DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Operativos: Trabajo Practico 1

October 31, 2021 Sistemas Operativos

Grupo 10

Integrante	LU	Correo electrónico
Chami, Uriel Alberto	157/17	uriel.chami@gmail.com
Strobl Leimeter, Matias Nicolás	645/18	matias.strobl@gmail.com
Fabian, Florencia	230/19	flor.fabian@hotmail.com

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

Contents

1 Introducción

El objetivo del presente trabajo consiste en el análisis de la gestión de concurrencia, una de los principales características que manejan los sistemas operativos. Para ello, se analizarán las técnicas para gestionar la contención sobre los recursos con el fin de evitar que se produzcan las denominadas condiciones de carrera, es decir, de evitar un comportamiento en el cual la salida dependa de un orden de ejecución que no se encuentra bajo control y que pueda provocar resultados incorrectos.

Este trabajo estará particularmente centrado en el uso de threads, que son una herrmienta muy importante provista por el sistema operativo que permite disponer de varios hilos de ejecución concurrentes dentro de un mismo programa.

2 Desarrollo

En la presente sección se explicarán el uso tanto de las estructuras ya definidas, como así tambien de las implementaciones de las funciones pedidas.

2.1 Lista atómica

La lista atómica consiste de una lista enlazada que luego será utilizada para almacenar los elementos de cada bucket de la tabla de hash.

La función principal con la que se debía lidiar era la de insertar, que recibía un valor por parámetro, y debía encargarse de crear un nodo con dicho valor y agregarlo al comienzo de la lista. El principal requerimento con el que se debía contar era que dicha función sea atómica ya que se debe asegurar que la inserción se realice en su totalidad o, que en el caso de ser interrumpida, pueda deshacer sus acciones de modo que fuese como si no hubiera realizado ninguna. Es decir, se debe procurar que si en la mitad del proceso de inserción surge alguna interrupción que provoque que dicho proceso no pueda ser finalizado, esto no modifique a la lista atomica.

Algorithm 1 Pseudocódigo insertar

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

Para lograr esto de manera efectiva, la función primero crea al nodo por fuera de la lista enlazada con el valor recibido por parámetro. Lo siguiente es asignarle el nodo que será su siguiente, que lógicamente será el primer nodo de la lista. Nótese que hasta aquí la lista no ha sido modificada, ya que el puntero a la cabeza de la misma sigue apuntando al mismo nodo que antes.

Para que el nuevo nodo esté insertado en la lista, la cabeza debe apuntar al nuevo nodo y el nodo que anteriormente cumplía el rol de cabeza debe ser el siguiente de la nueva cabeza. Lo que podría suceder es que tengamos más de un nodo queriendo ser insertado, en cuyo caso, ambos tendrán como siguiente a la cabeza ya que esas asignaciones no son atómicas. La funcion compare_exchange_weak nos asegura que si el newNode está apuntando a siguiente, entonces la cabeza sera asignada y por el contrario si el siguiente de newNode quedó apuntando a un lugar incorrecto (es decir, había dos nodos apuntando a la misma cabeza) se reapunte si siguiente a la nueva cabeza todo esto se hará de manera atómica. Con lo cual, se concluye que efectivamente la operación es atómica y además produce el resultado que buscamos.

Y como insertar es la única operación que modifica la lista, esta última entonces será atómica.

Sin embargo, a pesar de que dicha estructura es atómica, no queda protegido de condiciones de carrera, es decir que no garantiza que aplicando múltiples operaciones en cierto orden, el resultado respetará dicho orden.

2.2 HashMapConcurrente

Se cuenta con una estructura de datos llamada HashMapConcurrente, la cual consiste en una tabla de hash que gestiona las colisiones usando listas atómicas. Esta estructura se usará para procesar archivos de texto contabilizando la cantidad de apariciones de las palabras. Su interfaz de uso es la de un map donde las claves son las letras desde la 'a' a la 'z' (primera letra de la palabra) y los valores están representados por una ListaAtómica del tipo hashMapPair, siendo este una tupla de la palabra con su cantidad de apariciones.

La función "incrementar" toma de parámetro un string, y recorre la lista atómica correspondiente a la clave hash comparando la palabra de los hashMapPair con el string dado. Si se encuentra una coincidencia se incrementa la cantidad de repeticiones, si no, se inserta en la lista atómica el par <palabra, 1>.

A continuación colocamos el pseudocódigo de esta función para facilitar la explicación.

