de carrera sobre la lista que representa esa inicial. Pero si ponemos muchas palabras con la misma inicial en distintos archivos, estaríamos generando muchas esperas activas por parte de los diferentes threads. Un caso de lo descripto ocurre en el siguiente set de archivos, en donde todas las palabras son la misma:

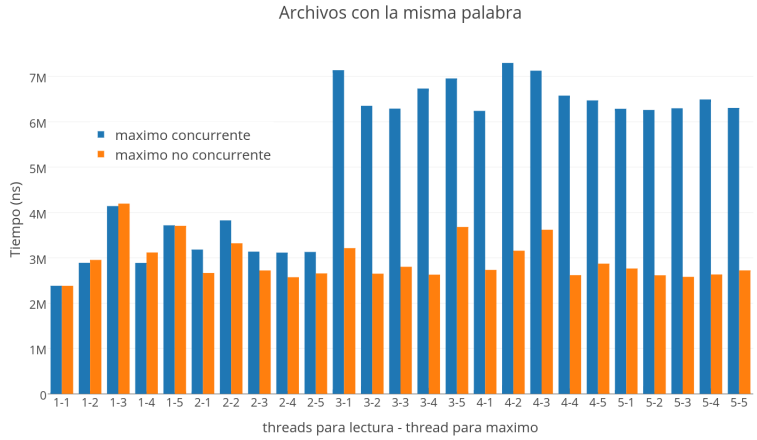

Figura 3.3: Archivos con la misma palabra repetida muchas veces

En la anterior figura podemos ver como mientras que el algoritmo sin concurrencia conserva su tiempo de ejecución a lo largo de las instancias, el otro algoritmo, siempre con peor perfomance que el no concurrente, empeora su ejecución.

Siendo que los archivos tienen la misma palabra, el algoritmo concurrente, por cada thread, impacta en el mismo nodo de la misma lista de la misma letra, tantas veces como repeticiones de la palabra haya: Como cada una de estas listas para ser accedidas requieren de un lock, se genera una enorme cantidad de peticiones de locking todo el tiempo, ralentizando notoriamente el tiempo de ejecución.

El algoritmo no concurrente forma por cada archivo un ConcurrentHashMap separado. Esto quiere decir que en su tarea de ingresar todas las mismas palabras presentes en un mismo archivo, puede hacerlo sin realizar ninguna condición de carrera sobre ningún lock (la estructura no es compartida) de forma muy rápida. La estructura resultante contendrá 1 sola lista (la que corresponde a la inicial de la palabra repetida) y 1 solo nodo en la lista (representando a la única palabra) con la cantidad total de repeticiones en el archivo. Al unir sus resultados con el ConcurrentHashMap compartido, utilizando la función *merge*, se realiza 1 sola inserción concurrente (se pide 1 sola vez el lock sobre la lista), disminuyendo al minimo el bloqueo de la estructura compartida.