DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP: Listas Sincronizadas

Primer Bimestre, 2023 — Programación Concurrente

Integrante	LU	Correo electrónico
Cappella Lewi, F. Galileo	653/20	galileocapp@gmail.com
Leo Mansini	318/19	leomansini2000@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

http://www.exactas.uba.ar

Abstract

Las estructuras de datos concurrentes son una forma de encarar el problema de recursos compartidos entre threads corriendo en paralelo. En este trabajo se implementará una red social llamada **Twiner**, que en su backend usará distintas implementaciones de listas enlazadas concurrentes, para luego comparar el rendimiento de cada implementación.

Contents

1 Introducción

En este informe se quiere analizar y comparar la efectividad de diferentes métodos de sincronización sobre listas enlazadas.
Para poder hacerlo en un contexto realista se definió el programa **Twiner**, explicado en la sección 1.1.
Luego en la sección 2 se presenta el código implementado, y también se explica la funcionalidad detrás de los diferentes métodos analizados.
Y finalmente en las secciones 3 y 4 se analizan los resultados obtenidos.

1.1 Twiner

Como fue mencionado, se analizan los métodos de sincronización sobre Twiner. Twiner es una plataforma muy popular en las redes donde las personas publican y leen *twines*. Normalmente Twiner recibe cientos de pedidos por segundo, por lo que es muy importante responderlas de forma concurrente.
En este informe se implementa la base de datos de Twiner usando listas enlazadas.

La concurrencia ocurre cuando les usuaries interactúan con la plataforma. Haciendo una de tres diferentes acciones:

- Iniciando sesión (o *logué andose*),
- cerrando sesión (o *deslogueándose*),
- o mandando un pedido API.

Al iniciar sesión se les provee una *session ID*. Y tanto para cerrar sesión como para hacer un pedido API necesitan proveerla para verificarse.

Entonces Twiner está implementado con un diccionario implementado sobre una lista enlazada de usuarios logueados con sus session ID.

1.2 Tipos de Listas

Se van a probar tres métodos de sincronización para listas enlazadas:

- Locks de granularidad (o *Fine-Grained locks*).
- Optimista.
- Sin locks (o *Lock-Free*).

Lo esperado es que los métodos mejoren en eficiencia en el orden mencionado, ya que cada uno permite más interacciones simultáneas que el anterior.

2 Métodos

2.1 Implementación

El programa fue implementado en Java 17. Y compliado sin especificar ninguna flag de optimización.

2.2 Twiner

Para implementar Twiner y los datos de los usuarios, se crearon las clase `Twiner` y `User`.
La clase Twiner presenta una interfaz para loguearse (`logIn(userId)`), desloguearse (`logOut(userId, sessionId)`), y recibir un pedido (`apiRequest(userId, sessionId)`) que representa a cualquier otra interacción de un usuario. Internamente, Twiner posee una estructura concurrente, un diccionario implementado sobre una lista enlazada, que almacenará los usuarios logueados ordenados por user ID. Los usuarios serán los valores y sus user ID serán las claves en este diccionario.
Al llamar `logIn` se agrega al usuario pedido al diccionario, y se provee una session ID.
Al llamar `logOut` primero se busca al usuario pedido en el diccionario y se revisa que la session ID usada sea la misma que le fue provista, y luego se lo saca del diccionario.
Y al llamar `apiRequest` se repite la revisión en el diccionario.

2.3 Simulación

2.3.1 UserThread

Para modelar el comportamiento de usuarios, se tiene la clase `UserThread`.
Al definir una instancia de `UserThread`, se le setea el tipo de acción que realiza (cada `UserThread` realiza un solo tipo de acción), y la cantidad de veces que realizará esa acción.

Los tipos de acción que puede realizar son:

- `logIn`: Agrega un usuario aleatorio a Twiner.
- `logOut`: Elimina, un usuario aleatorio de Twiner.
- `apiRequest`: Realiza la validación de si existe un usuario aleatorio de Twiner.

Estas acciones implican generar el usuario o el `userId` necesarios para luego llamar al método homónimo de Twiner.

Como el nombre indica, cada instancia de `UserThread` representará un thread, y todos los threads interactuarán con un `Twiner` compartido, haciendo uso, simultáneamente, de su estructura concurrente que almacena datos de usuarios.

