

UNIVERSIDAD TECNOLÓGICA NACIONAL

FACULTAD REGIONAL SANTA FE

Programación Concurrente

Profesores:

Silvina Meinero

Alumnos:

Bernard Maximiliano

Bracalenti Tomas

Paggi Santino

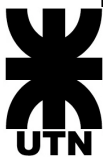
Comisión A

2024

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 2 de 12
		TPN 1	Comisión A
			TPN2

Índice

Introducción.....	3
Implementación de listas.....	3
Lista con sincronización de granularidad fina.....	3
Lista con sincronización optimista.....	3
Lista con sincronización no bloqueante.....	4
Escenarios.....	4
Resultados.....	5
Implementaciones de libro.....	6
Implementaciones modificadas.....	7
Lista con sincronización de granularidad fina.....	8
Lista con sincronización optimista.....	9
Lista de sincronización no bloqueante.....	10
Conclusiones.....	11

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 3 de 12
		TPN 1	Comisión A
			TPN2

Introducción

El trabajo práctico consiste en la utilización de mediciones empíricas para comparar el rendimiento obtenido con distintos tipos de listas enlazadas diseñadas para soportar operaciones concurrentemente. Para la implementación de estas estructuras y su testeo se utilizará el lenguaje Java.

El repositorio con el código fuente de este proyecto puede ser encontrado en: <https://github.com/TomasBracalenti/Concurrente>

Implementación de listas

Inicialmente la implementación de los distintos tipos de listas estaba basada en las mostradas en el libro del curso y en las presentaciones de clase. Sin embargo, los resultados obtenidos con estas no resultaron convincentes por ir en contra de lo esperado. Por ello, se decidió usar las implementaciones exactas propuestas por el libro de la cátedra. Más adelante se hará una comparación entre ambas.

Se definió una interfaz `SynchronizedList<T>` que define un protocolo con los métodos `add()` y `remove()` para un objeto genérico de tipo `T`. Además se definió un método `print()`, que imprime el contenido de la lista, para ayudar a la revisión del código. Los tres tipos de lista analizados implementan esta interfaz y son:

Lista con sincronización de granularidad fina

En esta implementación cada nodo (modelado mediante la clase `Node<T>`) de la lista tiene un lock (instancia de la clase `ReentrantLock`) que permite restringir el acceso a cada uno por separado. Debido a esto, la navegación de la lista se hace mediante lo que se llama "hand-over-hand", esto es, se toma el lock del siguiente nodo y se libera el del anterior hasta llegar a la posición deseada. Esta técnica es usada tanto en la adición como en la eliminación de objetos. La principal ventaja de este algoritmo es su simplicidad, mientras que su punto débil es el hecho de que, durante la navegación, los procesos deben solicitar muchos locks, lo que podría bloquearlos en exceso.

Lista con sincronización optimista

Esta implementación es similar a la anterior, usando el mismo tipo de nodo. La diferencia aparece a la hora de buscar el nodo sobre el que se hará la adición o eliminación, ya que esta se hace ignorando los locks de los nodos. Una vez encontrado el nodo deseado, se toma el lock de este y

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 4 de 12
		TPN 1	Comisión A
			TPN2

el de su predecesor y se valida, nuevamente recorriendo la lista desde su comienzo, que ambos nodos sigan siendo alcanzables desde el head. Si esto se cumple, se procede con la operación.

Esta opción destaca sobre la anterior gracias a que los procesos navegan la lista sin bloquearse. Sin embargo, la validación requiere volver a recorrer la lista desde su comienzo, lo que podría impactar significativamente en el rendimiento.

Lista con sincronización no bloqueante

Esta lista, a diferencia de las anteriores, está compuesta por nodos sin locks (modelados con la clase `LocklessNode<T>`) y que referencian a su sucesor mediante una instancia de la clase `AtomicMarkableReference<V>`. Esta clase proporciona métodos de comparación y asignación que se ejecutan atómicamente, lo que la vuelve ideal para programas multihilo. Adicionalmente, cuenta con una bandera que resulta útil para eliminaciones lógicas.

