DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Análisis empírico de implementaciones concurrentes de listas enlazadas

Programación concurrente
1er cuatrimestre 2023

Integrante	LU	Correo electrónico
Pérez Bianchi, Paula	7/20	paulitapb@live.com.ar
Wodka, Valeria Elizabeth	39/20	valewodka@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón 0/ Pabellón I)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Metodología- Implementaciones concurrentes

En este trabajo realizamos un análisis de la performance de tres implementaciones concurrentes de lista enlazada para representar conjuntos. Las implementaciones consideradas fueron: *Listas de granularidad fina*, *Listas con sincronización optimista* y *Listas Lock-Free*. Las primeras dos implementaciones utilizan locks para el manejo de la concurrencia mientras que la última utiliza herramientas atómicas. Estas diferencias hace que sea interesante analizar el tiempo de ejecución en distintos escenarios. Implementaciones cada una de las listas en Java. Para poder realizar la experimentación de manera más cómoda diseñamos una jerarquía polimórfica donde `ConcurrentList` es una clase abstracta y cada una de las subclases son las distintas implementaciones de listas.

Dado que la concurrencia hace que la lista resultante no sea única es imposible aplicar las estrategias de testing habituales. Para poder detectar si nuestras implementaciones tenían fallas implementamos el chequeo de un invariante de representación. Este consiste en verificar que luego de que cada uno de los threads termine su ejecución, la lista resultante cumple:

- El nodo centinela tail es alcanzable desde el head si recorremos la lista usando los punteros next.
- El size de la lista es correcto. Es decir que la cantidad de elementos encontrados al recorrer la lista usando los punteros next es igual al size que mantiene la lista. Para el caso de la lista lock-free hay que no contar los nodos marcados.
- No hay nodos duplicados. Es decir que no hay dos elementos con el mismo hash en la lista.
- Los nodos están ordenados por el hash.

Como usamos estas listas para representar conjuntos la condición del orden no es crucial sin embargo es una propiedad que mantenemos por nuestra implementación.

Para la experimentación planteamos distintos escenarios donde se realizan operaciones concurrentemente sobre listas de enteros y medimos el tiempo de ejecución de cada una de las implementaciones. Al utilizar listas de enteros nos aseguramos que el hash coincide con valor del elemento de la lista por como funciona HashCode en Java. Repetimos 1000 veces cada experimento. Corrimos los experimentos sobre un procesador Intel(R) Core(TM) i5-6200U CPU@2.30GHz de 4 cores con 8 GB de RAM.

2. Experimentos y Resultados

Para la experimentación exploramos que pasa al hacer distinto tipo de operaciones (`add()` y `remove()`) y combinarlas de distintas formas. Además variamos la cantidad de operaciones que realiza cada thread sobre la lista, las zonas de la lista donde trabaja cada thread y la cantidad de threads totales. Para llamar a las operaciones hacemos uso de *ThreadAdd* y *ThreadRemove* (y algunas variaciones de ellos) donde cada thread se dedica exclusivamente a realizar un tipo de operación. A continuación detallamos los experimentos que realizamos y comparamos la performance y comportamiento para cada implementación de lista:

2.1. Experimentos 1 y 2: Para cada tipo de operación y misma cantidad de operaciones por thread variamos la cantidad de threads

Para una cantidad fija de operaciones por thread variamos la cantidad de threads trabajando concurrentemente sobre la lista. En el experimento 1 todos los threads agregan elementos mientras que el experimento 2 todos eliminan. Los elementos que le corresponde eliminar o agregar a cada thread va a depender con la cantidad total de threads. Cada thread i opera sobre los elementos $x \in (0, |Threads| * cantOperaciones)$ que cumplen $x \equiv i \mod (|Threads|)$.

El objetivo de este experimento es observar la variación del tiempo de ejecución, considerando que esta ligado a la cantidad de Threads. Creemos que la implementación de *FineGrainedList* va a ser la más afectada al aumentar la cantidad de threads ya que al hacer el hand-over de los locks para llegar hasta el elemento a agregar o eliminar se va a ver muy afectado al aumentar la competencia entre los threads. Por otro lado las *OptimisticLists* y las *LockFreeLists* van a tener una performance superior dado que la competencia se da en cada nodo.

Para estos experimentos consideramos 250 operaciones por thread y variamos la cantidad de threads con los valores 2, 4 y 8. Medimos el tiempo de ejecución total que lleva correr todos los threads y luego para comparar las distintas variantes usamos el tiempo por operación.