2.3.2 ThreadPool

Para crear y manejar los threads se diseñó el módulo `ThreadPool`.
En este módulo se capturan los argumentos en la llamada al proyecto, los cuales son:

- `mode`: El módo de sincronización del diccionario de Twiner (`free`, `optimistic`, o `fine-grained`).
- `actions`: La cantidad de acciones que realizará cada thread, independiente de qué tipo de acción realice.
- `logIn`: Cantidad de threads que realizarán la acción `logIn`.
- `logOut`: Cantidad de threads que realizarán la acción `logOut`.
- `apiRequest`: Cantidad de threads que realizarán la acción `apiRequest`.

Con estos argumentos se crean la cantidad y tipos de UserThread deseados. Durante su creación esperarán a una barrera implementada con un semáforo, y cuando todos hayan sido creados, comenzarán a realizar sus acciones correspondientes.
El ThreadPool también contiene a la instancia de Twiner, y una lista con las combinaciones válidas de userId y sessionId, ambas compartidas por los threads.

2.4 Experimentación

Se desean medir tiempos de ejecución de cada operación, variando cantidad de threads en total, cantidad de threads por tipo de operación, y cantidad de veces que repiten su acción los threads. Para ello, cada thread mide cuando tarda realizar cada operación y retorna esos tiempos. Para reducir variabilidad no debida a la concurrencia, se toman todos los tiempos que tardaron las operaciones de cada tipo y se calcula el promedio.

El primer conjunto de pruebas fue correr el proyecto variando la cantidad de acciones de 1 a 10, y proporción de threads para cada acción, pero manteniendo fijos valores de threads totales. Se tomaron los tiempos y para los valores de distinta cantidad de acciones se tomó un promedio. Esto no hace que se pierda análisis dado que para distintas cantidades de acciones por thread los gráficos mostraban tendencias similares.

En el segundo set de pruebas, se corrieron experimentos variando cantidad de threads y de acciones realizadas por cada thread, pero manteniendo la cantidad total de acciones fija, por lo que al aumentar los threads se tenían threads con menos operaciones.

Ambos experimentos se realizaron para los 3 modos de concurrencia pedidos, fine-grained, free y optimist, y cada caso se corrió 10 veces, promediando los tiempos antes del análisis. Las ejecuciones se realizaron en una máquina con procesador `Intel i3 10100F` con 32GB de RAM, reemplazando el proceso de init con el experimento.

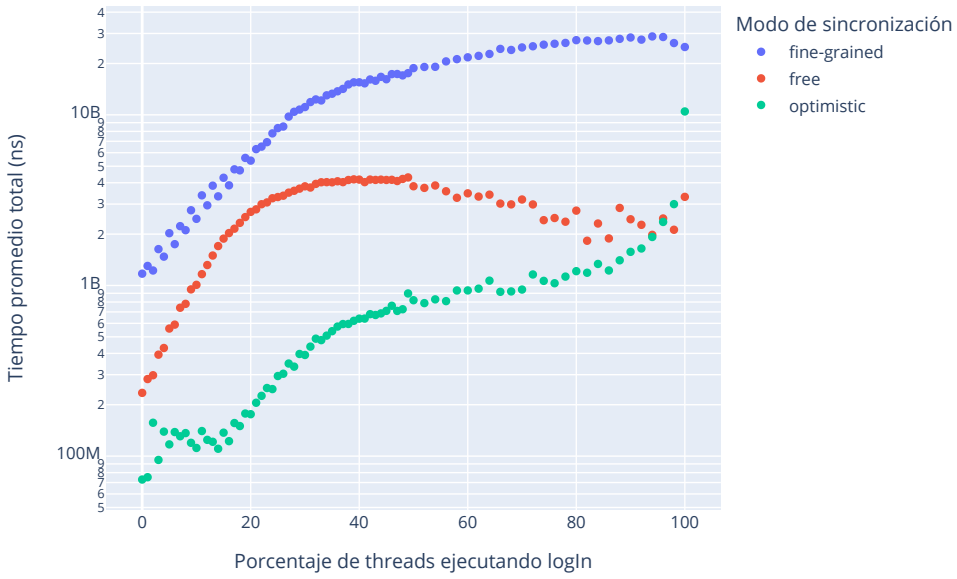
3 Resultados

Comenzamos analizando la operación logIn, que realiza inserciones en la estructura concurrente. Se puede ver en el gráfico 1a que a medida que crece la proporción de threads que realizan esta operación, es decir, hay menos de las otras operaciones, que remueven o buscan en la estructura, el tiempo promedio que tarda cada inserción crece. Esto puede explicarse por la cantidad de nodos que hay en la estructura: A medida que hay mayor proporción de threads agregando en la estructura se tienen más nodos y el tiempo que se tarda en insertar es mayor, ya que hay que recorrer una estructura más grande.

