DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico 1: Threading

Sistemas operativos

Integrante	LU	Correo electrónico
Martín Lei	578/16	martinlei96@hotmail.com
Lucia Fernandez	832/19	lucif268@gmail.com
Valeria Arratia Guillen	467/21	arratiavale@gmail.com
Matias Cozzani	915/19	matcozzani@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
https://exactas.uba.ar

1. Introducción

Desde que aparecieron las primeras computadoras hasta el día de la fecha, se ha buscado incrementar la optimización de la utilización de los recursos con los que contamos. Para ello, no solo ha habido mejoras notables en el aspecto físico de la computación: las mejores implementaciones a nivel electrónico han dado lugar a procesadores que han ido incrementando el poder de cómputo significativamente, año a año, incluso hasta el día de hoy. Pero también se han mejorado substancialmente en el campo conceptual de la computación: dados ciertos recursos ¿Cuál es la mejor manera de aprovecharlos? ¿Qué estructuras adicionales podemos construir por sobre la CPU en sí, que nos permitan generar incluso mejores resultados? Los sistemas operativos no son la excepción. Han evolucionado con el resto del paradigma computacional dando lugar a sistemas mejores, más óptimos, más seguros y más transparentes.

En el presente trabajo práctico buscaremos profundizar en una de las características principales que aparece en el estudio de los sistemas operativos: la concurrencia. Se detallará brevemente qué es la concurrencia, cómo se relaciona con la multiprogramación y qué ventajas ofrece. Estudiaremos cómo razonar sobre la ejecución concurrente de programas y las técnicas para gestionar la contención sobre los recursos y evitar que se produzcan condiciones de carrera.

Nos centraremos especialmente en el uso de **threads**, una herramienta provista por los sistemas operativos que nos permite disponer de varios hilos de ejecución dentro de un mismo proceso.

Realizaremos una implementación de un HashMap-Concurrente. Se trata de una tabla de hash abierta, donde cada entrada de la tabla es representada con una lista enlazada. A su vez, la función de hash no será otra cosa que obtener la primer letra de la palabra en cuestión, permitiendo solo contener iniciales minúsculas. Por tanto, nuestro Hashmap tendrá 26 entradas. Su interfaz de uso es la de un map o diccionario, cuyas claves serán strings y sus valores, enteros no negativos. La idea central es estudiar qué sucede sobre una estructura de datos preparada para manejar paralelismo cuando múltiples agentes -threads en este caso- trabajan sobre ella. En particular, buscaremos estudiar los comportamientos concurrentes realizando conteos de entradas en cada una de las listas asociadas a las entradas, así como también analizar cómo se comportarán los agentes al momento de tener que popular la tabla.

2. Problemas a Resolver

2.1. Condición de Carrera

Una *condición de carrera* o *Race Condition* es una circunstancia muy común en el contexto de la programación paralela. Se da cuando múltiples hilos de ejecución compiten por acceder a un recurso compartido. Decimos que hay race-condition cuando el resultado de la ejecución depende del orden en que se ejecutan los hilos, teniendo resultados distintos según qué hilo llegó a realizar ciertas operaciones primero. Para evitar esto, se recurre a diferentes técnicas de sincronización que nos permiten coordinar la ejecución de los hilos y asegurarnos de que solo uno acceda al recurso compartido en cada momento.

Las condiciones de carrera se evitan básicamente protegiendo las variables que concentran información crítica accedida y compartida por los threads. Si podemos pensar que un proceso es una abstracción de la ejecución de un programa, entonces un thread es una abstracción de un hilo de ejecución de ese proceso. Los threads entonces permiten mantener -dentro del mismo espacio de proceso- distintos hilos de ejecución activos. Como igualmente viven dentro del mismo espacio de proceso, entonces comparten porciones de memoria. Este contexto da lugar a que cada uno pueda acceder a las estructuras que recaudan información critica de nuestro programa al mismo tiempo.

Por sí mismo, esto no sería un problema ya que es precisamente el motivo por el que los threads son una herramienta interesante. El problema es cuando esa información crítica sufre modificaciones sin ningún tipo de criterio, dando pie a situaciones que podrían afectar la consistencia de los datos.

En nuestro contexto esto puede ocurrir en diferentes escenarios, como en la inserción de un nuevo nodo en la lista: ¿Qué pasa si un nodo está insertando una palabra en una entrada y se acaba su tiempo de CPU y otro quiere leer la entrada completa? Como el primero no terminó su tarea, el segundo tendrá información inconsistente. O, ¿Qué pasa al momento de incrementar la cantidad de elementos para una clave concreta en el HashMap, si varias ejecuciones intentan cambiar simultáneamente este valor?

