DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

HashMapConcurrente

11 de junio de 2023 Sistemas Operativos

1

Integrante	LU	Correo electrónico
Damburiarena, Gabriel	889/19	gabriel.damburiarena@gmail.com
Guastella, Mariano	888/19	marianoguastella@gmail.com
Espil, Victoria	843/19	viespil@gmail.com
Silva, Ignacio Tomas	410/19	ignaciotomas.silva622@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

El mundo de la tecnología está en constante evolución y hoy ese cambio nos lleva a la virtualización del hardware para los usuarios y a estar todos conectados a la nube. Esto nos trae problemas y preguntas como: ¿Qué sucede si varios usuarios quieren acceder al mismo sistema de almacenamiento? ¿Cómo funcionaría una estructura de datos con varios procesos distintos haciendo operaciones sobre ella? ¿Podemos entre todos aportar capacidad de cálculo para resolver un mismo problema? Intentaremos responder esas dudas, entre otras, en el siguiente informe.
Vamos a tratar los temas de estructuras de datos y concurrencia en el sistema operativo. En particular, vamos a implementar un diccionario concurrente sobre una tabla de hash. A continuación introducimos conceptos útiles a los cuales hacemos referencia constantemente en el informe.

1.1. Concurrencia en el sistema operativo

Dado la creciente complejidad de los sistemas y aplicaciones para ejecutar en computadoras, los sistemas operativos tuvieron que brindar una respuesta para que el usuario tenga una sensación de simultaneidad entre sus aplicaciones con tiempos de respuestas razonables. Teniendo en cuenta que el procesador es el recurso principal y que suele ser escaso (en un primer momento había un solo procesador por máquina) la respuesta de los sistemas operativos es alternar la carga de trabajo de diferentes aplicaciones. Si esta alternancia ocurre lo suficientemente rápido, se logra, para el usuario, la sensación de simultaneidad. En los sistemas modernos multiprocesador, se puede incluso separar la carga de trabajo entre diferentes procesadores. Esto último resulta fundamental para aplicaciones que requieren grandes tandas de procesamiento.

1.2. Threading

La concurrencia en el sistema operativo se consigue mediante dos abstracciones: Procesos y threads. Un proceso es un programa en ejecución, que incluye a los valores actuales de su program counter y registros, al igual que sus áreas de memoria y código asociadas, entre otros. En los sistemas operativos modernos es deseable tener múltiples unidades de ejecución que corran en el mismo espacio de direcciones, acá aparece el concepto de threads. Podemos pensar en los threads como versiones más livianas de los procesos, ya que cada uno tiene su stack y sus registros pero no necesitan un espacio de direcciones nuevo. También permite que dentro de un mismo proceso se puedan paralelizar tareas, esto último es importante ya que a veces crear varios procesos no resuelve este mismo problema de paralelización.

1.3. Hashing

Hashing es un método criptográfico que transforma datos de tamaño variable en valores más compactos y de tamaño fijo. Es usado regularmente en aplicaciones de seguridad, bases de datos y algoritmos de búsqueda eficientes. Hay una función de Hash que transforma los datos y luego una tabla donde se almacenan. Nosotros vamos a utilizarlo para hacer un diccionario que servirá como base de datos.

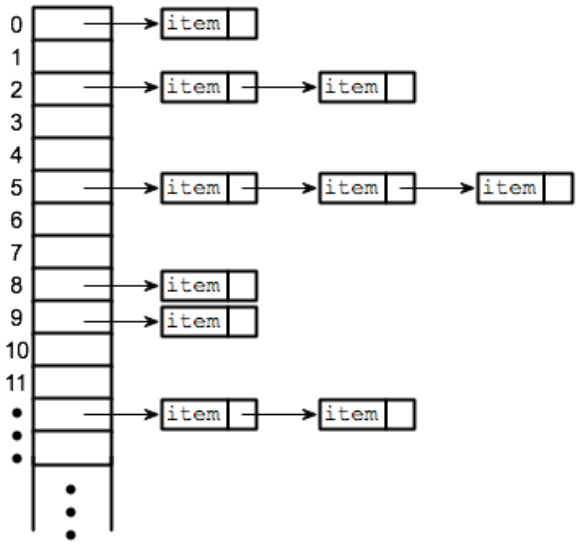

Figura 1: Ejemplo tabla de Hash

Ahora exploremos el problema que le da propósito a este informe.