Algorithm 2 Pseudocódigo incrementar

```
1:  function incrementar(clave)
2:      claveHash ← hashIndex(clave)
3:      disponibilidadPorLetra[claveHash].lock()
4:      for &p : *tabla[claveHash] do
5:          if p.first == clave then
6:              p.second++;
7:              disponibilidadPorLetra[claveHash].unlock()
8:              return
9:      nuevaEntrada ← hashMapPair(clave, 1)
10:     a ← tabla[claveHash]
11:     a.insertar(nuevaEntrada)
12:     disponibilidadPorLetra[claveHash].unlock()
13:     return
```

2.2.1 Manejo de la concurrencia

DisponibilidadPorLetra es un arreglo de mutex, con un mutex por cada una de los hashes (letras). Nos piden garantizar la contención solo en caso de colisión de hash. Al hacer el lock del mutex de la clave hash correspondiente antes de acceder a la lista atómica, permite que solo un thread recorra esa lista a la vez, pero puede haber tantos threads incrementando concurrentemente como buckets en la tabla.

Con esta implementación el código está libre de condiciones de carrera. Evitamos usar un único mutex para no generar más contención de la pedida, así con la idea del arreglo sólo se bloquean los threads en caso de colisión de hash. Antes de que la función retorne se produce el unlock del mutex evitando así posibles deadlocks.

La funcion "claves" se encarga de recorrer toda la tabla, agregando en un vector de strings todas las claves de cada lista atómica y retornandolo.

Dada una clave, la funcion "valor" recorre la lista atómica correspondiente a su primer letra y compara las claves de los hashMapPair con la dada. En caso de encontrarla devuelve la cantidad de apariciones, si no existe retorna 0.

Tanto en claves como en valor no hace falta garantizar consistencia en caso de que se ejecuten junto con incrementar, por lo que se evitó el uso de primitivas de sincronización obteniendo así concurrencia máxima y siendo ambas no bloqueantes y libres de espera.

Para la función máximo, nos daban un código que calculaba el máximo de toda la tabla sin usar ninguna primitiva de sincronización. Esto permitía que máximo e incrementar se ejecuten concurrentemente.

Algo que podía pasar, es que máximo devuelva un resultado que nunca fue el máximo de la tabla. Miremos el siguiente ejemplo:

- incrementar(ARBOL)
- incrementar(JARABE)
- incrementar(ARBOL)
- Se inicia el cálculo del máximo. Este recorre la lista atómica correspondiente a 'a', y guarda el nuevo máximo 'ARBOL' con 2 apariciones. Mientras recorre la lista 'b' se producen las siguientes llamadas a incrementar:
- incrementar(ARBOL)
- incrementar(ARBOL)
- incrementar(JARABE)
- incrementar(JARABE)

- Una vez que máximo termina de recorrer la tabla, devuelve 'JARABE'. Este en ningun momento fue un máximo válido.

La solución a este problema fue hacer un lock de todos los mutex de disponibilidadPorLetra antes de acceder a la tabla, evitando que más de un thread tenga acceso a ésta a la vez.

Así evitamos las inconsistencias producidas por ejecutar incrementar y máximo concurrentemente.

2.2.2 Funcion maximoParalelo

La función maximoParalelo procesa de manera paralela el máximo del HashMap con la cantidad de threads pasados por parámetro. Para la implementación de la misma, se utilizaron recursos compartidos, como un puntero a la próxima lista a recorrer para encontrar el máximo, y un mutex llamado afectarMaximo que controla el acceso a donde se almacena el valor máximo, dicha primitiva garantiza la exclusión mutua al modificar esta variable, de esta manera evitando condiciones de carrera.

Un detalle a considerar de la implementación de maximoParalelo llevada a cabo es que este devolverá el máximo del momento exacto en el cual se empezó a ejecutar la función, que no necesariamente va a concidir con el máximo del momento en el cual se retorna el resultado.

Aclaramos que hay pérdida de memoria en varias funciones de hashMapConcurrente que se solucionaría con un destructor de esta clase. De todas formas consideramos que no es algo que se evalua en este trabajo por lo que lo omitimos.

2.3 Carga de archivos

cargarArchivo es una función que lee un archivo pasado por parámetro y carga todas sus palabras en el HashMap también pasado por parámetro. Ésta utiliza la función incrementar, que como se dijo anteriormente, garantiza que hay contención en caso de colisión de hash.

