DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: Concurrencia en listas enlazadas

Programación concurrente

Integrante	LU	Correo electrónico
Juan Pablo Miceli	424/19	micelijuanpablo@gmail.com
Victoria Antonella Varani	451/19	victoriavarani@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

1. Introducción

En el ámbito de la computación, la gestión y manipulación eficientes de las estructuras de datos son de suma importancia. A medida que escala la complejidad de los sistemas informáticos, surge la necesidad de técnicas de programación concurrentes para aprovechar el poder del paralelismo y mejorar el rendimiento general del sistema. Una de esas estructuras de datos cruciales para este fin es la lista enlazada.

Las listas enlazadas tienen la característica de ser simples y flexibles a la hora de almacenar y recuperar datos. Sin embargo, su naturaleza secuencial, muchas veces, limita su potencial de rendimiento en procesos concurrentes. Cuando varios procesos quieren acceder y modificar simultaneamente la lista, las implementaciones tradicionales pueden generar cuellos de botella, lo que degrada la performance de la lista, y por ende el rendimiento general del sistema.

La idea central detrás de las listas enlazadas concurrentes es introducir mecanismos de sincronización que aseguren la correctitud y la consistencia y al mismo tiempo permitan operaciones concurrentes. Se han desarrollado varias estrategias y técnicas de sincronización para abordar los desafíos asociados con el acceso simultáneo, como garantizar la atomicidad, prevenir la corrupción de datos y evitar condiciones de carrera.

Un enfoque popular es la sincronización basada en *locks*, en la que se emplean *locks* para garantizar el acceso exclusivo a las secciones críticas de la lista enlazada. Este enfoque proporciona exclusión mutua, lo cual garantiza que solo un proceso puede modificar la lista a la vez.

En este trabajo buscamos profundizar el estudio de las listas enlazadas para programas concurrentes, explorando varios mecanismos y estrategias de sincronización. Analizaremos los pros y contras asociadas a diferentes enfoques y discutiremos los desafíos inherentes al diseño e implementación de los algoritmos propuestos. Por último, haremos un análisis de los resultados empíricos obtenidos en escenarios que utilizan diferentes implementaciones de este tipo de listas.

2. Granularidad Fina

La idea más intuitiva para implementar listas concurrentes es usar un solo *lock* para todo acceso o modifcación de la lista, dandonos un algoritmo de granularidad gruesa, libre de inanición si el *lock* utilizado es fair. Este tipo de sincronización no es escalable a la hora en que varios procesos quieran acceder a la vez un recurso ya que genera un constante cuello de botella. Es por ello que, como primer acercamiento a algoritmos avanzados de listas para programas concurrentes, vamos a estudiar la técnica de sincronización con granularidad fina.

A diferencia de la sincronización de granularidad gruesa, que bloquea toda la lista durante una operación, la sincronización de granularidad fina parte tomando en cuenta que las operaciones que actúan sobre distintas partes no necesitan ejecutar en exclusión mutua. Por ello, se divide la lista en unidades más pequeñas (independientes), como nodos, y se aplica mecanismos de sincronización a un nivel más localizado para asegurarnos sincronización solo cuando se esté intentando acceder al mismo componente al mismo tiempo. Si utilizamos una sincronización basada en *locks*, podemos añadir un *lock* a cada nodo, junto con los métodos *lock()* y *unlock()*. De esta manera, a medida que un proceso atraviesa la lista, se bloquea cada nodo cuando es visitado y luego se suelta. Notar que, cada *thread* mantiene de a dos nodos consecutivos lockeados, en el caso del *remove*, son necesarios el nodo a eliminar y su predecesor, y en el caso del *add*, se necesitan el predecesor y el sucesor del nuevo nodo a insertar. Es importante que todos los métodos de la lista deben adquirir el *lock* en el mismo orden, empezando por la cabeza hacia la cola, y tomando el *lock* de un nodo solo si ya se tiene el *lock* sobre su predecesor, así garantizamos progreso y ausencia de *deadlocks*.

3. Sincronización optimista

Aunque la sincronización con granularidad fina representa una mejora con respecto a la sincronización con granularidad gruesa, el enfoque basado en adquisición y liberación de *locks* puede resultar ineficiente en escenarios donde se produce una larga cadena de *locks*. Esta limitación ha impulsado la búsqueda de métodos de sincronización que mejoren la concurrencia y el rendimiento, y es aquí donde entra en juego la sincronización optimista, ofreciendo varias ventajas significativas.

