DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Programación Concurrente
1er cuatrimestre 2023

Integrante	LU	Correo electrónico
Frassia, Fernando Nicolás	340/13	ferfrassia@gmail.com
Charabora, Iván	234/20	ivancharabora@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 práctico se realiza una evaluación empírica de distintas implementaciones de conjuntos concurrentes sobre listas enlazadas. La concurrencia nos da la capacidad de tener múltiples procesos o hilos trabajando sobre el conjunto al mismo tiempo, optimizando así los accesos y las tareas a realizar.

Se implementaron en Java distintas versiones de conjuntos sobre listas, como locks de granularidad fina, sincronización optimista y sin locks, con el objetivo de medir el desempeño de estas implementaciones frente a distintos escenarios. Estos escenarios varían en cantidad de hilos y operaciones donde nos permite juntar datos del tiempo en ejecución y analizar el rendimiento de cada implementación.

En las siguientes secciones podremos ver los algoritmos, cómo funcionan, los escenarios planteados para la experimentación y los resultados obtenidos.

2. Desarrollo

Como se mencionó en la sección anterior los algoritmos sobre listas implementados son los de granularidad fina, optimista y sin locks. En los 3 casos, una lista consiste en una cadena de nodos, donde el primero apunta al segundo y así sucesivamente. Por comodidad, todas las listas comienzan con un nodo centinela *head* que precede al primer nodo de la lista y cuyo item siempre será *null*.

En las implementaciones con locks un nodo contiene:

- Un item.
- Una key, creada por una función de hash y el item.
- Un lock reentrante.
- Una referencia al siguiente nodo.

En el caso sin locks sólo contiene:

- Un item.
- Una key, creada por una función de hash y el item.
- Una referencia atómica marcable al siguiente nodo. Dicha referencia estará marcada si el nodo será eliminado.

2.1. Lista con locks de granularidad fina

Una lista con locks de granularidad fina consiste en dividir la lista en nodos individuales, cada uno con su propio lock. En lugar de tener que realizar un lock en la lista completa (como la granularidad gruesa), se puede hacer un lock en un nodo cada vez que sea necesario, lo que permite el acceso y la modificación simultánea en diferentes partes de la lista

El agregado de un nodo consiste en iterar tomando los locks de cada nodo por donde itero. De hecho, para agregar necesito el lock previo y el siguiente a donde iría mi nodo. Esto se debe a que si tomo solo el lock anterior otro hilo podría modificar el siguiente haciendo la lista inconsistente. Por lo tanto, cuando estoy iterando nodos, voy haciendo los locks del nodo donde estoy parado y el anterior. Cuando tengo los nodos indicados, consiste en cambiar el next del previo al nodo nuevo y el next del nodo nuevo al otro.

El eliminar nodos es similar en el aspecto de que hay que llegar a los nodos correspondientes lockeando los nodos en el camino. También hay que lockear dos nodos, en este caso el que se quiere eliminar y el predecesor. Nótese que si solo se toma el lock del predecesor para cambiar el next, otro hilo podría tomar el lock del que se quiere eliminar y borrar el siguiente. En ese caso el predecesor se queda apuntando a un nodo eliminado. Por lo tanto, también hay que tomar el lock del nodo a eliminar así no se puede eliminar el nodo sucesor.

Para saber si un valor está en un nodo en el conjunto consiste en hacer la misma estrategia de iterar con dos nodos obteniendo los locks y fijandose si el valor guardado en el nodo es el buscado.

Esta implementación de lista es libre de inanición, libre de deadlocks (si se obtienen los locks en el mismo orden) pero es ineficiente ya que se toman y liberan muchos locks (por ejemplo si quiere agregar algo al final lockeo toda la lista al menos una vez).

2.2. Lista con sincronización optimista

Bajo la idea de que ir lockeando por todos los nodos a medida que voy iterando para encontrar el nodo buscado es costoso, aparece la idea de lockear solo los que necesito. Uno se tienta por simplemente hacer la idea anterior pero solo lockeando cuando llegue al nodo buscado. El problema de esto es que cuando un proceso toma el lock no puede asegurar que siga valiendo que los nodos lockeados son válidos. Sea el ejemplo de querer agregar un nodo previo a uno que quiero eliminar. Si suponemos que ambos procesos encontraron los nodos correspondientes y primero elimino el nodo, cuando le toca al proceso agregador va a agregar un nodo nuevo con next un nodo eliminado. Por lo tanto, el nodo que se quiso eliminar sigue en la lista.

