DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico

## Implementacion de Hash Map Concurrente

Sistemas Operativos

Segundo Cuatrimestre de 2021


| Integrante | LU | Correo electrónico |
|---|---|---|
| Lucas Iván Kruger | 799/19 | Lucaskruger10@gmail.com |
| Marco Sánchez Sorondo | 708/19 | msorondo@live.com.ar |
| Sebastián Cantini Budden | 576/19 | sebascantini@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 el siguiente informe vamos a analizar la implementacion de una estructura de datos llamada HashMapConcurrente en C++, la cual es una tabla de hash abierta que gestiona las colisiones utilizando listas enlazadas. Su interfaz de uso es la de un map o diccionario, cuyas claves serian strings y sus valores, enteros no negativos. La idea es poder aplicar esta estructura para procesar archivos de texto contabilizando la cantidad de apariciones de palabras (las claves seran las palabras y los valores, su cantidad de apariciones). Nuestro principal objetivo es estudiar la gestion de concurrencia en esta estructura, y comparar los resultados obtenidos en diferentes casos (como el uso de multiples threads). El analisis se llevara acabo mediante el uso del conocimiento adquirido en clase y con la ejecucion de diferentes experimentos.

# 2. Resumen del Problema Planteado:

Los procesadores proveen a sus usuarios la posiblidad no sólo de ejecutar varias tareas al mismo tiempo mediante distintos núcleos, sino de tambien dedicar un mismo núcleo a múltiples tareas. Hacer uso de éstos recursos implica que en la práctica las computadoras se vuelvan cada vez mas rápidas y capaces de realizar múltiples tareas al mismo tiempo. Sin embargo, el uso de múltiples 'threads' que comparten los mismos recursos y variables puede generar conflictos de uso de estas generando condiciones de carrera, corriéndose así el riesgo de obtener resultados incorrectos o instancias inconsistentes de la estructura. Nos proponemos explotar éstas capacidades para aumentar la velocidad de uso de la estructura de datos 'Hash Map' y obtener asi una estructura más veloz y eficiente pero que al mismo tiempo no sea capaz de devolver resultados erróneos.

# 3. Diseño de la Solución:

## 3.1. Preguntas a responder

### 3.1.1. ¿Qué significa que la lista sea atómica? Si un programa utiliza esta lista atómica ¿queda protegido de incurrir en condiciones de carrera? ¿Cómo hace su implementación de insertar para cumplir la propiedad de atomicidad?

Que la lista sea atomica significa que las operaciones que efectuemos sobre ella deben indivisibles dentro de su ejecución.

Una variable de tipo atómica, en general, no permite la ocurrencia de condiciones de carrera al modificársela. Esto quiere decir que al aplicarle funciones o al observar sus valores, no se producen resultados inconsistentes. Particularmente para la lista concurrente, significa que no puede darse la situación en que cuando múltiples procesos realizan una insercion sobre ella, alguno de los elementos no sea insertado, o se rompa con la estructura de una lista (por ejemplo que hayan dos o mas nodos que apunten al mismo nodo siguiente, o que la lista no acabe en nullptr).

La situación 'borde' en este caso sería la siguiente... Sea la lista $l = a \hookrightarrow b \hookrightarrow c$.

Sean los **P_1, P_2, ..., P_n** procesos que intentan insertar sobre la lista. Supongamos que el proceso **P_1** intenta insertar el nodo 'x' y al terminar de ejecutar **Nodo* iter = _cabeza;** ningun otro proceso inserto nada. En este caso, se ejecutaria **nuevo->_siguiente = iter;** de manera tal que nuestro nuevo nodo apunte a la vieja cabeza, y se actualizaria la cabeza al llamar a
**this->_cabeza.compare_exchange_weak(iter,nuevo)**, que al ver que iter es igual de this_cabeza, le asigna a **this_cabeza** el valor del nuevo nodo y se retornaria *true* (como está negada en la condición del bucle, no se vuelve a ejecutar) quedando así la lista como $x \hookrightarrow a \hookrightarrow b \hookrightarrow c$.