En la sincronización optimista, las operaciones en listas concurrentes se diseñan para recorrer la lista sin adquirir *locks*, lo que permite que los procesos avancen sin retrasos causados por conflictos. En lugar de bloquear el acceso a los nodos, se utiliza un enfoque basado en la validación. Cuando un nodo se encuentra, este es bloqueado temporalmente y luego se verifica que el componente no haya cambiado durante el intervalo entre el momento en que se inspeccionó y el momento en que se bloqueó. Específicamente se valida si los nodos están en la lista y si son todavía adyacentes. Si un conflicto de sincronización hace que se bloqueen los nodos incorrectos, se desbloquean y se comienza de nuevo. Normalmente, este tipo de conflicto es raro, por eso es que llamamos a esta técnica sincronización optimista.

Notemos que este algortimo trae menos adquisiciones y liberaciones de *locks* por lo que es mas performante y permite una mayor concurrencia pero requiere recorrer la lista dos veces y no es libre de inanición. Esto último se debe a que un *thread* podría ser demorado por siempre si nuevos nodos son añadidos y removidos constantemente. De igual manera se espera que este algoritmo performe bien en la práctica.

4. Sin Lock

En un enfoque sin *locks*, no se requiere que los procesos adquieran *locks* exclusivos en la estructura de datos para realizar operaciones. En su lugar, emplean algoritmos y técnicas que aseguran el progreso incluso en presencia de modificaciones simultáneas por parte de otros procesos.

La sincronización sin *locks* depende en gran medida de las operaciones atómicas, por ejemplo *compareAndSet()*, que son indivisibles y proporcionan un comportamiento coherente y fiable frente a la concurrencia. Las operaciones atómicas permiten que los procesos modifiquen la lista compartida sin necesidad de bloqueos. Al aprovechar la atomicidad, se garantiza que las operaciones se ejecuten por completo o no se ejecuten en absoluto, lo que evita estados inconsistentes intermedios. En el caso de que el *compareAndSet()* falle, se debe reintentar la operación desde el inicio, lo cual implica que este método, al igual que el anterior no es libre de inanición.

Este método de sincronización trae un problema al actualizar un nodo cuando este fue eliminado. Para subsanar este posible error se usa sincronización *lazy*. La sincronización *lazy* en métodos sin *locks* aborda el problema al aplazar la sincronización hasta que surjan conflictos o condiciones específicas. Una operación se divide en dos fases: el componente es removido lógicamente seteando una bandera y luego, en otro momento, el componente es removido físicamente deslinkeandolo del resto de la lista. Para la implementación es importante mantener el invariante de que cada nodo no marcado es alcanzable. En esta implementación, el borrado físico es realizado al comienzo de cada operación mientras se avanza al nodo objetivo.

Se intentará borrar físicamente cada nodo encontrado que haya sido borrado lógicamente en un paso previo. En el caso de que este borrado físico falle, se reinicia la operación y se recorre la lista desde el inicio nuevamente.

5. Experimentación

Para evaluar el rendimiento y la eficacia de los algoritmos de sincronización en la programación concurrente, los experimentos empíricos son cruciales. En esta sección, analizamos los experimentos realizados para comparar el rendimiento y el comportamiento de los algoritmos de sincronización de granularidad fina, sincronización optimista y sin locks. Estos experimentos brindan información sobre cómo funcionan estos métodos de sincronización en escenarios del mundo real y ayudarán en un futuro a seleccionar el algoritmo más adecuado para los requisitos de concurrencia específicos.

Se programaron los siguientes tres métodos:

- **GF**: Algoritmo de lista enlazada concurrente con granularidad fina, de la sección 2
- **SO**: Algoritmo de lista enlazada concurrente con sincronización optimista, de la sección 3.
- **LF**: Algoritmo de lista enlazada concurrente sin *locks*, con sincronización lazy, de la sección 4.

Los experimentos fueron diseñados para evaluar los algoritmos de sincronización en el contexto de listas enlazadas concurrentes, una estructura de datos común utilizada en varias aplicaciones. A continuación detallaremos

5.1. Configuración de los experimentos

- Se estableció un escenario con una cantidad fija de hilos que se crearon al inicio del programa y compartieron una única estructura de datos concurrente inicialmente vacía.
- Todos los hilos realizaron la misma cantidad fija de operaciones.
- Se definieron dos tipos de hilos: agregadores y removedores. Cada hilo se dedicó exclusivamente a un tipo de operación, aunque diferentes hilos pudieron realizar operaciones diferentes en la lista.
- Los experimentos se llevaron a cabo en una misma PC (*CPU Intel Core i7 @ 2.6 GHz con 6 cores, 12 hilos ya que el mismo cuenta con hyperthreading y 16 GB de memoria RAM*) utilizando Java como lenguaje de programación.

