DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Sistemas Operativos
Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Milton Ariel Herrera Bauleo	828/18	mhbauleo@dc.uba.ar
Gabriel Sánchez Do Santos	574/17	documentodt@hotmail.com
Tomás Sujovolsky	113/19	tsujovolsky@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

El objetivo de este trabajo es indagar en la gestión de la concurrencia. La idea es profundizar en los métodos en los cuales distintos procesos utilizan los recursos compartidos del computador sin interferirse entre sí, o para mejorar su rendimiento. Para esto, vamos a utilizar herramientas como variables atómicas, *mutex* y/o *threads*, con un énfasis particular en esta última herramienta.

1.1. Conceptos Claves

A continuación, daremos una breve explicación de ciertos conceptos claves que utilizaremos a lo largo del trabajo:

- Variables Atómicas: denominadas así debido a que proveen operaciones de las que el sistema no distingue estados intermedios, estas variables proveen una forma segura de manejar recursos compartidos e impiden comportamiento indefinido, o *race condition*. Entre sus métodos se tiene: *compare_exchange*, *fetchAdd*, *testAndSet*, etc.
- *mutex*: se utilizan para garantizar que sólo un proceso entre a la sección crítica.
- *threading*: técnica que permite que en un mismo proceso se realicen distintas ejecuciones en paralelo. Estas ejecuciones se llaman *threads* y comparten el espacio del proceso.

1.2. Explicación del Problema

El problema que este trabajo plantea resolver es la implementación de una estructura de datos llamada *HashMapConcurrente*. Se trata de una tabla de hash abierta, que gestiona las colisiones usando listas enlazadas atómicas. Su interfaz de uso es la de un *map* o diccionario, cuyas claves serán 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 serán las palabras y los valores, su cantidad de apariciones)[1].

Sin embargo, hay que tener en cuenta que esta estructura admite concurrencia. Por ejemplo, se admite que dos procesos intenten agregar elementos al mismo tiempo, o buscar el máximo al mismo tiempo. Por eso, al implementar las funciones para interactuar con la estructura, será necesario tener en cuenta e idear métodos de contención, tanto para proteger la integridad de los datos, como para realizar la funciones de una manera más eficiente. El objetivo es que estas funciones estén libres de condiciones de carrera, *deadlocks* e inanición.

Las implementaciones más importantes, y en las cuales nos enfocaremos, son las funciones para calcular máximo y las funciones para cargar los archivos, ya que estas poseen 2 implementaciones, una orientada a la concurrencia, a través de los *threads*, y otra sin concurrencia.

Finalmente, mediante diferentes tipos de experimentos, evaluaremos la *performance* de las implementaciones anteriormente mencionadas, elaboraremos hipótesis para ellas y analizaremos los resultados para llegar a una conclusión.

[1] Esta es la explicación proveniente de la introducción del enunciado del tp.

2. Implementaciones

2.1. Inserción para Listas Atomicas

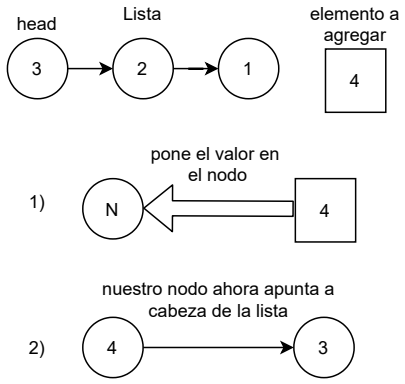
En principio, que una lista sea atómica implicaría que sus operaciones son atómicas, es decir, si se ejecuta una operación sobre la lista, el sistema no distingue estados intermedios de la lista durante la ejecución de esta operación, solo distingue el estado anterior y posterior a la ejecución de la operación. Esto es fundamental para la inserción de elementos en la lista de manera concurrente, ya que la inserción puede traer problemas de condición de carrera. Por ejemplo, si una inserción se hace en medio de la ejecución de otra inserción, es posible que los nodos se pisen unos con otros al momento de decidir quién es la cabeza de la lista.