Supongamos ahora que el proceso $P\_2$ intenta insertar un elemento z, y justo al terminar de ejecutarse la sentencia **Nodo* iter = cabeza;** ocurre que $P\_2$ inserta un nodo de valor d antes, asi queda la lista

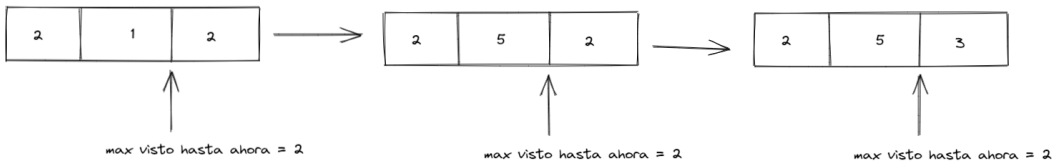
como $d \hookrightarrow a \hookrightarrow b \hookrightarrow c$ y la 'cabeza' asignada al nodo 'iter' ya no es mas la cabeza, por lo que al ejecutarse el bucle se le asigna como nodo siguiente a 'z' al nodo 'a' lo cual es incorrecto pues ahora deberia apuntar a 'd'. Pero luego de esto se ejecuta **this−>_cabeza.compare_exchange_weak(iter,nuevo)** (recordemos que es una funcion que hace una comparacion entre valores mas una asignacion de manera atomica) y al ver que iter no es igual a cabeza, se le asigna a iter el valor de **this_cabeza**, por lo tanto en la siguiente iteracion el siguiente de 'z' pasara a ser d, y en la proxima ejecucion del **compare_and_exchange** se actualizara la cabeza con el valor 'z'. En resumen, siempre que querramos insertar una cabeza y se de la situacion en que ∗**iter** apunte a la cabeza desactualizada, el bucle no terminara hasta que este no sea el caso y la cabeza sea igual al iterador.

### 3.1.2. Expliquen cómo lograron que la implementación esté libre de condiciones de carrera, justificando sus elecciones de primitivas de sincroniza- ción. Expliquen también qué decisiones debieron tomar para evitar generar más contención o espera que las permitidas.

Si permitieramos que los distintos threads pudiesen incrementar concurrentemente sin restriccion, podrian generarse casos donde los distintos threads intenten incrementar sobre la misma palabra, esto generaria condicion de carrera, sobrescribiendo el valor de la palabra en simultaneo y dejando un valor inconsistente o generando dos nodos distintos en el arrelgo con la misma palabra. Para poder incrementar concurrentemente sin problemas como los que describimos, creamos un arreglo de 26 mutex (uno por cada clave de hash), de manera tal que al estar modificando el valor de (o agregando) una clave, no se permita la modificacion de la lista asociada al hash. Esto quiere decir que se pueden incrementar hasta 26 claves en simultaneo, cuando tienen distintas iniciales (osea que no hay colision de hash).

### 3.1.3. a) Que problemas puede ocasionar que maximo e incrementar se ejecuten concurrentemente?
### b) Describir la implementacion de maximoParalelo. Cual fue la estrategia para repartir el trabajo entre los threads? Que recursos son compartidos por los threads? Como hicieron para proteger estos recursos de condiciones de carrera?

a) El principal error posible **para cuando ambos se ejecutan concurrentemente** es que mientras se busca el máximo, se incrementen algunos valores devolviendo un maximo de uns instancia inexistente.




Viendo el ejemplo anterior, podemos ver como en la primer instancia, el buscador de maximos recorrio la primer parte del arreglo y se quedo con 2 como posible maximo, luego de un par de dos tandas de inserciones, se nos genera una nueva instancia, y para el momento para el cual vuelve a avanzar el maximo, encontrara un 3 al final del arreglo, devolviendolo como resultado. Como podemos ver, el 3, no es resultado correcto de ninguna instancia, mientras que 2 o 5 si lo son y las aceptamos como respuestas correctas de la funcion.

