DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Threading

Sistemas Operativos
Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Nicolás Bukovits	546/14	nicobuk@gmail.com
Julián Zylber	21/18	jzylber@dc.uba.ar

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 **3**

2. Lista Atómica **3**

3. Hash Map Concurrente: Incrementar, Claves y Valor **4**

4. Hash Map Concurrente: Máximo **5**

5. Cargar Archivos **5**

6. Experimentación **6**
6.1. Experimentación sobre función máximo 6
6.2. Experimentación sobre la carga de archivos 8

7. Conclusión **9**

1. Introducción

El objetivo de este trabajo es estudiar una de las principales características de los sistemas operativos: la gestión de la concurrencia. Para ello se utilizaran los threads, que nos proveen de varios hilos de ejecución concurrentes dentro de un mismo programa. En particular, se utilizara la interfaz para threads que forma parte de la biblioteca estándar de `C++`.

Se realizara una implementación de una estructura de datos a la que denominaremos `HashMapConcurrente`. Esta estructura de datos es una tabla de hash abierta que utiliza listas enlazadas para gestionar las colisiones. Como hash, se toma la primera letra de la palabra a ingresar. Se trata de un diccionario cuyas claves son strings y sus valores son enteros no negativos. La finalidad de esta estructura es poder contabilizar la cantidad de apariciones de palabras (obtenidas de procesar distintos archivos de texto) guardando la palabra como clave y su cantidad de apariciones como el valor. A su vez se implementara una función que calcule la palabra con la mayor cantidad de apariciones (el máximo). Las implementaciones tendrán en cuenta que algunas operaciones se pueden ejecutar concurrentemente por lo que garantizaran que las mismas estén libres de condiciones de carrera y deadlocks.

Finalmente se realizarán diversos experimentos, donde se ejecutaran las funciones implementadas variando el nivel de concurrencia (la cantidad de threads) como así también las instancias sobre las cuales se calculan la cantidad de apariciones de palabras (los archivos de texto).

2. Lista Atómica

La lista enlazada es utilizada para manejar las colisiones en el `HashMapConcurrente`. El `HashMapConcurrente` es un vector de 26 posiciones y la función de hash devuelve una posición de este vector en base a la primer letra de la palabra a la que se le aplica la función. En cada posición del vector hay una lista. En esa lista por lo tanto se encuentran todas las palabras que comienzan con las misma letra. La lista es atómica, lo que quiere decir que cada inserción en la misma se realiza de manera ininterrumpible y por lo tanto se encuentra libre de condiciones de carrera. A continuación se muestra el pseudocódigo de la función de insertar en la lista atómica:

Algorithm 1 Función de insertar en lista atómica.

```
1: function insertar(valor)
2:     nuevaCabeza ← new Nodo(valor)
3:     nuevaCabeza.siguiente ← cabeza.load()
4:     while !cabeza.compare_exchange_weak(nuevaCabeza.siguiente, nuevaCabeza) do
```

La función mencionada en el Algoritmo 1 garantiza la propiedad de atomicidad debido a las siguientes razones: La variable cabeza es una variable atómica y para recuperar su valor se llama a la función load que nos garantiza que se devuelve de manera atómica el valor de la cabeza actualmente. Luego utilizamos la función `compare_exchange_weak` que realiza lo siguiente de manera atómica también: Compara el valor del primer argumento (`nuevaCabeza.siguiente`) con cabeza y si es el mismo valor devuelve verdadero y actualiza el valor de la variable atómica cabeza con el valor del segundo argumento (`nuevaCabeza`). En caso de que en la comparación los elementos sea distintos, entonces se devuelve falso y no se realiza el cambio del valor de la variable pero se realiza el load en el primer argumento (`nuevaCabeza.siguiente`). Debido a este funcionamiento podemos asegurar que en caso de que otro proceso o thread haya cambiado la cabeza antes de haber actualizado la misma, la comparación va a retornar falso, se va actualizar el valor de `nuevaCabeza.siguiente` con la cabeza nueva (es decir con el valor que insertó el otro thread) y volverá a intentar a agregar el nuevo Nodo en la cabeza, ya que si el método devolvió `false`, quedara en el ciclo, hasta que pueda realizar el intercambio.