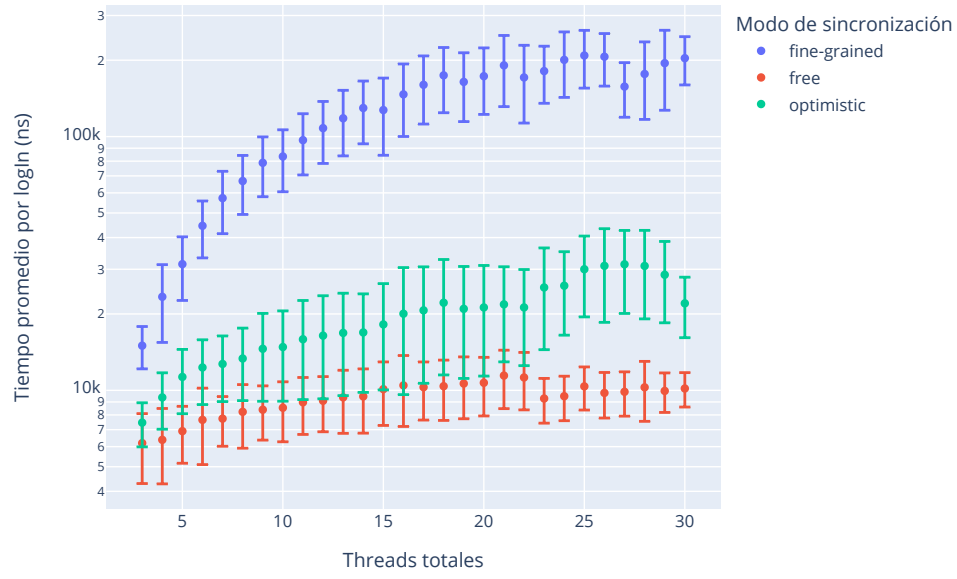
Es notable el caso del modo de concurrencia sin locks, que no sufre de peores tiempos cuando el porcentaje de threads realizando logIn alcanza $\approx 50\%$. Es posible que la razón de esto sea que esta operación experimenta demoras cuando dos threads agregan elementos que requieren modificar los mismos nodos existentes (predecesores en la lista). Es esperable que eso suceda si existen pocos elementos en la lista, pero si hay mayor cantidad de threads haciendo logIn, es menos probable que dos threads agreguen elementos que compartan predecesor o sean uno predecesor del otro.

Por otro lado, en el gráfico 1b, a medida que crece la cantidad de threads totales, el tiempo aumenta. Esto es esperable porque el recurso compartido esta siendo accedido por más threads.

En cuanto a la comparación entre los modos de concurrencia, los resultados son razonables con cómo maneja cada algoritmo los accesos concurrentes. Las listas concurrentes de granularidad fina usan muchos más locks que el método optimista, y por supuesto más que el método sin locks, lo cual genera esperas entre los threads y aumenta el tiempo promedio de las operaciones.