La función cargarMultiplesArchivos está basada en los mismos principios de maximoParalelo.

Algorithm 3 Pseudocódigo cargarMultiplesArchivos

```
1: function cargarMultiplesArchivos(hashMap, cantThreads, filePaths)
2:     atomic<int> ultimoFileLeido(0)
3:     threads[cantThreads]
4:     for 0 < i < cantThreads do
5:         threads[i] ← thread(codigoThread, ultimoFileLeido, filePaths, hashMap)
6:     for 0 < i < cantThreads do
7:         threads[i].join()
```

Para dicha implementación, se utilizó un recurso compartido denominado ultimoFileLeido, que particularmente cumple el mismo rol que indexLetraSinCalcular en la implementación de maximoParalelo.

3 Experimentacion

3.1 Aclaraciones generales

3.1.1 Promediado de las soluciones

Para tener más fiabilidad en los datos obtenidos, se ejecutarán 5 veces cada prueba y se promediarán los resultados de esa forma evitando outliers producidos por situaciones que se escapan de control (como injusticia en el scheduler)

3.1.2 Medición del tiempo

En esta ocasión, debido a problemas técnicos con el uso de clock_get_time utilizamos la librería std::chrono de la siguiente forma para obtener todos los tiempos que estudiamos a continuación

```
// Cargar archivos en paralelo

std::chrono::time_point<std::chrono::system_clock> startArchivos, endArchivos;
startArchivos = std::chrono::system_clock::now();

cargarMultiplesArchivos(hashMap, cantThreadsLectura, filePaths);

endArchivos = std::chrono::system_clock::now();
std::chrono::duration<double> tiempoArchivos = endArchivos - startArchivos;
printf( "%lf ", tiempoArchivos.count());


//Calcular maximo paralelo

std::chrono::time_point<std::chrono::system_clock> startMax, endMax;
startMax = std::chrono::system_clock::now();

auto maximo = hashMap.maximoParalelo(cantThreadsMaximo);

endMax = std::chrono::system_clock::now();
std::chrono::duration<double> tiempoMax = endMax - startMax;
printf( "%lf ", tiempoMax.count());
```

3.1.3 Especificaciones técnicas

Para asegurar que los tiempos sean comparables entre sí todos los experimentos sin excepción fueron ejecutados en el siguiente equipo:

- **Model:** MacBook Pro (16-inch, 2019)
- **Processor:** 2,3 GHz 8-Core Intel Core i9
- **Memory:** 16 GB 2667 MHz DDR4
- **Startup Disk:** Macintosh HD
- **Graphics:** Intel UHD Graphics 630 1536 MB

3.2 Carga de archivos - Aclaraciones/Introducción

Antes de hablar de las hipótesis, es importante aclarar las condiciones en las que pondremos a prueba este algoritmo.

3.2.1 Cantidad de archivos

Si la cantidad de archivos es inferior a la cantidad de threads, debido a que cada thread toma un archivo completo y lo lee, tendremos threads que serán creados sólo para luego ser destruidos sin haber hecho ningún trabajo computacional extra. Esto no es siquiera una hipótesis, es un hecho.
Por este motivo trabajaremos siempre con la $CantidadDeArchivos > CantidadMaximaDeThreads$ es decir, si $threads = 1, 2, .., 30$ entonces $CantidadMaximaDeThreads = 30$ esto significa que todas las ejecuciones procesarán $CantidadDeArchivos$ donde $CantidadDeArchivos \geq 30$ aclararemos en cada caso cuántos archivos son ejecutados.

3.2.2 Longitud de los archivos

La longitud de los archivos se refiere a la cantidad de palabras a cargar en el HashMap. Respecto de esta variable se utilizarán 3 variantes para la investigación: *corpus*, $test_1$ y *merge*, las primeras dos siendo las provistas por la cátedra y la tercera siendo la consecuencia de juntar $test_1$, $test_2$ y $test_3$. *corpus* posee 2498 entradas, $test_1$ cuenta con 6 y *merge* con 30. Más adelante se estudiarán las implicancias y particularidades que tiene cada data-set.