5.2. Ejecución y repetición

- Cada escenario se ejecutó 50 veces para obtener una muestra representativa de los resultados.
- Se promediaron los resultados de las repeticiones para obtener los datos a analizar y comparar.
- Para los experimentos que no varían la cantidad de agregadores y removedores, el ratio será de 50 % agregadores, 50 % removedores.
- Los hilos agregadores se inicializan con la cantidad de *adds* deseados y la lista sobre la cual se quiere operar. Estos hilos agregarán elementos aleatorios entre 0 y la cantidad de operaciones a realizar.
- Los hilos removedores se inicializan con la cantidad de *removes* deseados y la lista sobre la cual se quiere operar. Estos hilos removerán elementos aleatorios entre 0 y la cantidad de operaciones a realizar.

Se buscó repetir los experimentos múltiples veces y promediar los resultados para garantizar la robustez y la fiabilidad de los datos obtenidos. Esto permite obtener una visión más precisa y confiable del rendimiento de los algoritmos de sincronización en listas concurrentes.

5.3. Experimento 1: Variación en la densidad de operaciones

Presentación

Para llevar a cabo el experimento, se diseña un conjunto de casos de prueba para simular el acceso simultáneo a la lista por diez hilos. Cada hilo hará diez mil operaciones, ya sea añadiendo elementos a la lista o eliminando elementos existentes. Teniendo entonces un total de cien mil operaciones por caso de prueba. Variaremos la proporción de hilos agregadores y removedores en los diferentes casos de prueba para observar cómo afecta el rendimiento y el comportamiento de las distintas técnicas de sincronización, empezando con un hilo agregador y nueve hilos removedores y aumentando la proporción de hilos agregadores hasta obtener nueve hilos agregadores y un hilo removedor.

Es necesario notar, que al estar agregando y removiendo elementos generados de forma aleatoria en los casos donde haya una gran densidad de agregadores, muchos de esos *adds* terminarán abortando debido a que el elemento ya pertenecía a la lista. Lo mismo pasará para los *removes* en los casos donde haya una gran densidad de hilos removedores. Debido a esta compensación al variar la densidad de agregadores y removedores, se estima que esto no sesgará el propósito del experimento de ninguna forma.

El experimento tiene como objetivo proporcionar información sobre cómo funciona cada técnica bajo diferentes cargas de trabajo y ayudará a entender los pros y contras de cada mecanismo de sincronización evaluado para las listas.

Hipótesis

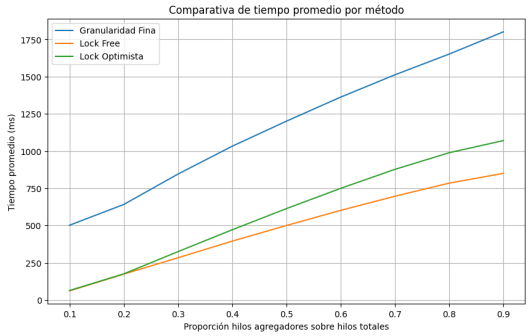
Dado que pensamos que para cada algoritmo, tanto la complejidad de *add* como la de *remove* es la misma, creemos que los tiempos de ejecución no se verán afectados por las diferentes densidades de operaciones.

Esto es porque se suele hacer la misma cantidad de *locks* u operaciones costosas tanto en el *add* como en el *remove*.

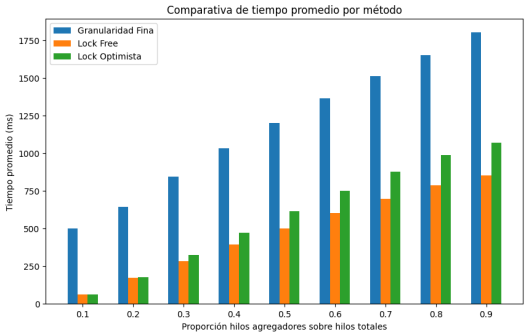
Hipótesis: los tiempos de ejecución no se verán significativamente afectados por las diferentes densidades, pero si ligeramente por el método de sincronización utilizado.

Resultados

A partir de los datos recopilados en las corridas de los algoritmos con diferentes densidades de *threads* agregadores generamos dos gráficos que sintetizan la información obtenida.