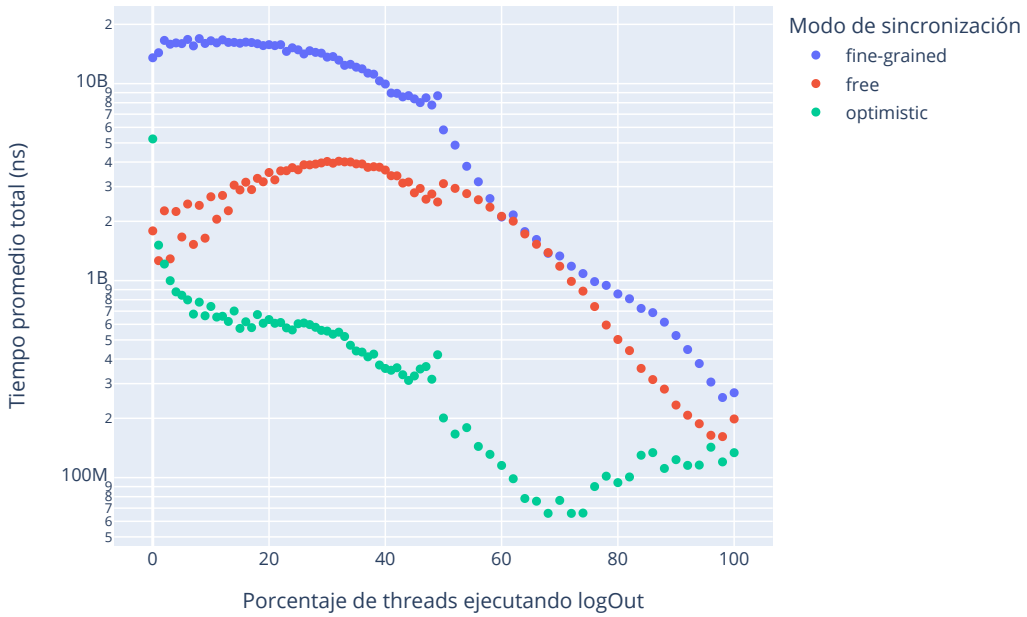


(a) Tiempo promedio total en función de la proporción de threads ejecutando login

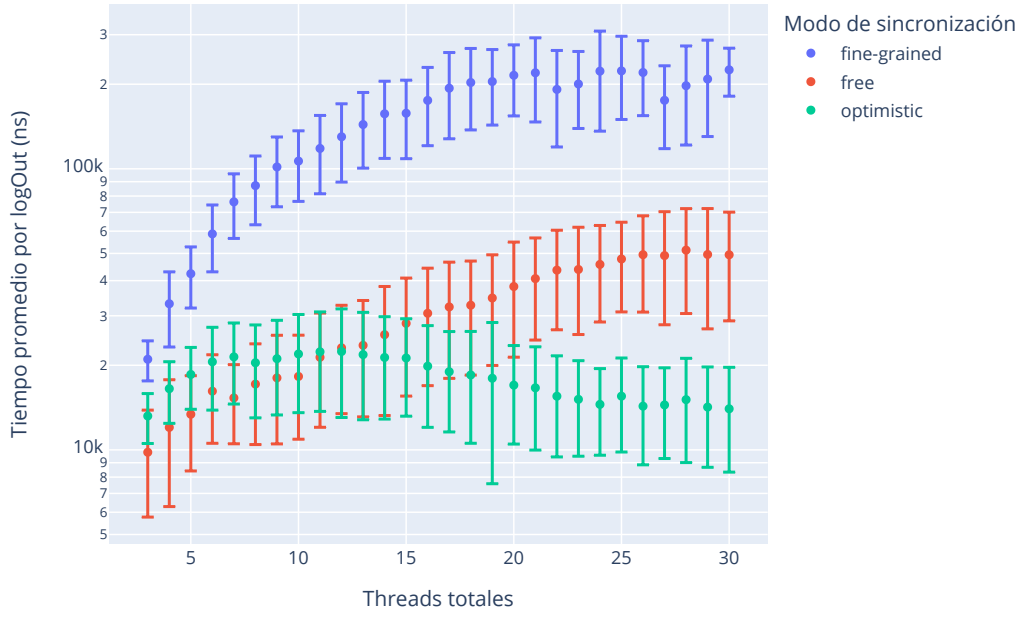


(b) Tiempo promedio para ejecutar un login en función de la cantidad de threads totales

Fig. 1: Tiempos de ejecución para la operación logIn



(a) Tiempo promedio total en función de la proporción de threads ejecutando logout