Entonces, la idea final de esta implementación es, una vez adquirido los locks, verificar que no hubo cambios en el medio y si hubo volver a empezar. La verificación consiste en que los nodos sigan en la lista (validación global) y que los nodos sigan siendo adyacentes (validación local). Explicado de una forma más algoritmica, se pueden hacer los cambios en la lista si se tienen los locks de dos nodos, si el primero es accesible desde head y si ese es el predecesor del otro.

Por lo tanto, los algoritmos de búsqueda, inserción y eliminación son similares a los de locks con granularidad fina exceptuando que no se toman los locks anteriores que no participan y luego de tomar los locks se llama a la función validadora.

Esta implementación usa menos locks que la anterior, lo que permite que se tenga mayor concurrencia y eficiencia pero requiere recorrer la lista dos veces.

2.3. Lista sin locks

Este algoritmo permite mejorar que no se recorra más de una vez la lista y no lockear nodos de más. Al igual que otros algoritmos concurrentes lock-free, esta implementación utiliza la operación atómica Compare And Set (CAS).

Nuevamente, no es tan sencillo como hacer un CAS entre nodos para cambiar el next. Por ejemplo, podríamos tener el problema de, al eliminar dos nodos consecutivos, que eliminemos el primer nodo usando CAS actualizando el next del predecesor y luego, al borrar el segundo, no se elimina porque el predecesor tiene como next al nodo. Entonces, la táctica consiste en marcar el nodo como eliminado en vez de eliminarlo realmente (eliminado lógico) y que el CAS falle si el nodo está marcado. Sin embargo, si no realizamos eliminaciones en algún momento, podríamos terminar con una lista que contiene muchos elementos marcados como eliminados, lo cual afectaría la eficiencia. Por lo tanto, la idea es que cada operación cada vez que encuentre un nodo eliminado lo elimine realmente. Para marcar nodos hacemos uso de la libreria AtomicMarkableReference.

La función correspondiente de eliminar el nodo es la función de encontrar o find. Esta es usada en la eliminación y el agregado de nodos para encontrar los dos nodos correspondientes y, de paso, siempre que encuentra un nodo marcado en el camino lo elimina. Igual que en sincronización optimista si no puede, o sea si falla el CAS, lo intenta otra vez.

El agregado consiste en el uso de la función de encontrar e intentar hacer el CAS en el next del nodo previo donde el esperado es el próximo sin estar marcado y el a cambiar es el nodo nuevo sin estar marcado. Nuevamente, si falla vuelve a intentar.

La eliminación primero también hace uso de la función de encontrar. Luego trata de marcar al nodo atómicamente y, si pudo marcarlo, se actualiza con CAS el next del nodo predecesor.

Para saber si un elemento está en el conjunto se recorre la lista hasta encontrar el nodo donde

debería estarlo. El elemento está en el conjunto si el nodo tiene el valor del elemento y no está marcado.

3. Experimentación

3.1. Experimentación cualitativa

Los experimentos diseñados para la experimentación cualitativa fueron elegidos con el objetivo de analizar el correcto funcionamiento de las implementaciones y, además, medir la performance de las mismas. Estos experimentos son:

- Experimento 1: Un hilo agregando diez mil números.
- Experimento 2: Un hilo sacando diez mil números.
- Experimento 3: Dos hilos agregando diez mil números donde uno agrega los pares y el otro a los impares.
- Experimento 4: Dos hilos sacando diez mil números donde uno elimina los pares y el otro a los impares.
- Experimento 5: Cuatro hilos agregando diez mil números donde dos agregan los pares (uno los números menores a 5000 y el otro los mayores) y el otro a los impares (uno los números menores a 5000 y el otro los mayores).
- Experimento 6: Cuatro hilos sacando diez mil números donde dos eliminan los pares (uno los números menores a 5000 y el otro los mayores) y el otro a los impares (uno los números menores a 5000 y el otro los mayores).
- Experimento 7: En una lista con los primeros 10 números, operan cuatro hilos donde:
  - Hilo 1: Agrega al 4 diez mil veces.
  - Hilo 2: Agrega al 6 diez mil veces.
  - Hilo 3: Saca al 4 diez mil veces.
  - Hilo 4: Saca al 6 diez mil veces.