Si bien la función *insertar* se compone de un par de instrucciones, el agregado en sí lo hace atómicamente debido al compare and swap (*compare_exchange* para c++) loop de nuestra función. *Compare_exchange* es una función provista para las variables atómicas, el cual realiza 2 acciones (una u otra) de manera atómica dependiendo del estado de la comparación, si el compare es true, se escribirá el valor nuevo en la variable atómica, de lo contrario se escribirá el valor actual de la variable atómica en el valor viejo.

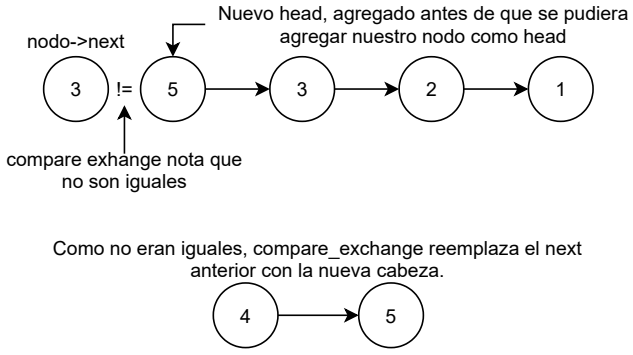
En nuestra implementación esta acción lo hace en un bucle constantemente, en donde busca que la cabeza de la *listaAtomica* en su estado 'actual' se corresponda con el nodo apuntado por nuestro nuevo nodo recién creado, si estos coinciden, pone como cabeza de la lista al nuevo nodo, terminando el loop y su ejecución; caso contrario, el nuevo nodo pasa a apuntar al nodo que apunta la cabeza, y mediante el loop vuelve a intentar el *compare_exchange* hasta que su comparación de true.

Esto previene cualquier tipo de *race condition*, ya que la atomicidad del CAS nos asegura que el cambio solo se efectuará antes o después de cualquier otro *insertar*, sin interrumpir ninguna otra inserción ni generando estados inconsistentes.

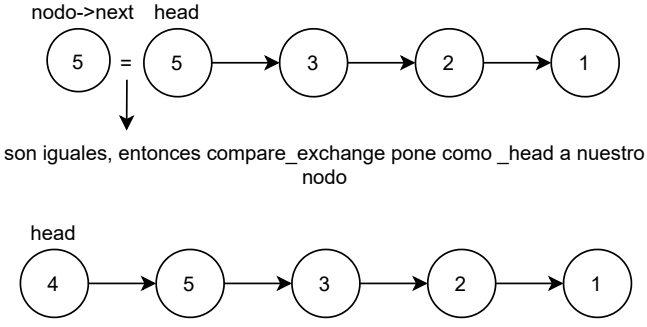
A modo de ilustración los gráficos muestran el proceso, dado que un elemento fue agregado antes que nuestro nodo. Contamos una lista con los elementos: 1, 2 y 3, con el 3 como cabeza de la lista; y queremos agregar el 4 al susodicho.



Inicio: creamos nodo N e inicializamos



Primer intento: falla



Segundo intento: éxito

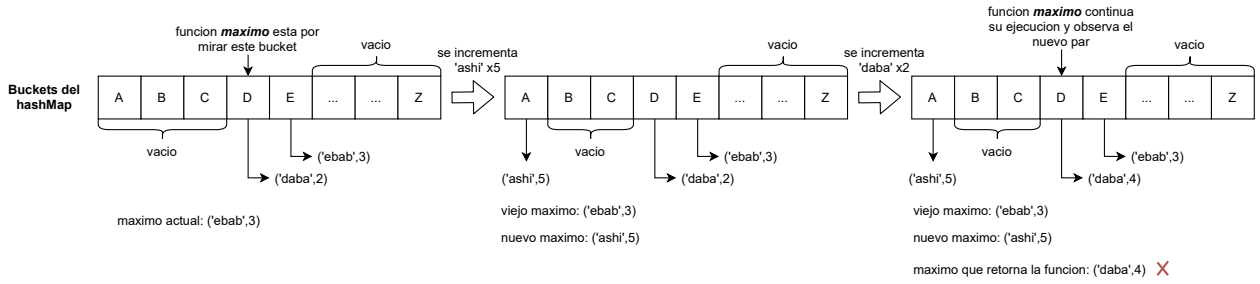
2.2. Implementación de *Incrementar, claves y valor* para hashMapConcurrente