En conclusión, si otro thread o proceso le "ganó" al proceso que esta intentando poner como cabeza su nuevo nodo, se quedará intentando hasta que pueda insertarlo. Cuando el esperado sea igual al actual en la cabeza, lo puede insertar correctamente en la lista al principio. Se puede

observar que si otros threads fueron insertando en el medio, el thread este insertara su nodo adelante de todos los nodos insertados. Notar que con esta implementación no puede ocurrir que durante la inserción, la lista quede en un estado con una cabeza que no tenga un siguiente (y que quedaría inconsistente con algún otro proceso que la este leyendo en ese momento al mismo tiempo, ya que vería solo la cabeza) y que tampoco puede ocurrir que se pierda algún nodo en la inserción (es decir que se pise la cabeza con otra y que la lista pierda algún elemento).

3. Hash Map Concurrente: Incrementar, Claves y Valor

La función incrementar del `HashMapConcurrente` recibe una clave como parámetro y debe insertar en el diccionario la clave con el valor de cantidad de apariciones igual a 1 (en caso de que no exista) o incrementar en uno el valor de la clave (en caso de que exista). Como puede haber varios threads intentando incrementar al mismo tiempo las claves se debe garantizar que esta operación esté libre de condiciones de carrera y sea consistente.

Para manejar la contención se utilizaron 26 `mutex`, definidos en un array. La idea es que haya un `mutex` por cada posición del vector sobre el cual esta implementado el `HashMapConcurrente` y de esta manera solo habrá contención en caso de colisión de hash. La función primero obtiene el valor de la posición correspondiente al vector, utilizando la función de hash, y luego toma el `mutex` correspondiente a esa posición. Itera todos los elementos de la lista de dicha posición y si encuentra la clave aumenta en uno la cantidad de apariciones. Si no lo encuentra. entonces utiliza la función de insertar para agregarlo a la lista. Una vez que terminó de realizar estas comprobaciones y operaciones, libera el `mutex` que tomó. De esta manera si otro thread esta intentando incrementar una clave pero que pertenece a otro de los buckets del `HashMapConcurrente`, podrá hacerlo sin problemas y sin esperar a que el primer thread termine. Solo en caso de que haya dos o mas threads queriendo incrementar claves cuyas letras iniciales sean las mismas (teniendo el mismo valor del hash), solo uno podrá hacerlo a la vez y los demás esperaran su turno. Una vez que uno termino, otro tomara el `mutex` y lo incrementara y así sucesivamente. El siguiente es el pseudocódigo de la función incrementar:

Algorithm 2 Funcion de incrementar en el HashMapConcurrente.

```
1:  function incrementar(clave)
2:      hash ← hashIndex(clave)
3:      mutexBuckets[hash].lock()
4:      found ← false
5:      for par ∈ tabla[hash] do
6:          if par.first == clave then
7:              par.second + +
8:              found ← true
9:              break
10:     if !found then
11:         tabla[hash].insertar((clave, 1))
12:     mutexBuckets[hash].unlock()
```

Las funciones de claves y valor no tienen que garantizar un correcto funcionamiento en caso que se ejecute de manera concurrente con la función de incrementar por lo que la implementación de ambas es sencilla y no usa ninguna primitiva de sincronización. Simplemente claves itera todas las posiciones del vector (los 26 buckets) y por cada uno itera también la lista correspondiente y va agregando las claves a un vector. Finalmente se devuelve dicho vector. La función de valor, que recibe por parámetro la clave, aplica la función de hash para obtener el bucket que debe revisar y recorre la lista correspondiente. En caso de encontrar la clave, devuelve su cantidad de apariciones, en caso contrario, retorna 0.

4. Hash Map Concurrente: Máximo

La función máximo del `HashMapConcurrente` se encarga de devolver el par (clave, valor) que representa la clave con la mayor cantidad de apariciones, es decir con el valor mas alto. La misma se puede ejecutar concurrentemente con la función incrementar, lo cual podría generar inconsistencias. Por ejemplo, puede suceder que cuando máximo haya leído los valores de las primeras `i` posiciones de la tabla, la función incrementar aumente una clave en alguna posición anterior a la iésima y el valor del máximo cambie. La función máximo nunca se va a enterar de este cambio y puede retornar un valor de un máximo no consistente con ninguna instancia del HashMap.