Los algoritmos primero comienzan buscando el nodo deseado y su predecesor. Durante esta etapa se realiza una eliminación física de los nodos marcados. Luego, se continúa con la adición o eliminación. Los métodos atómicos permiten que los procesos realicen estas operaciones sin la necesidad de bloquearse, lo que vuelve a esta una solución atractiva. La principal desventaja de este tipo de sincronización radica en la dificultad para comprender el algoritmo de búsqueda.

Escenarios

Siguiendo las pautas establecidas en la consigna, se definieron escenarios de ejecución en base a un conjunto de variables: cantidad de hilos, cantidad de operaciones realizadas por cada hilo, proporción de hilos que ejecutan operaciones de adición y tipo de lista compartida entre estos.

Los escenarios fueron modelados con la clase `Scenario`, que cuenta con un constructor donde se asignan todas las variables anteriormente mencionadas y un método `run()` que se encarga de su ejecución. Esta consiste de los siguientes pasos:

1. Sleep de 50 milisegundos para que el Garbage Collector de Java tenga el tiempo suficiente de eliminar todo proceso o variable que pudiera interferir en la ejecución.
2. Creación de los hilos según las variables que definen el escenario.
3. Medición de tiempo inicial.
4. Liberación de los hilos para que comiencen su ejecución.
5. Bloqueo en un semáforo hasta que todos los hilos finalicen su ejecución.
6. Medición de tiempo final y retorno.

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 5 de 12
		TPN 1	Comisión A
			TPN2

Los hilos encargados de interactuar con las listas fueron modelados mediante la clase `OperationThread`, que implementa la interfaz `Runnable`. El constructor de esta le asigna a este el tipo (adicción o eliminación) y la cantidad de operaciones que debe realizar, la lista sobre la que deben operar y los semáforos necesarios para coordinarse con la instancia de `Scenario` que lo creó. Su método `run()` sigue los siguientes pasos:

1. Bloqueo en un semáforo de partida para comenzar su ejecución al mismo tiempo que el resto de hilos del escenario.
2. Ejecución de la operación que le corresponde. Tanto si es una adición como una eliminación, se llama al método correspondiente de la lista con una instancia de la clase `Integer` con un valor aleatorio entre 0 y 1000.
3. Repetición del paso 2 la cantidad de veces establecida.
4. Liberación del semáforo de llegada para informar el fin de su ejecución al `Scenario` creador.

Los valores elegidos para las variables mencionadas anteriormente fueron:

Cantidad de hilos: 100, 1000 y 10000.

Cantidad de operaciones por hilo: 10, 100 y 1000.

Proporción de hilos que ejecutan operaciones de adición: 0%, 25%, 50%, 75%, 100%.

Tipo de lista usada: lista de sincronización de granularidad fina, lista de sincronización optimista, lista de sincronización no bloqueante.

Como para cada tipo de lista se probaron dos implementaciones diferentes, esta elección da como resultado 270 escenarios distintos. Además, cada escenario fue ejecutado 20 veces y se calculó su tiempo de ejecución promedio, lo que significa que se evaluaron un total de 5400 valores medidos.