Para la implementación de *incrementar*, se pide que haya contención solamente cuando hay colisión de índices, debido a esta exclusión mutua, optamos por utilizar *mutex* y, asimismo, como queremos que la exclusión sea solo cuando los índices coinciden, utilizamos un arreglo de *mutex* con la intención de diferenciarlos, uno por cada *bucket*.

Puesto que estamos utilizando *mutex* para las colisiones, no hay problemas de *race condition*, porque solo 1 hilo puede acceder a su sección crítica, y no nos preocupan los casos donde no hay colisiones, debido a que las inserciones se realizarían en listas distintas. Para las implementaciones de *claves* y *valor*, no utilizamos ninguna herramienta de sincronización. Al no usar herramientas de sincronización ni *syscalls* de ningún tipo, nos aseguramos de que ambas funciones sean no bloqueantes. A su vez, nos aseguramos de que ambas cumplan la propiedad *wait-free*, debido a que sólo deben recorrer la tabla y hacer sus operaciones correspondientes.

2.3. Modificación de *maximo* e implementación de *maximoParalelo*

Se tuvo que modificar *maximo* puesto que era posible ejecutar varios *incrementar* junto a *maximo* concurrentemente, y esto podría darnos un resultado inesperado e inconsistente. Por ejemplo, si para un *HashMapConcurrente* solo contamos con 2 elementos: (’daba’, 2) y (’ebab’, 3), supongamos que se esta por procesar (’daba’, 2), si en conjunto a *maximo* se logra ’incrementar’ 5 veces una clave anterior, como por ejemplo, ’ashi’, y luego se incrementa ’daba’ dos veces antes de que *maximo* lo procese. Podemos observar que (’daba’, 4) en ningún momento fue el máximo de la tabla, sin embargo, máximo procesó a partir de (’daba’, 4) y lo devolvió como máximo.



Hubo incrementos en conjunto a *maximo*, estos se hicieron antes de que *maximo* pudiera procesar el *bucket* D.

Como solución se extiende la interacción de *mutex* que hay en la implementación de *incrementar* junto a *maximo*. Primero, *maximo* lockea los *mutex* de todos los *buckets*, los mismos que usa *incrementar*, para así evitar inconsistencias por cualquier otro *incrementar*, luego en la medida que *maximo* va leyendo y comparando la información, este va desbloqueando los *mutex* de las listas que no va a volver a leer, permitiendo así que si un *incrementar* quiere usar ese *bucket*, pueda hacerlo. Lo que hacemos con esto es capturar el máximo del instante donde *maximo* bloqueó todos los *buckets*.

Para la implementación de *maximoParalelo*, en principio lockeamos los *buckets*, al igual que *maximo*, para evitar las mismas inconsistencias que este tenía. Por otro lado, la función en sí se encargará de crear los *threads* y ponerlos a calcular un máximo local a cada uno. Todos los *thread* comparten el *hashMap*, un atómico entero, el cual usaremos como índice; y una *listaAtomica* para máximos locales.

La idea es que cada *thread* calcule un máximo local para que, finalmente, una vez obtenidos todos los máximos locales, se comparen y de ahí obtener el máximo absoluto. Cada hilo trabajará sobre los *buckets* que le vaya asignando el índice atómico (el entero atómico ya mencionado) mediante un *fetchAdd*, este método es atómico y devuelve el valor actual del índice, y le suma al índice (en este caso 1), de manera tal que cada hilo obtenga un índice diferente, hasta recorrer todos los *buckets*.

Al igual que *maximo*, en la medida que el hilo termina con su *bucket*, este la desbloquea permitiendo que ese *bucket* ahora pueda ser modificado (por un *incrementar*, por ejemplo), y agrega el máximo obtenido a la lista atómica de máximos locales, para finalmente ser comparado una vez que todos los hilos terminen.

