Relatório do 2º Projeto da Disciplina de Concorrência e Paralelismo

Gonçalo Banha (45429) & Miguel Almeida (45526)

27 de Novembro de 2017

Resumo

O problema inerente ao trabalho prático da cadeira de *Concorrência e Paralelismo* consiste na aprendizagem e respetiva implementação de como melhorar um programa funcional, que usa uma *Linked List* que contêm um conjunto de inteiros, na linguagem Java. A solução passa por implementar diversos tipos de técnicas de sincronização de forma possibilitar a concorrência do programa. É também objetivo do trabalho avaliar a qualidade da solução implementada e o impacto da mesma no funcionamento do programa. Como resultado da nossa implementação conseguimos perceber o efeito dos diferentes tipos de *Lock*.

1 Introdução

Este relatório foi elaborado pelo grupo nº16 para o segundo projeto prático da unidade curricular de *Concorrência e Paralelismo.*

Neste projeto foi-nos pedido para implementar diferentes técnicas de sincronização de modo a permitir a execução concorrente de operações de *add, remove e contains* de uma *Linked List.* Um dos objetivos deste trabalho recai em perceber de que maneira diferentes técnicas de sincronização não só influenciam o número de operações por segundo como também a consistência e coerência dos resultados obtidos, analisando o funcionamento do programa. Foram implementadas diversas técnicas de sincronização nomeadamente *Coarse-Grained Synchronization, Coarse-Grained Read-Write Synchronization, Fine-Grained Synchronization, Optimistic Synchronization, Lazy Synchronization, Non-Blocking Synchronization* e uma sincronização nativa do Java.

2 Abordagem

No início deste projeto, foi fornecido um programa funcional, já paralelizado, que permite adicionar (*add*) e remover (*remove*) inteiros de uma *Linked List.* Quando o programa era executado, usando mais do que uma *thread*, o programa rebentava pois, ao mesmo tempo, eram feitas várias operações aos mesmos registos. Se houver duas operações distintas, como o *add* e o *remove*, feitas ao mesmo tempo, a operação de adicionar um elemento à lista pode ser feita no apontador (*next Node*) de um elemento que está para ser removido, acabando por não conseguir adicionar o primeiro à lista. Assim, de modo a combater os problemas de múltiplos acessos por parte de diferentes *threads* implementámos algumas técnicas de sincronização. Como já

referido anteriormente, existem diversas maneiras de sincronizar um programa, servindo este trabalho também para perceber qual a melhor técnica de implementação.

2.1 Técnicas de Sincronização

- ***Coarse-Grained Synchronization***

  Este tipo de sincronização é o mais básico de todas as sincronizações. O mesmo bloqueia o acesso à totalidade da lista sendo impossível duas *threads*, ao mesmo tempo, acederem à mesma lista. Esta sincronização funciona bem quando existe um número relativamente pequeno de *threads*, que tentam aceder ao mesmo objeto, uma vez que existe um único *Lock* partilhado pelas diferentes *threads*. Caso contrário, haverão muitas *threads* em fila de espera para aceder objeto, provocando um engarrafamento (*bottleneck*).

- ***Coarse-Grained Read-Write Synchronization***

  Esta sincronização não difere muito da primeira implementação utilizando *Coarse-Grained Synchronization*, contudo, esta possui dois tipos de *locks*, um *lock* para operações de leitura e outro *lock* para operações de escrita, deste modo, sempre que uma *thread* desejar escrever na lista, esta fica com o *lock* de escrita e não existe mais nenhuma outra *thread* que consiga escrever na lista enquanto esta não libertar o *lock*, contudo, isto possibilita que exista outra *thread* a ler a lista naquele exato momento, isto trás a vantagem de diminuir o efeito de *bottleneck* da técnica de *Coarse-Grained Synchronization*, contudo ainda deixa muito a desejar no que toca a eficiência.

- ***Fine-Grained Synchronization***

  Esta sincronização, ao contrário da *Coarse-Grained Synchronization*, que bloqueia o objeto inteiro, divide-o em componentes que podem ser sincronizadas individualmente, garantido que duas *threads* não possam aceder à mesma componente (objeto). Neste caso, em vez de bloquear toda a lista e todos os acessos à mesma, são bloqueados nós individuais. São bloqueados dois nós iniciais, e à medida que a lista é percorrida, à procura do nó pretendido, os dois *locks* acompanham esse percurso, bloqueando sempre dois nós simultaneamente. Esta condição de bloquear dois nós, garante que, por exemplo, não haja um nó inacessível após uma remoção de dois nós consecutivos. Bloqueando apenas dois nós por *thread*, permite que várias *threads* percorram a lista ao mesmo tempo, sem entrarem em conflito de acesso nos objetos.

