DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: Threading

Sistemas Operativos

Integrante	LU	Correo electrónico
Torsello, Juan Manuel	248/19	juantorsello@gmail.com
Lavalle Cobo, Ignacio	282/19	ignacio.lavalle@gmail.com
Yazlle, Maximo	310/19	myazlle99@gmail.com
Escudero, Federico Martín	788/19	fedeescudero301@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 este trabajo se buscará profundizar sobre uno de los problemas principales que aparecen al estudiar los sistemas operativos: el manejo de la concurrencia. Analizaremos la ejecución concurrente de programas y las técnicas para gestionar la contención sobre los recursos. Para ello se brindará una nueva implementación de una de las estructuras de datos más populares, el HashMap. A nivel funcional no hay grandes diferencias entre nuestra implementación y otras, la interfaz es la típica de un diccionario. La particularidad de esta tabla de Hash es la posibilidad de realizar operaciones concurrentemente.
Para trabajar sobre el problema de la concurrencia utilizaremos tres herramientas principales: threads, mutex y operaciones atómicas.

- Un **thread** es una herramienta provista por los sistemas operativos que nos permite disponer de varios hilos de ejecución concurrentes, dentro de un mismo programa.
- Un **mutex** es un tipo específico de semáforo que sirve para sincronizar el acceso de los distintos threads a un recurso compartido, dándonos exclusión mutua.
- Las **operaciones atómicas** son operaciones que corren independientes de cualquier proceso. En este trabajo en particular, utilizaremos las variables atómicas dadas en C++ por la librería *atomic*, que son thread-safe.

El análisis sobre las decisiones de diseño tomadas para lograr la concurrencia compone el grueso del informe.

2. Desarrollo

Esta sección explicaremos en detalle las estructuras e implementaciones realizadas.

2.1. Lista Atomica

Se implementará mediante una **lista enlazada**, en donde cada nodo es un par $< clave, \#Apariciones >$. Utilizaremos esta lista para almacenar los elementos de cada bucket de la tabla de hash.

La única función que modifica esta lista es **insertar**, que recibe un valor por parámetro, y debe encargarse de crear un nodo con dicho valor y agregarlo al comienzo de la lista. El principal requerimiento con el que se debe contar es 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, que no modifique a la lista atómica.

```
1 void insertar(T valor)
2         Nodo* nuevaCabeza =  new Nodo(valor)
3         nuevaCabeza->_siguiente =_cabeza.load()
4         while (!(_cabeza.compare_exchange_weak(nuevaCabeza->_siguiente,
              nuevaCabeza)))
```

Primero se crea el nuevo nodo, el cual va a apuntar a la anterior cabeza de la lista y se define este nuevo nodo como el primero, o sea la nueva cabeza de la lista. Esto podría traer problemas, por

ejemplo, si tuviésemos mas de un nodo queriendo ser insertado ambos tendrán como siguiente a la cabeza ya que esas asignaciones no son atómicas.

Para poder realizar la inserción de manera atómica, la lista utiliza la función **compare_exchange_weak**, esta nos asegura que si la nueva cabeza quedo apuntando al lugar incorrecto, es decir que al ser interrumpida y quedasen dos nodos apuntando a la misma cabeza, se corrija y apunte correctamente a la nueva cabeza, además utiliza **atomic_load** que hace una lectura atómica, lo que nos asegura que es el valor que tiene en ese momento.

Por lo tanto la lista es atómica, ya que insertar es la única operación que realiza modificaciones sobre la lista. Además esto nos ayuda a evitar race conditions.

2.2. HashMap Concurrente

Se cuenta con una estructura de datos llamada HashMapConcurrente, la cual consiste en una tabla de hash abierta que gestiona las colisiones utilizando listas enlazadas. 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' hasta la 'z' (haciendo referencia a la 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.

2.2.1. Incrementar

La forma que provee nuestra implementación para modificar los valores del HashMap es a través de incrementar. La función recibe una clave de tipo string, si esta ya pertenecía a las claves del diccionario se debe incrementar en uno el valor previo. De lo contrario, se agrega un nuevo par <clave,1> en el bucket correspondiente al hash de la clave.

¿Qué pasa si más de un thread intenta agregar una clave?

Nuestra implementación permite paralelismo solo en el caso de que las claves tengan distintos hashes, en caso contrario, no se permite agregar concurrentemente elementos a una misma lista.