2.2. Deadlocks

Otro problema con el que nos podemos encontrar en la programación concurrente es cuando dos o más agentes se quedan esperando indefinidamente a que otro para libere recursos que necesitan para continuar su ejecución. En otras palabras, es una condición en la que los hilos o procesos están atrapados en un estado de bloqueo mutuo, sin poder avanzar -sin progreso-.

Una primera aproximación para estudiar la presencia de deadlocks en un programa es analizar una serie de condiciones conocidas como "las condiciones de Coffman", que son:

- **Exclusión mutua:** Al menos un recurso debe estar asignado en modo exclusivo (solo puede ser utilizado por un hilo o proceso a la vez).
- **Posesión y espera:** Un hilo o proceso debe tener al menos un recurso asignado y estar esperando para adquirir otros recursos que están en posesión de otros hilos o procesos.
- **No asignación y espera:** Un hilo o proceso debe estar esperando para adquirir al menos un recurso que está siendo poseído por otro hilo o proceso.
- **Espera circular:** Existe una cadena circular de hilos o procesos en la que cada uno está esperando a que el siguiente libere un recurso.

Cuando estas cuatro condiciones se cumplen simultáneamente, se produce un deadlock y los hilos o procesos quedan bloqueados sin poder avanzar. Esto puede llevar a un bloqueo completo del sistema, donde ninguna tarea puede progresar y se requiere una intervención externa para resolver el deadlock.

2.3. Paralelización

Buscamos aprovechar al máximo el paralelismo y la ejecución simultánea de tareas. En un sistema concurrente, el objetivo es permitir que múltiples hilos, procesos o tareas se ejecuten de manera eficiente, segura y efectiva, lo que puede mejorar el rendimiento y la capacidad de respuesta del sistema. Es decir, deben poder ejecutar tareas de manera simultánea, pero la respuesta final debe ser equivalente a la que obtendríamos en caso de no utilizarlos.

Para esto se tiene que tener varias cosas en cuenta:

1. **Modularización del problema:** El problema o la tarea principal se divide en subproblemas más pequeños e independientes que pueden ser ejecutados concurrentemente. Esto permite que varios hilos o procesos trabajen en paralelo, procesando diferentes partes del problema de forma simultánea.
2. **Sincronización adecuada:** Es necesario utilizar mecanismos de sincronización para coordinar la ejecución de los hilos o procesos concurrentes. Estos mecanismos permiten establecer un orden de ejecución, evitar condiciones de carrera y garantizar la consistencia de los datos compartidos.

3. **Granularidad apropiada:** La granularidad se refiere al tamaño de las tareas que se ejecutan concurrentemente. Si las tareas son demasiado grandes, puede haber una falta de concurrencia, ya que los hilos o procesos estarían esperando mucho tiempo para adquirir y liberar recursos. Por otro lado, si las tareas son demasiado pequeñas, puede haber un alto costo de sincronización. Es importante encontrar un equilibrio adecuado y dividir las tareas en unidades significativas que permitan un buen balance de carga y aprovechamiento de recursos.

4. **Evitar bloqueos y deadlocks:** Los bloqueos y deadlocks pueden reducir drásticamente la concurrencia efectiva de un sistema.

3. Desarrollo del problema

3.1. Atomicidad e Inserción

El uso de estructuras atómicas garantiza que una secuencia de operaciones se ejecute de manera indivisible y sin interrupciones, asegurando la consistencia de los datos. Esto es especialmente importante en entornos concurrentes, donde múltiples hilos o procesos pueden acceder y modificar los mismos datos al mismo tiempo.

En este contexto, la utilización de una lista atómica como estructura de datos proporciona operaciones atómicas, basadas en el uso de instrucciones atómicas, garantizando la protección ante condiciones de carrera. En particular, los nodos de la lista enlazada se establecen atómicos y son manipulados a través de operaciones atómicas, tales como *.load*(), suministradas por la librería atomic de $C++$.

Dado que la lista atómica que se diseña para esta solución es en sí una estructura de datos compleja, es difícil poder implementar una atomicidad real sin ayuda del hardware: no podemos asegurar que a lo largo de una operación de inserción, ese thread nunca perderá la CPU. Sin embargo, si es posible utilizar estructuras atómicas que nos aseguren que, incluso si lo hace, no habrá resultados inconsistentes.