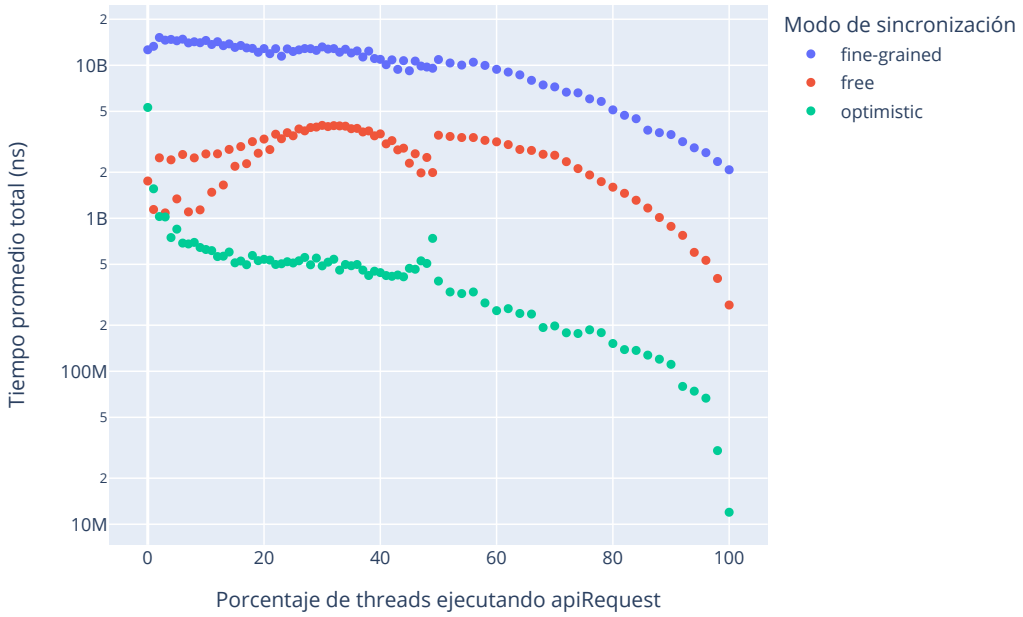
(b) Tiempo promedio para ejecutar un logout en función de la cantidad de threads totales

Fig. 2: Tiempos de ejecución para la operación logOut

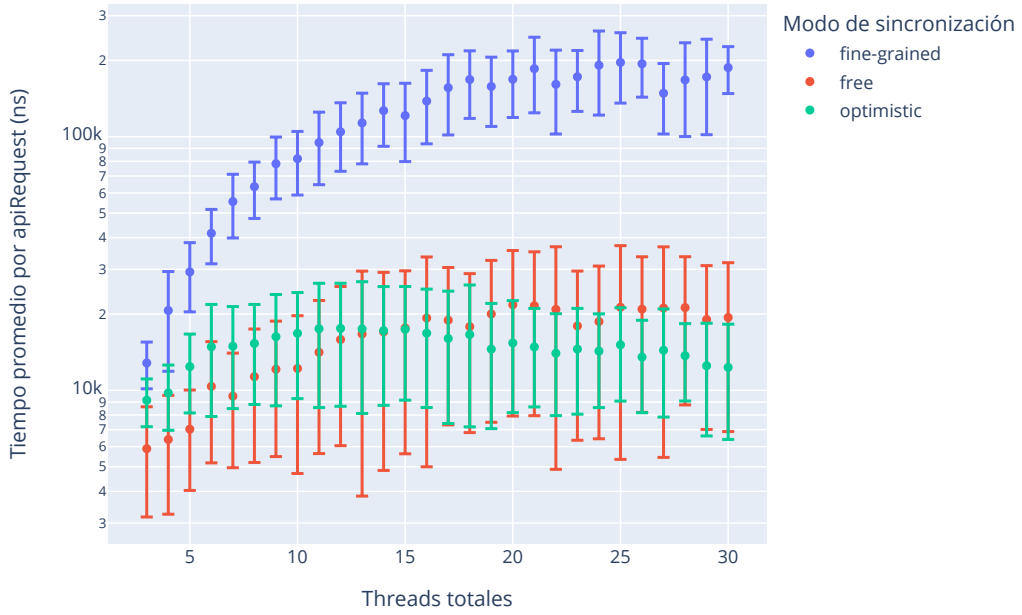
Para el caso de la operación logOut, a medida que hay mayor proporción el tiempo disminuye. En este caso, el efecto en el gráfico 2a es opuesto al de logIn, en 1a, ya que al haber más proporción de threads borrando elementos, hay menos nodos en la estructura, y por lo tanto se recorre en menos tiempo.
Y por cómo se experimentó, es más posible que se intente borrar un elemento no perteneciente a la estructura.

En el caso del gráfico 2b, este es más comparable al gráfico 1b. Esto es porque se puede ver que a mayor cantidad de threads, estos colisionan más en sus interacciones con la estructura concurrente, resultando en mayor tiempo de ejecución, sin importar la operación.

También se puede ver que en el caso del modo optimista, el tiempo para los logOut comienza a bajar al alcanzar ≈ 10 threads. Esto no se condice con nuestras hipótesis, ya que se esperaba que el tiempo creciera a mayor threads, ni se le encontró una explicación.