Hablando solamente sobre la sincronización: la contención para los hilos es realizada a través del índice atómico y el *fetchAdd*, que garantiza que cada hilo siempre tendrá un índice diferente sobre el que

trabajar, el unlock sobre el *bucket* cumple el mismo objetivo que el unlock de *maximo*, el insert de la listaAtomica garantiza que no habrá condición de carrera al agregar los máximos locales.

2.4. Completamos *cargarArchivo* e implementamos *cargarMultiplesArchivos*

Al implementar *cargarArchivo* no utilizamos ninguna herramienta de sincronización debido a que sólo debemos agregar las palabras del archivo a la tabla. La propia función *incrementar* se encarga de manejar la concurrencia.

Para la función *cargarMultiplesArchivos* utilizamos un vector de flags atómicos, uno por cada archivo a procesar. Usamos *testAndSet* para 'bloquear' los archivos y evitar que se procesen más de una vez.

Cada *thread* intenta acceder a cada archivo utilizando *testAndSet* en el flag correspondiente al archivo. El primer *thread* que bloquee un flag será el que procesará el archivo correspondiente a ese flag. En conjunto, los *threads* procesarán concurrentemente todos los archivos.

3. Experimentación y Resultados

Para comenzar con la experimentación, primero declararemos las características de la computadora donde se corrieron los experimentos, luego explicaremos cómo generamos las instancias de experimentación y las funciones para experimentar y, por último, las implementaciones de cada experimento, con sus análisis correspondiente.

Los experimentos fueron corridos desde una VM de VirtualBox corriendo una distribución de Linux Ubuntu 18.04 LTS que tiene asignados:

- 2 núcleos de un procesador Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz.
- 8gb de memoria RAM de 2666Mhz.
- Utilizando una ssd como disco principal.

3.1. Generador de Instancias

Para generar las instancias donde correr los experimentos, utilizamos las palabras del diccionario inglés[2] de la biblioteca *nltk* en python. Esta biblioteca contiene 236736 palabras, que guardamos en el vector *word_list*.

Para conseguir palabras repetidas y también aumentar la cantidad de palabras, seleccionamos aleatoriamente ciertas palabras[3] y las appendeamos al vector *word_list*, aumentando efectivamente su tamaño en una cantidad igual a las palabras agregadas. Luego, si queríamos utilizar menos palabras, realizábamos un shuffle del vector y tomábamos un rango que contuviera la cantidad que queríamos. El shuffle lo realizamos ya que aumenta las probabilidades de que haya palabras que comiencen con todas las letras en el corte que tomábamos.

Por último, este vector lo ordenamos de distintas maneras y separamos en distintos archivos dependiendo del experimento. Las distintas configuraciones son las siguientes:

- random: ordenamos la lista de manera al azar y luego la dividimos entre 1 y 50 archivos.
- sorted: ordenamos alfabéticamente la lista y luego la dividimos entre 1 y 50 archivos.
- sorted x archivo: ordenamos alfabéticamente la lista, la separamos con respecto a sus letras iniciales correspondientes y después distribuimos acordemente las palabras entre 1 y 50 archivos.

3.2. Implementación de los Experimentos

Para la experimentación, implementamos la función tiemposDeExperimento(), que se encarga de correr las distintas implementaciones y de calcular sus tiempos de ejecución. Esta función recibe como parámetros un HashMap, la cantidad de *threads*, un vector de archivos, la función de la que se quiere calcular el tiempo de ejecución, y un entero; este último representa la cantidad de veces que se correrá una función dada. La función tiemposDeExperimento() devuelve todas las mediciones de tiempo realizadas sobre la función pasada por parámetro. Para unificar la aridad de las distintas implementaciones de las que queremos medir su tiempo y poder llamarlas de manera sencilla desde tiemposDeExperimento(), utilizamos 4 funciones wrapper, una por cada implementación. Para la realización de las mediciones, utilizamos la biblioteca chrono de C++.