- ***Optimistic Synchronization***

  O *Optimistic Synchronization*, para baixar os custos de sincronização, realiza uma pesquisa sobre a lista, sem usar os *locks*. Quando encontra o nó que precisa, bloqueia o nó atual e o seguinte e confirma que o nós bloqueados estão corretos. Se ocorrer uma operação de outra *thread* que origine um erro nos nós bloqueados, estes são libertados e a sincronização começa novamente. Este conflito é pouco usual o que faz com que a sincronização ocorra quase sempre à primeira. Este tipo de implementação faz sentido

se o custo de percorrer a lista duas vezes sem *locks* é significativamente menor do que percorrer a lista apenas uma vez com *locks*.

- ***Lazy Synchronization***

  Esta técnica de sincronização tem como objetivo tornar o método *Contains() wait-free* e fazer com que os métodos *add()* e *remove()* percorram a lista apenas uma vez. Para isso, é adicionado um campo *Boolean* a cada nó para indicar se o nó está para ser apagado ou não. Ao contrário da *Optimistic Synchronization*, não é necessário bloquear o nó pretendido e não é preciso valida-lo atravessando toda a lista novamente. Em vez disso, o algoritmo assume que todos os nós com a variável *Boolean* a *false* são acessíveis. Se ao percorrer a lista, o nó não for encontrado ou for encontrado com o variável a *true*, então esse nó já não pertence à lista. Com esta técnica, o *add()* percorre a lista até encontrar o antecessor do novo nó, bloqueando-o e adicionando o novo nó. O método *remove()* é separado em duas partes: A primeira em que, após encontrar o nó, altera a sua variável deixando-o logicamente inacessível, e a segunda parte, em que o remove fisicamente.

- ***Non-Blocking Synchronization***

  Esta técnica de sincronização utiliza um campo diferente para indicar o proximo elemento da lista, utiliza algo chamado *Atomic Markable Reference*, que consiste em juntar a referencia do apontador do próximo elemento e um *boolean* que indica se o nó está para remoção ou não, isto acontece de modo a que possa assegurar que um campo do nó não seja alterado depois deste ser removido logicamente, assim, com a utilização de *Atomic Markable References* conseguimos garantir isso. O facto deste tipo de sincronização não utilizar *locks* torna-a um pouco diferente das restantes, no que toca ao método *contains* este não é muito diferente do mesmo método utilizado na *Lazy Synchronization*, contudo, no caso do Add vamos supor que uma *thread* deseja adicionar um elemento à lista, esta percorre a lista, para localizar o sitio onde inserir, se o valor a inserir já estiver na lista então retorna falso, caso contrario adiciona o elemento, atualizando a sua referencia para o nó a seguir, como o método chama ainda o *compareAndSet()* e este verifica a referencia e o *boolean* a adição só é bem sucedida se o elemento atrás do adicionado não estiver marcado para sair e a sua referencia for o nó à frente, caso contrario, o método começa novamente. No caso do *remove* o método localiza o elemento a remover e tenta marcar o elemento como logicamente removido, isto só funciona se não tiver ocorrida outra *thread* marcar o elemento como removido antes, se isto correr bem então o método retorna *true* e é feita uma única tentativa de remoção física do nó. Não existe necessidade de tentar repetir a tentativa pois o nó será removido pela próxima *thread* que atravessar aquele segmento da lista, lendo que o nó está marcado e então remove o mesmo da lista.

Grande parte da implementação do programa foi sustentada recorrendo a *slides* da disciplina e à bibliografia da mesma, de forma a perceber como cada tipo de sincronização deveria ser implementado. Os problemas que encontrámos ao longo da implementação da solução foram essencialmente na implementação da sincronização sem *locks* (*Non-Blocking Synchronization*). Estas dificuldades ficaram a dever-se ao facto de a maneira de aceder à lista ser um pouco

diferente, devido à falta de *locks*, contudo, e depois de fazer as apropriações necessárias conseguimos implementar este tipo de sincronização segundo aquilo que constava na documentação.