(a) Tiempo promedio total en función de la proporción de threads ejecutando apiRequest



(b) Tiempo promedio para ejecutar una apiRequest en función de la cantidad de threads totales

Fig. 3: Tiempos de ejecución para la operación apiRequest

Para la operación apiRequest, en el gráfico 3a vemos una disminución del tiempo a medida que incrementa la proporción de esta operación, que simplemente busca en la estructura (no agrega ni remueve). Como en el caso del gráfico 2a, esto es esperable porque a mayor proporción de threads de apiRequest, menor proporción de threads de logIn que agregan nodos a la estructura, y por lo tanto recorrer la lista es más rápido. Si bien en ambos casos decrece el tiempo, en el caso de logOut es claro que los tiempos se reducen en mayor magnitud, lo cual hace sentido ya que en ese caso se remueven elementos, mientras que en apiRequest no se agregan pero tampoco se remueven.

En el gráfico 3b, se ve que a más threads totales aumenta los tiempos en fine-grained, como es el caso en 1b y 2b, se mantiene la condición en donde muchos threads compiten para acceder a los mismos locks. En el caso de free se ve un ligero crecimiento en los tiempos de ejecución aunque los errores son muy grandes. Las desviaciones en los tiempos de free y optimistic son más grandes, lo cual nos es razonable ya que las demoras en este método dependen de los nodos a los que acceden los demás threads por lo que esa variabilidad es esperable. Sin embargo, no tenemos explicación de porqué ese comportamiento no es tan notorio en las demás operaciones, por lo que se puede deber a variabilidad en la medición.

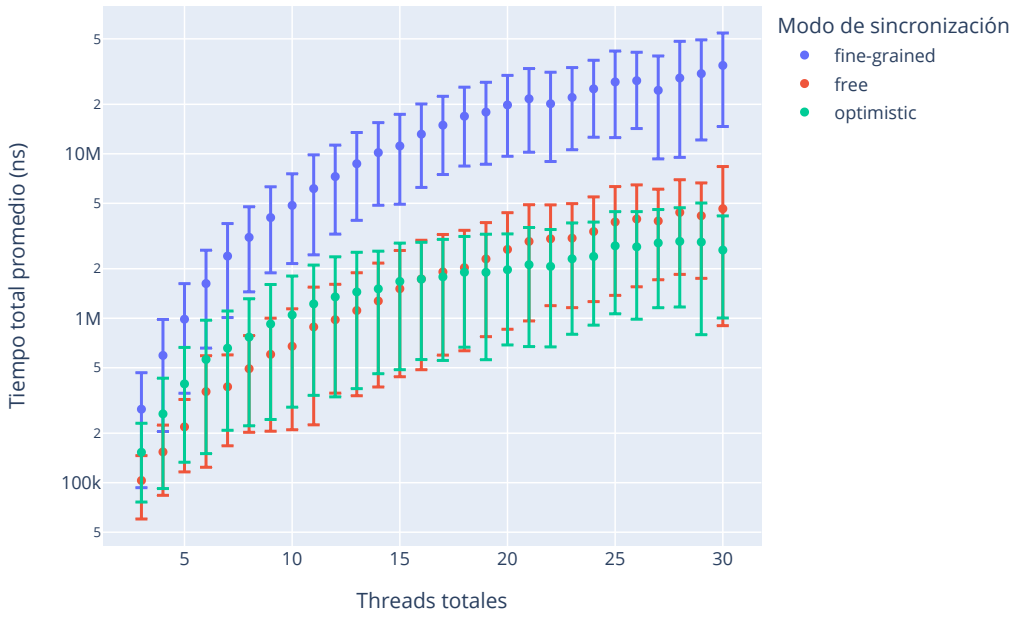
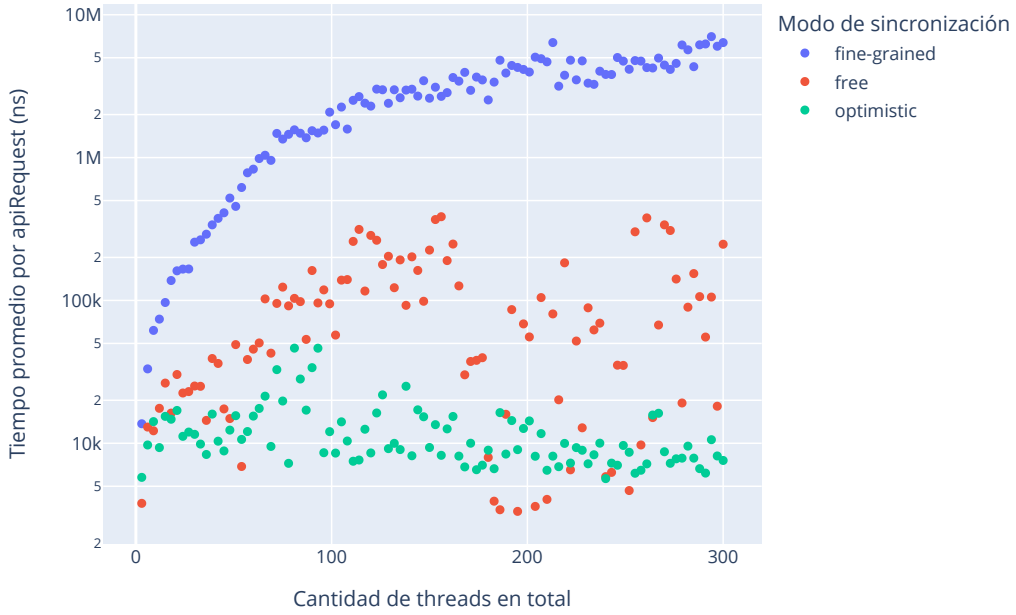