Para lograr esto nuestra solución hace uso de un arreglo de mutex de tamaño 26, la cantidad de posibles resultados de nuestra función de hash. Entonces, cuando un thread incrementa una clave se hace dueño del mutex asociado al hash index de la misma, de esta manera, si otro intentase incrementar una clave correspondiente a esa lista, debería esperar hasta que el proceso que se haya adueñado del mutex lo libere. Al no estar en disputa un recurso compartido, esto es lo que permite que dos threads con claves que no colisionan puedan ejecutarse en paralelo optimizando el funcionamiento del HashMap.

A continuación podemos observar un pseudocódigo de nuestra solución, donde en la línea 2 se ve que lo primero que hace es calcular hashIndex para realizar el *lock()* del mutex correspondiente, cuando lo consigue llama a la función incrementarEnLista que simplemente la recorre en busca de la clave si la encuentra la aumenta, si no agrega un nuevo par. Al finalizar se libera el mutex.

```
1 void incrementar(string clave)
2     int bucketIndex = hashIndex(clave)
3     mutexes[bucketIndex].lock()
4     incrementarEnLista(tabla[bucketIndex], clave)
5     mutexes[bucketIndex].unlock()
```

2.2.2. Claves y Valor

La operación *claves* devuelve todas las claves que se encuentran en la tabla. *valor* recibe una clave y devuelve el valor asociado en la tabla. A diferencia de la función incrementar en ambas operaciones priorizamos la eficiencia antes que la consistencia, si alguna de las dos se ejecuta en simultáneo con incrementar no se garantiza que el resultado sea consistente. Por esta razón, ninguna de las dos implementaciones hace uso de primitivas de sincronización.

```
1  vector<string> claves()
2      vector<string> claves
3      for (fila: tabla)
4           for(element: fila)
5               claves.push_back(element.first)
6      return claves
7
8  unsigned int valor(string clave)
9      ListaAtomica<hashMapPair> fila = tabla[hashIndex(clave)]
10     for(element : fila)
11         if(element.first == clave)
12             return element.second
13     return 0
```

Como se puede observar en el pseudocódigo para estas funciones, la primera recorre todas las claves de la tabla guardándolas en un arreglo. Por otro lado, *valor* obtiene la lista atómica a la cual corresponde la clave que se está buscando y la recorre para obtener el valor, en caso de no pertenecer a la tabla, retorna 0.

2.2.3. Máximo

El método máximo devuelve el par <clave, valor> con el valor más alto de la tabla.
Para evitar un funcionamiento incorrecto, lo primero que se pide sobre esta función es evitar el paralelismo con incrementar. A continuación se muestra lo que podría pasar en caso de que estos métodos puedan ejecutarse en paralelo:

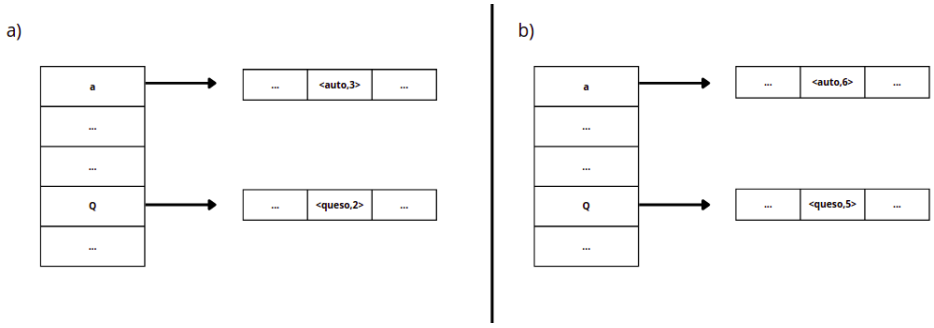

Figura 1: Escenario donde el resultado de la función máximo, nunca lo fue.

Supongamos que se comienza a ejecutar máximo en el instante (a), el resultado debería ser <auto,3>, entonces se recorre la primera lista y se obtiene ese resultado parcialmente. Ahora, mientras se recorren el resto de las listas entre "b" y "q" se llama 3 veces incrementar(auto) e incrementar(queso), entonces no solo no se va a obtener el resultado correcto para el instante (b) que sería

<auto,6>, sino que se va a obtener <queso,5>. Esto se debe a que máximo se va a seguir ejecutando sin volver a recorrer la lista "a", entonces cuando llegue a la lista "q" va a comparar contra el resultado anterior, por lo que las 5 apariciones de "queso" van a ser mayor a las 3 apariciones que tenía "auto".