3.3 Carga de archivos - Hipótesis

Lo que inicialmente puede pensarse es que si se divide el trabajo entre más procesos, indudablemente, se tardará menos. Esto muestra no ser así en la vida real, donde 5 albañiles pueden levantar un muro 5 veces más rápido que un único albañil pero 100 de ellos no levantarán el muro mucho más rápido porque además se necesitará de alguien que se dedique exclusivamente a organizar tantas tareas. Lo que tratamos de observar es que existe un límite de subdivisión del trabajo en el que la mera organización de tareas es tan costosa en tiempo que no vale la pena sumar más recursos porque en efecto seremos más lentos (además de más caros). Es por esto, que pasada esta intuición inicial, nuestra hipótesis es que veremos una mejoría en el rendimiento directamente proporcional a la cantidad de threads hasta cierto punto, pero pasado ese punto, agregar threads estancará o empeorará el rendimiento.
En términos de código, estos *costos organizativos* son por un lado la creación y la gestión en general de los threads por parte del sistema operativo. Además de tener syscalls, los cambios de contexto y todo lo que implica para el sistema operativo como carga extra que existan n ramas de ejecución en vez de una, en segundo punto los recursos compartidos, particularmente en la carga de archivos hay dos recursos compartidos: la variable atómica $ultimoFileLeido$ y el HashMap. La variable atómica se aumenta cada vez que un thread comienza la lectura de un archivo, mientras que el HashMap se utiliza cada vez que se lee una palabra, sin dudas la fuente más grande de serialización.

3.4 Carga de archivos - Experimento 1 - *corpus* hasta 30 threads

En este caso ejecutaremos la carga de 30 copias de *corpus* en paralelo con $threads = 1, 2, ..., 30$.

3.4.1 Análisis de resultados

En la figura 1.a se puede observar claramente cómo el agregado de threads mejora indudablemente el rendimiento. Lo que se debe observar es que la cantidad de palabras en el HashMap es $2498 \times 30 = 71940$, es decir que nos encontramos frente a un trabajo computacional *grande*. Volviendo a la analogía de los albañiles, si tuviesemos que construir un edificio completo, probablemente 100 obreros sigan siendo más apropiados que 5 aunque haya que organizarlos.

3.5 Carga de archivos - Experimento 2 - $test_1$ hasta 30 threads

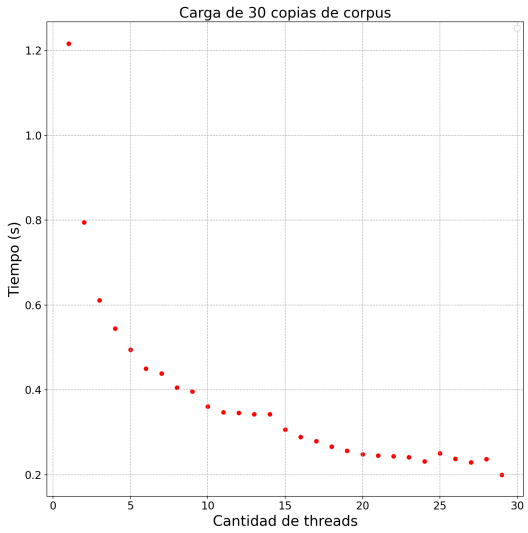
Ejecutaremos la carga de 30 copias de $test_1$ en paralelo con $threads = 1, 2, ..., 30$. En este caso, hipotetizamos encontrar ese punto de disminución del rendimiento agregado por cada thread ya que la carga es mucho más pequeña ($6 \times 30 = 180$ palabras)

3.5.1 Análisis de resultados

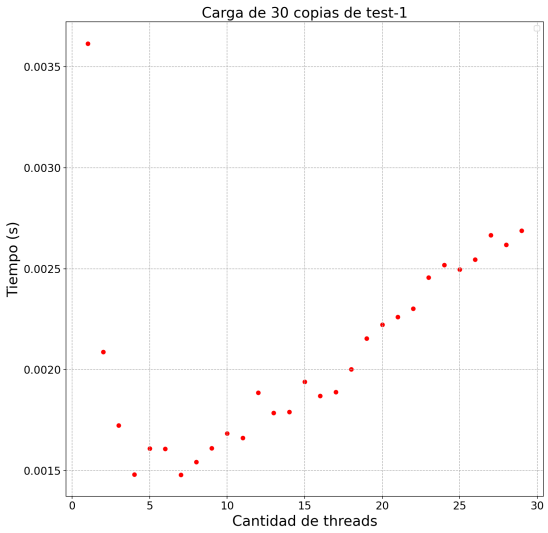
Podemos ver en la figura 1.b que efectivamente el rendimiento mejora radicalmente entre 1 y 5 pero luego de 5 todas las otras cantidades de threads **producen más overhead que beneficio**.