Fig. 4: Tiempo promedio para ejecutar todas las acciones

Para calcular los tiempos totales graficados en 4, se sumó lo que tomaron los threads para todas las operaciones. Como en los tiempos discriminados por operacion, se ve que fine-grained es la más lenta. Luego, free y optimistic compiten pero a mayor threads optimistic es ligeramente más rápido, lo cual no sucedía si se tenía en cuenta solamente los tiempos de logIn.

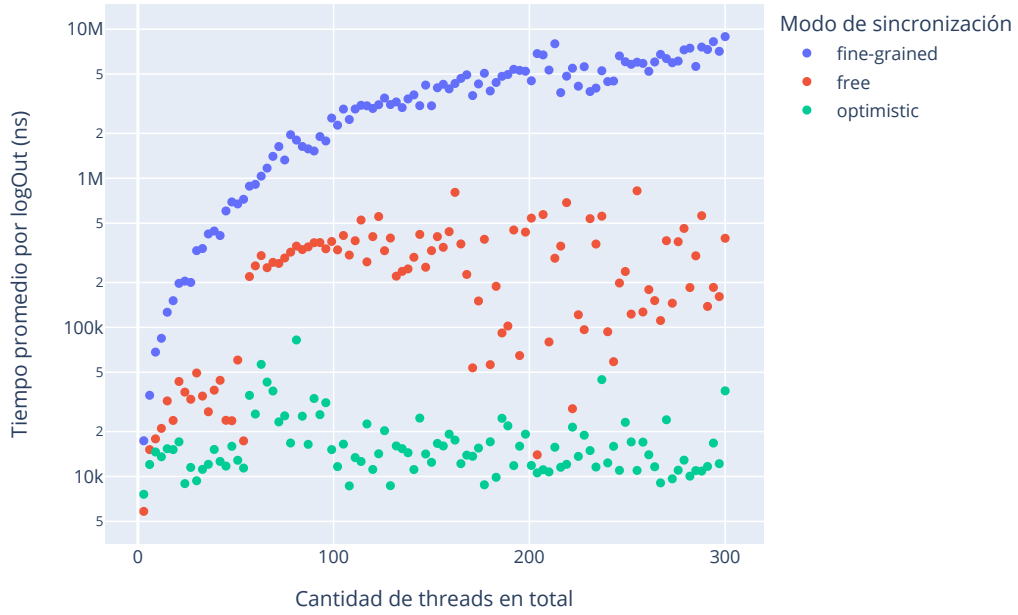
Con los resultados analizados se puede notar que la variable que más afecta al tiempo de ejecución, tanto total como para cada operación, es la cantidad total de threads independiente de qué operaciones ejecutan. Esto se ve reflejado en la figura 4.