3 Validação

Ao longo do desenvolvimento da nossa solução foram efetuados vários testes, de forma a perceber de que formas estávamos a progredir, teste estes efetuados numa máquina (8 * dual-core) disponibilizada pelo docente da cadeira.

Foram realizados testes de diferentes perfis. Certos testes tinham como objetivo perceber se as alterações que fazíamos ao nosso projeto estavam, de algum modo, a alterar o *output* do programa e outros testes serviam para perceber até que ponto o programa estava a ser otimizado. Para isto, corremos vários testes em que testamos todos os tipos de sincronização explicados no ponto anterior, e para cada um desses tipos de sincronização corremos testes onde alterávamos o número de *threads* utilizadas (1, 2, 4, 8 e 16), e para cada número de *threads* fizemos também variar a percentagens de *writes* executada pelo programa (25%, 50% e 75%). Conseguimos assim, com todos estes valores perceber de que modo o aumento do número de *threads* e o aumento da percentagem de *writes* para cada número de *threads* utilizada fazia variar os resultados dos diferentes tipos de sincronização.

Foi também adicionado ao programa certas validações que permitem concluir que o programa está a executar corretamente. Ao inicializar o programa a lista já vem preenchida com alguns valores, ou seja, não vem vazia. Se não colocássemos invariantes não podíamos garantir que num programa que começasse com a lista vazia estava a funcionar corretamente, foram por isso inseridos invariantes de maneira a confirmar que o numero de *removes* nunca é superior ao numero de *adds*.

4 Avaliação

Do ponto de vista da avaliação de resultados obtidos foi necessário efetuar diversos testes. Os testes de devem-se ao facto de necessitarmos de ter a noção do impacto que a utilização de diferentes números de *threads* e percentagens de *writes* têm no programa.

De modo a ter uma ideia mais clara de como o fator *thread* e percentagem de *writes* influenciam o resultado final, fizemos vários testes para cada tipo de sincronização. De notar que todos os gráficos apresentados com os resultados dos testes estão na escala de 1e7, bem como o facto de que a técnica de *Fine-Grained Sincronization* não tem o valor de zero, mas sim um valor bastante baixo quando comparado com os outros métodos de sincronização.