Consideramos las siguientes dos posibilidades como resultados consistentes:

- Si llamemos a **maximo()** antes de **incrementar()**, pero que por alguna razon el incremento se de sobre una variable que pasa a ser maxima antes de terminar de hallar el maximo, y que el maximo que se retorne no sea el maximo de la estructura al momento de haberlo pedido.
- En el caso que **incrementar()** se llame antes que **maximo()**, pero que el maximo sea hallado antes de terminar de incrementar y que el maximo sea el del estado anterior a incrementar la clave.

Consideramos estos dos resultados como consistentes ya que para alguna instancia, el resultado devuelto es el maximo (siempre y cuando no genere problemas como el mencionado anteriormente).

b) Para garantizar que los threads hagan busquedas en listas distintas, y que no se produzcan busquedas repetidas en ellas, definimos una variable atomica compartida (**hashIndex**) utilizada para indexar la tabla de hash. Esta variable es incrementada de manera atomica por cada thread, y tambien es incrementada por cada thread al terminar la busqueda del maximo en una lista, pasando asi el thread a buscarlo en otra. Al terminar con cada lista, se almacena el maximo de cada una en un arreglo de pares ($max\_value$, de tamaño 26, tambien compartida e indexada con **hashIndex**) y una vez que se terminan de ejecutar todos los threads se procede a encontrar el maximo par dentro de $max\_value$.

#### 3.1.4. a) ¿Fue necesario tomar algún recaudo especial, desde el punto de vista de la sincronización, al implementar cargarArchivo? ¿Por qué?
**b) Describan brevemente las decisiones tomadas para la implementación de cargarMultiplesArchivos.**

a) No, porque de ello nos encargamos en cargarMultiples archivos.

b) Creamos un vector de threads del tamaño indicado por parametro e inicializamos un entero atomico 'count' en cero, que usaremos para indicar el numero de archivo a cargar. Esta variable es la que utilizamos como recaudo para poder incrementar de manera atomica y que threads distintos no carguen el mismo archivo. Esta variable es pasada POR REFERENCIA al thread 'stmf' junto con el vector de filePaths de los distintos archivos (indexado por la variable atomica recien descripta), asi llamamos desde cada thread a cargarArchivo con archivos distintos desde distintos threads.

# 4. Diseño de Pruebas:

**Caracteristicas del entorno de ejecucion: Se utilizo un procesador AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx (cuatro nucleos de 2.1GHz c/u y dos threads por nucleo).**

## 4.1. Experimento 1: Variación del tiempo de carga para distintas cantidades de threads y palabras

**Hipótesis:** Creemos que mas threads va a ser más rapido sin importan cant palabras. Tambien esperamos un incremento lineal mientras aumentamos la cantidad de palabras. Creemos que este crecimiento sera asi ya que, aunque el insertar es concurrente, insertar dos palabras que comienzen en el mismo caracter no lo es por lo cual insertar $n$ palabras que comienzan con el mismo caracter tiene complejidad de $O(n)$.

**Preparacion:** Este experimento tiene dos partes, una donde se analiza el impacto que tienen distintos threads sobre archivos con igual cantidad de palabras y luego sobre archivos con distintas.Para la primer parte se creo una carpeta llamada exp_unif donde hay un total de 30 archivos con 2600 palabras cada uno. Para la segunda parte se utilizaron diez carpetas las cuales tienen entre mil y diez mil palabras equitativamente distribuida entre diez archivos.

## 4.2. Experimento 2: Variacion del tiempo de busqueda del maximo para distintas cantidades de threads y palabras

**Hipotesis:** Similar al Experimento 1, creemos que al aumentar la cantidad de threads, se va a disminuir el tiempo de ejecucion de nuestro programa. Adicionalmente, al aumentar palabras vamos a ver un crecimiento lineal ya que dentro de cada lista atomica de la estructura hacemos una busqueda lineal.

**Preparacion:** Utilizamos los mismos archivos que preparamos para el Experimento 1.

## 4.3. Experimento 3: Tiempo en carga según distribución de las entradas

**Hipótesis:** A menor uniformidad en un conjunto de claves (carpetas) a insertar, mayor costo temporal a la hora de ejecutar debido a una mayor secuencialidad en las inserciones.