2.1.1. Experimento 1: Agregar Concurrentemente

Los resultados de este experimento los podemos ver en la Figura 1 donde tenemos un gráfico por cada cantidad de threads y en cada uno observamos el tiempo de ejecución por operación separando por el tipo de lista. Como tomamos promedios de las mediciones de repetir el experimento también agregamos el desvío con error-bars. Como esperábamos los tiempos de las *FineGrainedLists* son consistentemente mayores que los de las demás implementaciones. Además podemos ver que al aumentar la competencia entre los threads el tiempo de ejecución es mayor. Esto tiene varias razones, en principio hay un overhead al agregar más cantidad de threads y por otro lado al aumentar la cantidad de threads creas mayor competencia entre los threads por el CPU time. Además en el caso de por ejemplo de la lista lock-free donde los threads solo son evacuados por el scheduler igual van a malgastar su quantum.

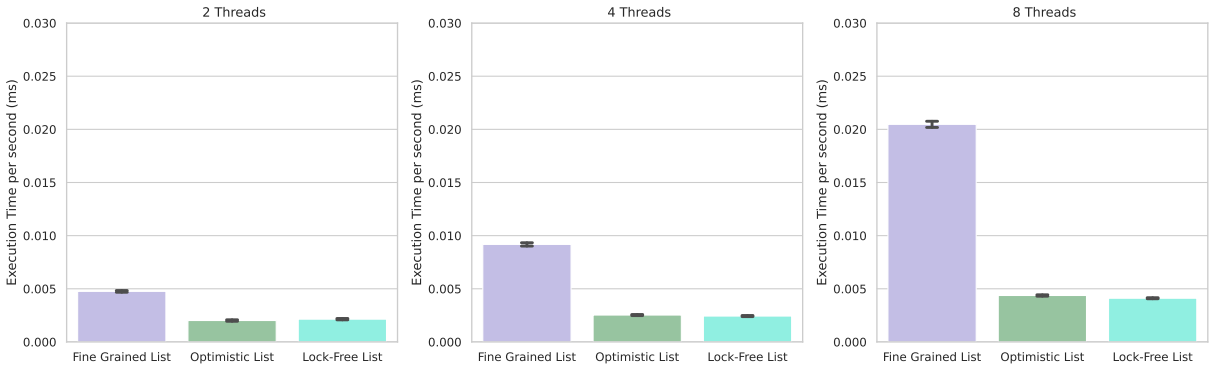

Figura 1: Tiempo de ejecución por operación agregar para cada tipo de lista, variando la cantidad de threads

2.1.2. Experimento 2: Eliminar Concurrentemente

En este experimento la lista comienza con $250 * |Threads|$ y luego se van eliminando de forma concurrente como ya explicamos. Los resultados los podemos observar en la figura 2. Son similares a lo que observamos en el experimento anterior salvo que se puede observar que el tiempo de ejecución por operación es mayor. También se observan las mismas relaciones donde las *FineGrainedLists* tardan más que las otras listas en realizar operaciones. Sin embargo a diferencia del caso anterior podemos observar que la diferencia es algo menor a lo que observabamos en el experimento anterior.

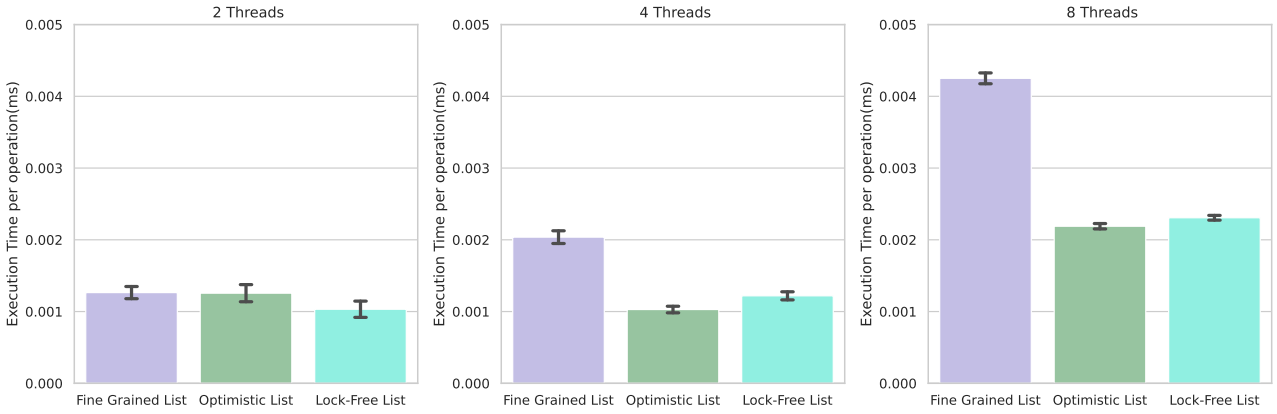

Figura 2: Tiempo de ejecución por operación eliminar para cada tipo de lista, variando la cantidad de threads