  El objetivo de este experimento es ver que el 5 sigue perteneciendo a la lista final y no se vea afectado por una modificación de un next incorrecto debido a una condición de carrera.

Tras correr los experimentos observamos que las tres implementaciones pasaron todos los tests, lo cual nos da cierta certeza de su correcto funcionamiento.

En el siguiente cuadro podremos ver la performance por experimento y por implementación, habiendo corrido los experimentos 500 veces y tomado los valores en promedio.

	Granularidad fina (ms)	**Optimista (ms)**	**Lock free (ms)**
Experimento 1	6814.28	3836.87	3788.21
Experimento 2	6785.27	3692.83	3658.86
Experimento 3	5274.98	3128.86	3500.82
Experimento 4	12.12	6.78	6.32
Experimento 5	4689.64	2720.35	3134.33
Experimento 6	4.17	4.38	3.09
Experimento 7	44.93	24.41	10.15

Cuadro 1: Tiempo promedio en milisegundos por experimento y por algoritmo

En resumen, los resultados obtenidos de los experimentos muestran que las implementaciones optimista y lock-free tienden a ser más eficientes en términos de tiempo de ejecución en la mayoría de los escenarios. Estas implementaciones aprovechan el uso de operaciones atómicas y marcado lógico de nodos eliminados. Por otro lado, la implementación con granularidad fina presenta tiempos de ejecución más altos debido a que adquiere y libera mayor cantidad de locks.

Otra observación es que los experimentos 4, 6 y 7 muestran tiempos muy bajos para todas las implementaciones. Estos experimentos involucran la eliminación de números, y las implementaciones optimista y lock-free logran aprovechar eficientemente las operaciones atómicas y el marcado lógico de nodos eliminados.

También se puede ver que los tiempos más grandes en cada implementación se dan cuando hay sólo un hilo trabajando. Esto afirma en números que la concurrencia mejora la eficiencia.

3.2. Experimentación cuantitativa

Ya habiendo hecho la experimentación cualitativa y los tests correspondientes asumiremos de ahora en más que las implementaciones funcionan correctamente, por lo que sólo nos centraremos en su performance.

Para paliar los aspectos no controlables de los experimentos decidimos correr cada escenario para cada algoritmo 100 veces y tomar promedio. Tomaremos esa medida para hacer un análisis de performance de las implementaciones en distintos escenarios.

3.2.1. La variación en la proporción de hilos que ejecutan determinadas operaciones, manteniendo constante la cantidad de hilos totales y la cantidad de operaciones por hilo.