3.6 Carga de archivos - Experimento 3 - *merge* hasta 30 threads

Luego de haber visto un ejemplo claro de que "mucho trabajo repartido entre muchos actores tiene buen rendimiento" y otro de "poco trabajo repartido entre muchos tiene mal rendimiento", nos gustaría terminar de validar las hipótesis observando un data-set un poco más balanceado con 30 entradas y leyendo 90 instancias de las mismas. Es decir $30x90 = 2700palabras$ esto constituye un 3% de la muestra *corpus* leída 30 veces. Consideramos que esta visión intermedia nos permitirá ver con toda claridad el patrón de mejora hasta un punto de deminish of returns y comienza a empeorar. Esperamos encontrar este patrón de forma menos abrupta.



a Carga de 30 copias de *corpus* en paralelo con $threads = 1, 2, ..., 30$.

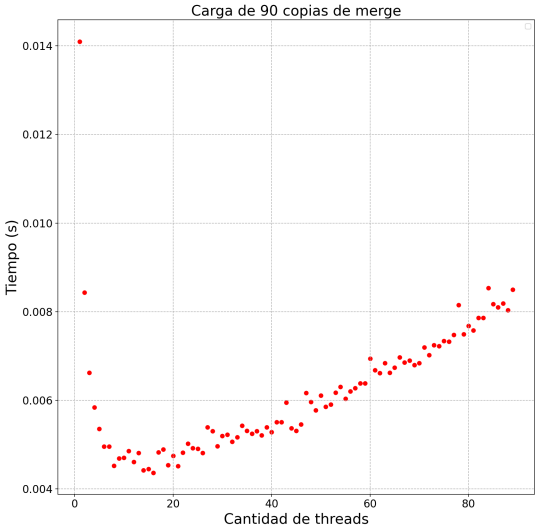


b Carga de 30 copias de $test_1$ en paralelo con $threads = 1, 2, ..., 30$.

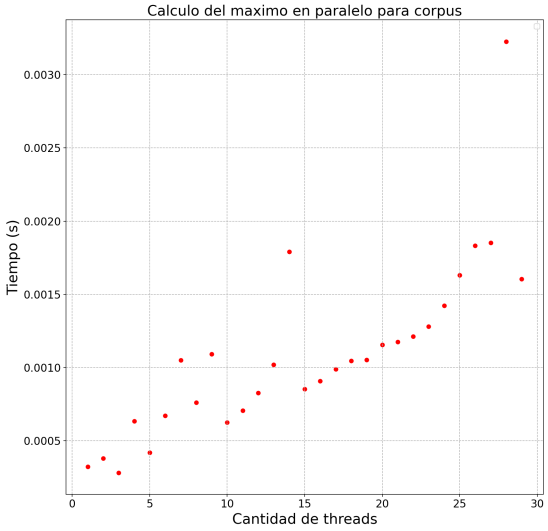
Figure 1

3.6.1 Análisis de resultados

Podemos ver claramente en la figura 2.a lo que buscábamos. En el rango de 10 a 20 threads el rendimiento es óptimo mientras que si aumentamos la cantidad de threads el rendimiento paulatinamente disminuye hasta que llegamos a valor similares (y hasta superiores) que haciédolo con un único thread.



a Carga de 90 copias de *merge* en paralelo con $threads = 1, 2, ..., 90$.



b Calculo del maximo en paralelo con $threads = 1, 2, ..., 30$ para el data-set *corpus*.

Figure 2

3.7 Máximo - Aclaraciones/Introducción

3.7.1 Limite de 26 threads

Debido al funcionamiento del algoritmo de *maximoParalelo* la máxima cantidad de threads activos y útiles al mismo tiempo es 26 ya que cada uno toma una lista asociada a un hash y nuestra función de hash opera en módulo 26. Por este motivo y similarmente a *cargarArchivos* nos concentraremos en el estudio de los valores $threads = 1, 2, ..., 26$

3.7.2 Máximo - Diccionario

Fue descargado un diccionario de internet de la siguiente fuente. Se cortaron y se modificaron levemente debido a los límites de memoria de la computadora. Los archivos modificados y usados para la experimentación se pueden encontrar en la carpeta /data del proyecto entregado. Se utilizaron las letras desde la a hasta la m.