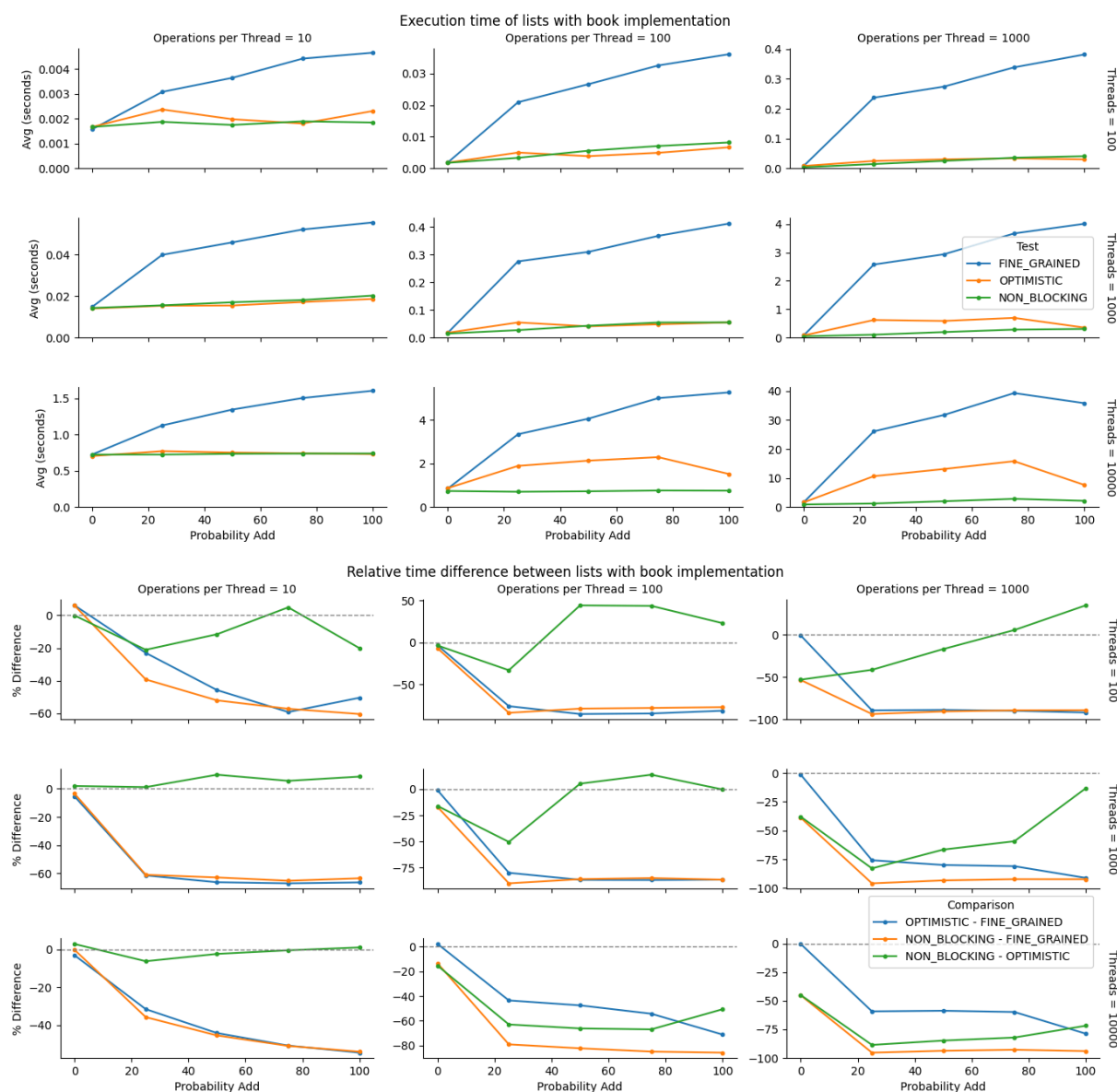
Resultados

El programa en Java ejecuta los tests, obtiene el tiempo de ejecución promedio de cada escenario y genera dos archivos de salida en formato `.csv`, uno con los resultados de las listas implementadas idénticamente a las del libro y otro con los de las listas modificadas. Estos archivos luego son interpretados por un programa hecho en Python que toma los datos y devuelve una matriz con los gráficos de tiempos y otra con la diferencia porcentual entre las listas.

Los siguientes resultados fueron obtenidos usando una PC con un procesador Intel Core i5-12600K, 32 gb de memoria RAM DDR4 a 3200 Mhz y Windows 11 Pro.

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 6 de 12
		TPN 1	Comisión A
			TPN2

Implementaciones de libro



La primera conclusión que puede obtenerse es que la lista con sincronización de granularidad fina tiene un rendimiento notablemente inferior a las otras, llegando esta a tardar hasta 20 veces más tiempo en ejecutar el mismo escenario. Claramente esto se debe a la elevada cantidad de veces que esta lista hace uso de las primitivas `lock()` y `unlock()`.

Las diferencias entre las listas con sincronización optimista y no bloqueante comienzan a notarse recién en los escenarios que realizan más de 10000 operaciones, siendo esta última la de mejor rendimiento en prácticamente todos los casos, algo esperable sabiendo que esta evita los bloqueos por completo.