¿Cómo evitamos el paralelismo entre máximo e incrementar?

Cuando un thread quiere calcular el máximo lo primero que hace es adueñarse de los 26 mutex que se usaban previamente en incrementar.

En el caso que un thread quiere calcular el máximo y hay otros incrementando este se va a bloquear esperando que liberen los mutex asociados a las claves que están incrementando. En el caso inverso, si un thread busca incrementar una clave y otros se encontraban calculando el máximo debe esperar a que se libere el mutex asociado a la clave que busca incrementar.

¿Que sucede luego de obtener todos los mutex?

Al tener todos los mutex se garantiza que no hay otro thread incrementando, entonces se puede calcular el máximo sin tomar recaudos extra, simplemente recorriendo las claves de cada bucket de manera iterativa .

Por último, se liberan todos los mutex.

```
1 <clave, valor> maximo()
2     for(auto mtx : mutexes)
3         mtx.lock()
4     <clave, valor> maximo = calcularMaximo()
5     for(auto mtx : mutexes)
6         mtx.unlock()
7     return maximo;
```

2.2.4. Máximo Paralelo

Para calcular el máximo la función de la sección anterior recorría cada fila secuencialmente. En cambio, *maximoParalelo* distribuye el trabajo de recorrer cada fila en distintos threads. Este método recibe una cantidad de threads como parámetro y se encarga de coordinar para que en todo momento cada thread este calculando el máximo de una fila determinada.

Los threads comparten un entero atómico que representa el índice de la próxima fila en la que se debe calcular el máximo, filas con índices menores a este ya han sido o se encuentran siendo recorridas por un thread.

Cuando un thread quiere resolver que lista debe calcular, se asigna el valor de la variable compartida en una variable local y luego la aumenta en uno. Como la variable es atómica, toda la operación de asignación y suma se realiza atómicamente, garantizando que ningún otro thread actualiza la variable en simultáneo, ni se asigna el mismo valor. De esta manera es seguro que ninguna lista sea salteada o se calcule su máximo para una misma fila más de una vez.

Todos los threads comparten un par <clave,valor> que representa el máximo hasta el momento. Cuando un thread termina de calcular el máximo, en caso de haber encontrado una palabra que aparezca más veces que el actual, se debe actualizar.

Es importante que la sección donde se actualiza sea atómica, de lo contrario, si dos o mas threads en simultaneo buscan actualizar el valor actual porque han encontrado valores superadores, nada nos garantiza que esos cambios se produzcan en un orden en que el ultimo que pise el valor sea el mayor.

Para evitar esto compartirán un mutex que pedirán al momento de querer actualizar un máximo, esto lo podemos ver en la línea 9 del pseudocódigo.

```
1  void buscarMaximoThread(atomic<int> actual, <clave,valor> maximoActual, mutex
       mtx_maximo){
2
3      while(actual < cantLetras){
4
5          int indice = actual++;
6
7          if(indice >= cantLetras) break;
8
9          max = maximoLista(indice);
10
11         mtx_maximo->lock()
12         if( esMayor(max , maximoActual) )
13             actualizarMaximo(max,maximoActual);
14         mtx_maximo->unlock();
```

Esta función es la que ejecuta cada thread creado en maximoParalelo, donde primero determina cuál es la lista que le toca calcular, luego llama a una función auxiliar encargada de calcular el máximo y por último actualiza el máximo parcial.

2.3. Carga de archivos

Esta se realiza mediante la función cargarArchivo que lee un archivo pasado por parámetro y carga todas sus palabras en el HashMap también pasado por parámetro. Esta se implementa mediante la función incrementar, quien es la encargada del manejo de las colisiones de hash, solucionando los posibles problemas de concurrencia.

Además, se cuenta con la función cargarMultiplesArchivos la cual toma tres parámetros, un vector de archivos, la cantidad de threads entre los que se van a dividir el trabajo y el HashMap al cual se van a cargar las palabras. Para esta función se utilizó una técnica similar a la de maximoParalelo en donde cada thread utiliza cargarArchivo y actualizando un índice que indica cuál es el siguiente archivo a cargar en el HashMap.

3. Experimentación

3.1. Introducción

3.1.1. Aclaraciones generales

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 el scheduler.

Para las mediciones de tiempo se utilizó la librería std::chrono de C++ mediante la función system_clock::now().