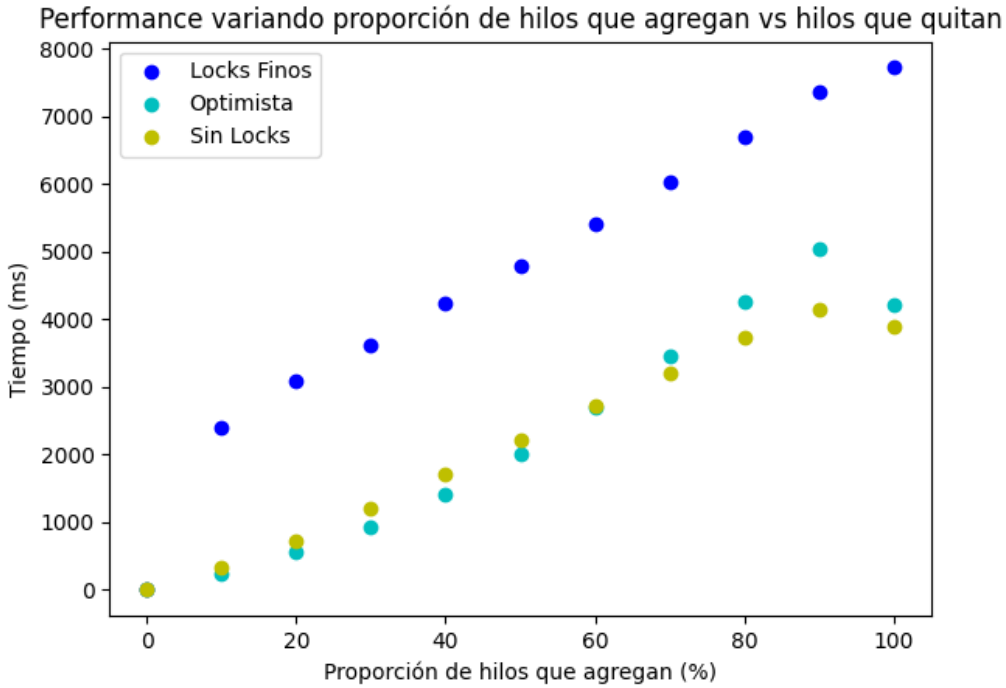
Para este grupo de experimentos decidimos tomar todas las combinaciones posibles de hilos para analizar proporciones y tiempos. Entonces realizamos tres experimentos llamados agregarVsQuitar, agregarVsPertenecer y quitarVsPertenecer.

3.2.2. Agregar vs Quitar

Decisiones tomadas para este experimento: La cantidad total de hilos se fijó en 30 y la cantidad de operaciones por hilo se fijó en 10.000.

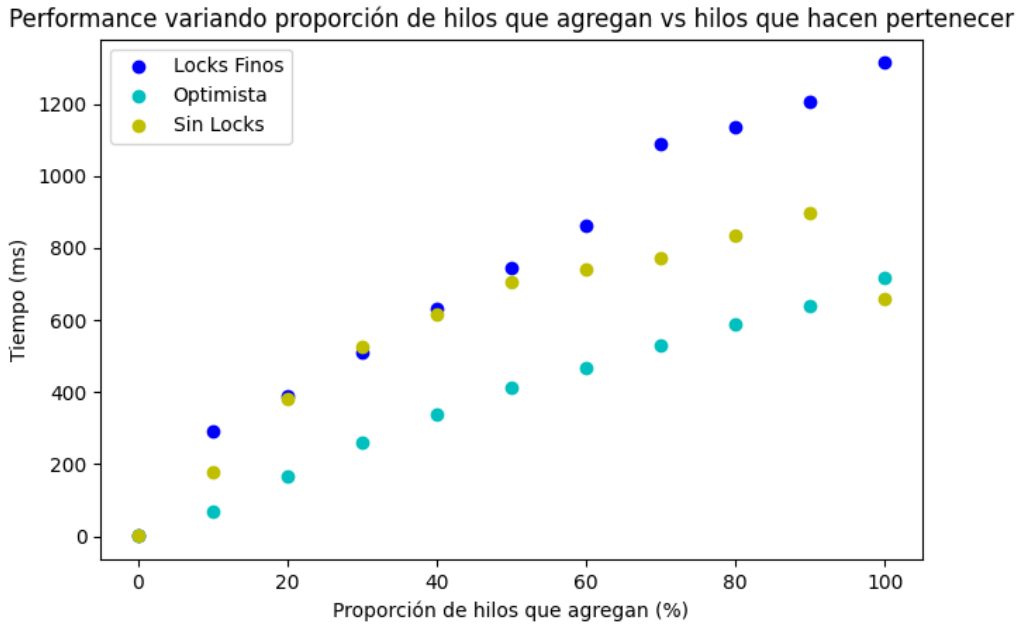
Tras realizar el experimento vemos una correlación inversa entre la performance y la proporción de hilos que agregan; es decir, a mayor proporción de hilos que agregan peor es la performance. Dicha correlación parece mantenerse constante para las tres implementaciones. Por otro lado, cabe mencionar que en cada escenario la implementación de *locks finos* tomó un tiempo considerablemente mayor a las otras dos implementaciones, esto lo atribuimos a que en ese caso se toman y se liberan una gran cantidad de locks, por eso el incremento de tiempo. Otra observación destacable es que de 0 % a 60 % la implementación *optimista* tiene una leve mejora en performance contra la *sin locks*, y de 70 % en adelante es peor y su curva parece estar disparándose un poco.