Siempre que utilicemos la función de tiemposDeExperimento(), tomaremos la mediana del vector resultante como dato a guardar. esto lo hacemos porque es el dato con menor sesgo e interferencia de los valores outliers. Aclararemos en cada experimento cuántas muestras tomamos para evaluar el tiempo de una función con determinados parámetros.

[2] Utilizamos las del diccionario inglés porque son acordes a la indexación de la estructura *hashMapConcurrente*.
[3] Generalmente, una cantidad de palabras igual a un múltiplo de la cantidad total del diccionario.

3.3. Experimento 1: Cargar Archivos en Relación a la cantidad de archivos y su distribución

La idea de este experimento es analizar el comportamiento de las funciones que cargan archivos con respecto a la cantidad de archivos a cargar, sobre las distintas configuraciones antes mencionadas. Para ello tomamos 39000 palabras ordenadas de las tres maneras (random, sorted y sorted x archivo) y por cada iteración las distribuimos acordemente en 1 archivo más, hasta llegar a que estén distribuidas en 50 archivos. Para la función CargarArchivo simplemente la ejecutamos una cantidad igual a la de los archivos en esa iteración. Y, para CargarMultiplesArchivos, utilizamos un nuevo *thread* por archivo. Utilizamos 30 mediciones en la función tiemposDeExperimento(), ya que las funciones de cargar archivos tardan un tiempo no negligible[4].

Notemos que en la iésima iteración de las corridas de experimentación, se corrieron los algoritmos con y sin concurrencia para las tres instancias. Por lo tanto, por cada iteración en la cantidad de archivos, se corrieron 6 mediciones en total.

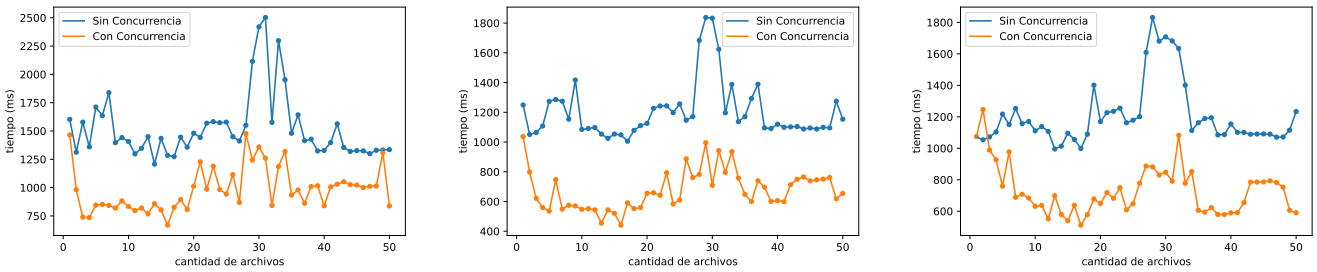
3.3.1. Hipótesis

Esperamos ver un comportamiento uniforme con respecto a cargarArchivo y las instancias ya que al no realizar la carga de forma concurrente, la distribución de la instancia no lo debería afectar. Asimismo, esperamos ver que la instancia sorted se comporte como el mejor caso para la función con varios archivos, ya que en general cada *thread* cargaría palabras que comiencen con letras distintas y no se interferirían entre sí en el *HashMapConcurrente*.

De la misma manera, la distribución de sorted x archivo debería comportarse como el peor caso, ya que los *threads* tendrían partes de igual cantidad de palabras por letra, lo que llevaría a interferencia entre sí. En particular, creemos que mientras menor sea la cantidad de *threads* con las que trabajar, más se incrementa la probabilidad de que se interfieran, y mientras mayor sea la cantidad de *thread* menos incidencia tendrá la distribución.

Por último, creemos que la instancia random obtendrá resultados que se encuentran entre medio de las otras dos, porque las probabilidades de que se quieran agregar dos palabras en el mismo *bucket* consistentemente bajan por la distribución aleatoria. En general, creemos que todas la configuraciones de cargarMultiplesarchivos se comportarán de manera más rápida que cargarArchivo, por la capacidad de distribuir la carga en más de un hilo y así aprovechar más recursos de los que ofrece el procesador.