3.1.2. Especificaciones tecnicas

Para asegurar que los tiempos sean comparables entre si, todos los experimentos se ejecutaron con la misma PC :

- **Modelo:** Dell inspiron 3583 (08CA).
- **Procesador:** Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz.
- **Memoria RAM:** 16 GB 2400 MHz DDR4 (AMD).
- **cantidad de threads:** 8.

3.2. Experimento 1 - Distribución uniforme - Máximo Paralelo

Para este experimento, se utilizó el archivo "igualCantidadDePalabrasXLetra.txt" el cual contiene 4.000 palabras para cada letra de la "a" hasta la "l". De esta manera se busca que dividiendo el trabajo en mayor cantidad de threads, los resultados sean más eficientes.

¿Siempre que agreguemos más threads el resultado va a ser mejor?

En primer lugar, están las limitaciones del hardware, ya que una PC puede manejar una cantidad limitada de threads en simultáneo. Además, dado el funcionamiento de nuestro HashMap, no tendría sentido tener más de 26 threads, que es la cantidad de listas que contiene la tabla.

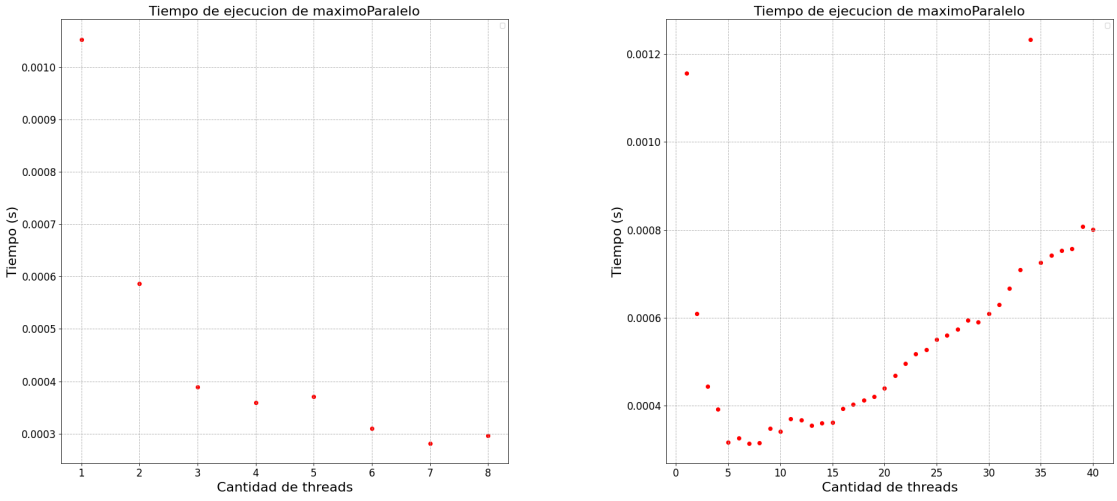

Figura 2: Tiempo de ejecución de la función maximoParalelo hasta 8 (a) y 40 (b) threads respectivamente.

Una vez corridos los experimentos, en la Figura 2.a podemos observar que entre 6 y 8 threads es donde se obtienen los mejores resultados. Esto sucede dado que es el punto en donde obtenemos una mayor paralelización en la distribución de las listas. Al tener 12 listas con claves cargadas, con 6 threads ya alcanzaría para que cada thread tenga que trabajar sobre 2 listas. Considerando que la PC con la que trabajamos solo puede manejar 8 threads simultáneamente, tiene sentido que quizás utilizando más de esta cantidad, no se obtengan resultados mejores. Yendo al otro extremo, en la Figura 2.b podemos notar como además, a medida que se crean mayor cantidad de threads, no solo no mejora el funcionamiento, sino que cada vez se vuelve más lento. Creemos que esto se debe al trabajo

adicional que tiene no solo la creación de threads que nunca van a ser utilizados, sino el trabajo del scheduler de ir alternando entre tantas tareas.

3.3. Experimento 2 - Distribución No Uniforme - Máximo Paralelo

Buscamos analizar que sucede cuando no hay una distribución uniforme en la letra inicial de las palabras. Nuestra hipótesis plantea que si unas pocas filas de la tabla concentran la mayor porción del total, el aumentar la cantidad de threads más haya de un punto no debería mejorar la performance. En particular, el punto de quiebre debería ser cercano a esa cantidad de filas que agrupan la mayor parte de las palabras.

Para poner a prueba esta hipótesis, utilizamos un dataset con 82.937 palabras en el que las filas a,b y c concentran el 91 % del total. En 6 podemos observar la distribución de las letras.