2. Presentación del problema

Realizaremos una implementación de una estructura de datos llamada HashMapConcurrente, que se basa en una tabla de hash abierta y utiliza listas enlazadas para manejar las colisiones. Esta estructura tiene una interfaz similar a la de un map o diccionario, donde las claves son cadenas de texto. El propósito de esta implementación es poder utilizarla para procesar archivos de texto y contar la cantidad de apariciones de distintas palabras. En este caso, las palabras serán utilizadas como claves y sus respectivas cantidades de apariciones como valores asociados.

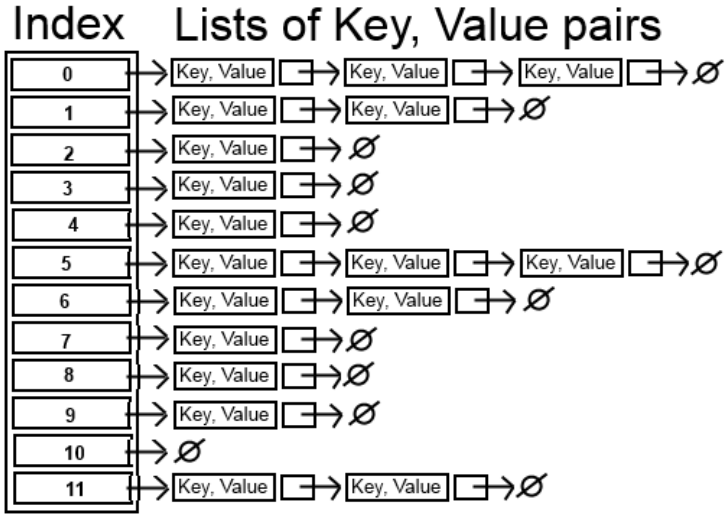

Figura 2: Representación de la estructura de datos, siendo los indices uno por cada letra del abecedario

2.1. Lista Atómica

Esta lista es como cualquier otra lista enlazada pero con una diferencia, el insertar es atómico. No es necesario incluir un mutex en las otras operaciones de la lista ya que tanto en cabeza, i-ésimo y longitud devolvemos los valores en el momento que se solicitaron.

- void insertar(const T &valor): Esta función inserta un nuevo nodo al comienzo de la lista de manera atómica.

2.2. HashMapConcurrente

Sobre este HashMapConcurrente vamos a implementar las siguientes funciones para proveer al usario:

- void incrementar(string clave): Esta función introduce una nueva clave en el diccionario, si ya esta presente, aumenta su contador en 1.
- vector<string>claves(): Esta función retorna todas las claves presentes en la estructura.
- unsigned int valor(string clave): Esta función retorna el valor asociado en la clave, que se corresponde con la cantidad de veces con las que esa clave se introdujo en el diccionario.
- pair<string,unsigned int>maximo(): Esta función retorna el máximo valor asociado a una clave.
- pair<string,unsigned int>maximoParalelo(unsigned int cantThreads): Esta función retorna el máximo valor asociado a una clave, pero distribuyendo la carga del cómputo entre la cantidad de threads que se toma como parámetro. Se encarga de crear los threads, organizar la estrucura para paralelizar el cálculo y luego de eliminarlos.

2.3. Cargar los archivos

Vamos a proporcionar dos maneras de cargar las tablas de hash:

- int cargarArchivo(HashMapConcurrente &hashMap, string filePath): Carga un archivo solamente.
- void cargarMultiplesArchivos(HashMapConcurrente &hashMap, unsigned int cantThreads, vector<string>filePaths): Carga varios archivos a la vez utilizando la cantidad de threads pasada por parámetro.

3. Enfoque para la resolución del problema

Para resolver el problema vamos a utilizar la libreria estandar de C++ para threading y un esqueleto de código provisto por la catedra. La idea central es lograr un código que sea consistente en cuanto al orden de operaciones pero lockeando la menor cantidad de memoria posible, para asi maximizar la concurrencia.