2.2. Experimento 3: Mantenemos fija la cantidad de elementos totales a agregar y variamos la cantidad de threads

Para una cantidad fija de elementos a agregar variamos la cantidad de threads que se reparten dicha tarea, es decir que cada thread agrega $\frac{cantAdds}{|Threads|}$. Queremos observar como varia el tiempo total de ejecución al agregar más threads. Cada thread van a agregar de forma alterada, como esta explicado en el Experimento 1 [2.1]

Si bien al aumentar la cantidad de threads cada uno va a agregar menos elementos también se va a aumentar la competencia y contención entre los threads. Suponemos que a mayor cantidad

de threads se ejecutará con mayor velocidad, pero esto va a depender el overhead generado por agregar más cantidad de threads y de la política de scheduling particular del Sistema Operativo. Además creemos que la *FineGrainedList* va a ser claramente la más afectada porque al realizar el hand-over de los locks la competencia sobre los mismos va a ser muy grande.

Para este caso trabajamos sobre 1000 operaciones y variar la cantidad de threads entre 2, 4 y 8. Es decir, tener 2 threads que agregan 500 elementos cada uno, 4 threads que agregan 250 elementos cada uno y 8 threads que agregan 125 elementos cada uno.

En los resultados de este experimento los podemos observar en la figura 3 donde graficamos el tiempo total de ejecución de todos los threads para cada tipo de lista. Pudimos observar que el tiempo de ejecución para *OptimisticLists* y las *LockFreeLists* fue bastante similar para la distinta cantidad de threads, solo en el caso de 2 threads se vio un tiempo de ejecución algo peor aunque la direfencia no es significativa. Además, se hace evidente que luego de 4 threads para *FineGrainList* el tiempo de ejecución empeora por el overhead de agregar tantos threads. Sumando a esto, nuestras computadoras no pueden ejecutar los 8 threads en simultaneo por lo que usar esta cantidad de threads no nos va a ofrecer mejoras.

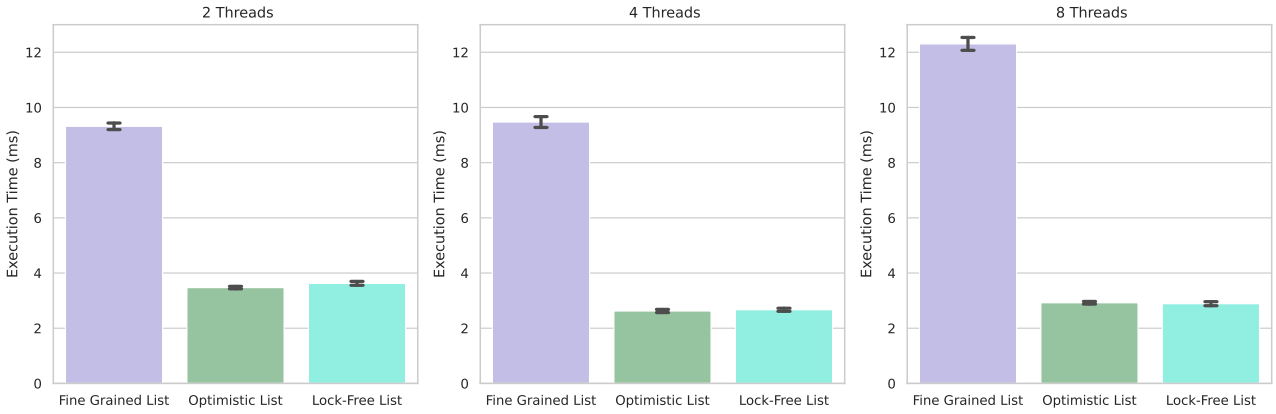

Figura 3: Tiempo de ejecución total de agregar 1000 elementos para cada tipo de lista, variando la cantidad de threads

2.3. Experimento 4: Misma cantidad de adds por thread, variando que elementos agrega cada thread

Para una cantidad fija de operaciones por thread y de threads variamos que elementos debe agregar cada thread. En el primer caso (usado también para el Experimento 1 [2.1]), cada thread agrega los elementos $x \in (0, cantOperaciones)$ que cumplen $x \equiv i \mod (|Threads|)$ donde i es el numero de thread. En el segundo caso, cada thread esta encargado de agregar $\frac{cantOperaciones}{|Threads|}$ elementos consecutivos a la lista. Nos interesa comparar el tiempo de ejecución al aumentar la contención entre threads para cada una de las implementaciones.