	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 7 de 12
		TPN 1	Comisión A
			TPN2

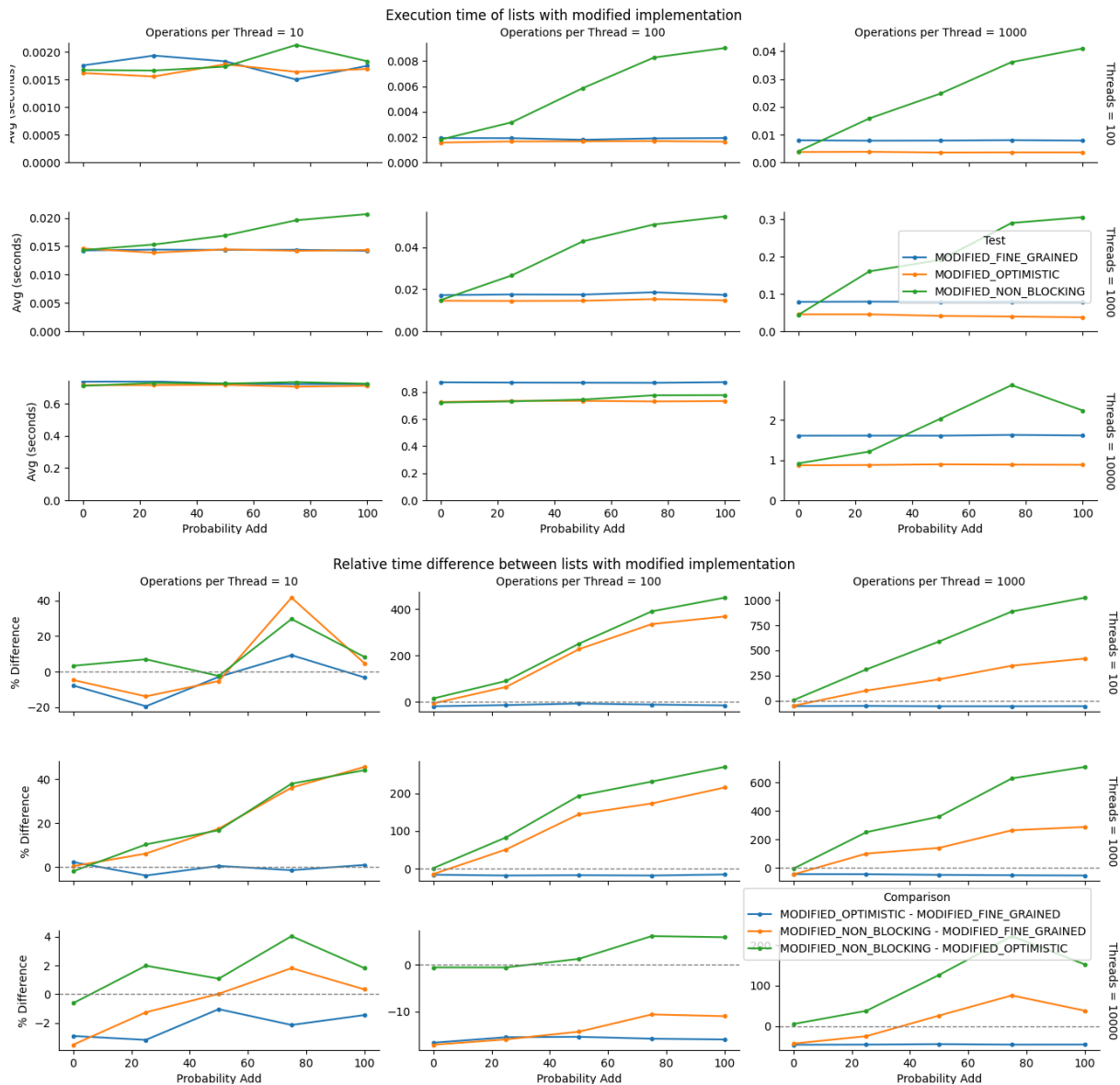
Respecto a cómo varían los tiempos de ejecución según las características de cada escenario se observa que:

- El aumento en la proporción de hilos que realizan operaciones de adición mostró una perceptible correlación con el aumento del tiempo de ejecución.
- El aumento en la cantidad de hilos manteniendo constante la cantidad de operaciones realizadas por cada hilo supuso un esperable aumento en el tiempo de ejecución, como puede observarse comparando los gráficos pertenecientes a una misma columna.
- La variación en la cantidad de hilos manteniendo constante la cantidad total de operaciones mostró un comportamiento interesante. Se puede ver que los subgráficos presentes en cada diagonal secundaria de la matriz muestran escenarios con la misma cantidad de operaciones totales. Dentro de cada una de estas diagonales se observa que el aumento en la cantidad hilos supuso un aumento en el tiempo de ejecución, que fue mayor para las listas con sincronización optimista y no bloqueante. Una hipótesis sobre el porqué de este fenómeno es que la lista de sincronización de granularidad fina ya incurre en una alta cantidad de bloqueos en los escenarios de menor cantidad de hilos, lo que haría que el aumento de estos no le sea tan notable como sí lo es para los otros tipos de listas. Sin embargo, como excepción a este fenómeno aparecen los escenarios de 100 hilos que ejecutan 1000 operaciones y de 1000 hilos que ejecutan 100 operaciones, entre los que no se percibe diferencia alguna.


Implementaciones modificadas

A continuación se mostrarán los resultados obtenidos para los mismos escenarios pero, esta vez, comparando las implementaciones modificadas, las cuales alteran pequeños pero sensibles detalles de las provistas por el libro.

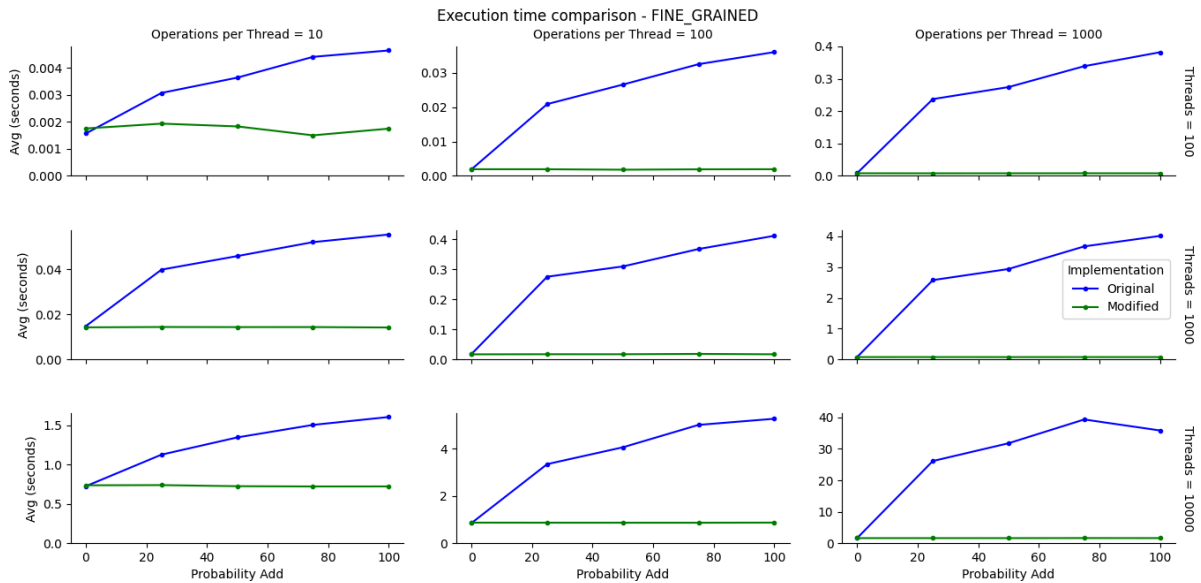
	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 8 de 12
		TPN 1	Comisión A
			TPN2



Como puede observarse, los resultados son totalmente diferentes a los mostrados anteriormente. De repente la lista con sincronización no bloqueante pasa a ser la peor de las tres por amplio margen, mientras que la de granularidad fina se acerca mucho a la optimista, que aparece como la mejor de todas. Para comprender mejor la situación veamos unos gráficos que comparan las dos implementaciones para cada tipo de sincronización.