3.1. Lista Atómica

3.1.1. Insertar

```
1 Function insertar(const T &valor)
2 |   m.lock();
3 |   Nodo *nuevo_nodo = new Nodo(valor);
4 |   nuevo_nodo->_siguiente = _cabeza.load();
5 |   _cabeza.store(nuevo_nodo);
6 |   m.unlock();
7 end
```

Tuvimos que modificar *insertar* para hacer que la lista fuera atómica. Que sea atómica quiere decir que las operaciones sobre ella se realizan de manera indivisible e ininterrumpible, incluso con múltiples threads ejecutándose simultáneamente. Para ello agregamos un mutex a la clase ListaAtomica y lo usamos en *insertar*. Si un programa utiliza esta lista atómica queda protegido de incurrir en condiciones de carrera ya que utilizar lock() al principio y un unlock() luego de agregar el elemento a la lista se asegura que solo un thread entre a la sección crítica a la vez y que tampoco pueda ser interrumpido por el scheduler.

3.2. HashMapConcurrente

3.2.1. Incrementar

Algorithm 1: HashMapConcurrente::incrementar

```
1  Function incrementar(clave)
2  |   bucket ← hashIndex (clave)
3  |   lock mutexBucket[bucket]
4  |   items ← tabla[bucket]
5  |   iteradorLista ← items.crearIt()
6  |   while iteradorLista.haySiguiente() do
7  |   |   if iteradorLista.siguiente().first = clave then
8  |   |   |   iteradorLista.siguiente().second += 1
9  |   |   |   unlock mutexBucket[bucket]
10 |   |   |   return
11 |   |   end
12 |   |   iteradorLista.avanzar()
13 |   end
14 |   items.insertar(*(new hashMapPair(clave, 1)))
15 |   unlock mutexBucket[bucket]
16 end
```

Para este algoritmo, vamos a implementar algo similar a como sería una función clásica de incrementar en un hashmap, con la variación de que vamos a aplicar un lock en el bucket (cada bucket tiene su propio mutex para manipularlo) donde se inserta esta clave hasta insertar la misma. De esta manera logramos que se pueda insertar claves en varios buckets en simultáneo sin condiciones de carrera, ya que se agregan siempre en el mismo orden sin importar la traza.

3.2.2. Claves

Algorithm 2: HashMapConcurrente::claves

```
1  Function claves()
2      claves ← empty vector of strings
3      actual ← empty string
4      mutexClaves.lock()
5      for i in 0 to cantLetras − 1 do
6          mutexBucket[i].lock()
7      end
8      for i in 0 to cantLetras − 1 do
9          items ← tabla[i]
10         iteradorLista ← items.crearIt()
11         while iteradorLista.haySiguiente() do
12             actual ← iteradorLista.siguiente().first
13             claves.push_back(actual)
14             iteradorLista.avanzar()
15         end
16         mutexBucket[i].unlock()
17     end
18     mutexClaves.unlock()
19     return claves
20 end
```

Para el algoritmo de claves, se vuelve más complicado permitir accesos concurrentes a la estructura mientras se esta ejecutando la función, ya que requiere información que esta en toda la estructura. Al principio vamos a aplicar un lock a todos los buckets (cada bucket tiene su propio mutex). Esto es para que no se pueda incrementar cuando llamamos a *claves* ya que si esto pasara podriamos tener distintos resultados dependiendo en la traza. Vamos a ir liberándolos a medida que obtenemos la información de ellos. De esta manera logramos que el valor de retorno son las claves almacenadas en el momento en que se llamó a la función, y no lockeamos por completo la estructura durante la ejecución de la función. Vamos a permitir que este método se ejecute solo de a una vez usando mutexClaves para así evitar la condición de carrera.

