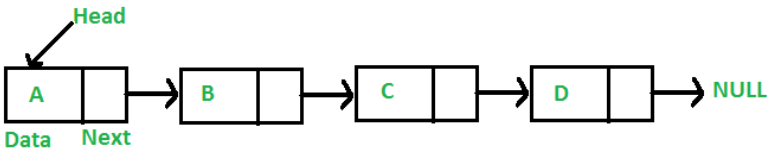
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

LinkedList



Daniel Murteira nº 46833
Ricardo Fernandes nº 42548

Abstrato

Este trabalho pretende comparar diferentes execuções concorrentes de um programa que implementa uma lista ligada e que contém três operações distintas: read (contains), write (adiciona um nodo) e remove (remove um nodo). Para tal existem várias implementações da mesma lista com diferentes aproximações no que se refere ao sistema de locking de cada uma.

Introdução

O objetivo deste trabalho passa por executar uma lista ligada de forma concorrente com a melhor performance possível. Para tal foram implementadas várias versões de listas ligadas (linked lists) com diferentes alternativas de locking (incluindo uma versão sem locks). Cada lista contém três operações distintas: read (contains), write (insere um novo nodo na lista) e remove (remove um nodo da lista). Cada uma destas versões foi testada várias vezes através de um script que recebe como parâmetros o tempo de execução, o número de threads a ser utilizado no teste e a percentagem de escritas. Durante o teste são executadas o máximo de operações possíveis (das três já acima mencionadas). No final a lista é validada e obtemos os resultados em termos do número de operações realizadas para cada uma das threads, o total de operações realizadas e o número de operações por segundo, o que nos irá permitir comparar a eficiência das várias implementações.

Aproximação

Este programa baseia-se na implementação de uma lista ligada em que os nodos têm um valor inteiro (que é único na lista) e uma referência para o próximo nodo. A lista encontra-se sempre ordenada e não é possível que existam dois números iguais na mesma. A lista faz uso de nodos sentinela, um para o início com o menor inteiro que o java aceita e um para o final com o maior inteiro que o java aceita.
Partindo daqui, foram testadas várias implementações concorrentes desta mesma lista:

- Em primeiro foi implementada da lista em que todos os seus métodos se encontram sincronizados. Para que tal aconteça basta acrescentar “synchronized” ao cabeçalho dos métodos usados pela lista. Isto garante que apenas uma thread de cada vez consegue executar métodos que se encontrem implementados na class da lista.

- Em segundo foi implementada uma versão com um lock global. Este é um lock do tipo ReentrantLock que é uma variável global da class onde se encontra implementada a lista. Para qualquer um dos métodos (read, write, remove) é necessário dar lock a toda a lista (todos os nós). Em qualquer um dos métodos, damos unlock à lista após acabarmos a operação.

- Em terceiro foi implementada uma segunda versão do lock global mas com um lock do tipo ReentrantReadWriteLock. Esta variável possui um par de locks, um para Reads (leituras) e outro para Writes (escritas). Assim sendo, os métodos de escrita (write e remove) adquirem locks do tipo write sobre a lista e os métodos de leitura (contains) adquirem locks do tipo read. Os locks do tipo

write são exclusivos, ou seja, apenas uma thread de cada vez consegue obter este lock. Os locks do tipo read podem ser partilhados entre várias threads, tornando assim possível fazer reads concorrentes sobre a lista, desde que esta não tenha um lock do tipo write ativo (que tem prioridade em relação aos de read).

- Em quarto foi implementada uma versão da lista com locks sobre os nodos em vez de sobre a lista. Assim sendo foi adicionada uma variável aos nodos do tipo ReentrantLock. Nesta versão, nos métodos de adicionar e remover começa-se por fazer lock aos dois primeiros nodos da lista e vamos depois avançando na lista nodo a nodo, removendo o lock ao nodo com menor valor e adicionando um lock ao nodo a que agora chegámos. Desta forma percorremos a lista até encontrar a posição que pretendemos, tendo sempre dois nodos locked. No método de leitura (contains) usamos a mesma estratégia.