### 4.3.1. Distribucion en tres archivos

La implementación que se hizo para poder llevar a cabo varias inserciones al mismo tiempo nos impide llevar a cabo la inserción paralela de dos elementos con la misma inicial (es decir la misma clave del hashmap subyacente del diccionario), por lo tanto si se llega a la situacion de querer insertar varios elementos al mismo tiempo de una misma clave de hash, ésta inserción se llevará a cabo secuencialemente a pesar que sean varios los threads que las ejecutan.

**Desarrollo del experimento:** Se crearon 3 carpetas con 3 archivos cada una. En la carpeta ABC, hay 1 archivo donde todas las palabras tienen inicial 'a', otro donde todas tienen inicial 'b' y el tercero con 'c'. En la carpeta AAB' hay 2 archivos con palabras que tienen el caracter inicial 'a' y uno con palabras que inician con 'b', y en la ultima carpeta (AAA) los 3 archivos contienen palabras que inician con el caracter 'a'.

Cada archivo de palabras contiene 20.000 palabras donde todas son distintas de las del resto de los archivos en la misma carpeta, el experimento fue corrido 10.000 veces para cada carpeta. Se utilizó la misma cantidad de threads que de archivos por carpeta, de manera tal que la cada carpeta pueda procesar todos los archivos concurrentemente.

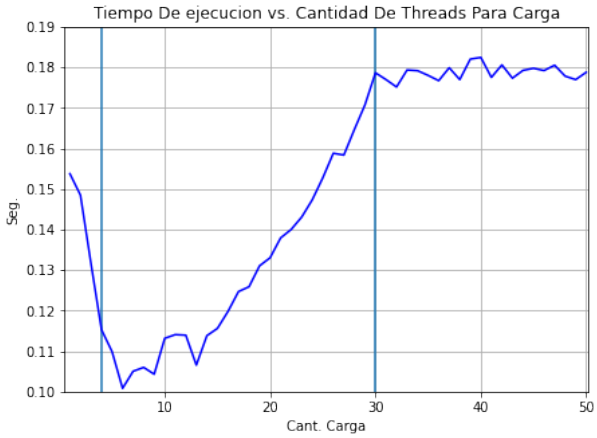
### 4.3.2. Distribucion en 26 archivos

En este experimento se opto por crear la siguiente instancia: 26 carpetas con 26 archivos cada una. Cada archivo tiene 5200 palabras.

Las distintas carpetas tienen distintas distribuciones en los archivos de palabras que comienzen con un caracter particular. Mas en detalle, la primer carpeta tiene un archivo con palabras que todas empiezan con la letra A, luego otro con la letra B, y asi sigue. Luego la segunda carpeta tiene en el primer archivo palabras que empiezan con A y B, luego otro con B y C, etc. Asi llegamos a la ultima carpeta para la cual todos los archivos, contienen palabras que comienzan con cada letra. Notemos que para cada archivo, tiene una distribucion uniforme de las palabras que comienzan con cada letra. El algoritmo de carga sera corrido con 4 threads.

# 5. Analisis de los Resultados:

Antes de empezar con los analisis, hace falta aclarar que la escala de tiempo es muy chicas por lo que tenemos que tomar encuenta que el ruido de fondo generado por otros procesos (otros procesos ocupando el procesador en vez de nuestros propios threads y los cambios de contexto de esos procesos) puede afectar nuestro resultados causando pequeñas variaciones como tambien alguna que otra anormalidad en los resultados. Adicionalmente, cerramos la mayor cantidad de procesos posibles para disminuir el ruido de fondo. Esto nos permitira tener un ambiente mas ideal para ideal para correr experimentos.