3.2.3. Valor

Algorithm 3: HashMapConcurrente::valor

```
1  Function valor(clave)
2      bucket ← hashIndex (clave)
3      mutexBucket[bucket].lock()
4      items ← tabla[bucket]
5      iteradorLista ← items.crearIt()
6      while iteradorLista.haySiguiente() do
7          if iteradorLista.siguiente().first == clave then
8              mutexBucket[bucket].unlock()
9              return iteradorLista.siguiente().second
10         end
11         iteradorLista.avanzar()
12     end
13     return 0
14 end
```

Este algoritmo sigue la misma lógica que el de incrementar. El funcionamiento es bastante simple, lockea solamente un bucket (cada bucket tiene su propio mutex para manipularlo) y lo itera para obtener su valor. Al solo bloquear el bucket del cual queremos obtener la información, no bloqueamos toda la estructura y evitamos al mismo tiempo la condición de carrera ya que, de nuevo, siempre se ejecuta en el mismo orden sin importar la traza y evitamos que incrementar pueda interrumpir para manipular el bucket.

3.2.4. Maximo

Algorithm 4: HashMapConcurrente::maximo

```
1  Function maximo()
2      hashMapPair *max = new hashMapPair()
3      max->second = 0
4      mutexMaximo.lock()
5      for unsigned int i = 0; i <cantLetras; i++ do
6          mutexBucket[i].lock()
7      end
8      for unsigned int index = 0; index <HashMapConcurrente::cantLetras; index++ do
9          for auto it = tabla[index]->crearIt(); it.haySiguiente(); it.avanzar() do
10             if it.siguiente().second >max->second then
11                 max->first = it.siguiente().first
12                 max->second = it.siguiente().second
13             end
14         end
15         mutexBucket[index].unlock()
16     end
17     mutexMaximo.unlock()
18     return *max
19 end
```

Este algoritmo es bastante simple. Itera toda la estructura y se queda con el máximo. Para evitar que dos *maximo* se ejecuten a la vez utilizamos un mutex. Luego aplicamos un lock a todos los buckets uno por uno para que nadie más los pueda cambiar. Esto se hace para evitar que *incrementar* se ejecute concurrentemente con *maximo* ya que llevaría a que la información no sea consistente. Podría pasar el caso donde se incrementan claves que todavía no evaluó la función *maximo* y esto llevaría a devolver un máximo que no lo era ni al momento de llamar al método ni al momento de comenzar la ejecución del mismo. A medida que leemos los buckets, los liberamos para no clausurar toda la estructura mucho tiempo.

3.2.5. Maximo Paralelo

```
1  Struct args_struct:
       // estructura de variables compartidas entre threads
2      int bucketActual
3      ListaAtomica<hashMapPair>*maximosBuckets
4      HashMapConcurrente *context
5  end
6  Function maximoLista(lista)
       // simplemente toma el máximo bucket de la lista
7      max ← new hashMapPair()
8      max.second ← 0
9      for each item in lista do
10         if item.second > max.second then
11             max.first ← item.first
12             max.second ← item.second
13         end
14     end
15     return max
16 end
17 void* maximoThread(arguments)
       // código a ejecutar para cada thread
18     args ← (struct args_struct *)arguments
19     while args.bucketActual < 26 do
20         args.context.mutexMaxCounter.lock()
21         miBucket ← args.bucketActual
22         args.bucketActual++
23         args.context.mutexMaxCounter.unlock()
24         if ¬(miBucket < 26) then
25             break
26         end
27         bucket ← args.context.tabla[miBucket]
28         max ← maximoLista (bucket)
29         args.context.mutexBucket[miBucket].unlock()
30         if ¬max.first.empty() then
31             args.maximosBuckets.insertar(max)
32         end
33     end
34     return nullptr
35 end
```

```
1  Function maximoParalelo(cantThreads)
       // crea las variables compartidas, inicializa los threads, cuando terminan los threads con
          todos los buckets, los cierra y toma el máximo de entre todos los devueltos por los
          threads
2      mutexMaximo.lock()
3      maximosBuckets ← new ListaAtomica<hashMapPair>
4      args ← new args_struct
5      args.bucketActual ← 0
6      args.maximosBuckets ← maximosBuckets
7      args.context ← this
8      threads ← empty vector of pthread_t
9      for i in 0 to cantLetras − 1 do
10         mutexBucket[i].lock()
11     end
12     pthread_create_retry ← 0
13     while threads.size() < cantThreads do
14         if pthread_create_retry > 3 then
15             break
16         end
17         ptid ← new pthread_t
18         if pthread_create(ptid, NULL, maximoThread, (void *)args) = 0 then
19             threads.push_back(ptid)
20             pthread_create_retry ← 0
21         end
22         else
23             pthread_create_retry++
24         end
25     end
26     for each thread in threads do
27         pthread_join(thread, NULL)
28     end
29     max ← maximoLista (maximosBuckets)
30     mutexMaximo.unlock()
31     delete maximosBuckets
32     delete args
33     return max
34 end
```