3.3.2. Resultado



(a) Instancia Random (b) Instancia Sorted (c) Instancia Sorted X Archivo

Figura 3: Comparaciones por instancia de las funciones cargarArchivo y cargarMultiplesArchivos

Hay varias cosas para analizar de los resultados, ya que tenemos propuestas que se confirmaron y otras que no.

En primer lugar, indaguemos en las hipótesis que se confirmaron. La instancia Sorted efectivamente demostró tener los mejores resultados, creemos, debido a la explicación presentada anteriormente. Otra hipótesis que se confirmó es el comportamiento de cargarMultiplesArchivos en la instancia de Sorted x

[4]En el orden de los segundos.

Archivo. Si se compara la velocidad a la que baja el tiempo en relación con la cantidad de archivos (otra manera de leerlo podría ser el tiempo en relación a la cantidad de *threads*, en este caso) de esta instancia con la de las otra dos, se puede ver que esta llega a su punto de establidad después de los 10 archivos, mientras que las otras lo alcanzan alrededor de las 5 archivos. Esto confirma el hecho de que cuando se utilizan pocos *threads* en esta instancia, esos pocos *threads* se interfieren entre sí por estar intentando cargar palabras en el mismo *bucket*. Por último, se confirmó la idea general de que el algoritmo que aprovecha el paralelismo se comporta mejor que el que no. La única excepción es la instancia sorted x archivo, donde con dos archivos tarda más que el algoritmo sin concurrencia pero esto se explica con la interferencia antes mencionada.

En segundo lugar hablemos de la hipótesis que se demostró incorrecta. Con esto nos referimos a los tiempos que tardó la instancia random. Esta instancia obtuvo los peores tiempos de las tres, y creemos que es por la misma razón porque pensamos que sería la mejor. Las otras intancias agrupan las palabras en relación a su orden, elemento que tiene mucho efecto en la concurrencia del algoritmo. Al tener archivos que trabajen distintas letras, entre ellos no se interfieren y si hay muchos archivos que tienen el mismo orden, a medida que se desincronicen los *threads*, cada uno operará con palabras distintas. Por otro lado, la distribución random no garantiza nada de esto, por lo que en realidad no podemos afirmar nada con respecto a la probabilidad de que se intenten cargar palabras del mismo tipo en el mismo *bucket*.

Por último, analicemos la irregularidad de los gráficos. Los tres gráficos presentan un pico alrededor de los 30 archivos las dos veces que corrimos el experimento (fueron dos porque la primera vez nos pareció un fenomeno raro). Nosotros no creemos que se deba a un efecto de la cantidad de archivos sobre las funciones, si no que creemos que es atribuible a algunas variables que no tuvimos en cuenta. Este experimento tardó horas en ejecutarse, y se ejecutó en una laptop con un procesador de frecuencia de clock variable y con un mal sistema de *cooling*. Si bien no creemos que se haya experimentado un *thermal throttle*, creemos que la frecuencia variable y la temperatura pudieron haber afectado las mediciones mas haya de determinado tiempo.

3.4. Experimento 2: Incidencia de los *threads* en maximo Paralelo

La idea de este experimento es analizar la relación entre la cantidad de *threads* que se usan para calcular el máximo y el tiempo que tarda. Idealmente, buscamos encontrar una cantidad óptima de *threads* y una cantidad pésima para utilizar como cota superior e inferior en el siguiente experimento, donde compararemos con la versión de maximo que no utiliza paralelismo.

Este experimento fue implementado de la siguiente manera: primero, creamos un *HashMapConcurrente* con un conjunto de archivos que contienen en total alrededor de 710000 palabras aleatorias, dejando al *HashMapConcurrente* con la mayor cantidad de palabras posibles. Luego, corrimos la función de máximo paralelo una vez por cada cantidad de *threads* que permite, es decir, 26 veces en total, con la función de tiempos tomando 200 mediciones por cada cantidad de *threads*.