Para la operación de inserción, primeramente, se debe construir el nuevo nodo. Esto implica establecer el valor del nodo, el cual es pasado por parámetro, y establecer el nodo siguiente al que apunta, la actual cabeza de la lista. Luego, es preciso restablecer la cabeza de la lista como el nuevo nodo que se está insertando. Aquí es donde surge un problema crítico de concurrencia: ¿Qué pasa si justo antes de establecer la cabeza de la lista como este nuevo nodo insertado, el **thread A** que lleva acabo tal operación pierde la CPU y, en su lugar, lo recibe otro -**thread B** que también debía terminar de insertar un elemento? ¿Cuál es la nueva cabeza de la lista? ¿El nodo insertado por **A** o el nodo insertado por **B**?

Para resolver este problema, se hace uso de la primitiva de sincronización:

compare_exchange_weak(T esperado, T deseado)

Esta primitiva permite comparar el valor de una variable atómica con un valor esperado. **Si los valores son iguales entonces reemplaza el valor de la variable atómica por el valor deseado**, caso contrario, **el valor esperado se actualiza para obtener de manera atómica el valor actual de la atómica**. En nuestra implementación, es establecer el valor de la variable esperada como la cabeza de la lista y el valor deseado como el nuevo nodo. De esta manera, si son iguales el valor del nuevo nodo reemplaza el valor de la cabeza de la lista, y si son distintos, se obtiene el valor actual.

Algorítmicamente, podemos pensar la primitiva del siguiente modo:

```
if variableAtomica == esperado then
    variableAtomica ← deseado
    return True
else
    esperado ← variableAtomica
    return False
```

En nuestro contexto de uso:

```
if cabecera == nuevoNodo− > siguiente then
    cabecera ← nuevoNodo
    return True
else
    nuevoNodo− > siguiente ← cabecera
    return False
```

De este modo, podemos aprovechar la primitiva para iterar todas las veces que haga falta hasta que efectivamente el estado del programa nos permita asegurar que el siguiente del nuevo nodo que estamos insertando sea la cabeza actual de la lista -puede o no ser la misma que cuando empezamos a chequear- en cuyo caso se realizará el reemplazo atómicamente.

3.2. Incrementar, claves y valor:

Las tres operaciones deben estar libres de condiciones de carrera y de deadlocks. Para lograrlo se decidió implementar para cada entrada de la hashtable un esquema de multiple-readers single-writer con mutex. Esto es, por cada entrada i de la tabla se cuenta con dos mutex ($readers[i]$, $writers[i]$) y un contador de la cantidad de lectores en esa entrada ($cantLectores[i]$).

Esta implementación permite granularizar la contención de la siguiente manera:

Escritor:

- Si un escritor quiere escribir y no hay nadie leyendo, entonces puede hacerlo. En este momento bloquea la entrada para que no se pueda leer mientras que él escribe.
- Si un escritor quiere escribir pero actualmente hay lectores, se bloqueará hasta que el último lector termine de leer.
- Si un escritor quiere escribir pero actualmente hay lectores, se bloqueará hasta que el escritor termine de escribir.

Lector:

- Si un primer lector quiere leer y no se está escribiendo, bloquea la escritura y procede a leer.
- Si múltiples lectores quieren leer podrán hacerlo luego de incrementar el contador correspondiente, sin contención entre ellos.
- Cuando el último lector termine con la sección crítica -esto es, el acceso a la entrada de la tabla sobre la que necesita trabajar- desbloquea la escritura.

Dado que este esquema está implementado **por entrada de la tabla**, no hay contención alguna en caso de inserciones sobre entradas distintas: esto es, cada instancia de implementación MRSW en la $entrada_i$ es independiente de la instancia de implementación MRSW de la $entrada_j$ siempre que $i \neq j$.

3.3. Búsqueda del Máximo

Si las operaciones de buscar el máximo e incrementar se ejecutan concurrentemente sin ninguna protección adicional, pueden ocurrir problemas como condiciones de carrera y resultados incorrectos.

Veamos el siguiente escenario de ejecución posible:
Supongamos un caso donde la función máximo comenzó a correr sobre una cierta instancia de datos. Imaginemos además, un snapshot del estado del programa en ese momento.
En este snapshot del programa, dentro de la tabla de hash, aquella clave con máxima cantidad de apariciones es la palabra "casa", con un total de 50 apariciones, y que la palabra "baba" tiene 49 apariciones. Si en este momento, la operación máximo está analizando las entradas de la letra "d" -ergo, ya recorrió completamente la entrada de la letra "b" por lo que no volverá a mirarla-. El problema surge si, por ejemplo, se producen dos incrementos sobre la palabra "baba", por lo que la palabra "casa" ya no será el máximo. Sin embargo, la ejecución del máximo ya no volverá a recorrer esas entradas por lo que devolverá una respuesta incorrecta.