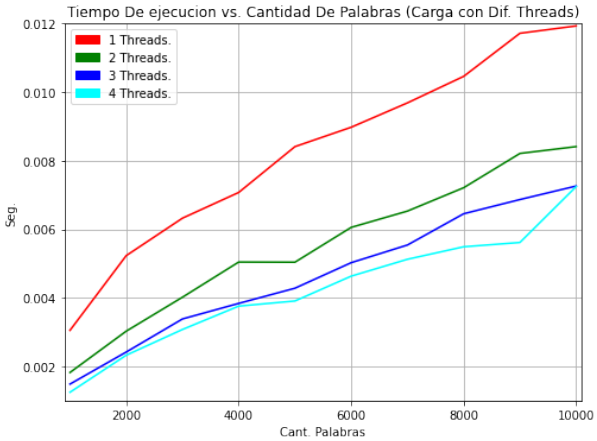

## 5.1. Experimento 1

### 5.1.1. Sin variacion de palabras:

Tal como dijimos en la preparacion, utilizamos treinta archivos, lo cual se ve muy claro en el grafico ya que luego de treinta threads no vemos ningun tipo de cambio en los resultados mas que variaciones. Algo que tambien notamos es que mas threads no implica un menor tiempo. Sino que luego de seis threads, empieza a crecer, mostrandonos que existe una cantidad optima de threads. Nosotros esperabamos que el punto optimo sean cuatro threads en vez de seis. Sin embargo, recordemos que con una escala temporal tan chica, el ruido de fondo mencionado anteriormente afecta nuestros resultados, y pudo haber generado esta inconsistencia.

La razon para esto es que como solo tenemos cuatro nucleos, luego de que estos esten ocupados, el resto de los procesos estan en Ready esperando su turno. Esto implica que no se puede tomar beneficio de la concurrencia en su totalidad ya que tenemos un cuello de botella generado por el limite de nucleos. Tener mas procesos implica mas cambios de contexto y mas tiempo de un cuantum al siguiente de un mismo thread (aunque depende mucho de la impletmentacion del scheduler). Por lo que tener muchos threads adicionales logra ser contraproducente, tal como se muestra en el grafico.

### 5.1.2. Con variacion de palabras:




Como podemos ver en el grafico anterior, un incremento en la cantidad de threads implica una carga de archivo mas rapida (dado que son multiples archivos). Como ya sabemos que los distintos archivos tienen la misma cantidad de palabras y cada thread carga un archivo a la vez, nos parecia claro que la cantidad de palabras no iba a afectar el resultado esperado sino que solo iba a mostrar un crecimeinto

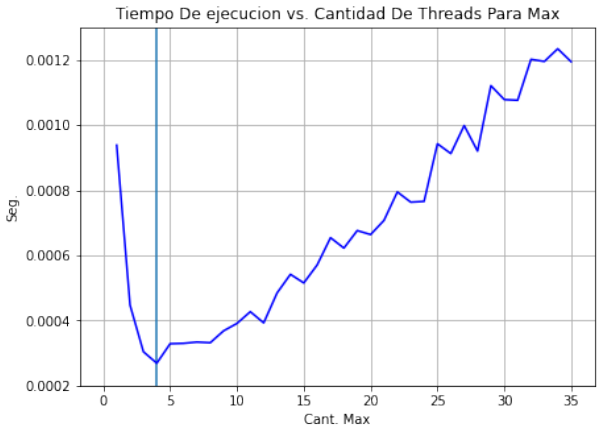
lineal en los resultados pero respetando nuestra hipotesis anterior. Cada thread adicional muestra una mejora exponencialmente mas chica que la anterior y como se puede ver la diferencia entre tres y cuatro threads es casi nula, pero aun asi hay una leve mejora, por lo que cuatro threads es lo mas optimo para este procesador con esta implementation.

A pesar de todo, es importante notar que el incremento de las palabras tuvo un impacto de crecimiento lineal en los resultados tal como esperabamos. Esto ocurre ya que las palabras son insertadas en el hash map una a la vez (en un mismo thread). Aunque nuestra estructura sea concurrente, no afecta el rendimiento de cada thread individual y sabemos que como se ejecuta una insercion por cada palabra de cada archivo, sabemos que insertar todos los archivos tiene una complejidad de $O(n)$ sin importar la cantidad de threads, con $n$ siendo la cantidad de palabras en total.