(a) Comparación de tiempos promedio de ejecución de GF, SO y FL por cantidad de hilos agregadores.



(b) Comparación de tiempos promedio de ejecución de GF, SO y LF por cantidad de hilos agregadores.

Figura 1: Comparación entre los métodos GF, SO y LF para datasets con número de hilos agregadores variable.

Como se puede ver en la figura 1 los tres métodos obtuvieron un crecimiento lineal en su tiempo promedio ante el aumento en la densidad de hilos agregadores. Esto contrasta con parte de la hipótesis planteada previamente y puede deberse a que no se tuvo en cuenta que entre más hilos agregadores, más grande es en promedio la lista, llevando a que recorrer la lista sea cada vez más costoso, principalmente al usar granularidad fina la cual tendrá que realizar un *lock* y un *unlock* por nodo, operaciones muy costosas en relación a simples modificaciones de memoria.

Por otro lado, se puede observar de forma clara como granularidad fina es el método de sincronización con peor rendimiento de los tres. Sus contrapartes, quienes tienen algoritmos más elaborados presentan tiempos de ejecución sustancialmente menores como planteamos en la hipótesis. Por último, también podemos observar una leve diferencia entre sincronización optimista y lock free, creemos que se debe a que la implementación de la sincronización sin locks se hizo con un método lazy, permitiendo que no requiera recorrer una segunda vez la lista al momento de validar, bajando la complejidad del algortimo.

En resumen, los resultados de los experimentos respaldan la idea de que la granularidad fina tiene un rendimiento inferior en comparación con los métodos de sincronización optimista y lock-free en listas enlazadas concurrentes. Estos últimos métodos, con enfoques más sofisticados y menos dependencia de locks, demuestran una mejor eficiencia y rendimiento. Sin embargo, se debe tener en cuenta que el desempeño puede variar dependiendo de las características del escenario de uso, en este caso, entre más operaciones de agregar los algoritmos tienden a enlentecer.

5.4. Experimento 2: Variación en la cantidad de hilos totales, operaciones totales constantes

Presentación

En este experimento, nuestro objetivo es investigar la relación entre el número de procesos y el rendimiento de las operaciones de listas enlazadas cuando se emplean métodos sincronizados. Específicamente, enfocaremos en las tres técnicas de sincronización mencionadas previamente.

Para llevar a cabo el experimento, configuramos un escenario en el que un número variable de hilos acceden a una misma lista y la modifican al mismo tiempo agregando o removiendo elementos. La cantidad de *threads* involucrados empezará en dos, y aumentará con paso dos, hasta llegar a diez hilos. La cantidad de operaciones totales por caso de prueba será de 96.000 y serán divididas equitativamente entre todos los *threads* participantes.

Al medir el tiempo de ejecución de las operaciones de la lista en diferentes escenarios, podemos analizar cómo la técnica de sincronización y el número de hilos influyen en el rendimiento. Nos centramos en dos aspectos clave:

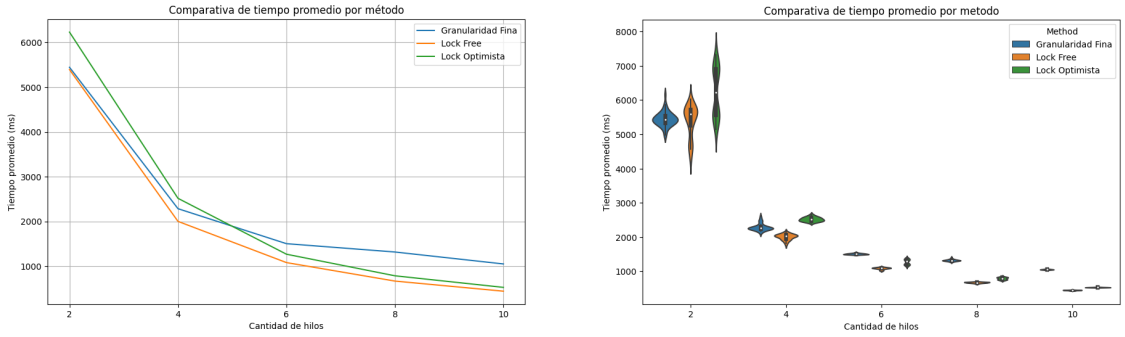
- Escalabilidad: se busca investigar si al aumentar la cantidad de hilos mejora el rendimiento general. La escalabilidad se refiere a la capacidad del sistema concurrente para manejar de manera eficiente una mayor carga de trabajo con hilos adicionales. Analizamos cómo los métodos sincronizados manejan la creciente concurrencia y si utilizan efectivamente el paralelismo para mejorar el rendimiento.
- Contención: a medida que aumenta la cantidad de hilos, la sobrecarga de contención y sincronización también puede aumentar. Las contenciones ocurren cuando varios hilos compiten por los mismos recursos, lo que lleva a un mayor bloqueo y serialización, lo que puede limitar la escalabilidad. Evaluamos la capacidad de las técnicas de sincronización para minimizar la contención.