3.4.1. Hipótesis

Esperamos ver un máximo y mínimo local, es decir, esperamos que exista una incidencia entre la cantidad de *threads* y el tiempo de cómputo de la función. Lo que no sabemos, y nos gustaría responder, es si tienen una relación conocida, por ejemplo una relación lineal. Tampoco sabemos cual será el mínimo local, aunque intuimos que el máximo será 1 *thread* ya que no aprovecha el paralelismo de la función.

3.4.2. Resultado

Como se puede apreciar en la Figura 4, acertamos que el máximo se daría en uno. Esto confirma lo que planteamos en la hipótesis. El mínimo se dio en 4 *threads*, lo que, teniendo en cuenta que en la VM 2 núcleos físicos del procesador con hyper*thread*ing (permitiendole atender a 2 subprocesos por nucleo) en cuestión son tomados como 4 nucleos lógicos, hay una relación entre el mínimo y la cantidad de cores o hilos que el procesador otorga. Utilizar más de 4 *threads* para realizar la busqueda del máximo parecería aumentar el tiempo que tarda de forma lineal.

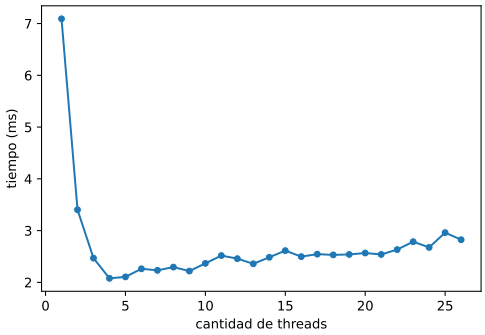


Figura 4: Tiempo de Computo por cantidad de *threads* de la Función MaximoParalelo

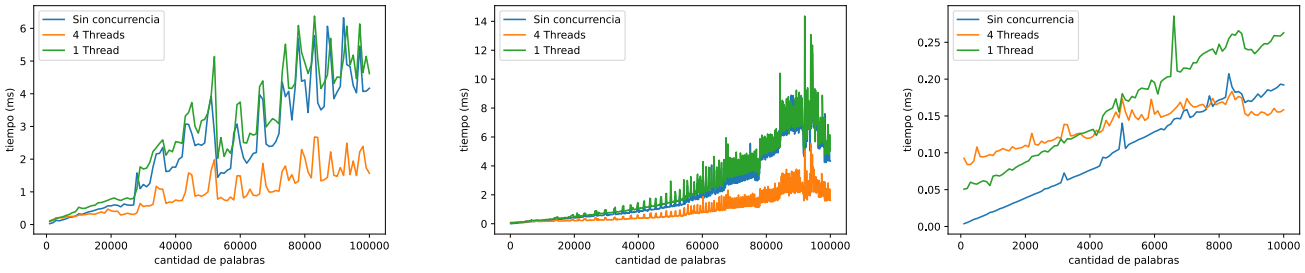
3.5. Experimento 3: Maximo paralelo vs maximo normal

La idea de este experimento, como ya se dijo anteriormente, es comparar el efecto que trae consigo el uso del paralelismo. Para ver eso, vamos a ir agregando de a 1000 palabras a un hashmap hasta llegar a 100000 palabras y en cada iteración vamos a calcular el máximo con la función sin concurrencia, con la función paralela con un *thread* y con la función paralela con 4 *threads*. Esto nos va a dejar comparar al maximoParalelo con su mejor y peor parámetro con la función sin concurrencia. Luego, realizaremos lo mismo pero agregando de a 100 palabras para ver una progresión más detallada. Notemos que en cada iteración, se calculan los tres máximos, uno después del otro y que utilizamos 1000 mediciones por cada función.