A continuación, los gráficos de la figura 5 muestran tiempos de ejecucion promedio para cada operación y totales, para distinta cantidad de threads pero cantidad constante de operaciones totales (igual a 100). Eso significa que a mayor cantidad de threads, cada thread realizó menos operaciones.

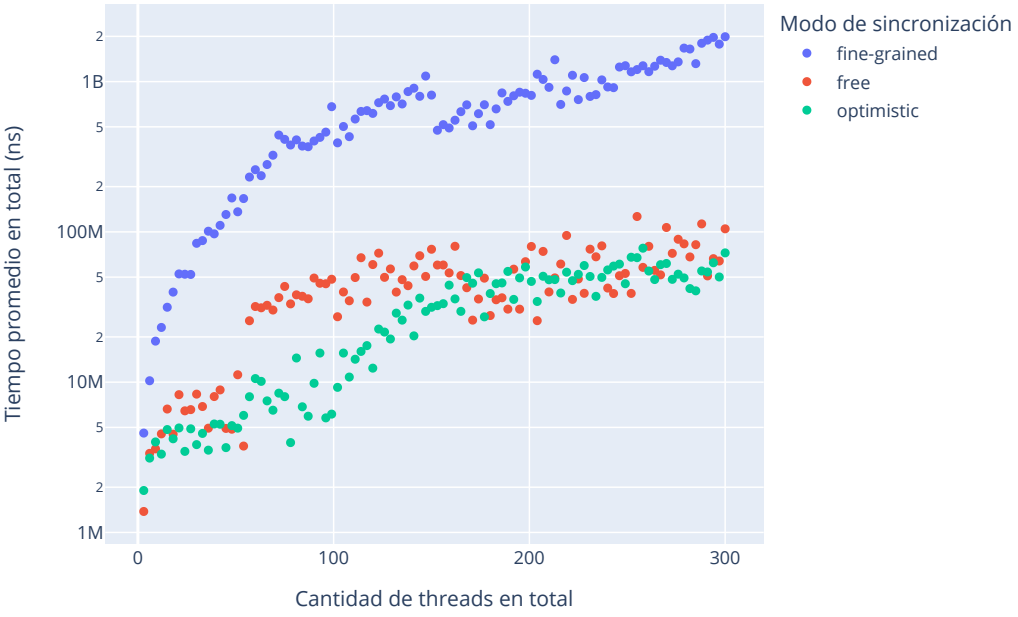


(a) Tiempo promedio para ejecutar una apiRequest en función de la cantidad de threads totales

(b) Tiempo promedio para ejecutar un logIn en función de la cantidad de threads totales



(c) Tiempo promedio para ejecutar un logOut en función de la cantidad de threads totales



(d) Tiempo total de todas las operaciones en función de la cantidad de threads totales

Fig. 5: Tiempos para ejecutar operaciones, con la cantidad de operaciones totales constantes. Es decir, a mayor cantidad de threads, cada thread realizó menos operaciones.

En la figura 5b se puede ver que a medida que hay más threads, los tiempos aumentan hasta los 150 threads, y luego se mantiene relativamente constante, al menos en free y optimistic. Este comportamiento no se replica en los tiempos para apiRequest (5a) o logOut (5c). Al ver los tiempos totales en la figura 5d, vemos que se tiene un efecto similar al de logIn, si bien no tan creciente.

4 Conclusión

En este trabajo se realizaron experimentos para comparar algoritmos de concurrencia en listas, variando la cantidad de threads involucrados y sus acciones en la lista (leer, borrar, agregar).

Se verificó que el algoritmo fine-grained es peor en tiempos de ejecución que optimistic y free, sin importar cantidad de threads, si bien la diferencia crece con la cantidad de threads totales. Por otro lado, los algoritmos free y optimistic muchas veces presentaron performance similar, con muchos casos donde optimistic fue superior.

Fue notoria la diferencia en la naturaleza de cada una de las operaciones: al haber mayor proporción de threads agregando elementos en la estructura concurrente, los tiempos de ejecución aumentaban, al tener una estructura más grande. Si, en cambio, había más threads removiendo elementos, en este caso los tiempos disminuian, ya que la estructura era cada vez más rápida de recorrer.

El análisis de los resultados no comprobó totalmente las hipótesis, la comparación entre optimistic y free no esta clara y las barras de error, seteadas por la desviación estandar de las mediciones, podrían ser menores si se realizara una experimentación a mayor escala (corriendo más veces cada caso).