En lo que respecta al presente análisis, se decide aprovechar la implementación de mutexes generada para incrementar y también bloquear la entrada antes de leer los datos de la misma. Esto claro nos reduce -en algún punto- las posibilidades de obtener una respuesta inconsistente con los datos, pero sigue quedando abierto a algunas inconsistencias: si la modificación de incrementar se produce sobre otra entrada -como vimos en el ejemplo planteado- maximo potencialmente devuelve resultados incorrectos.

Por otro lado, podemos mejorar la búsqueda del máximo repartiendo el trabajo, es decir, usando múltiples threads. Se implementa la función *maximoParalelo* de la siguiente manera:

La función recibe por parámetro el número de threads que serán utilizados para repartir la carga de trabajo. Cada uno de estos se encarga de realizar una búsqueda lineal del máximo en una fila de la tabla. Una vez que lo obtiene, lo agrega a un vector de máximos locales. Luego de procesar una fila, cada thread obtiene un nuevo índice correspondiente a una fila que aún no ha sido procesada y repiten este proceso hasta que todas las filas del HashMap hayan sido recorridas.

Este proceso de obtener un nuevo índice es crítico. La obtención del índice debe realizarse de manera tal que nunca dos threads obtengan exactamente el mismo valor, pero que ninguna de las entradas de la tabla quede sin revisar. Para garantizar estas condiciones, se utiliza un entero atómico compartido: *indexParalelo*. Cada thread obtiene el valor actual, e incrementa de manera atómica. Esto se hace aprovechando la función $fetch\_add$. Esta variable compartida designa por un lado qué entrada de la tabla de hash debe revisar ese thread en particular, por otro lado establece en qué índice de *maximosParciales* debemos almacenar el valor máximo local de esa entrada. Todos los threads acceden y modifican este vector de forma concurrente. Sin embargo, dada la estrategia utilizada, donde cada thread procesa filas distintas de la tabla, no habrá conflictos al acceder y modificar el vector de forma concurrente.

De esta manera, para ambos recursos compartidos, se evitan condiciones de carrera y no hay riesgo de pérdida de información.

3.4. Carga de Archivos

En la implementación de *CargarArchivo* no fue necesario tomar ningún recaudo especial dado que la función incrementar, que utiliza *CargarArchivo*, ya aplica mecanismos de sincronización.

Por otro lado, al momento de cargar múltiples archivos podemos utilizar múltiples hilos para repartir el trabajo. Podemos hacer que cada hilo se encargue de un archivo a la vez, teniendo en

cuenta las mismas consideraciones tomadas para la función *MaximoParalelo*.

La versión **paralela** de *CargarArchivo* hace uso de la misma idea que *MaximoParalelo*: un entero atómico compartido le indica a cada thread que archivo debe procesar.

4. Experimentación

La experimentación fue corrida minimizando lo máximo posible la incidencia del ruido del sistema sobre los resultados, para ello, se dejó correr la experimentación sin ningún otro proceso de fondo mas que aquellos indispensables para el kernel y su funcionamiento. Las características del sistema que corrieron la experimentación son:

- **CPU:** Intel Core i7-1165G7 @ 2.80GHz
- **RAM:** 16GB
- **SO:** Ubuntu 22.04.2 LTS
- **Compilador:** G++ standard C++20

Además creamos una función ContarPalabrasExperimentar que no es mas que una copia de ContarPalabras pero con algunas modificaciones mínimas a fin de poder realizar los experimentos con mayor control sobre los argumentos de las funciones.

Para las experimentaciones utilizaremos 10 archivos con 5.000 palabras, generados en python desde una jupyter notebook. Cada uno de los experimentos correrá 500 veces a fin de obtener una cantidad de muestras coherente.

Se plantean 3 hipótesis a verificar:

1. **Experimento 1:** Aumentar la cantidad de threads de lectura en función de la cantidad de archivos a leer mejora el rendimiento.
2. **Experimento 2:** Aumentar la cantidad de threads de lectura en archivos ordenados alfabéticamente empeora el rendimiento.
3. **Experimento 3:** El rendimiento de la función *MaximoParalelo* mejora si aumentamos la cantidad de threads de ejecución.