Máximo paralelo es un algoritmo más complejo. A efectos de claridad vamos a explicar la esencia del algoritmo. Se basa en crear la cantidad de threads pasados por parámetro y en una variable compartida que va contando que buckets se revisaron, el hashMap con el contexto y una lista con los máximos de cada bucket. La idea es que cada thread revisa un bucket y cuando termina de revisarlo, sigue con el siguiente que este disponible para revisar. Este proceso se realiza con todos los buckets y la carga se reparte de manera equitativa entre los threads, ya que mientras haya suficientes buckets sin revisar, la carga se reparte entre todos. Cabe destacar que esta manera es más óptima que la de dividir la carga de manera igualitaria al principio, ya que si por alguna razón un thread tarda mucho tiempo en procesar un bucket, los otros pueden seguir trabajando y hacer el trabajo que este no puede hacer por quedarse colgado. Cuando ya no quedan buckets por revisar, se cierran todos los threads y se toma el máximo de la lista con todos los máximos de cada bucket.

3.3. Cargar Archivos

3.3.1. Cargar Archivo

```
1  Function cargarArchivo(hashMap, filePath)
       // Abre el archivo
2      fstream file
3      int cant = 0
4      string palabraActual
5      file.open(filePath, file.in)
6      if !file.is_open() then
7          cerr ← "Error al abrir el archivo" + filePath
8          return -1
9      end
10     while file → palabraActual do
11         hashMap.incrementar(palabraActual)
12         cant++
13     end
14     if !file.eof() then
15         cerr ← "Error al leer el archivo"
16         file.close()
17         return -1
18     end
19     file.close()
20     return cant
21 end
```

El funcionamiento de cargar archivos es bastante simple. Toma el nombre de un archivo y para cada palabra en el realiza un incrementar en el hashmap. En cuanto a concurrencia, su funcionamiento es correcto porque la función incrementar esta libre de race conditions ya que no se modifica el archivo en ningun momento, por lo tanto no sería posible tener distintos resultados al cargar las palabras del mismo.

3.3.2. Cargar multiples archivos

```
1  Struct args_struct:
       // estructura de variables compartidas
2      vector<string>filePaths
3      HashMapConcurrente *hashMap
4      mutex *mutexFiles
5      int fileActual
6  end
7  void* cargarArchivoStruct(argumentos)
       // modifica las variables compartidas
8      args ← (struct args_struct *)argumentos
9      while args.fileActual < args.filePaths.size() do
10         args.mutexFiles.lock()
11         file ← args.fileActual
12         args.fileActual++
13         args.mutexFiles.unlock()
14         filePath ← args.filePaths[file]
15         cargarArchivo(*args.hashMap, filePath)
16     end
17     return nullptr
18 end
```

```
1  Function cargarMultiplesArchivos(hashMap, cantThreads, filePaths)
       // crea las variables compartidas, los threads pedidos y carga los archivos
2      threads ← empty vector of pthread_t
3      mutexFiles ← new mutex
4      args ← new struct args_struct
5      args.mutexFiles ← &mutexFiles
6      args.fileActual ← 0
7      args.filePaths ← filePaths
8      args.hashMap ← &hashMap
9      pthread_create_retry ← 0
10     while threads.size() < cantThreads do
11         if pthread_create_retry > 3 then
12             break
13         end
14         ptid ← new pthread_t
15         if pthread_create(ptid, NULL, cargarArchivoStruct, (void *)args) = 0 then
16             threads.push_back(ptid)
17             pthread_create_retry ← 0
18         end
19         else
20             pthread_create_retry++
21         end
22     end
23     for each thread in threads do
24         pthread_join(thread, NULL)
25     end
26 end
```