## 5.2. Experimento 2

### 5.2.1. Sin variacion en la cantidad de palabras:

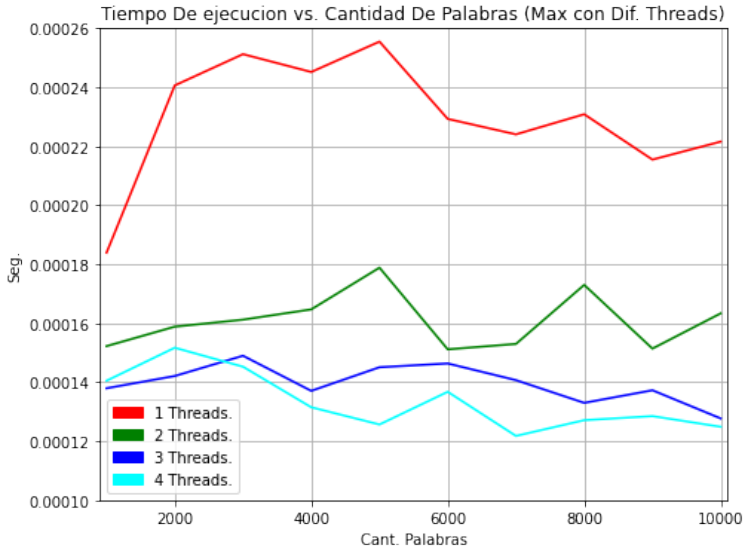
Podemos ver que al buscar el maximo hay un gran descenso en el tiempo hasta llegar a los cuatro threads (siendo el minimo aqui), aunque, contrario a nuestra hipótesis, luego observamos un incremento temporal sostenido con el aumento de la cantidad de threads.




El incremento de tiempo ocurre por razones similares a las discutidas en el experimento anterior. El codigo permite que cada thread vea una lista que contiene todas las palabras que comienzan con la misma letra. Como utilizamos un abecedario que contiene veintiseis letras, utilizar mas de este numero de threads no nos daria ningun beneficio. Esto no significa que veintiseis threads sea optimo. Como explicamos antes, tener mas threads que la cantidad de nucleos puede ser contraproducente, y esto es lo que se ve claramente en el grafico. Adicionalmente, sabemos que dependiendo cuanto puede recorrer un thread en un cuantum, un mismo thread puede empezar a buscar en una segunda lista en un mismo cuantum, logrando que menos de veinitseis sea el numero maximo con sentido.

Se puede notar que antes de cuatro threads hay un decremento exponencial en tiempo, lo cual muestra la importancia de aproximar la cantidad de procesos a la cantidad de nucleos. Recordemos que como los experimentos fueron corridos en una computadora sin muchos procesos adicionales. Por ende, no tomamos en cuentas situaciones donde hay suficientes procesos para que nuestro programa pueda utilizar tres o menos nucleos en simultaneo.

### 5.2.2. Con variacion de palabras:




Muy similar a 5.1.2, podemos ver como mas threads mejoran en rendimiento pero, nuevamente, agregar mas threads tiene una mejora cada vez menor hasta el menor tiempo con cuatro threads.

Lo que no esperabamos es que el aumento de palabras no muestren ningun impacto. Tal como dijimos en la hipotesis esperabamos un incremento lineal en los resultados originado de el aumento en cantidad de palabars. Creimos esto ya que al buscar maximo hacemos una busqueda lineal en cada una de las seis listas atomicas. Sin embargo los resultados muestran lo contrario, cuantas mas palabras se ve una pequeña mejora.

Hay dos razones principales por la cual estos resultados fueron tan inesperados. Primero, como podemos ver en la escala, cada ejecucion del programa tardo muy poco tiempo, permitiendo que el otros procesos tengan efecto mas impactante en el lo que tarda nuestro programa. Segundo sabemos que tantas palabras estan distribuidas en veintiseis listas distintas por lo que amortigua el creciemiento, ya que las estas se ven concurrentemente.