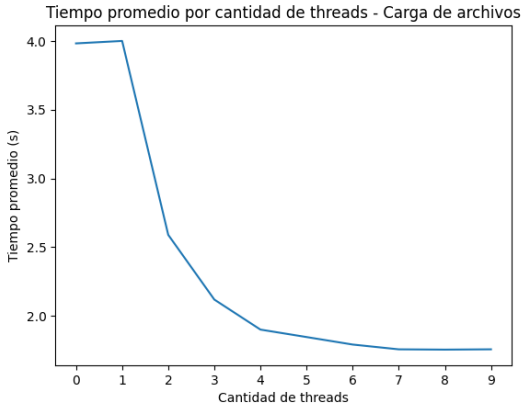
4.1. Experimento 1: Threads en función de archivos

La idea es ir variando la cantidad de threads de uno en uno hasta diez, esperando ver una relación lineal entre el aumento de hilos y el descenso del tiempo de ejecución.
Por otro lado, también analizaremos empíricamente el comportamiento de la lectura de archivos según la distribución de los datos, es decir, si las palabras están ordenadas alfabéticamente, o desordenadas pero con palabras repetidas, o desordenadas y sin repetir.

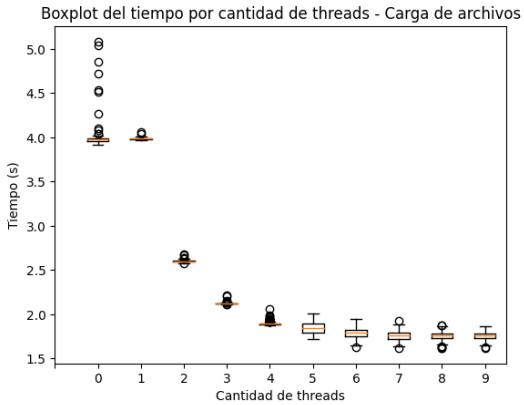
4.1.1. Ordenado alfabéticamente

En este experimento, la hipótesis es que como los threads estarán accediendo a palabras con puntos de acumulación similares entonces no podrán aprovechar la concurrencia y la mayoría de ellos caerán en esquemas secuenciales, lo que implica que competirán por el acceso a la misma entrada de la tabla debiendo esperar a que se libere. Por ende, no veremos mejoras significativas al incrementar la cantidad de threads. Más aún, posiblemente veamos que aumentar la cantidad de threads sea contraproducente, puesto que muchos de ellos comenzarán a bloquearse entre sí.

Se determina, por cuestiones asociadas a la implementación del código, que no tiene sentido disparar más threads que archivos, dado que cualquiera de aquellos threads extra que se creen no tendrán índice que mirar dentro del arreglo de archivos, no harán nada y terminarán.



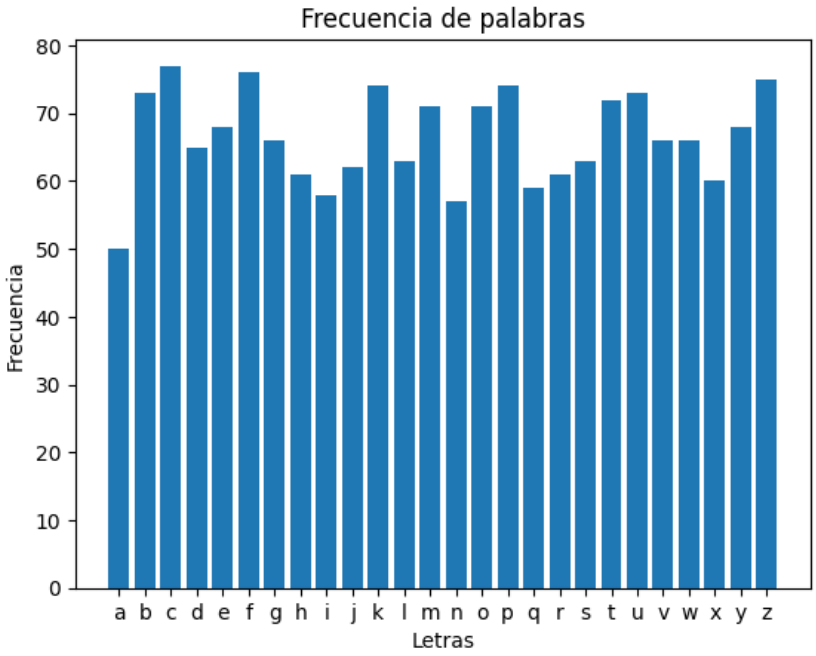
(a) Tiempo promedio por cantidad de threads.



(b) Boxplot del tiempo por cantidad de threads.

Los gráficos nos muestran una mejora significativa a medida que se van agregando threads a la ejecución, lo que nos permite refutar nuestra hipótesis principal en esta experimentación. Éste resultado fue sumamente inesperado ya que todo apuntaba que al ordenar las palabras, iban a haber mas bloqueos debido a las políticas de contención implementado en la estructura de HashMap.