- Em quinto foi implementada uma versão otimista da lista, novamente com locks sobre os nodos (ReentrantLock). Desta vez, nos métodos de adicionar e remover não são feitos locks enquanto percorremos a lista à procura do valor que pretendemos. Em vez disso, fazemos lock apenas quando encontramos o valor diretamente abaixo daquele que andamos à procura, dando lock a este nodo e ao próximo. Após termos feito lock aos nodos, é necessário ver se estes não foram alterados (visto que não tínhamos qualquer tipo de lock na pesquisa). Assim sendo verifica-se se os nodos ainda se encontram na lista e se não foram alterados. Em caso negativo, volta-se a executar a pesquisa até conseguir passar na verificação. Em caso afirmativo procede-se à escrita nos nodos. No final retiram-se os locks aos nodos. O contains funciona de igual forma.

- Em sexto foi implementada uma versão "lazy" da lista, ainda com locks (ReentrantLock) ao nível dos nodos. Os nodos têm agora uma marca (boolean) que indica se já foram removidos ou não (lógicamente). As remoções estão agora divididas em duas fases distintas: uma fase lógica em que se marca o nodo e uma fase física em que se remove o nodo da lista. Tal como na versão otimista, as pesquisas não usam locks, tanto no add, no remove ou contains. Depois de se executar uma pesquisa é necessário uma verificação em que vemos se nenhum dos nodos (par de nodos) que estão locked estão marcados (removidos lógicamente). Se estiverem, voltamos a tentar executar a ação em questão, começando novamente pela pesquisa. O contains, no entanto, funciona de forma diferente, executando a pesquisa apenas uma vez e retornando sempre, não usando qualquer tipo de lock. Procuramos até encontrar o valor logo abaixo do pretendido e vemos se o próximo não se encontra marcado (logicamente removido) e se o seu valor corresponde ao que procuramos.

- Em sétimo foi implementada uma versão sem qualquer tipo de locks sobre a lista. Os nodos têm agora uma AtomicMarkableReference com o valor da marca (boolean) e a referência para o próximo nodo, para além do seu valor. Criou-se uma nova class chamada Window que é chamada através do método privado find. Este método recebe o primeiro nodo da lista (no nosso caso o nodo sentinela com o menor valor) e o nodo que pretendemos encontrar. Este método percorre a lista até encontrar um nodo com valor maior ou igual ao pretendido (ou chegar ao final da lista) e devolve uma Window que tem como variáveis dois nodos (o que queríamos encontrar e o anterior). No entanto, se o find encontrar um nodo logicamente removido (com a marca a true) este tenta remove-lo fisicamente através do compareAndSet aplicado ao nodo anterior. Se conseguir remover então continua o ciclo. Caso contrário começa o ciclo do início até o nodo ser removido ou alguém o remover. Desta forma, o add e o remove ficam bastante mais simples, utilizando o find para encontrar o pretendido. No add o ciclo corre até que se consiga adicionar o nodo (executando uma nova pesquisa a cada iteração) ou até encontrar um nodo com o mesmo valor. No remove a lógica a mesma mas apenas até remover logicamente o nodo pretendido ou a pesquisa não encontrar o nodo a

remover. Se o método conseguir remover logicamente o nodo (marcá-lo) tenta depois removê-lo fisicamente, embora mesmo quando isso n sucede o método acabe sendo a sua remoção posteriormente feita por uma das threads que chame o find e encontre o nodo.