Hipótesis

A medida que aumenta el número de *threads*, el rendimiento de las operaciones de lista variará entre las diferentes técnicas de sincronización. La sincronización GF puede mostrar rendimientos decrecientes, mientras que se espera que la SO y el enfoque LF ofrezcan una mejor escalabilidad y un rendimiento mejorado con una concurrencia creciente.

Hipótesis: el rendimiento mejorará a mayor número de *threads*. De igual manera, esta mejora no alterará el patrón observado: GF tiene el peor rendimeinto mientras SO y LF tienen rendimientos parecidos.

Resultados



(a) Comparación de tiempos promedio de ejecución de GF, SO y LF por cantidad de hilos.

(b) Tiempos promedio de ejecución de GF, SO y LF por cantidad de hilos y su varianza.

Figura 2: Comparación entre los métodos GF, SO y LF para datasets con operaciones totales constantes y cantidad de hilos variable.

Como se puede ver en la figura 3 efectivamente el rendimiento ante un mayor número de *threads* es mejor sin importar el método. Esto indica que la concurrencia tiene un impacto positivo en el tiempo de ejecución, ya que múltiples *threads* pueden trabajar simultáneamente en la lista enlazada. Tambien se puede ver que la tasa de mejora en el rendimiento disminuye a medida que se aumenta la cantidad de *threads*, esto se puede adjudicar a que empieza a ser cada vez mayor el overhead causado por el cambio de contexto entre *threads* y su compentencia por la CPU.

Por otro lado, podemos observar un resultado que se contradice con la hipótesis planteada. Ya que GF comenzó teniendo rendimiento mejor que SO y similar a LF, al momento de usar dos *threads*. Esto llama

la atención, ya que era esperado que el rendimiento de GF se encuentre siempre por debajo de los otros métodos. Para evaluar una posible causa de este resultado, se evaluó con más detalle la figura 2b, aquí se puede ver que GF es muy consistente en sus resultados, presentando una varianza reducida, mientras que SO tiene una alta varianza, lo cual indica que este algoritmo no es muy confiable para un bajo número de *threads*. Este patrón se revirtió conforme aumentamos la cantidad de hilos, teniendo por primera vez GF como el peor algoritmo al momento de usar seis hilos.

Para resumir, la concurrencia demuestra ser beneficiosa, aunque es necesario evaluar cual es el punto donde el costo de aumentar la cantidad de *threads* deja de ser relevante con relación a la mejora en performance obtenida. También se pudo concluir, que pese a que un método sea muy óptimo para un número alto de *threads*, su performance puede no ser óptima en programas con poca concurrencia.

5.5. Experimento 3: Variación en la cantidad de hilos totales, operaciones por hilo constantes

Presentación

En este experimento se busca evaluar el impacto de la variación en la cantidad total de hilos en el rendimiento de los métodos de sincronización GF, SO y LF, manteniendo constante el número de operaciones por hilo. Cada hilo realiza el mismo número fijo de operaciones en la lista concurrente. Esto se hace para mantener la carga de trabajo constante y poder comparar el rendimiento de los métodos de manera equitativa.

Para este experimento la configuración del escenario es parecido al experimento 2. Un número variable de hilos acceden a una misma lista y la modifican al mismo tiempo agregando o removiendo elementos. La cantidad de *threads* involucrados empezará en dos, y aumentará con paso dos, hasta llegar a diez hilos. La cantidad de operaciones totales por caso de prueba será de 10.000 operaciones por thread, por lo que en el caso de 2 *threads* serán 20000 operaciones y en 10 serán 100.000 operaciones. Nuevamente nos enfocaremos en analizar como el método de sincronización y el número de hilos y por lo tanto de operaciones influye en el rendimiento centrandonos en escalabilidad y contención.