Suponemos que para *FineGrainedList* el tiempo de ejecución sea similar ya que al realizar el hand-over de los locks para llegar al elemento aun en los casos donde los threads operen sobre partes "distintas" de la lista igual tienen que competir por los locks de anteriores. Para el caso de *OptimisticList* suponemos que el tiempo de ejecución para el agregado por bloques va a ser menor ya que la competencia por un mismo lock se da únicamente en los elementos borde de los bloques. El resto del tiempo los threads van a trabajar en distintas secciones de la lista. La

situación de *LockFreeList* es muy similar a la de *OptimisticList* pero en vez de que se quede en en lock, como esta implementación no trabaja con locks, lo que va a ocurrir es que se va a quedar reintentando hacer el `CompareAndSet`.

Para generar este experimento, consideramos 4 threads con un total de 1000 operaciones de agregar un elemento a la lista.

Se puede observar en la Figura 4 que para *FineGrainedList* para ambas variantes el tiempo de ejecución es similar como preveíamos. Para *OptimisticLists* y las *LockFreeLists* se obtienen menores tiempos de ejecución al agregar los elementos de forma alternada lo cual nos parece sorprendente ya que no es lo que esperábamos. Creemos que esta diferencia puede darse por particularidades del scheduler, como por ejemplo que para el caso de *OptimisticList* como el los threads no van a dormirse nunca pues la mayoría de las veces pueden acceder directamente a los locks reciban algún tipo de penalización del scheduler.

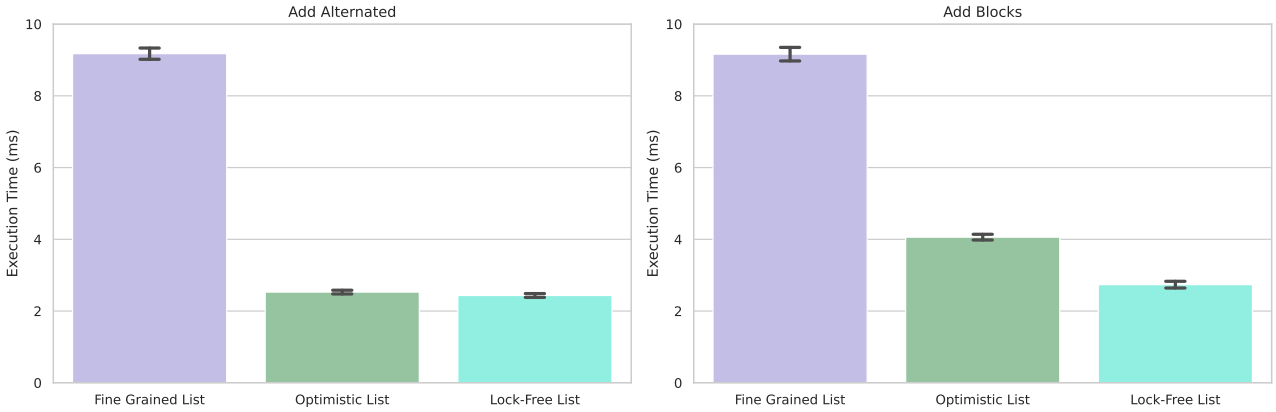

Figura 4: Tiempo de ejecución total de agregar 1000 elementos para cada tipo de lista variando el orden en el que se agregan los elementos

2.4. Experimento 5: Misma cantidad de elementos a agregar y eliminar, variando la cantidad de threads dedicados a cada operación

Para una cantidad fija de elementos variamos la cantidad de threads que agregan y eliminan elementos concurrentemente. Cada thread de un tipo de operación estará encargado de los elementos $x \in (0, cantOperaciones)$ que cumplen $x \equiv i \mod (|ThreadsDeMismaOperacion|)$ Sabiendo la operación eliminar solamente elimina al elemento si este pertenece a la lista queremos observar como varia la longitud de la lista para poder determinar cuantas eliminaciones fueron exitosas. Esperamos que a mayor cantidad de threads que eliminan elementos, menos eliminaciones sean exitosas.