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 9 de 12
		TPN 1	Comisión A
			TPN2

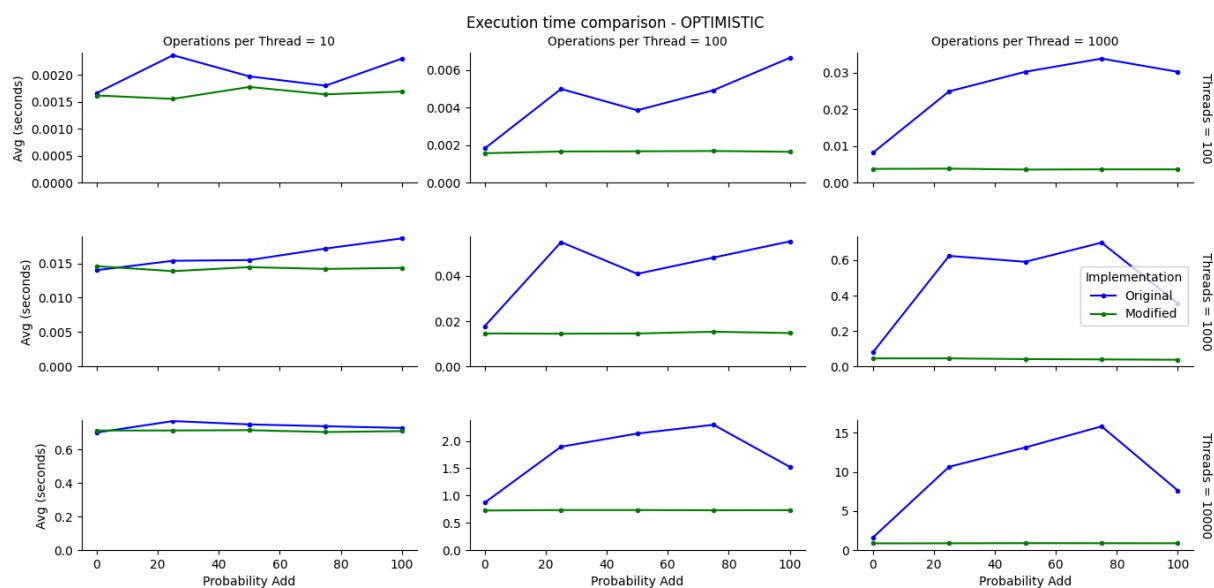
Lista con sincronización de granularidad fina



	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 10 de 12
		TPN 1	Comisión A
			TPN2

Para la lista con sincronización de granularidad fina el cambio en el rendimiento es superlativo, de hasta más de 40 veces en algunos casos. La diferencia entre estas implementaciones radica simplemente en dos sentencias if. La primera simplemente se asegura de que el argumento pasado a los métodos no sea null (este cambio no afecta al rendimiento, pero igualmente merece ser mencionado). La segunda, causante del significativo aumento en el rendimiento, verifica que el nodo que actualmente se está analizando (la variable curr) no sea la cola de la lista. Este simple cambio es responsable por la gran mejora en el rendimiento. No se ha llegado a una convincente explicación de por qué esto es así. La más creíble sería pensar que la temprana liberación de los locks del nodo actual y de su predecesor ayuda a reducir el tiempo que los procesos están bloqueados esperando para hacer sus operaciones, pero resulta improbable que tan pequeña diferencia pueda ser tan significativa.