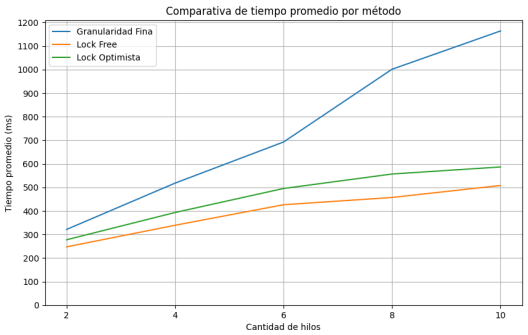
Hipótesis

Al aumentar el número de operaciones totales, se puede esperar que el tiempo total de procesamiento incremente también, ya que todos las operaciones se llevan a cabo sobre la misma lista, y esto puede incurrir en contenciones al trabajar sobre nodos adyancentes. Además, debemos tomar en consideración el tiempo perdido en cambio de contexto entre los distintos hilos.

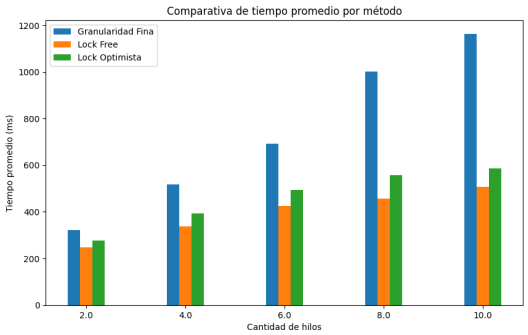
De igual manera, se considera que la ganancia en performance obtenida mediante la concurrencia es mayor a la pérdida discutida en el párrafo anterior, por lo que, duplicar la cantidad de operaciones totales, no implicará que el tiempo tomado sea el doble, si no, significativamente menor.

Hipótesis: El tiempo de ejecución aumentará conforme aumenten la cantidad de operaciones totales a realizar, pero duplicar la cantidad de operaciones no duplicará el tiempo de ejecución.

Resultados



(a) Comparación de tiempos promedio de ejecución de GF, SO y LF por cantidad de hilos agregadores.



(b) Comparación de tiempos promedio de ejecución de GF, SO y LF por cantidad de hilos agregadores.

Figura 3: Comparación entre los métodos GF, SO y LF para datasets con número de hilos agregadores variable.

En la figura 3 se puede observar que los resultados obtenidos respaldan la hipótesis planteada: a medida que se aumenta la cantidad de hilos, el tiempo de ejecución también aumenta. Además, se puede apreciar que el enfoque GF presenta una peor escalabilidad en comparación con los enfoques de SO y LF.

Este fenómeno puede ser atribuido a la contención que se produce cuando múltiples hilos compiten por los mismos recursos. En el caso de GF, se requiere bloquear y desbloquear la lista durante el recorrido, lo cual puede generar una mayor serialización y bloqueo de operaciones. Estos factores impactan directamente en la capacidad de escalar eficientemente el uso de sincronización en GF.

Por otro lado, los enfoques de SO y LF, al no requerir bloquear y desbloquear la lista durante el recorrido, presentan una menor contención y, por lo tanto, exhiben una mejor escalabilidad en comparación con GF. Mientras los aumentos de tiempos eran del doble para SO y LF, para GF era casi del cuadruple.

6. Conclusiones

Con este trabajo práctico vimos tres métodos distintos para implementar listas concurrentes: Granularidad Fina, Lock Optimista y Lock Free. El primero, requiriendo una cantidad de *locks* lineal con respecto al tamaño de la lista, siendo mejorado por el segundo, que resuelve el problema realizando *locks* solo en el entorno del nodo con el cual se quiere trabajar, a costas de la posibilidad de que haya que reiniciar la operación, y por último un método que elimina completamente el uso de locks, usando operaciones atómicas, introduciendo también el overhead de verificar la consistencia de las operaciones.

Esta comparación empírica de los algoritmos de sincronización Granularidad Fina, Lock Optimista y Lock Free destaca la importancia de considerar la escalabilidad al elegir un mecanismo de sincronización apropiado. En particular si uno quiere garantizar la no inanición o poca varianza en los resultados conviene optar por Granularidad Fina aunque su rendimiento en lo práctico sea peor. Pero si se prefiere métodos mas performantes, por los resultados obtenidos, conviene elegir Lock Free o Lock Optimista. Teniendo en cuenta que para Lock Free arrojó resultados ligeramente mejores, pero es necesario contar con variables atómicas en el sistema.

En última instancia, la elección del algoritmo de sincronización debe basarse en los requisitos específicos del sistema, teniendo en cuenta factores como el nivel de contención, la escalabilidad deseada y la cantidad esperada de subprocesos simultáneos.