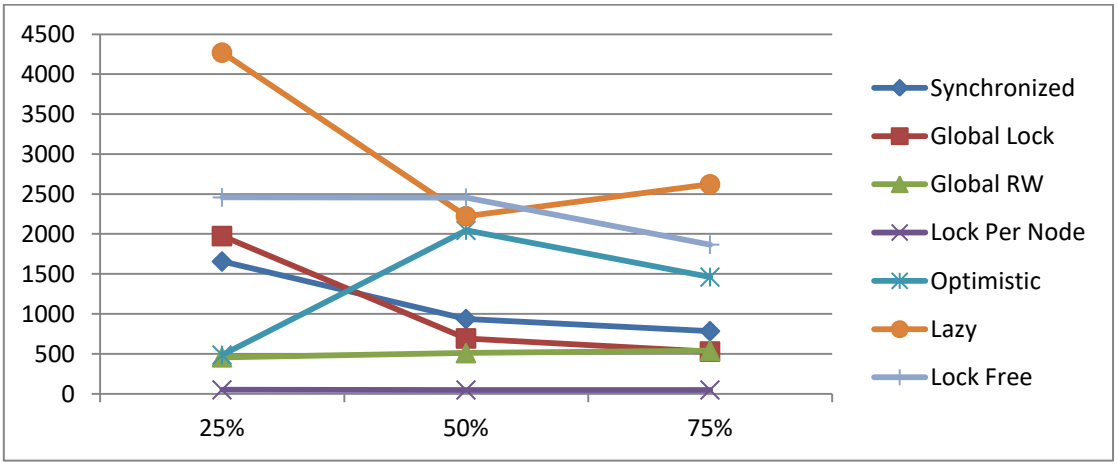
Quando todas as threads acabam de correr é executada a validação final da lista. Esta deve estar ordenada, sem duplicados e o número de elementos presentes na lista deve ser igual ao número de elementos inicial da lista a somar ao número de nodos adicionados, subtraindo o número de nodos removidos.

Validação

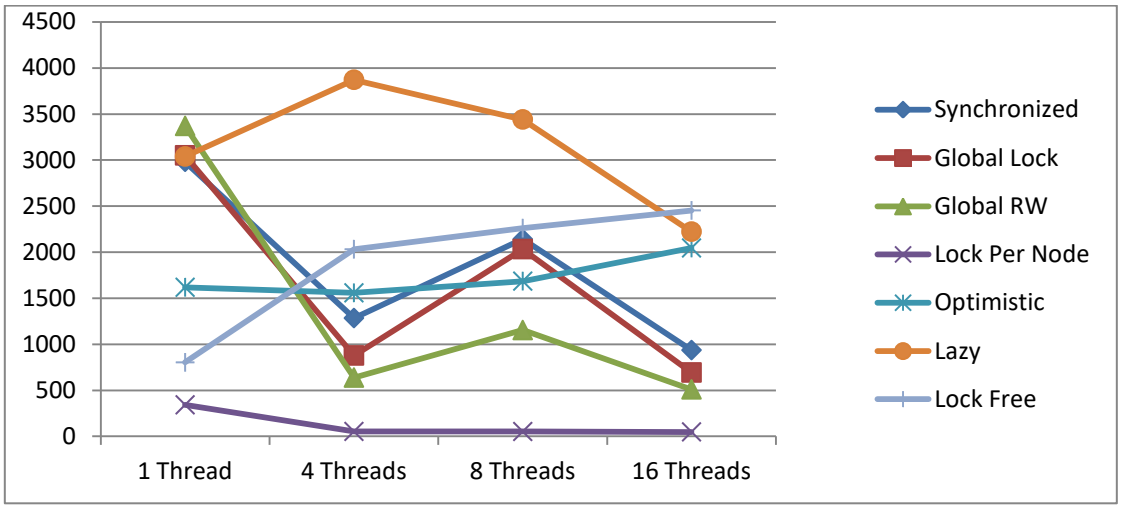
Depois do programa correr é sempre feita a validação da lista através de asserções. As asserções definidas verificam que a lista não contém valores repetidos, está ordenada e que o tamanho final da lista é igual ao tamanho inicial da mesma a somar ao número de nodos adicionados e a subtrair os nodos removidos. Cada uma das implementações da lista foi executada várias vezes, variando o tempo, o número de threads usadas e a percentagem de escritas de forma a ter a certeza que a implementação passa pelas validações sobre qualquer tipo de condição.

Avaliação

Desempenho com 16 threads e várias percentagens de escritas



Desempenho com diferente número de threads e 50% escritas



Os gráficos acima apresentados são o resultado dos testes efetuados numa máquina com 16 cores. O primeiro representa o número de operações por segundo que uma dada implementação conseguiu atingir durante a sua execução com uma certa percentagem de escritas, sempre com 16 threads. O segundo apresenta o número de operações por segundo das várias implementações quando executadas para 50% de escritas e um número variado de threads. Cada um dos resultados apresentados é a média de um conjunto de testes executado para cada uma das implementações. De notar que os resultados podem estar adulterados em relação à realidade pois os testes não foram executados de forma isolada (haviam mais testes a correr na mesma máquina).