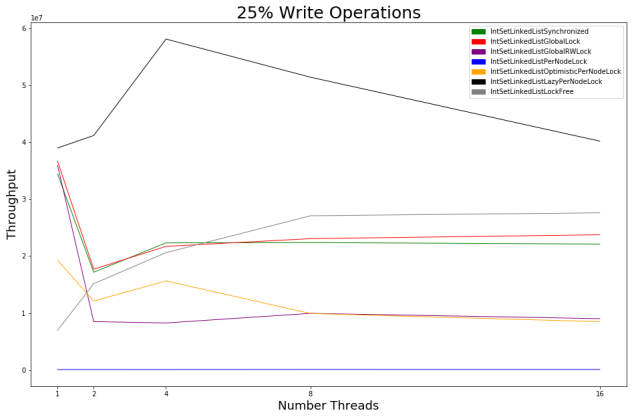

Figura 1: Teste com 25% das operações sendo *writes*

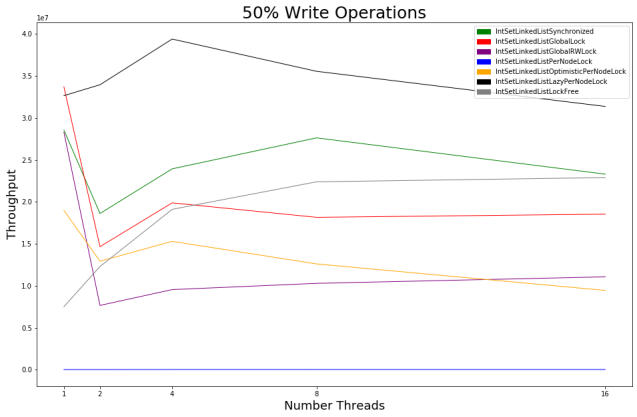

Figura 2: Teste com 50% das operações sendo *writes*

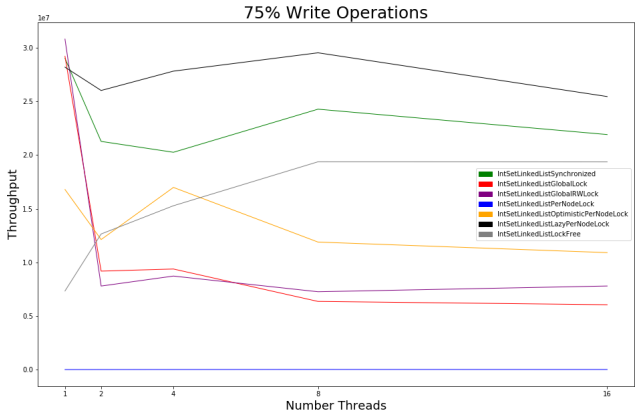

Figura 3: Teste com 75% das operações sendo *writes*

Como é visível nas figuras acima, o número de operações por segundo (*throughput*) com 4 e 8 *threads* (dependendo do tipo de sincronização em analise) ficou um pouco acima do numero de operações com 16. Isto deve-se ao facto da máquina utilizada para correr os testes ser uma máquina partilhada com os outros estudantes da cadeira, por isso, ao correr testes quando existem outros grupos a correr os seus testes, resulta em desvios ligeiros nos resultados. Isto, acrescido ao facto de quando se executam testes com 16 *threads* todos os cores da máquina estão ocupados, não havendo por isso margem para que mais instruções sejam executadas pela máquina, o que faz com que,se existir algum acesso à máquina por parte de outros grupos os resultados dos testes terão resultados flutuantes.

Observando a gráfico da imagem 1 conseguimos ver que existe um *ponto critico* em que o numero de operações por segundo deixa de aumentar e passa a diminuir, sendo esse ponto quando o numero de *threads* é igual a 4. No caso das técnicas de sincronização com *global locks*, isto fica a dever-se ao facto de só existir uma única *thread* a executar o programa de cada vez, ao aumentar o numero de *threads* disponível só vai fazer aumentar a fila de espera, e não

o numero de operações executadas por segundo. No caso da sincronização *lazy* e *optimistic* o valor do *throughput* desce pois como estes métodos percorrem a lista sem adquirir *locks* e só quando chegam ao nó desejado é que verificam se este pode ser alterado, se existir muitas *threads* a executar simultaneamente, a probabilidade de um nó ter sido alterado no entretanto é maior, obrigando assim a que o método não execute a sua operação e comece novamente.

No que toca à linha representativa da técnica de *Fine-Grained Synchronization*, esta tem um valor anormal. Contra aquilo que estávamos à espera, esta tem um desempenho bastante inferior quando comparada com as restantes técnicas de sincronização. O valor de *throughput* desta técnica deveria ser superior ao valor obtido nas técnicas que utilizam *global locks* uma vez que esta permite que mais *threads* executem as suas operações em simultâneo, reduzindo assim a fila de espera. Isto pode ficar a dever-se a um erro na implementação da técnica.

Como é visível nos gráficos, a técnica de sincronização de *Lazy Synchronization* apresenta um maior valor de operações realizadas por segundo. Isto é expectável pois é uma técnica que permite a execução de diferentes *threads* em simultâneo bem como o facto de ser *wait-free* e dos seus métodos *add* e *remove* percorrerem a lista apenas uma vez, tornando a técnica mais rápida.

5 Conclusões

Inicialmente pensávamos que quantas mais *threads* um programa tivesse para utilizar, mais operações por segundo faria. Isto não foi observado e fica a dever-se à maneira como algumas técnicas de sincronização são implementadas, pois estas criam filas de espera, o que impossibilita as diferentes *threads* de executarem o seu serviço. Outro factos que contribui para este facto é, como já referido anteriormente, a máquina de testes ser uma máquina partilhada. Contudo, no caso da *Non-Blocking Syncronization* o aumento do número de *threads* levou a um aumento do número de operações. Verificamos também que os resultados da *Fine-Grained Synchronization* produzem um número de operações bastante inferior aos demais, como já foi explicado anteriormente.

Com a utilização de diferentes técnicas de sincronização passa a ser possível garantir que diferentes *threads* a executar ao mesmo tempo não prejudicam o funcionamento de outras *threads* nem das suas operações. É possível com estas técnicas garantir a coerência e consistência de um programa.

6 Bibliografia

Nir Shavit, The Art of Multiprocessor Programming (2008);