3.8 Máximo - Hipótesis

Similar a lo sucedido en *cargarArchivos* consideramos que existirá un punto de disminución del rendimiento debido a un **overhead** producido por el costo de sincronización y de la creación/gestión de threads. En este caso los puntos principales de congestión serán el *indexLetraSinCalcular* pero por sobre todo el mutex *afectarMaximo* que será pedido por todos los threads cada vez que encuentren un nuevo máximo.

Una aclaración a hacer es que en el mejor de los casos el primer valor es máximo y nosotros tenemos una mejora de eficiencia que aumenta el rendimiento en dicho caso. Y si consideraramos una muestra balanceada, en promedio sucederá que el 50% de las palabras no requerirán tomar un mutex.

De todos modos no cambiaremos el balance de las muestras, para preservar esta condición y que las métricas no engañen.

3.9 Máximo - Experimento 1 - *corpus* con 26 threads

Analizaremos la búsqueda del máximo con $threads = 1, 2, ..., 26$ en el data-set *corpus* de 2498 entradas. Esperamos ver una mejoría en el rendimiento con cada thread agregado debido al volumen de los datos.

3.9.1 Análisis de resultados

En la figura 2.b podemos ver lo sucedido para *corpus*, nuestro data set más grande. Esto nos dice que nuestra implementación de búsqueda de máximo no performa nada bien al aumentar la cantidad de threads. 2 o 3 threads parecieran ser la cantidad óptima pero tampoco hay una diferencia enorme con ejecutarlo en un único thread. Si *corpus* muestra este comportamiento, entonces los resultados de $test - 1$ y *merge* son predecibles. El escalado será aún peor.

3.10 Máximo - Experimento 2 - data-set extra grande con 26 threads

Estos resultados pueden significar 2 cosas:

- Que nuestro manejo de la concurrencia es malo
- Que el data-set no es suficientemente grande como para que se pueda apreciar una mejora.

Probaremos con un data-set aún no utilizado hasta este momento: *diccionario*. Este data-set es más voluminoso que *corpus* en términos de cálculo de máximo ya que tiene más palabras *distintas*, aproximadamente 10.000

3.10.1 Analisis de resultados

Podemos ver como frente a un gran volumen de palabras el uso de múltiples threads muestra ser superior en rendimiento. Sin embargo al superar el mejor candidato (5 threads) la cantidad de threads se correlaciona negativamente con el rendimiento, llegando al caso más extremo donde la cantidad mínimo de concurrencia (1 thread) toma el mismo tiempo que la cantidad máxima (26 threads).

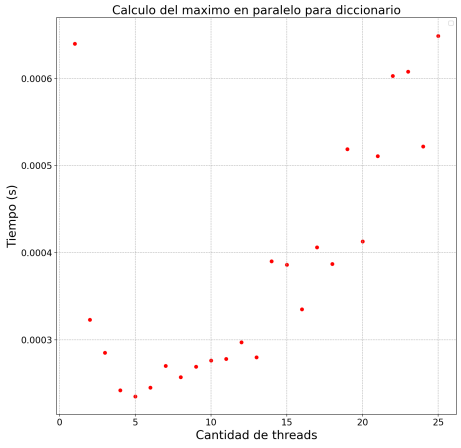


Figure 3: Calculo del maximo en paralelo con $threads = 1, 2, ..., 26$ para el data-set *diccionario*.

4 Conclusiones

La concurrencia y el trabajo paralelizado es un herramienta sumamente poderosa, muchas de las tareas de **alta carga de procesamiento** podrían sacar provecho de dividir dicho prcoesamiento entre algunos threads.
Sin embargo, aunque sea una solución tentadora, paralelizar tiene un costo, tanto en rendimiento como en términos de horas de desarrollo. Para ejecutar procesamientos en paralelo requerimos cierta sincornización entre estas particiones, esta sincronización no sólo debe ser pensada y trabajada además de testeada (y este es un punto sensible, el testeo de ejecuciones paralelas es muy complejo), si no que también cuando ejecutamos el programa, las condiciones de sincornía regulan el rendimiento de dicho proceso.

Como si esto fuera poco, muchas veces el dominio mismo del problema nos establece particiones mínimas y éstas a su vez la cantidad máxima de threads, como es el caso de las listas por cada hash en el HashMap-Concurrente, aunque técnicamente sea posible dividir este recurso en términos prácticos es mucho más sencillo mantenerlo como recurso de uso exclusivo.
Está claro que más threads no es siempre mejor, se trata de un equilibrio.