O lock ao nível dos nodos (Lock Per Node no gráficos) teve o pior desempenho de entre as implementações apresentadas em qualquer um dos casos. Isto deve-se ao facto de ser necessário atravessar a lista dando lock a dois nodos de cada vez para qualquer que seja a operação, ou seja, se tivermos uma operação a ser realizada nos dois primeiros nodos, nenhuma das outras threads conseguirá executar qualquer pesquisa sobre a lista, pois não conseguirão obter os locks sobre os primeiros elementos. Como se vê, o impacto na performance é grande e as consequências podem ser observadas em ambos os gráficos. Para execuções concorrentes (com mais de 1 thread) as duas melhores implementações são a Lock Free (sem locks) e a Lazy. A Lock Free, não tendo qualquer tipo de lock, beneficia claramente do uso de mais do que uma thread, crescendo com o aumento do número de threads usadas (embora exista sempre um limite em relação ao crescimento máximo de performance em relação ao número de threads usadas). Apesar disso permanece abaixo da Lazy. Isto deve-se ao facto da Lock Free também executar removes no seu find (que é usado no remove e no add), ou seja, sempre que o find encontrar um nodo logicamente removido, este vai tentar removê-lo e não irá avançar até que este tenha sido removido ou até que alguém o tenha conseguido remover. Isto introduz um grande custo nesta operação quando comparado ao Lazy.

No caso da versão Lazy, estes métodos têm apenas de esperar até obter locks, depois disso a operação termina sempre, já no caso do Lock Free, mesmo quando acedemos aos nodos, pode ser necessário repetir a operação pois não existe garantia que vamos conseguir realizar a escrita.

Na versão otimista nota-se claramente o custo de uma operação falhar e ter de voltar a executar a pesquisa e a validação até conseguir por fim executar a ação pretendida. Este processo acaba por ter um grande impacto no desempenho final desta implementação mantendo-se por isso sempre abaixo da versão Lazy e Lock Free. No entanto mantém-se sempre acima das versões Global Lock pois esta utiliza locks globais e as suas escritas e leituras implicam que mais ninguém consiga aceder à lista. O mesmo acontece para a versão sincronizada, visto que apenas uma thread de cada vez consegue aceder a métodos da lista. O mesmo se repete para a versão GlobalRW pois as escritas são exclusivas e só a thread que se encontra a escrever é que tem acesso à lista enquanto estiver a executar um add ou remove. Para além disso o

ReentrantReadWriteLock não se encontra otimizado para realizar pequenas operações, o que é o caso visto que as operações sobre a lista são rapidamente executadas, o que faz com que o custo de usar este tipo de lock se sobreponha aos benefícios que este traz.

Conclusões

A aproximação a um problema deste género deve ter sempre em conta as condições oferecidas, em termos do número de threads disponíveis e da percentagem de escritas (em média) que vamos ter. Com poucas threads disponíveis não vale a pena recorrer por exemplo ao Lock Free devido à sua grande complexidade, já que este não retira grande benefício ao ser executado com poucas threads. Numa versão com apenas uma thread disponível torna-se desnecessário o uso de qualquer tipo de locks pois a execução será sempre sequencial e os locks apenas irão introduzir uma pior performance (não havendo sequer necessidade de garantir exclusão mútua). Para programas em que as threads executem um maior número de leituras que de escritas e/ou em que as operações realizadas pelas threads sejam complexas o suficiente de modo a atenuar o custo, devem também ser substituídos os ReentrantLocks por ReentrantReadWriteLocks de modo a obter uma melhor performance (visto que os reads são partilhados) nos casos em que se usam locks nas leituras.
Numa versão com várias threads disponíveis é preferível o uso da implementação Lazy pois é bastante menos complexa que a Lock Free e a sua performance é, no geral, até superior à do próprio Lock Free.

Reconhecimentos

André Catela, nº 42643 pela discussão de resultados obtidos e apuramento de conclusões ao longo do desenvolvimento do trabalho.

Bibliografia

As implementações foram baseadas no livro:
Maurice Herlihy – The Art of Multiprocessor Programming. Elsevier

Para a implementação do GlobalRW:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

Para a implementação do Lock Free:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicMarkableReference.html

Sobre ReadWrite Locks:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html