Cabe mencionar que la distribución de la cantidad de las palabras por cada archivo juegan un rol fundamental en el tiempo de ejecución, ya que si las cantidades son dispares, algunos threads terminarán antes que otras en procesar las letras y esto conllevaría al destape del cuello de botella que se esperaría generar. Esto no sería el caso, ya que usamos *random.choice*() de python que selecciona elementos de una secuencia y utiliza una distribución uniforme para las palabras.



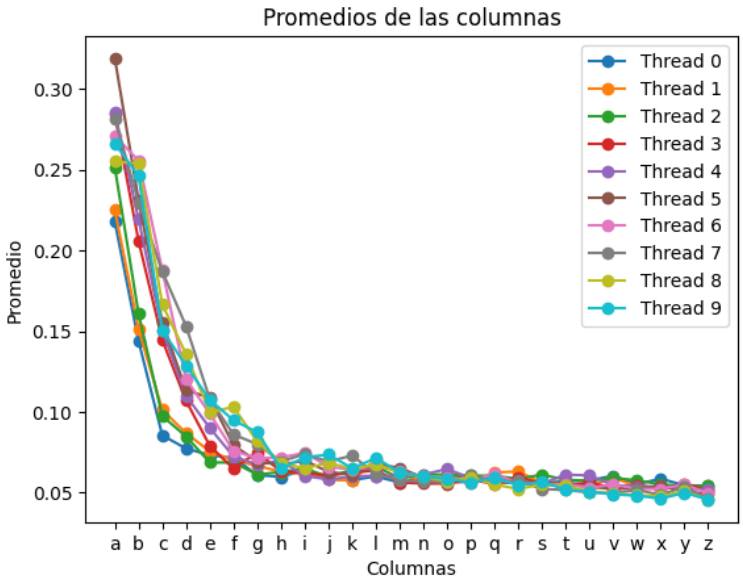
Dado que el resultado no fue el esperado, nos pareció prudente analizar más en profundidad este comportamiento y se propuso el experimento detallado a continuación.

4.1.2. Experimento 1.2: Puntos de acumulación y corrimiento.

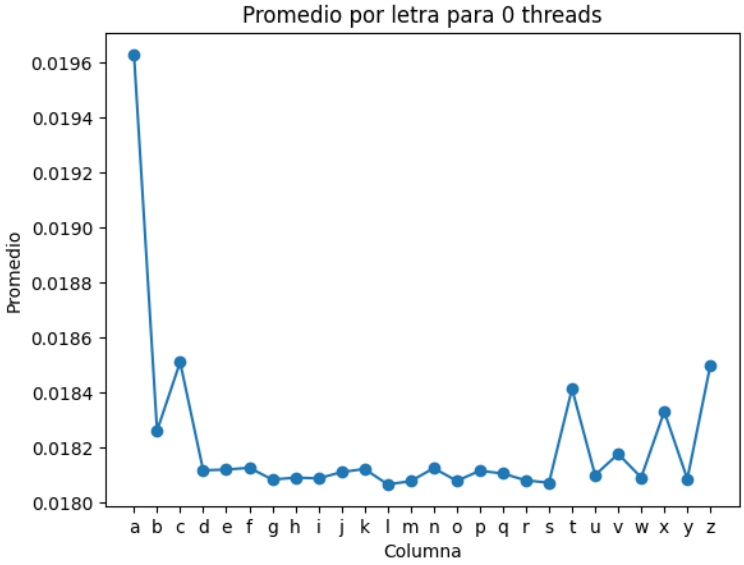
La hipótesis detrás de este experimento es la siguiente: aumentar la cantidad de threads que leen los archivos sobre archivos ordenados alfabéticamente y con misma cantidad de palabras por letra mejora el rendimiento debido a que los threads se acumulan al principio, pero a medida que obtienen CPU van teniendo corrimientos lo que vuelve a permitir esquemas paralelos.

Dado que las palabras están ordenadas y que un solo thread se encarga de un archivo particular, inicialmente todos irán primero a procesar las palabras con A, luego con B, etc. A priori, esto podría parecer ineficiente dado que las entradas de nuestro HashMap concurrente están protegidas con un mutex para accesos concurrentes. Esto implicaría que $thread_0$ comenzaría a procesar el $archivo_0$ en la entrada A y el $thread_1$ comenzaría a procesar el $archivo_1$ en la entrada A, y no tendrá acceso pues el $thread_0$ llegó primero, por lo que se bloqueará. Esto podría con todos los threads que quieran iniciar con la letra A. Sin embargo, aquí es donde entra en juego nuestra hipótesis: cuando un cierto thread llegue a terminar su batch de palabras iniciadas por una cierta letra, pasará al próximo batch de palabras de una inicial distinta. A medida que esto ocurra, los threads exhibirán un "corrimiento" dentro de los archivos, descomprimiendo la presión ejercida sobre las diferentes entradas de la tabla y permitiendo que los threads que "quedaron atrás" inserten más libremente dentro de la tabla.