La idea de este algoritmo es similar a la de máximo concurrente. Se crean la cantidad threads que se recibe por parámetro y se le asigna a cada uno un archivo. Hay una variable compartida que es el índice del último archivo que un thread esta procesando, por lo que cuando un thread termina, revisa esa variable y sigue con el archivo siguiente. También comparten el hashMapConcurrente donde cargan las palabras. Al funcionar casi igual a maximoParalelo, tampoco hay problemas a la hora de la sincronización o condición de carrera.

4. Hipótesis

Antes de pasar a experimentar, vamos a plantear cómo creemos que se debería comportar cada algoritmo que programamos para algunos casos específicos que consideramos pertinentes.

Carga de Archivos No debería haber diferencia en performance al variar la cantidad de threads si usaramos el algoritmo de CargarMultiplesArchivos con un único archivo, pero ¿Qué pasaría si variamos la cantidad de threads al cargar más de un archivo? En este caso creemos que al aumentar la cantidad threads va a mejorar el tiempo de carga considerablemente hasta llegar a tener la misma cantidad de threads que archivos por cargar. Si la cantidad de threads supera la cantidad de archivos, creemos que el tiempo de ejecución dejaría de reducirse y en su lugar se estancaría.

Máximo Paralelo Pensamos en dos casos distintos para los cuales podremos analizar los tiempos de ejecución en función de la cantidad de threads utilizados.

En primer lugar, el caso en el cual tenemos palabras que comienzan con todas las letras, de la 'A' a la 'Z', con una distribución similar. Es decir, que hay una cantidad similar de palabras que comienzan con cada letra. Para este caso creemos que el tiempo de ejecución va a mejorar constantemente a medida que agregamos threads, hasta llegar a 26 threads, que es la cantidad total de letras que admitimos en nuestro hash map, y luego se estancaría la performance. Esto es porque, en el caso de tener 26 threads, tendríamos un thread calculando el máximo de cada bucket.

En segundo lugar, planteamos casos donde sólo cargamos palabras que comienzan con una o dos letras distintas, de forma que queden cargados sólo uno o dos de los buckets del hash map. Éstos casos sirven para ver efectivamente el momento en el cual los threads dejan de mejorar la performance. Para el primer caso, donde sólo cargamos un bucket, creemos que usar más de un thread no tiene sentido, y no mejorará el tiempo de ejecucion. Para el caso donde cargamos dos buckets, suponemos que usando más de dos threads no encontraremos mejoras de tiempos de ejecución.

5. Resultados

Dataset Para la siguiente etapa de experimentación, vamos a usar un dataset que contiene 394000 palabras distintas, aproximadamente. El mismo lo vamos a cargar como un único archivo en algunos casos, o dividido en 8 archivos distintos en otros casos.

Especificaciones Los experimentos fueron realizados en una computadora con las siguientes especificaciones: Intel® Core™ i5-10210U CPU @ 1.60GHz × 8 (soporta hasta 8 hilos), 8GB de RAM, SSD.

Listamos algunos resultados obtenidos al cargar archivos con 1 thread o múltiples threads(cargarArchivo y cargarMultiplesArchivos):

- 1 archivo 394k palabras distintas. Tiempo de carga: 95.4799
- 8 archivos de 50k palabras usando CargarMultiplesArchivos con 8 threads. Tiempo de carga: 26.8235
- 1 archivo, 24k palabras que empiezan con A. Tiempo de carga: 7.04522
- 1 archivo, 42k palabras que empiezan con A o B. Tiempo de carga: 10.6145

Luego otros resultados representados en gráficos:

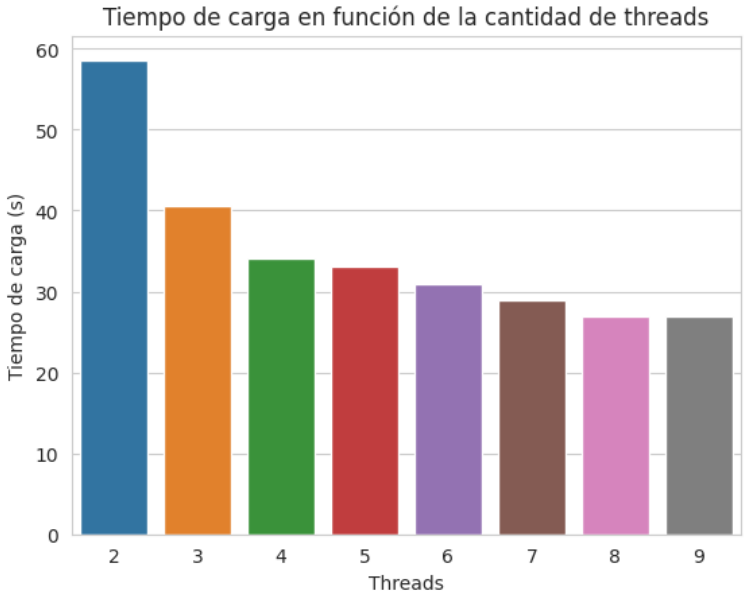

Figura 3: cargarMultiplesArchivos con 8 archivos y variando los threads

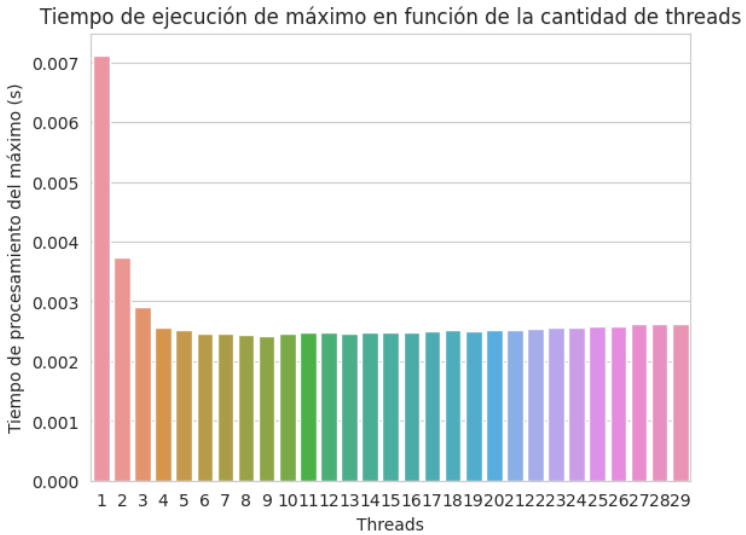

Figura 4: Cómputo del máximo en promedio de 100 casos variando la cantidad de threads

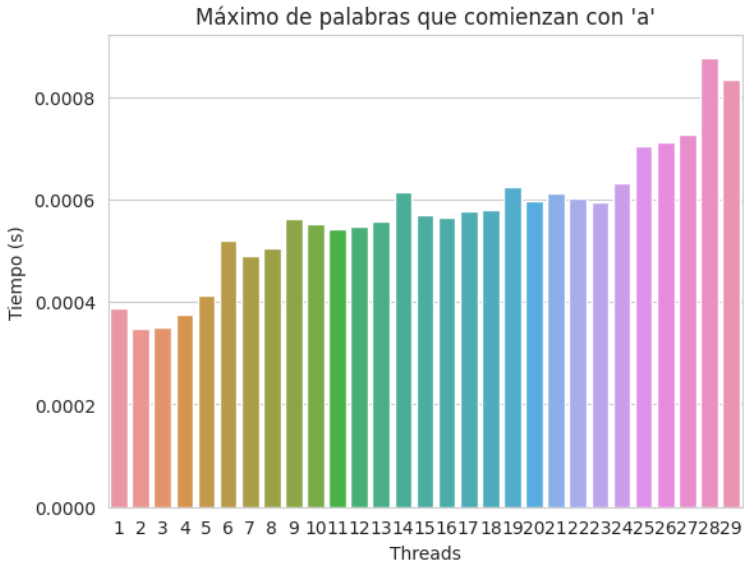

Figura 5: Cómputo del máximo solo teniendo en cuenta palabras que comienzan con la letra A variando la cantidad de threads