Para poder correr este experimento consideramos 1000 como la cantidad de elementos y variamos la cantidad de threads de la siguiente forma:

1. Un thread para agregar elementos y otro para eliminarlos.
2. Cuatro threads para agregar elementos y uno para eliminarlos.
3. Un thread para agregar elementos y cuatro para eliminarlos.

En la Figura 5 se visualizan los resultados de ejecutar el experimento mil veces y calcular la mediana del tamaño de la lista al finalizar la ejecución de todos sus threds. Se puede notar que a mayor proporción de *ThreadAdd*'s sobre *ThreadRemove*'s se pueden realizar más eliminaciones exitosas. Entendemos que esto ocurre ya que cuantos más elementos lleguen a ser agregados antes de que se arranque a eliminar luego van a poder ser eliminados exitosamente de la lista, mientras que el resto se va agregar luego de que ya se intento eliminarlos. Podemos ver que *FineGrainedList* es la que tiene menos eliminaciones exitosas si hay pocos *ThreadAdd*'s esto tiene sentido ya que es la implementación más lenta como vimos en los experimentos anteriores. Por otro lado *LockFreeList* es consistente en que siempre se pueden eliminar todos los elementos de la lista. Esto se da porque agregar elementos es muy rápido.

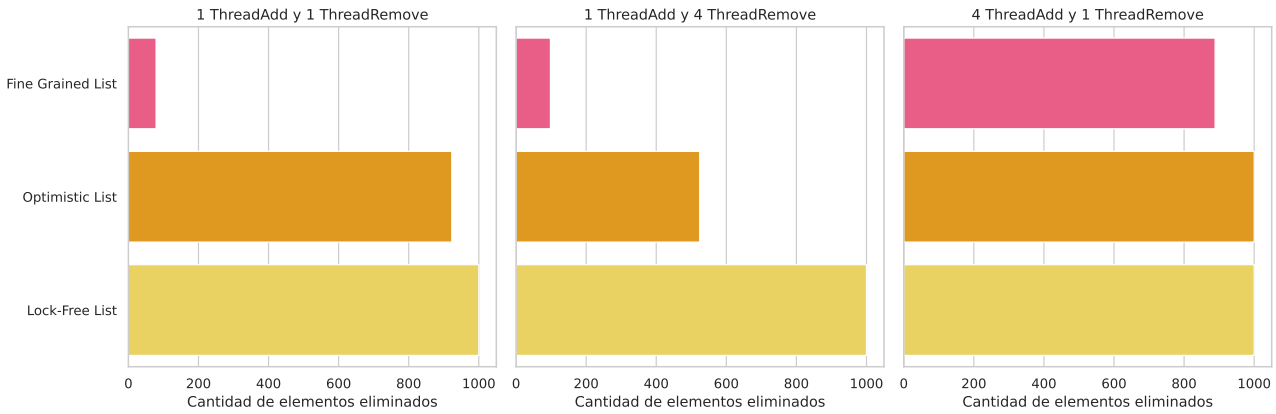

Figura 5: Cantidad de elementos eliminados por cada tipo de lista según cuantos threads están dedicados a cada operación

2.5. Experimento 6: Concurrencia entre threads agregando *sleep_time*

Nos interesa observar como se ejecutan las distintas operaciones que realiza cada thread si forzamos que alguno de ellos se duerma por un tiempo dado, es decir si forzamos que un thread no compita para ejecutar en un momento dado. Consideramos el Caso 1 del Experimento 5 [2.4] y sobre este agregamos la función `sleep(1000)` donde forzamos al thread que elimina los elementos a detenerse por un segundo después de eliminar algún elemento en particular. Estudiamos el caso donde el thread que elimina arranca durmiéndose, el caso donde después de eliminar un cuarto de la lista se duerme y el que duerme luego de eliminar la mitad de la lista. Estimamos que en el primer caso va a poder tener mayor cantidad de eliminaciones exitosas ya que durante el *sleep_time* del *ThreadRemove*, el thread que agrega elementos va a estar ejecutándose sin interrupciones forzadas por la obtención de locks en los casos donde se utiliza esta herramienta de sincronización, agregando así mayor cantidad de elementos antes de que arranquen a ser eliminados.