3.5.1. Hipótesis

Esperamos ver que para *HashMapConcurrentes* chicos[5], la función sin paralelismo sea la más rápida, ya que no tiene que esperar a que se le otorguen *quantums* a los *threads*. A medida, que el tamaño del *HashMapConcurrente* crezca, esperamos ver que la implementación con paralelismo de 4 *threads* sea la más veloz. En general, creemos que la función con paralelismo de un *thread* va a ser siempre más lenta que la función sin concurrencia y que se diferenciarán por un valor constante que se le podría atribuir a esperar que se le asigne el quantum al *thread* correspondiente. Creemos esto porque ambas realizan lo mismo, pero la paralela se lo asigna a un *thread* y no lo realiza ella misma.

3.5.2. Resultado



(a) Aumentando de a 1000 palabras (b) Aumentando de a 100 palabras (c) Zoom al principio de la figura

Figura 5: Maximo Sin Concurrencia VS maximoParalelo con 1 *thread* VS maximoParalelo con 4 *threads*

Como se puede apreciar en las figuras, en este experimento se cumplieron las hipótesis planteadas. Eso demuestra que nuestras explicaciones estaban en lo correcto. Sin embargo, algo para tener en cuenta es lo ruidosa que es la muestra. Tenemos dos teorías de porque puede estar sucediendo esto. La primera

[5]Con poca cantidad de palabras.

es que se debe a la interferencia de otros procesos ajenos al experimento; teniendo en cuenta que tardan milésimas de segundos, cualquier cambio puede resultarles porcentualmente grande, además, teniendo en cuenta que los tres máximos se calculan en la misma iteración, uno después del otro, tiene sentido que los tres presenten los picos en los mismos momentos. La segunda teoría, en el caso de que no sea la primera, es que se deba a una relación entre el máximo y la cantidad de palabras; si bien no la creemos tan probable.

4. Conclusión

Hay dos temas en los cuales queremos hacer hincapié para concluir el trabajo. El primero es nuestra reflexión acerca de la contención y el uso de variables y recursos compartidos. El segundo es acerca del aprovechamiento de los recursos del procesador para optimizar algoritmos.

En primer lugar, queremos mencionar que en un sistema operativo, en el cual pueden existir varios procesos que pueden intentar acceder a una misma variable en distintos tiempos (o al mismo tiempo en el caso de procesadores de más de un núcleo) accesos no controlados pueden derivar en que se invalide la variable, como se explica en la sección 2.1. Como una variable no confiable invalida todos los algoritmos que la utilicen, es claro que el problema de contención es un eje fundamental de la programación de sistemas operativos. Al mismo tiempo, hay tener en cuenta que la contención es un limitante de la eficiencia de los algoritmos, ya que hay que esperar turnos si se están usando *mutex* o esperar a un valor específico en el caso del *compare_exchange*. Esto se puede ver en la motivación de la sección 2.3. Si bien no experimentamos sobre el máximo sin contención, es claro que al no tener *mutex* es marcadamente más veloz que al tenerlos, pero sus resultados se volverían inconsistentes.

Sin embargo, hay una manera de trabajar alrededor de las limitaciones mencionadas y mejorar la *performance*. Esta es el uso de *threads*, aprovechando así los múltiples recursos que nos ofrece el procesador. En este trabajo, el principal ejemplo es el maximoParalelo. Al agregar contención a la función de máximo inconsistente, se debía resolver, de un *bucket* a la vez, los máximos locales. Como se ve en el experimento 2 y 3, al agregar *threads*, se pueden aprovechar hasta los 4 hilos (producto de los 2 nucleos con hyper*thread*ing) que tenía la VM y así mejorar su performance. No queremos dejar de mencionar que el uso de *threads* no se limita a mejorar la *performance* en algoritmos que tienen contención. Este fue un caso particular, pero su uso para optimizar algoritmos es mucho más amplio, como por ejemplo utilizarlos para realizar más de una operación compleja necesaria en una función concurrentemente.

Por último, queremos relacionar estos conceptos con una frase que repetimos mucho en las clases, 'los problemas en los sistemas operativos se tienen que resolver bien y rápido'. Bueno, para que estén bien, tiene que haber contención, y para que sea rápido, hay que utilizar todos los recursos disponibles.