Dada una determinada cantidad de threads y de archivos ordenados alfabéticamente medir el promedio de tiempo que cada thread tarda en registrar en el HashMap cada porción de palabras que inician con la misma letra. A fin de garantizar un ambiente más controlado para el experimento se asume que o bien correrá un thread para todos los archivos, o cantidad de archivos threads. Por otro lado, cada archivo generado tiene exactamente la misma cantidad de palabras, más aún, la misma cantidad de palabras por letra: 2600 palabras, 100 por letra. Para llevar acabo el experimento, se modificó ligeramente el código de algunas funciones -definidas como funciones auxiliares para que no entren en conflicto con las originales-. Aprovechando que los threads comparten memoria con el proceso que los dispara, se define de forma global un vector de vectores de tupas de timespec -estructura utilizada por la librería time.h para medir tiempos-. Es decir, para cada thread se inicializa un vector de 26 tuplas de las cuales se podrá extraer el tiempo para cada letra. Es importante aclarar que se guardan las estructuras en sí y no los tiempos calculados para disminuir la carga de trabajo extra sobre los threads. Cada thread toma tiempos para cada letra y espera a que se produzca un cambio de letra para terminar de medir el tiempo y asignarlo en la posición correspondiente de su vector de tiempos.



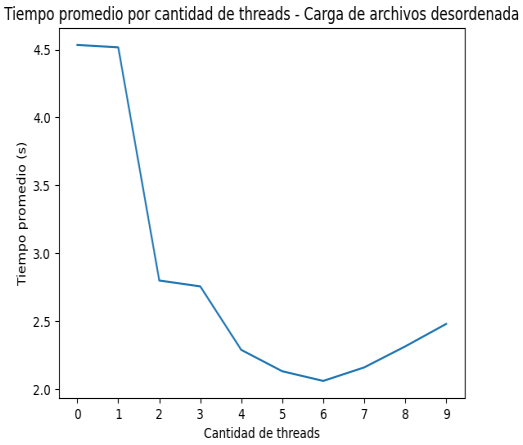
En el gráfico anterior podemos observar los tiempos de procesamiento para cada letra por cada thread en un color distinto. Como podemos ver, hay una mejora notable en el tiempo de ejecución por letra a medida que avanzamos en el abecedario: como conjeturamos anteriormente, esto se debe a que hay un "corrimiento" de los hilos a medida que avanzan en las entradas de la tabla y liberan las anteriores. Todos los threads se estancan en la letra A esperando a que el primero termine, sin embargo, una vez que el primero avanza, va abriendo lugar a que los anteriores avancen también.



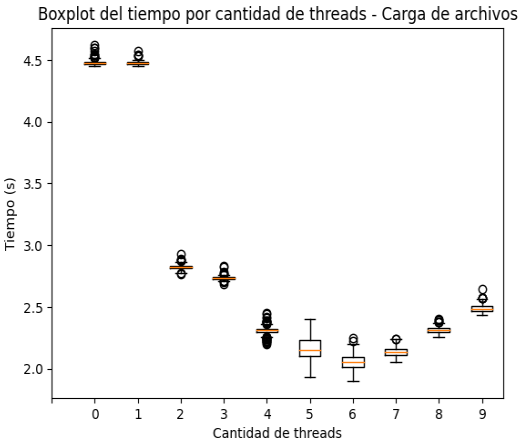
Ahora podemos ver claramente para que sirve la paralelización: incluso siendo que hay estructuras de contención en el recurso compartido, la paralelización nos optimiza mucho el rendimiento de la carga de datos. Si bien al principio es probable que tarden un poco más en cargar las palabras, luego esa demora se desvanece y los threads pueden avanzar muy rápidamente. Como además desplegamos un thread por archivo, el costo de procesar cada chunk de palabras iniciadas con una cierta letra se paga una sola vez. En el caso de disparar solamente un thread que procese todos los archivos, el mismo debe pagar el costo de un determinado chunk, cantidad de archivos veces.

4.1.3. Experimento 1.3: No ordenados alfabéticamente

En este caso utilizaremos archivos que no están ordenados alfabéticamente. La hipótesis es que cuando los threads empiecen a cargar los archivos, como estos no tienen puntos de acumulación de entradas similares -que empiezan por la misma letra- entonces los threads se ven menos comúnmente forzados a caer en esquemas secuenciales -porque no entran en juego las estrategias de contención de concurrencia- y por ende se puede aprovechar de mejor manera el paralelismo, teniendo mejores resultados.