En la Figura 6 se muestran los resultados de correr mil veces el experimento y calcular la mediana del tamaño de la lista al finalizar la ejecución de sus threads. Se nota claramente que cuanto antes se ejecute el `sleep(1000)` para el *ThreadRemove*, mayor cantidad de elementos van a poder ser eliminados exitosamente, en particular esto afecta a la implementación de *FineGrainList*. Por otro lado se ve que las listas *OptimisticList* y *LockFreeList* como son más rápidas no necesitan dormir al *ThreadRemove* para poder tener una buena cantidad de eliminaciones exitosas.

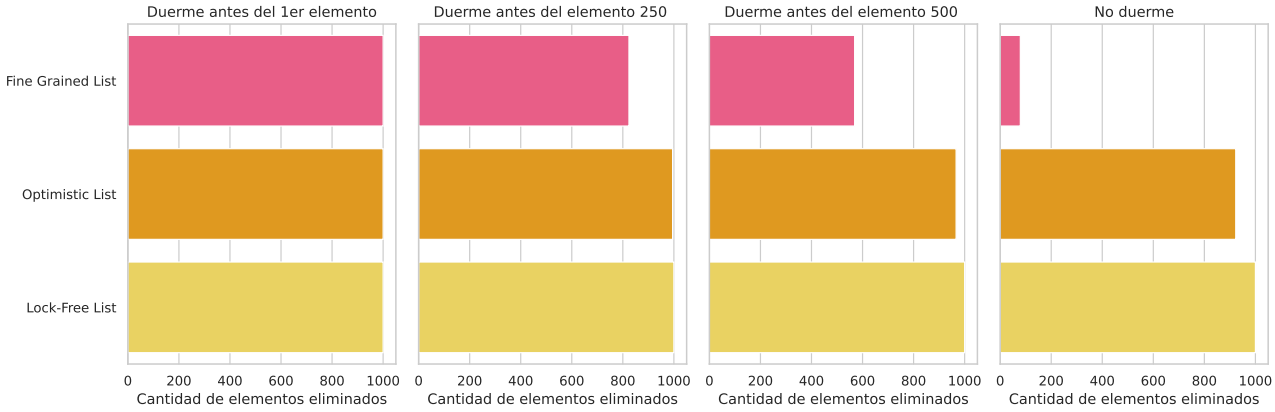

Figura 6: Cantidad de elementos eliminados por cada tipo de lista según cuando duerme el thread que elimina elementos.

3. Conclusiones

Gracias a los experimentos y los resultados obtenidos pudimos analizar las diferencias que presentan las distintas implementaciones concurrentes y llegamos a las siguientes conclusiones principales.

- Se puede concluir que en todos los casos la *FineGrainedList* es el que más tarda en ejecutarse. Lo cual tiene sentido porque tenemos que hacer el hand-over y conseguir muchos locks para poder realizar un tipo de operación por lo que la contención entre los threads siempre va a ser un problema.
- Comparando *OptimisticList* y *LockFreeList* restantes entendemos que suelen dar resultados muy similares, teniendo una leve mejora la implementación de *LockFreeLists*
- Al agregar elementos de a bloques vs. alternadamente nos sorprendió que esta última tuvo mejores resultados. Tal como ya mencionamos suponemos que esto se debe a optimizaciones propias del scheduler y de políticas de fairness entre threads.
- En varios experimentos se percibe que el tiempo de ejecución con únicamente dos threads suele ser muy bueno. Suponemos que esto ocurre ya que si al comienzo se genera un desfasaje entre los threads deja de haber concurrencia de los mismos por un mismo nodo y que además el overhead de hacer threading es menor.
- Al realizar los gráficos notamos que el tiempo de ejecución disminuye a medida que aumentan las repeticiones por lo que creemos que puede ser que el scheduler tenga una política que se adapta para optimizar la ejecución de los threads.
- La verificación del invariante de representación fue útil a la hora de testear que las implementaciones fueran correctas. Creemos que es una herramienta válida para testear el funcionamiento de distintas estructuras concurrentes ya que no podemos asegurar de forma determinística un estado final único.

4. Referencias

Todo lo analizado fue producto de los temas vistos en clase y basado en las diapositivas provistas por la cátedra. A continuación listamos el material de consulta que utilizamos:

- The Art of Multiproccesor Programming by Maurice Herlihy & Nir Shavit
- Diapositivas de listas enlazadas (Campus)