La forma en que se solucionó esto es mediante los mismos 26 `mutex` que se usaron en incrementar. La idea es la siguiente: Se empieza a recorrer la primer lista en la posición 0 y se toma el `mutex` de la primera posición. Luego cuando se avanza a analizar la siguiente lista se toma el `mutex` de la siguiente posición, sin liberar el anterior. Y así sucesivamente hasta llegar a la ultima posición donde van a estar todos los `mutex` tomados. Una vez que se analizaron todas las posiciones y se encontró al máximo se liberan todos los `mutex` juntos.

La idea es que esta implementación devuelve el valor del máximo al momento de **finalizar** la ejecución de la función. Es decir, no se van permitiendo cambios en las posiciones anteriores, solo la función incrementar puede ir incrementando concurrentemente posiciones que la función máximo todavía no reviso. Osea, puede haber cambios solo en lugares a los que el máximo todavía no tuvo en cuenta. Solo cuando se revisaron todas, va a haber un instante pequeño donde estén todos los `mutex` tomados, pero son liberados inmediatamente.

La implementación de `maximoParelelo` tiene que resolver lo mismo que la función máximo pero usando la cantidad de threads especificada por parámetro. La estrategia utilizada para repartir el trabajo entre los n threads fue la siguiente: Se crean n threads. Estos tienen las siguientes variables compartidas: un par (clave,valor) llamado `max`, un `mutex` llamado `mutexMaximo` y una variable atómica del tipo entera denominada `indiceARevisar`. Cada thread realiza lo siguiente indefinidamente: aumenta en uno atómicamente la variable `indiceARevisar`. Esta variable representa el índice dentro de la tabla que va a revisar el thread. Si el valor es igual o mayor a 26 (la cantidad de entradas de la tabla), entonces el thread termina su ejecución, por que significa que ya las 26 posiciones fueron tomadas por threads. En caso contrario va a revisar lo que la función `fetch_add(1)` devolvió, que es el índice que le corresponde revisar. Para comparar los valores de la fila que va revisar con el máximo encontrado hasta el momento, tiene que también tomar el `mutex` correspondiente al bucket (recordar que máximo se podía ejecutar concurrentemente con incrementar). Una vez que encontró el valor mas alto dentro de su fila, toma el `mutex` de la variable `mutexMaximo`. Dicho `mutex` se usa para garantizar la exclusión mutua en la comparación y actualización del máximo de la fila del thread con el máximo global (la variable `max`). Se actualiza si corresponde la variable `max` y luego inmediatamente libera el `mutexMaximo`. De la misma forma que en la función de máximo, los 26 `mutex` de los buckets (filas) no se liberan hasta que se analizan todas las filas. Una vez que se analizaron todas las filas se liberan todos los 26 `mutex`.

5. Cargar Archivos

La función `cargarArchivo` es la encargada de leer el archivo de texto que se le pasa como parámetro e insertar en el `HashMapConcurrente` las claves de las palabras encontradas con su respectiva cantidad de apariciones. Para implementar esta función no fue necesario tomar ningún recaudo especial, desde el punto de vista de la sincronización. Como se utiliza la función de incrementar explicada anteriormente, que se demostró que está libre de condiciones de carrera. Por lo tanto, simplemente hay que usar esa función por cada palabra del archivo y no es necesario hacer nada más. La función de incrementar es la que se encarga de evitar inconsistencias.

Para la función de `cargarMultiplesArchivos`, que recibe además de los archivos de texto, la cantidad de threads entre los cuales tiene que repartir el trabajo, se tomo solo el siguiente recaudo. Debido a que se debe garantizar que cada thread lea un archivo a la vez se utiliza una variable atómica entera denominada `indiceARevisar`. Cada uno de los n threads entonces

aumenta atómicamente esa variable en uno y el índice que devuelve la función de `fetch_add(1)` es el índice en el vector de archivos de texto que va a cargar el thread. Por lo tanto simplemente el thread llama a la función de `cargarArchivo` mencionada anteriormente con el archivo que le tocó.