Finalmente, vemos una bajada de tiempo para el caso donde todos los hilos son de agregar en las implementaciones *optmista* y *sin locks*, pero no podemos atribuirle una razón en particular.



3.2.3. Agregar vs Pertenecer

Decisiones tomadas para este experimento: La cantidad total de hilos se fijó en 30 y la cantidad de operaciones por hilo se fijó en 1.000.

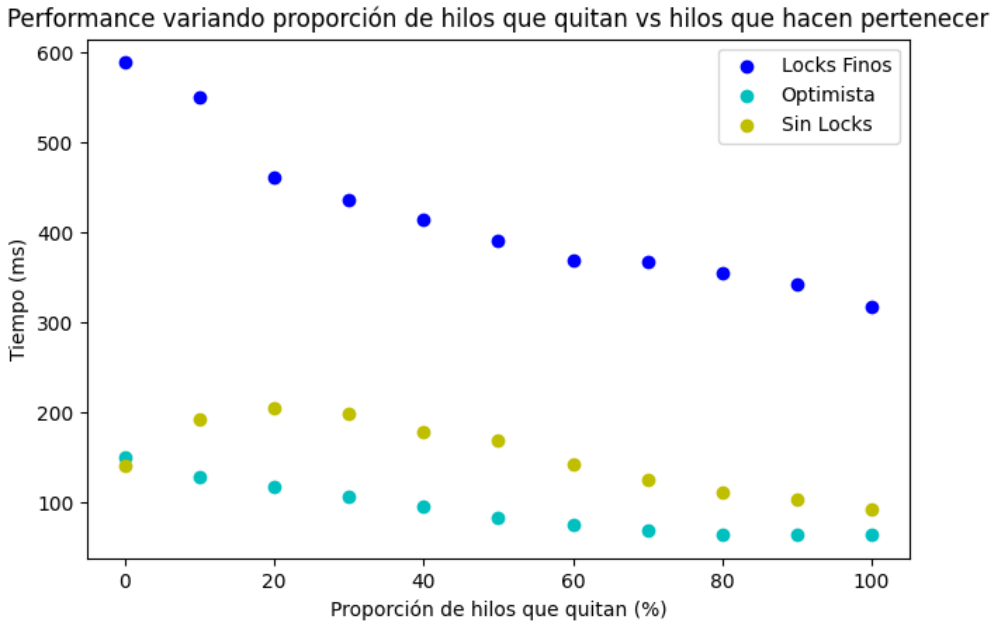


En este experimento vemos nuevamente una correlación inversa entre la performance y la proporción de hilos que agregan, aunque estos resultados guardan ciertas diferencias con los resultados del experimento anterior. Por ejemplo, las implementaciones *optimista* y *sin locks* ya no están tan cerca una de la otra en performance, siendo ésta primera la mejor en todo el experimento. Del 20 % al 50 % *sin locks* y *locks finos* se asemejan bastante, aunque luego *sin locks* parece reducir taza su incremento de tiempo a partir del 60 %. Nuevamente vemos la anomalía

en *sin locks* cuando todos los hilos son de agregar, aunque cabe observar que ahora *optimista* no presenta dicha anomalía.

3.2.4. Quitar vs Pertenecer

Decisiones tomadas para este experimento: La cantidad total de hilos se fijó en 30 y la cantidad de operaciones por hilo se fijó en 1.000. Al inicio de cada corrida el conjunto posee los elementos consecutivos del 1 al 10.000, esto es para que haya elementos para eliminar. Los hilos eliminar y pertenecer usan números en el rango del conjunto inicial.