Los tiempos de ejecución resultantes se pueden observar en 6, notamos que los resultados se comparten acorde a nuestra hipótesis.

La peor performance se produce con una cantidad de threads menor a 3, esto sucede porque no permite paralelizar el trabajo de calcular las 3 filas. Luego, a partir de 3 threads los tiempos de ejecución bajan significativamente.

En particular con 3 y 4 se producen los mejores resultados , a partir de ese numero el tiempo de ejecución aumenta. Esto se debe a que el tiempo de ejecución que le tomara a los threads que calculan la fila a,b y c es mayor al tiempo que tarda un solo thread en calcular el máximo de las otras filas. En la figura 6 podemos observar una comparación entre el tiempo de ejecución de calcular las filas a,b y c con 3 threads, contra calcular con 1 thread solamente el resto de las filas.

Entonces, como ya tenemos un overhead de esas 3 filas el aumentar la cantidad de threads no mejoraría en nada, es mas terminando afectando la eficiencia dado que la creación y coordinación de threads no es gratis.

Como conclusión de esta experimentación nos llevamos que la performance de nuestra estructura esta muy ligada a la distribución de las palabras. El aumentar la cantidad de threads no garantiza en absoluto una mayor eficiencia.

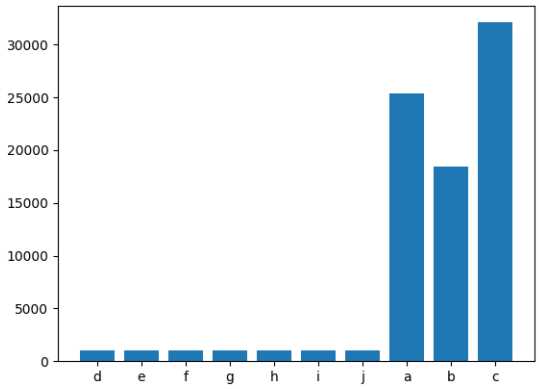

Figura 3: Cantidad de palabras por letra inicial

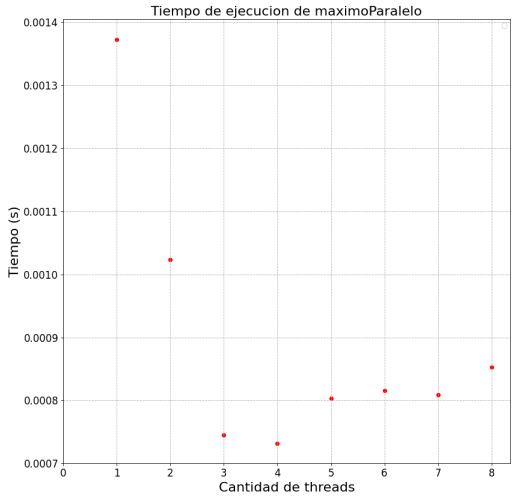

Figura 4: Tiempo de ejecución de maximoParalelo de 1 a 8 threads

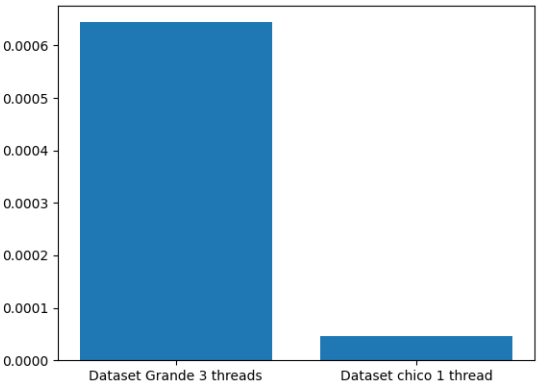

Figura 5: Comparación entre el tiempo de ejecucion de calcular las filas a,b y c con 3 threads y calcular las restantes con 1 thread

Figura 6: Experimento 2

3.4. Experimento 3 - Carga de archivos - Archivos grandes

Para este experimento se busco analizar como varia la carga de archivos con distinta cantidad de threads. Se utilizo el archivo "igualCantidadDePalabrasXLetra.txt", como en el experimento 1, pero se dividió el archivo en 12 archivos cada uno con las 4000 palabras que tiene el original por cada letra.

Lo que se espera es que a medida que aumenten los threads debería mejorar el rendimiento de la operación, obviamente hasta los limites físicos del hardware utilizado.