(c) Tiempo promedio por cantidad de threads para para carga desordenada.



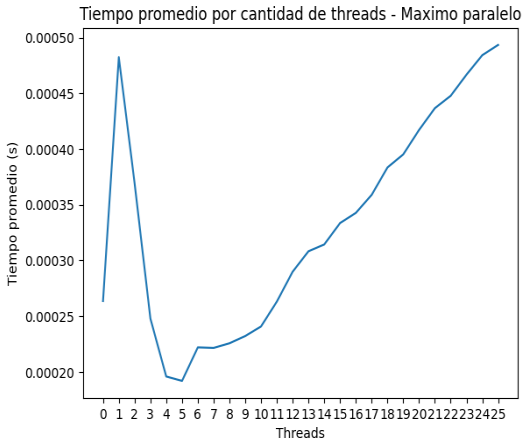
(d) Boxplot del tiempo por cantidad de threads para carga desordenada.

El gráfico refleja una clara mejora en el tiempo de ejecución a medida que se van agregando threads, y si bien es cierto que se corrobora la hipótesis, también es cierto que hay un mínimo en 6 threads. Esto demuestra claramente que el uso de threading debe ser cauteloso: no siempre más es mejor. En este caso, para esta implementación y estas características puntuales de hardware, la cantidad óptima de threads para esta tarea es 6, pero para distintos contextos bien podría ser otra.

4.2. Experimento 2: *MaximoParalelo* mejora con mayor cantidad de threads.

La función maximoParalelo se encarga de recorrer todo el HashMap de forma concurrente con el objetivo de encontrar cuál es la palabra con mayor cantidad de apariciones en los archivos leídos. Las características de los archivos utilizados son los mismos que para el experimento 1.3.

Notar que dado que un thread recorre una entrada de la tabla por vez, aumentar la cantidad de threads a mayor que 26, que es la cantidad de entradas totales resulta absurdo, pues el trabajo ya esta distribuido en su totalidad.



(e) Tiempo promedio por cantidad de threads para para carga desordenada.

(f) Boxplot del tiempo por cantidad de threads para carga desordenada.

Los datos nos muestran que en efecto hay una mejoría al aumentar la cantidad de threads, pero que a partir de un cierto momento -5 threads- comienza a empeorar. Nuevamente, nos encontramos ante un punto mínimo que optimiza la performance. Esto muy probablemente se deba a que como hay estructuras de contención, la cantidad de threads mas eficiente para recorrer la tabla es una tal que aprovecha la paralelización, pero también tal que la periodicidad con la que se paga el overhead del thread switch es minimal. Siguiendo esta idea, también es interesante meditar sobre que la cantidad de threads que maximiza la performance no es exactamente la cantidad de hilos que admite la CPU del sistema de experimentación.

Esto remarca que al momento de realizar una implementación threading es impresindible realizar un estudio correcto de la solución que se debe ofrecer, no solo desde el punto de vista del algoritmo, sino también de la complejidad añadida sobre el sistema a través de las estructuras de contención que se ha decidido implementar.

Por último, es importante aclarar que todas las conclusiones podrían cambiar si se cambia el paradigma de contención: en vez de bloquear una entrada entera de la tabla al momento de realizar un incremento sobre una determinada clave, solo se bloquea aquella clave que se está incrementando. Implementar mayor granularidad en las medidas de contención bien podría dar lugar a que la cantidad de threads efectivos para trabajar sobre una determinada tarea se modifique.

5. Conclusiones

En conclusión, la utilización de threads para dividir el trabajo entre distintos agentes es tan efectiva como situacional. En líneas generales, cada vez que nos enfrentemos a un problema computacionalmente denso sería coherente estudiar la posibilidad de sub-dividir ese problema en problemas más chicos, atacado cada uno de ellos con un thread particular.

- Incorporar al diseño de una solución determinada el uso de threads puede mejorar notablemente la respuesta de dicha solución.
- Los threads permiten hacer un uso muy efectivo de los recursos computacionales disponibles.
- No todo es color de rosas: es determinante para que la solución se comporte correctamente armar estructuras de contención robustas que protejan de bloqueos y aseguren la consistencia de los datos.
- Es crucial que las estrategias de contención entren en juego lo mínimo e indispensable posible para asegurar el punto anterior.

- No hay que olvidar que la existencia de threads agrega costo también, tanto computacional como de complejidad de razonamiento sobre la solución. Es importante determinar qué cantidad de threads serán los más óptimos para una cierta tarea siendo que existe un overhead significativo al momento de realizar context-switch, no siempre más es mejor.
- No necesariamente cualquier solución se beneficiará de la aplicación de threads. Todo depende de las características del trabajo a realizar.