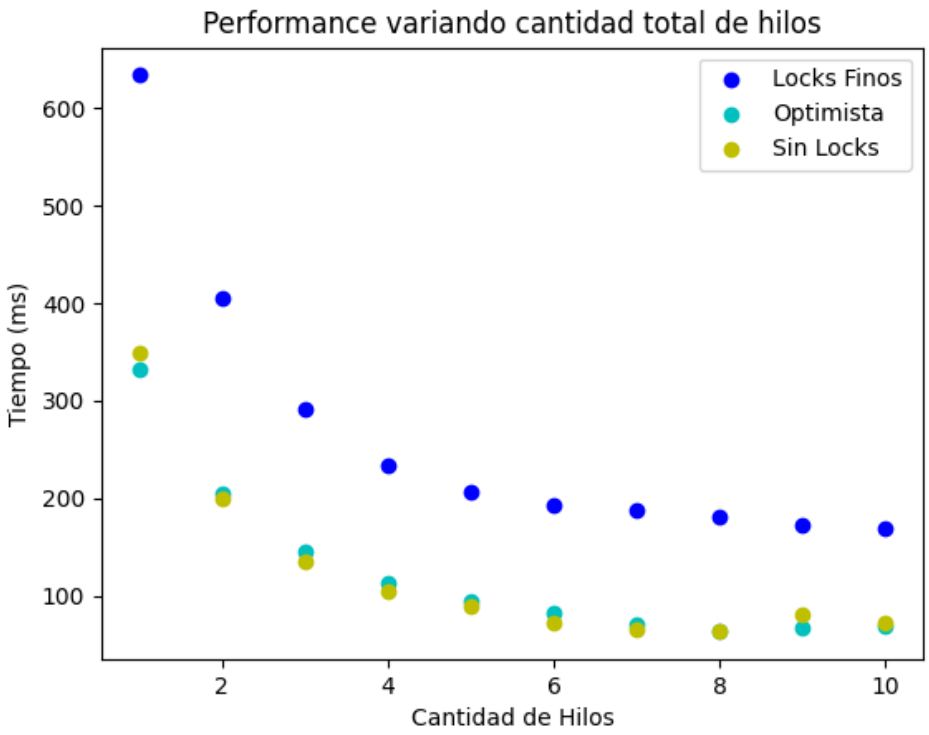


En este gráfico se puede ver que en *sin locks* y *optimista* se mantiene el tiempo o tarda un poco menos llegando a utilizar gran porcentaje para la eliminación. Se nos ocurre que puede ser porque ambas operaciones son similares. Sin embargo, en el *sin locks* el pertenece no tiene que iterar hasta poder realizar la acción y nos sorprendió que 100 % de hilos ejecutando pertenecer no sea lo más rápido. Pensamos que es porque la lista cada vez se queda con menos elementos y ejecuta en una estructura más chica. Los locks finos tienen implementaciones similares para las funciones pertenece y eliminación, entonces tiene sentido que cuanto más se elimine, más se achique la estructura y menos locks se toman; Por lo tanto menor tiempo.

3.2.5. La variación de la cantidad de hilos totales, si se preserva el número total de operaciones:

Este experimento fue pensado con la idea de que dado un *pool* de trabajo fijo, a mayor concurrencia mejor será la performance.

Las decisiones tomadas para este experimento fueron: Fijamos la cantidad máxima de hilos en 10 y el número total de operaciones en 10.000.