Como podemos ver en los resultados 7se obtiene una gran mejoría con el aumento threads reduciendo el tiempo, hasta 8 threads que es el limite de nuestro hardware, además después de 12 vemos mantiene una constancia ya que 12 threads son suficientes para los 12 archivos. Como referencia tenemos que cargar el archivo entero, o sea "igualCantidadDePalabrasXLetra.txt" sin divir, toma 2.24 segundos.

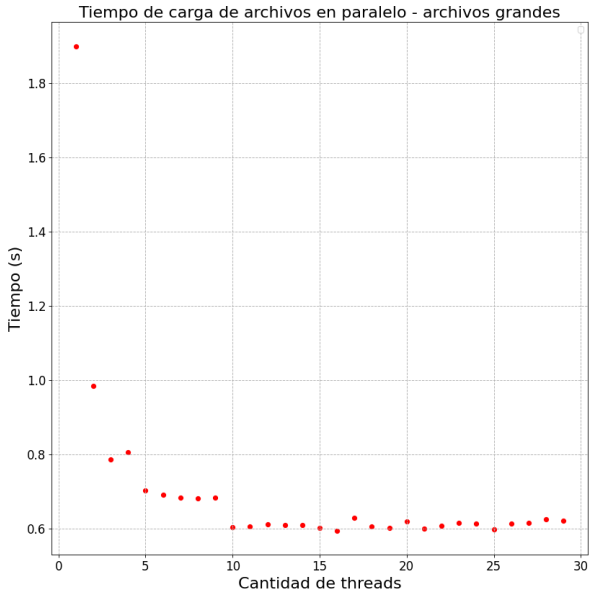

Figura 7: Experimento 3 - Tiempo de ejecución por thread hasta 30 threads

3.5. Experimento 4 - Carga de archivos - Archivos pequeños

Para este experimento utilizaremos los archivos "test-1", "test-2", "test-3"10 veces cada uno, o sea un total de 220 palabras, y los correremos hasta en 60 threads.

Nuestra hipótesis fue que como los archivos eran mas pequeños, los resultados se iban a ver mas afectados por otros costos, como son la creación de los threads y la organización de estos.

Como podemos ver en la figura 8, una vez superada la cantidad de 8 threads, que es el máximo de nuestro hardware, empieza a incrementar el tiempo de ejecución. Tal como planteamos en la hipótesis, creemos que esto se debe a que al tratarse de archivos pequeños, los costos de creación y organización marcan una diferencia en el rendimiento.

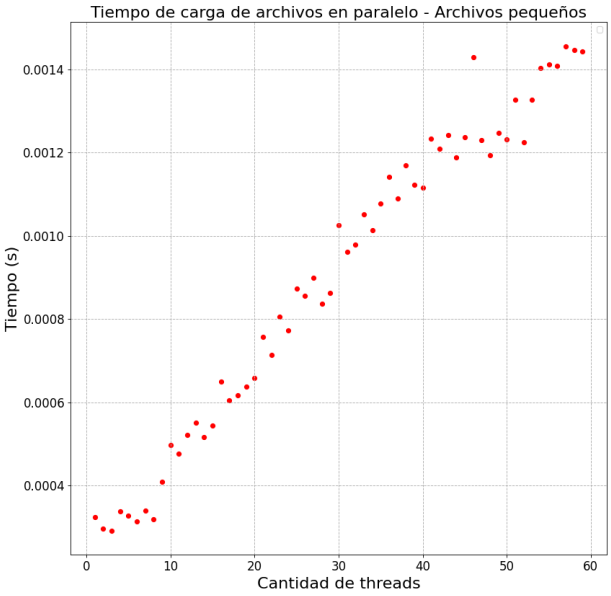

Figura 8: Experimento 4 - Tiempo de ejecución por thread hasta 60 threads

4. Conclusión

A partir de los experimentos realizados llegamos a la conclusión de que la concurrencia es una gran herramienta para realizar tareas con alta carga de procesamiento. Pero que en otros casos puede dar resultados mixtos, como vimos en el experimento 2, que con datasets con distribuciones mas concentradas en solo algunos puntos, a partir de cierta cantidad de threads empeora la performance o en el experimento 1 que el aumentar considerablemente el numero de threads también puede afectar negativamente el tiempo.

Además pudimos ver que, la dificultad de la programación paralela y llegar a algoritmos correctos es mucho mayor. Así como la dificultad para testear que un programa no tenga problemas de concurrencia.