6. Experimentación

Las características técnicas de la computadora donde se corrieron los experimentos son las siguientes:

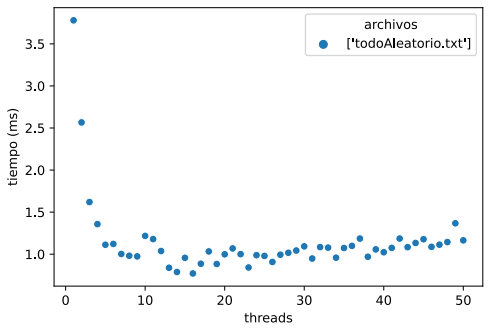
- **Sistema operativo**: macOS Big Sur version 11.3.1
- **Procesador**: Intel Core i9 de 8 Cores con Hyper Threading (16 threads) a 2,3 GHz
- **Memoria**: 16 GB 2667 MHz DDR4

La experimentación se realizó en dos partes. La primera consistió en medir y analizar por separado la performance del proceso de carga de los archivos en el `HashMapConcurrente`. La segunda etapa consistió en lo mismo pero para el proceso de computar el máximo. En ambas etapas se utilizaron las variables de la cantidad de threads y el tipo de archivo. Cada experimento se corrió 5 veces y se tomo la mediana del tiempo para una mayor fidelidad. Las instancias de los tipos de archivos se enumeran a continuación:

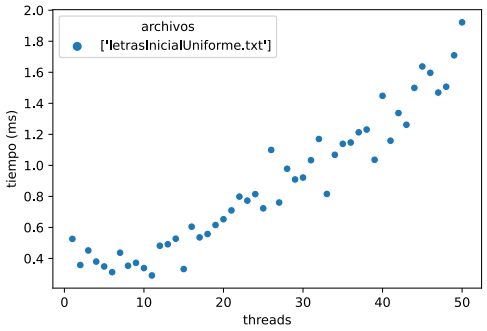
- **todosIguales**: Son archivos donde todas las palabras son iguales. Todas las palabras son "linux".
- **primeraLetraIgual**: Son archivos donde la primera letra de todas las palabras son iguales, el resto son distintas. Las palabras tienen todas una longitud de 11 caracteres. Todas empiezan con "l".
- **letrasInicialUniforme**: Archivos donde se distribuye de manera casi equitativa la cantidad de palabras con la misma letra inicial de todas las posibles letras iniciales. Es decir, $\forall x \in \text{letras} \Rightarrow \left\lfloor \frac{\text{total de palabras}}{26} \right\rfloor \leq \text{palabras que empiezan con } x \leq \left\lceil \frac{\text{total de palabras}}{26} \right\rceil$. Las palabras tienen una longitud de 11 caracteres.
- **todoAleatorio**: Archivos en donde todas las palabras se generaron con letras aleatorias. Las palabras tienen una longitud de 11 caracteres.
- **dummyText**: Archivo con palabras en inglés al azar.

6.1. Experimentación sobre función máximo

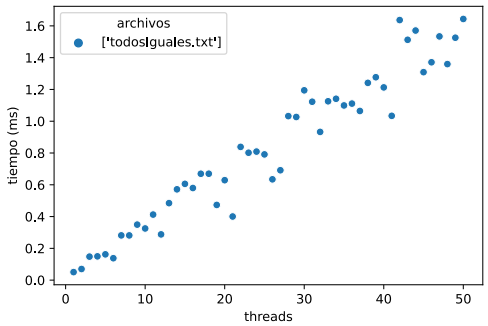
Se ejecutó el algoritmo de `maximoConcurrente` y se midieron los tiempos en *ms*, variando la cantidad de threads, desde 1 hasta 50. Las mismas ejecuciones se realizaron sobre los distintos tipos de instancias: Se utilizó el tipo de archivo *todosIguales* con una cantidad de palabras igual a 500000; el tipo de archivo *primeraLetraIgual* con 3000 palabras; el tipo de archivo *todoAleatorio* con 200000 palabras; y el tipo de archivo *letrasInicialUniforme* con 30000 palabras. La hipótesis es que para los tipos de archivos *todoAleatorio* y *letrasInicialUniforme*, la performance mejore al aumentar la cantidad de threads, hasta llegar hasta los 26. Cuando se supera esa cantidad de threads no se debería notar mejora en la performance ya que habrá threads que simplemente se crearán y finalizaran rápidamente ya que no tienen trabajo que hacer. Para los otros tipos de archivos es esperable que la performance no cambie, ya que habrá solo un bucket en uso, los demás van a estar vacíos, por lo que los threads no tendrán trabajo para hacer. Los resultados de los experimentos son los siguientes:



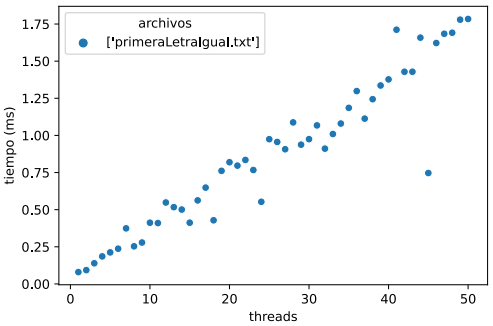
(a) Tiempo de ejecución de máximo sobre threads



(b) Tiempo de ejecución de máximo sobre threads)



(c) Tiempo de ejecución de máximo sobre threads

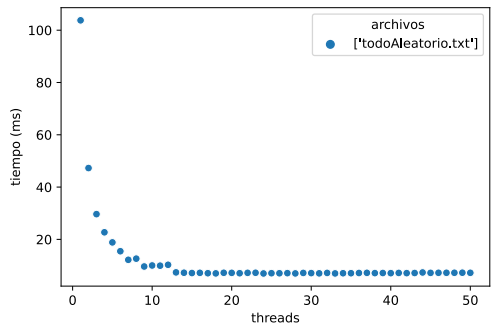


(d) Tiempo de ejecución de máximo sobre threads)

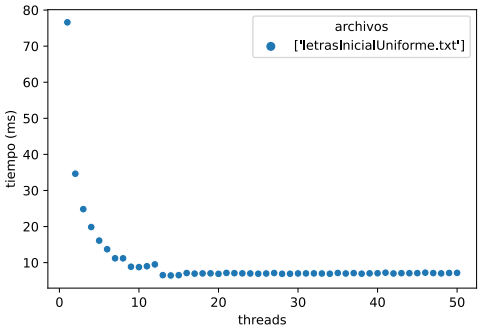
Figura 1: Comparación de máximo con distintos tipos de archivos y threads

De la Figura 1 se pueden obtener las siguientes conclusiones. Para el caso de la instancia de *todoAleatorio* se puede ver que la hipótesis planteada es correcta, ya que a medida que se aumenta la cantidad de threads, el rendimiento tiende a mejorar, hasta atascarse en un punto en el cual ya no mejora mas por mas que aumente la cantidad de threads. El resultado interesante es que para el tipo de instancia de *letrasInicialUniforme* se esperaba un comportamiento similar pero no fue lo que ocurrió. La explicación a este hecho puede ser que la cantidad de palabras tal vez es poco significativa (este tipo de instancia fue corrido con un total de 30000 palabras), por lo que la sobrecarga de crear los threads insume un tiempo considerable mientras que el tiempo que tarda en calcular el máximo es muy pequeño (notar que esta por debajo de los *2 ms*). Para los casos de *todosIguales* y *primeraLetraIgual* se esperaba que no cambie mucho el rendimiento pero sin embargo se aprecia que empeora el tiempo de computo al aumentar la cantidad de threads. La causa de este fenómeno puede ser la misma que la del anterior; se trata de tiempos de menos de *2 ms*.

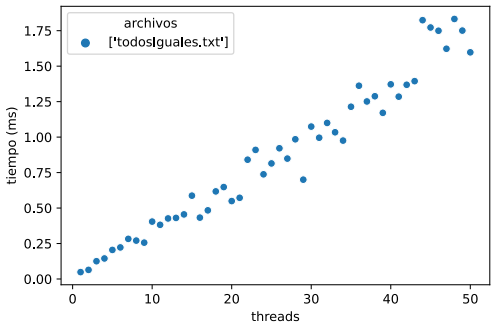
Por lo tanto, dados los resultados obtenidos en esta primera instancia se decidió correr los experimentos nuevamente sobre los mismos tipos de instancias pero esta vez con un tamaño mucho mas considerable de palabras: 1500000 en cada una de las instancias. La nueva hipótesis es que dado que esta cantidad de palabras es mucho mayor, se pueda apreciar mejor el comportamiento esperado ya que calcular el máximo va a tardar mas milisegundos y el overhead de crear el thread no consume mucho tiempo con respecto a calcular el máximo. Por lo tanto se espera que con estos nuevos experimentos el tipo de instancia de *letrasInicialUniforme* se comporte de manera similar al *todoAleatorio*. Es decir, que mejoren los tiempos de cómputo al aumentar los threads, hasta llegar a un limite donde no mejora mas. Ese límite debería estar cercano a los 26 threads. Y se espera que el rendimiento no cambie para el caso de *primeraLetraIgual* y solo tiene sentido que llegue a empeorar para el caso de *todosIguales*, donde hay una sola palabra repetida 1500000 veces. Los resultados de los nuevos experimentos son los siguientes:



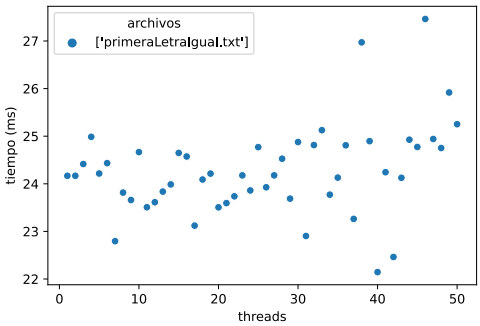
(a) Tiempo de ejecución de máximo sobre threads



(b) Tiempo de ejecución de máximo sobre threads)



(c) Tiempo de ejecución de máximo sobre threads)



(d) Tiempo de ejecución de máximo sobre threads

Figura 2: Comparación de máximo con distintos tipos de archivos y threads

De los últimos resultados se pueden comprobar las hipótesis. Los casos de *todoAleatorio* y *letraInicialUniforme* mejoran su tiempo de cómputo al aumentar los threads, llegando a un límite donde no mejoran mas. Se puede ver en Figura 2 que es alrededor de los 20 threads. A partir de ese momento los cambios en los tiempos de computo no son significativos.

Para el caso de *primeraLetraIgual* se puede ver que no hay una tendencia de mejorar el tiempo de computo al aumentar la cantidad de threads. Esto tiene sentido ya que hay solo un bucket en uso. Se puede ver que los tiempos fluctúan poco dentro del rango de los 27 a los 23ms.

Por último para el caso de una única palabra con muchas repeticiones, el rendimiento empeora al aumentar los threads, ya que no se puede paralelizar porque hay un solo elemento para recorrer. El crear threads solo introduce una sobrecarga innecesaria. El crecimiento por la curva parece ser lineal.

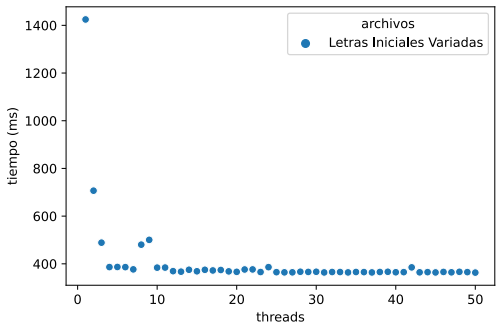
6.2. Experimentación sobre la carga de archivos

Para evaluar el impacto de los threads en la carga de archivos, se probaron cargando 26 archivos (26 ya que por las características del `insertar`, como máximo pueden ejecutar 26 insertar de forma concurrente) de 2000 palabras cada uno. Se probaron 3 tipos de instancias:

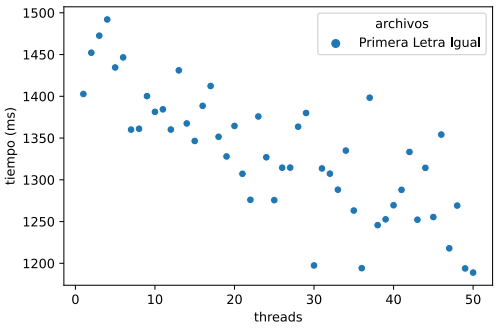
- **Todos los archivos con palabras con letra inicial distinta:** Esperamos que este sea el mejor caso, ya que cada thread usaría una única letra y no tendría que esperar a otro thread. Los tiempos deberían mejorar hasta los 26 threads, y luego estancarse.
- **Todos los archivos con palabras con la misma letra inicial:** Esperamos que este sea el peor caso, ya que todos compiten por el mismo recurso de insertar en el bucket del hash correspondiente a esa letra. Los tiempos deberían ser constantes, los threads no ayudan.

- **Todos los archivos con palabras al azar en inglés**[1]**:** Esperamos que sea un caso promedio, mejor que el peor caso, pero peor que el mejor. Esperamos una reducción de los tiempos hasta los 26 threads, y luego estancarse.