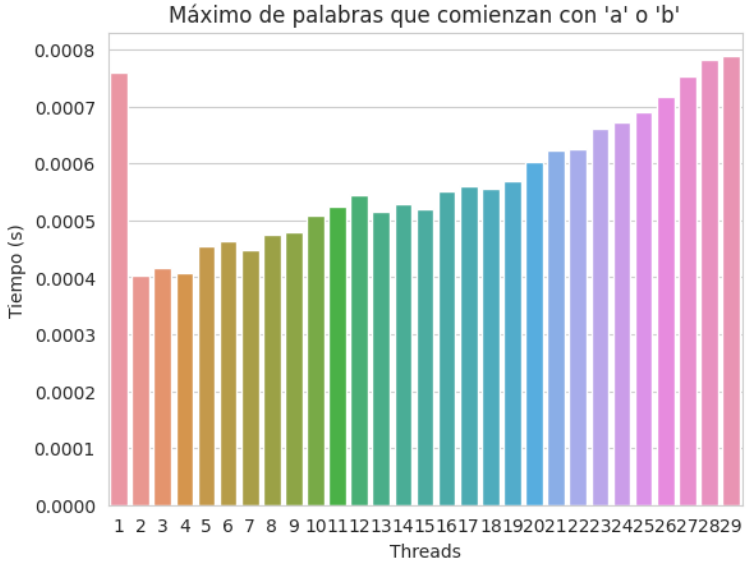

Figura 6: Cómputo del máximo solo teniendo en cuenta palabras que comienzan con las letras A o B variando la cantidad de threads

6. Análisis de resultados y síntesis

En el apartado de tiempo de carga, claramente por los resultados enlistados, para cargar múltiples archivos conviene utilizar threads, para un solo archivo no cambia. Pero a la hora de decidir cuantos threads utilizar para cargar múltiples archivos, llegamos a la conclusión por nuestros resultados, que como se ve en la figura 3 que a partir de 4 threads ya no sirve tener más threads. La mejora no es grande al agregar más threads ya que cambia lo mismo al pasar de 3 a 4 threads que de 4 a 9 threads. Eso si, de 1 thread a 4 aumenta considerablemente el rendimiento, tal como esperabamos.

Luego, en el tiempo de cómputo del máximo vemos que sucede algo similar. Como se puede ver en la figura 4, a partir del 4to thread ya la mejora no es sustancial, es más, a la larga empeora el rendimiento. Recordemos que esto es el promedio de 100 casos sobre la búsqueda del máximo.

A continuación vemos dos experimentos similares en cuanto a la modalidad. En ambos casos buscamos el máximo y aumentamos la cantidad de threads para encontrarlo. En el primero, como se ve en la figura 5, buscamos el máximo entre las palabras que comiencen con A, que es lo mismo que solo usar un thread. Esperabamos que el tiempo se estanque o empeore levemente a medida que se creaban más threads sin ningún propósito pero nos llevamos la sorpresa de que empeora considerablemente el rendimiento. Esto nos muestra que crear threads sin ninguna utilidad es bastante malo.

Luego, probamos para el caso de revisar solo 2 buckets, como se ve en la figura 6, tomando solo las palabras que comienzan con A o B y buscar el máximo entre ellas. El resultado es análogo al experimento anterior y luego de la mejora al agregar un segundo thread, el rendimiento cae y empeora hasta pasar incluso el solo tener un único thread.

7. Conclusiones

Para concluir, el uso de varios threads puede ser una estrategia efectiva para disminuir significativamente el tiempo de ejecución en un programa. Sin embargo, es importante tener en cuenta algunas consideraciones adicionales antes de decidir cuántos threads utilizar. Una de estas consideraciones es entender las características de la computadora en la que se está ejecutando el programa, ya que la capacidad del procesador y la cantidad de núcleos e hilos disponibles puede influir en el rendimiento final. Otra consideración es entender el programa y calcular mediante pruebas y distintos análisis cuando deja de ser óptimo agregar threads. A veces puede no convenir destinar más recursos y complejizar el código para obtener solo una pequeña mejora.