## 5.3. Experimento 3

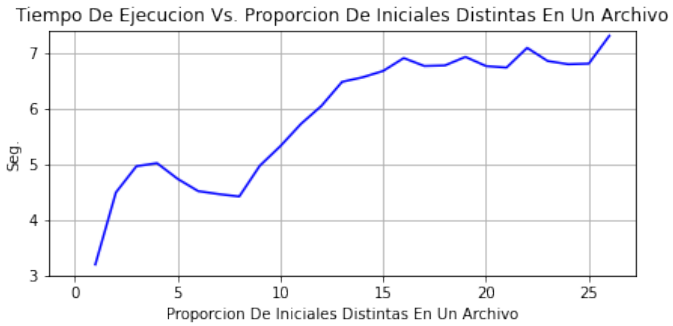
### 5.3.1. Distribucion en 3 archivos

| Carpeta | Tiempo(Seg.) |
|---|---|
| ABC | 0.000171 |
| AAB | 0.000253 |
| AAA | 0.010709 |

Como podemos observar en la figura , la corrida mas rapida fue sobre la instancia ABC porque cada thread estaba insertando en una lista distinta sin hacer esperar a otro. El segundo mas rapido fue sobre AAB ya que hay un choque entre dos threads para insertar, lo cual provocaba que se esperen, y uno insertando libremente. El mas lento fue AAA debido a que cada thread debia esperar al otro siempre para poder insertar.

Lo anterior apoya nuestra hipotesis y era esperable debido a nuestra implementacion particular de los mutex para las listas en la tabla de hash.

### 5.3.2. Distribucion en 26 archivos




Como podemos observar en la figura hay un aumento en el tiempo a medida que los archivos tienen mas palabras con iniciales distintas. Esto se debe a que en hay mas hilos queriendo usar los mutex de nuestra tabla de hash. En la primer carpeta cada hilo puede usar el mutex de la letra que le toco sin hacer esperar a otro. Mientras que en el ultimo estos choques haciendo esperar son mayores. Esto apoya nuestra hipotesis.

El grafico no muestra una linea estrictamente creciente ya que nosotros no sabemos como el scheduler ordena las tareas, por ende es probable que hayan situaciones donde, casualmente los threads esten insertando palabras con characteres iniciales diferentes logrando velocidades mas altas. Esto no siempre es asi ya que en una situacion contraria, puede ocurrir que los distintos threads intenten insertar palabras con caracteres iniciales identicos logrando un incremento de tiempo.

# 6. Sintesis de los Resultados:

- El rendimiento óptimo para inserción de palabras en paralelo se observa alrededor de la cantidad de núcleos del procesador. Las mejoras en tiempo de ejecución terminan a partir de allí.
- La inserción de palabras nuevas tiene un costo temporal superior a incrementar una clave ya insertada.
- Al igual que para la inserción, el tiempo de ejecución para máximo disminuye hasta la cantidad de núcleos de procesador (aproximadamente).
- A mayor uniformidad en las claves de las palabras insertadas, mejor tiempo de inserción.

# 7. Conclusiones:

- La cantidad óptima de threads dependen de la cantidad de núcleos del procesador.
- Es importante notar que cada scheduler tiene sus ventajas como desventajas, lo cual tambien afecta como corre el programa. Se debe conocer bien el procesador al momento de elegir la cantidad de threads y en un caso de desarrollo de un programa que utilice multithreading donde no se pueda elegir la cantidad de threads, es importante conocer bien el estándar de núcleos utilizado por la audiencia apuntada para poder tener una implementacion óptima.
- La mejora en el rendimiento debida al paralelismo está sujeta a la distribución de las claves de las palabras a insertar, a mayor uniformidad, mejor rendimiento de la estructura.
- Una alta repetición en las palabras mejora el rendimiento, y la aparicion de muchas palabras nuevas ralentiza las operaciones de búsqueda e inserción.