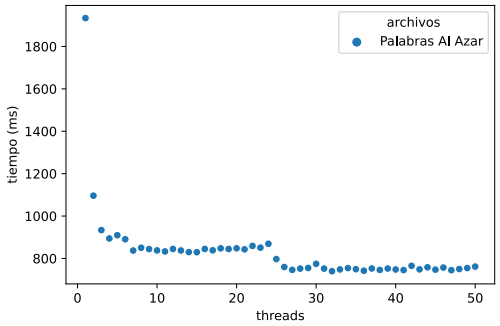
A continuación los resultados de la experimentación:



(a) Tiempo de ejecución de máximo sobre threads



(b) Tiempo de ejecución de máximo sobre threads)



(c) Tiempo de ejecución de máximo sobre threads)

Figura 3: Comparación de Cargar Archivos con distintos tipos de archivos y threads

Como podemos ver en la Figura 3, nuestras hipótesis se comprueban. El peor caso oscila en tiempos muy parecidos, parece una leve reducción a mayor threads. Si uno mira con detenimiento, la escala va de 1500 a 1200ms, demostrando que el tiempo no varía demasiado, es más ruido que otra cosa. Se mejora un poco el tiempo al distribuir las cargas de archivos, pero como todos esperan el recurso limitado, no es sustancial esta mejora.

El mejor caso, a pesar de que mejora hasta estancarse en los 26, los cambios son drásticos hasta 4 threads, donde a partir de ahí la mejora es poco significativa. Suponemos que como nunca hay colisiones en el hash, con pocos threads ya se mitiga el retraso por entrada/salida.

Por último, el caso promedio, podemos ver que también tenemos una reducción drástica con pocos threads, y luego se estanca, con una última mejoría significativa alrededor de los 24 threads. Podemos suponer que pocos threads mitigamos el efecto entrada salida, y hay algunas pocas colisiones al insertar, que ya con ≥ 24 threads no se ven casi.

7. Conclusión

En este trabajo se resolvieron algunos problemas de gestión de concurrencia con threads en una estructura propuesta para contabilizar cantidad de apariciones de palabras en archivos de

[1]No es completamente promedio si pensamos en un texto cohesivo, deberíamos tomar en cuenta la distribución promedio de las palabras en inglés

texto. Principalmente el enfoque estuvo en implementar los procesos de carga de las palabras en la estructura y el calculo del máximo de manera concurrente.

De la experimentación y análisis se puede concluir que para el caso de la función de máximo, si la cantidad de palabras no es muy significativa, no tiene sentido calcularlo con mas de un thread, de hecho la performance puede empeorar ligeramente por la sobrecarga de crear los threads. La mejor opción en esos casos, incluso cuando las palabras están repartidas equitativamente en la estructura propuesta es calcularlo con un solo thread. Solo cuando la cantidad de palabras cargadas en la estructura es bastante significativa se puede apreciar una mejora considerable en el computo del máximo. Igualmente dicha mejora tiene un limite, a partir del cual no tiene sentido crear mas threads para el calculo. Dicho limite en este caso en particular se estableció alrededor de los 20 threads. Igualmente si se trata de que las palabras se repiten mucho, es decir tienen una gran cantidad de apariciones tampoco conviene usar muchos threads. El uso de threads para máximo conviene cuando hay muchas palabras distintas y estas están repartidas de manera lo mas equitativa posible en la estructura.

En el caso de la función cargar archivos, sin demasiadas palabras ya pocos threads generan mejores sustanciales en el tiempo de carga. En un texto promedio, donde difícilmente todas las palabras empiecen con la misma letra, ya con pocos threads se consiguen mejoras significativas de tiempo ya que puede ejecutar otro thread mientras se realiza E/S. Con pocos threads ya es suficiente para buenos tiempos, y si se quieren mejorar aún más, se pueden llegar a 24 threads donde las colisiones en el hash son mucho mas infrecuentes.

Como cierre del trabajo, podemos concluir que los threads son una excelente herramienta para optimizar trabajo que puede realizarse de manera concurrente, especialmente relacionado con entrada y salida. Cargar palabras y sacar un máximo de una sencilla tabla de hash mejora su rendimiento sustancialmente al utilizar threads.