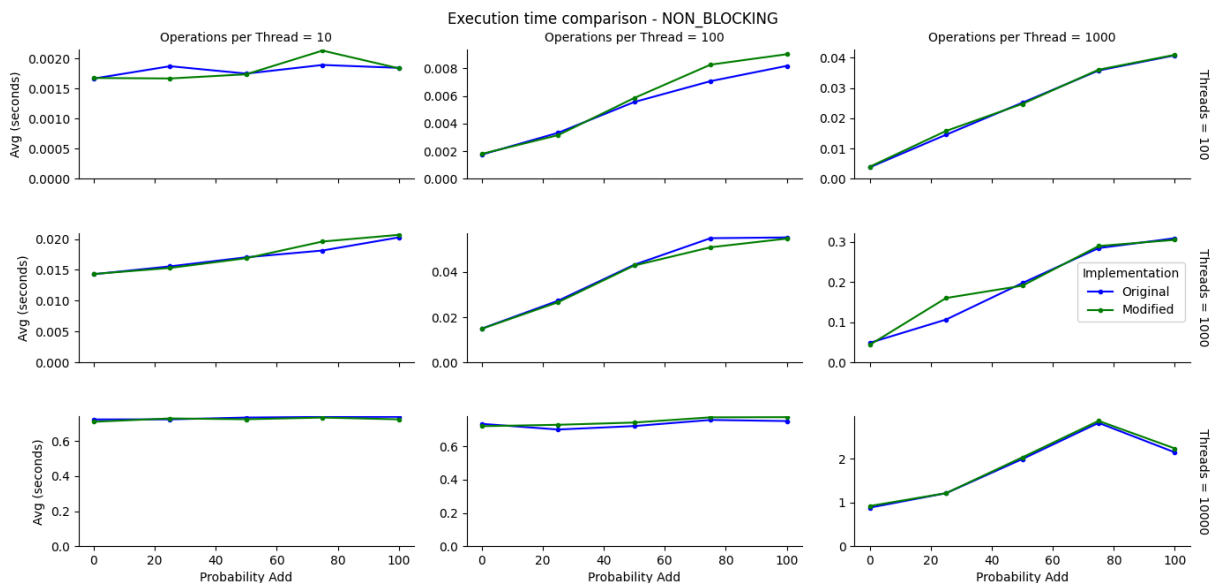
Lista con sincronización optimista



Para la lista con sincronización optimista se añadieron las mismas dos sentencias ifs mencionadas anteriormente. Similarmente, el cambio observado en el rendimiento es sorpresivo, aunque en este caso no fue tan significativo como con la lista con sincronización de granularidad fina.

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 11 de 12
		TPN 1	Comisión A
			TPN2

Lista de sincronización no bloqueante



Finalmente, para la lista con sincronización no bloqueante no se encontraron muchas modificaciones para hacer, más allá de reemplazar la clase Window por un arreglo de LocklessNode<T>. Tal cambio prácticamente no impactó en el rendimiento.

Cabe destacar que, mientras se investigaban las soluciones mostradas, surgió la idea de reemplazar la instancia de ReentrantLock de Node<T> por una de Semaphore. Fue total el asombro cuando se vió que el tiempo de ejecución de los escenarios se multiplicaba hasta por seis. Se desconocen los detalles de las implementaciones de ambas clases pero igualmente sorprende ver tales diferencias en rendimiento entre dos clases que ofrecen primitivas tan populares para la resolución de problemas concurrentes.

Los resultados arrojados por estas implementaciones levantaron sospechas en el grupo de trabajo, por lo que se decidió probar con las implementaciones presentadas en el libro para eliminar toda sospecha de haber introducido algún error en las mismas.

Otro problema encontrado durante la realización de las pruebas fue que las primeras ejecuciones tenían un mayor rendimiento que las últimas. Por esta razón se agregó un sleep() de 50 milisegundos entre ejecuciones, para darle suficiente tiempo al Garbage Collector de Java de "limpiar" la máquina virtual, pero aún así el fenómeno permaneció presente. Por ello se decidió crear un bucle donde cada escenario es ejecutado una única vez y repetirlo veinte veces, de manera que ningún escenario pudiera verse beneficiado por sobre los demás.

	Universidad Tecnológica Nacional Facultad Regional Santa Fe	Programación Concurrente	Página 12 de 12
		TPN 1	Comisión A
			TPN2

Conclusiones

Las listas enlazadas son una de las estructuras de datos fundamentales usadas por infinidad de programas de toda índole, por lo que es necesario encontrar una forma de lograr que programas multihilo puedan hacer uso de estas. En este trabajo se analizaron listas enlazadas adaptadas a entornos concurrentes de tres tipos de sincronización diferentes: de granularidad fina, optimista y no bloqueante. Los resultados arrojaron que las listas de sincronización no bloqueante salen victoriosas en esta comparación, mientras que las de sincronización optimista ocupan un cómodo segundo lugar no tan lejos de las primeras. Resulta difícil recomendar el uso de listas con sincronización de granularidad fina por lo inferiores que resultan ser en comparación con las otras. Sin embargo, resta seguir examinando y evaluando las distintas implementaciones existentes de estos algoritmos con la intención de optimizarlos y arrojar luz sobre los inconvenientes expuestos, puesto que pueden modificar críticamente nuestro entendimiento del tema.