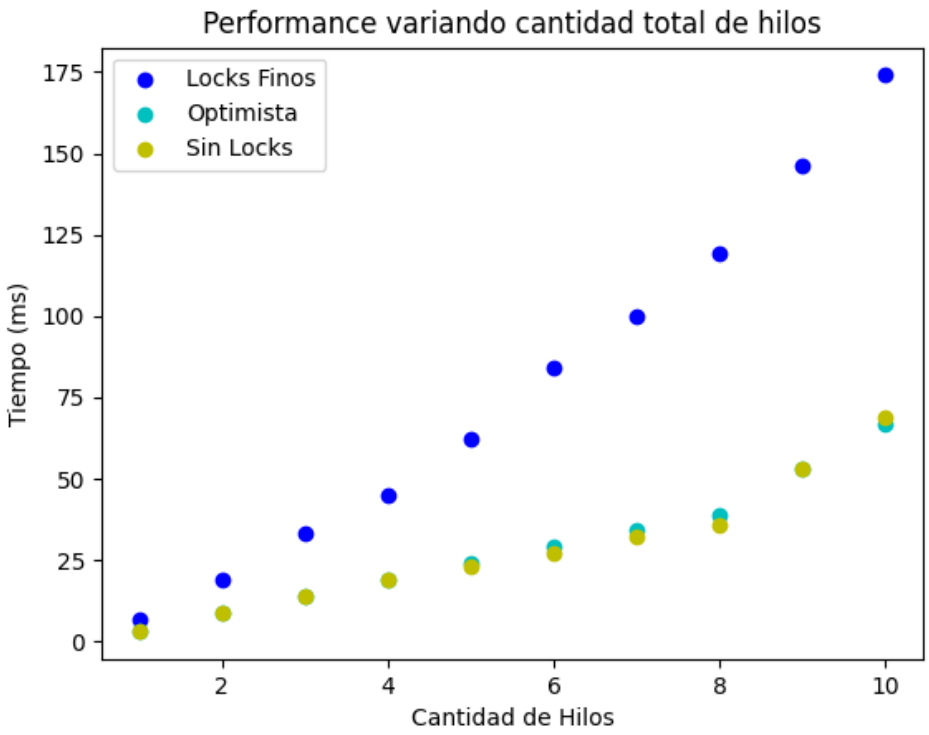


En este experimento la idea era analizar la performance a medida que aumenta la concurrencia para realizar una tarea en particular, en este caso agregar 10.000 elementos al conjunto. Tal como esperábamos las 3 implementaciones muestran su peor performance cuando hay un sólo hilo de ejecución, y a medida que se agregan hilos los tiempos bajan logarítmicamente hasta no presentar mejoras significativas (de 6 hilos en adelante). Una vez más notamos cómo la implementación *locks finos* es la peor a lo largo del experimento, y que las otras dos se asemejan mucho en estos escenarios.

3.2.6. La variación de la cantidad de hilos totales, si se mantiene constante la cantidad de operaciones que ejecuta cada hilo.

Decisiones tomadas para este experimento: Fijamos la cantidad máxima de hilos en 10 y el número total de operaciones en 1.000.

Como fijamos la cantidad de operaciones por hilo esperamos que a medida que aumenten los hilos aumente el tiempo en realizar las operaciones, simplemente porque se están realizando 1.000 operaciones más por hilo. Aunque la tasa de aumento dependerá de cada implementación, esperamos que la *locks finos* sea la que peor performance tenga, dado a que esperamos que tome y libere muchos locks.



Los resultados obtenidos respaldan nuestra hipótesis en cuanto a la correlación entre la cantidad de hilos y el tiempo. Esto lo atribuimos a que mientras más hilos haya trabajando, más probabilidades hay de estar bloqueado esperando un lock o si se modificó la lista en el medio y se debe volver a empezar.

También se ve, igual que en experimentos anteriores, que las implementaciones *optimista* y *sin locks* se asemejan mucho en sus resultados a lo largo de todo el experimento. Ambas muestran un aumento lineal en el tiempo y que parece dispararse un poco llegando a 9 y 10 hilos.

Por otro lado, como esperábamos también, la implementación de *locks finos* es la peor en cuanto a performance, mostrando una taza de incremento mayor a las otras dos, aunque también sigue una curva lineal.

4. Conclusión

En este trabajo práctico, se analizaron tres implementaciones de conjuntos sobre listas enlazadas concurrentes: *granularidad fina*, *optimista* y *sin locks*. Los experimentos realizados incluyeron diferentes escenarios de agregado, eliminación y pertenencia de elementos, con variaciones en la cantidad de hilos, operaciones por hilo y proporción de hilos.

Viendo los resultados obtenidos se vio que las implementaciones *optimista* y *sin locks* demostraron ser más eficientes en términos de tiempo. Además, se observó que la concurrencia mejoró significativamente la eficiencia, ya que los tiempos de ejecución disminuyeron cuando se utilizaron múltiples hilos en comparación con un solo hilo. Esto se puede ver tanto en la experimentación cualitativa como en la cuantitativa. Por último, se pudo observar que si se aumenta la cantidad de hilos, manteniendo la cantidad de operaciones por hilo, la performance empeora.