Concorrência e Paralelismo 2016-17 — Projeto
Ana Simão - 42978
Pedro Vieira - 42610

1. Introdução

O projeto da cadeira de Concorrência e Paralelismo consiste numa aplicação que simula um repositório de artigos científicos e que utiliza três *HashMaps*: um que associa o artigo ao seu *id* único (*byArticle*), outro que associa uma lista de artigos a cada autor (*byAuthor*) e outro que associa uma lista de artigos que contêm uma dada *keyword* (*byKeyword*).

O desafio passou por criar uma estratégia de sincronização que garante a consistência e a exclusão mútua, evitando *data races*, no acesso aos dados do sistema durante a utilização de 2 ou mais *threads* e que, para isso, foi implementado um mecanismo de *locking* em que cada entrada de cada mapa é protegido por um *lock*.

Outro aspeto que também foi necessário ter em consideração é facto dos *ids* dos artigos serem usados pelos *HashMaps byAuthor* e *byKeyword* como chaves externas dos *papers* do mapa *byArticle* e, por isso, a realização de inserções e remoções concorrentes poderá resultar numa inconsistência de dados em que, por ex., um dos *papers* da lista de artigos de um dado autor faz referência a um artigo que já foi removido.

2. *Approach*

Como referido no ponto anterior, a solução desenvolvida utiliza *fine-grained locking* através de *locks* ao nível de cada uma das entradas de cada um dos mapas. Este detalhe envolveu alterar a implementação da classe *HashMap* de forma a inicializar um *array* de *locks* do tipo *ReentrantReadWriteLocks* com uma cardinalidade igual ao do próprio mapa. Por sua vez, foram adicionados métodos de *acquire(key)* e *release(key)* ao *HashMap* com o intuito de obter/libertar o *lock* associado à *key* enviada nos argumentos.

Este tipo de *locks* é extremamente eficiente pois, ao realizar *acquire*, é possível explicitar se o estamos a realizar para ler ou escrever: no caso em que múltiplas *threads* tentam obter o mesmo *lock* para leitura, este permitirá que todas elas prossigam e entrem na *critical section* pois não existe *data races* quando apenas são feitas leituras concorrentes; por outro lado, quando uma *thread* realizar *acquire* de um *lock* com o intuito de alterar dados, nenhuma outra poderá aceder à *critical section* enquanto esta não o libertar (garantindo a *mutual exclusion*). Como é expectável que uma utilização real do sistema proposto tenda a ter mais acessos do que escritas, este tipo de *locks* tem um enorme impacto no aumento da performance da solução.

A classe *Repository*, que funciona como um ponto de controlo de obtenção e gestão dos *locks* a usar para os três *HashMaps* em cada operação, tem a seguinte especificação:

- **Leituras (*findArticleByAuthor* e *findArticleByKeyword*):** antes de se realizarem os *gets* individuais ao *byAuthor* e ao *byKeyword* é invocada a operação *acquireReadLock()* para a chave da pesquisa (nome do autor e *keyword*, respetivamente) no *HashMap* correspondente; de seguida a *thread* realiza o *get* propriamente dito; executa a sua lógica com o objeto obtido e, antes de passar ao próximo objeto, como já não necessita mais dele, liberta o *lock* anterior através da operação *releaseReadLock()*. Como foi especificado previamente, é possível que *diversas* threads acedam aos mesmos dados para leitura com o mesmo *lock* de forma concorrente desde que não exista já uma *thread* a escrever ao mesmo tempo; se isso acontecer, a *thread* que está a escrever tem acesso exclusivo ao recurso.

- **Escritas (*insertArticle* e *removeArticle*):** no início de cada um destes métodos é realizado o *acquireWriteLock()* da entrada do artigo que foi passado como argumento e no final do método, antes do *return*, é chamado o correspondente *releaseWriteLock()*. Este comportamento foi implementado de forma impedir que mais do que uma *thread* crie ou a

remova o mesmo artigo, acções que são causadoras do problema de inconsistência de dados referido na introdução. De resto, a implementação é semelhante ao descrito nas leituras só que obtêm-se e libertam-se os locks através de *acquireWriteLock()* e *releaseWriteLock()*, respetivamente, para cada recurso.

Esta solução é também *deadlock-free* pois é garantido que cada *acquire* de um *lock* está emparelhado com um *release* e a própria implementação implica que nunca existirá um *circular-wait*.

3. Validação

Antes de se ter iniciado o processo de implementação da solução propriamente dita, o grupo tentou validá-la conceptualmente através de histórias com diversos *interleavings* a uma escala reduzida. De seguida, prosseguiu-se à criação de invariantes adicionais que são verificadas sempre que o programa inicia o seu processo de validação ou no final da sua execução. Considerando *#articles* o número de artigos distintos existentes no sistema e *#remove* o número de remoções bem sucedidas (foi necessária a alteração da assinatura do método de remoção para retornar um *boolean*):

$$1) \quad \#put \geq \#remove$$

$$2) \quad \forall_{i \in byAuthor}\ i \neq \bot \Rightarrow 1 \leq \#i.value \leq \#articles$$

$$3) \quad \forall_{i \in byKeyword}\ i \neq \bot \Rightarrow 1 \leq \#i.value \leq \#articles$$

$$4) \quad \forall_{i \in byAuthor\ :\ i \neq \bot}\ i.value \subseteq byArticleId.values \wedge (\forall_{art \in i.value}\ art \subseteq byKeyword.get(art.keywords))$$

$$5) \quad \forall_{i \in byKeyword\ :\ i \neq \bot}\ i.value \subseteq byArticleId.values \wedge (\forall_{art \in i.value}\ art \subseteq byAuthor.get(art.authors))$$

Para além disso, o grupo utilizou um pequeno *script* que executou repetidamente a solução com a *flag DO_VALIDATION* a *true* onde colocou em prática as invariantes e verificou-se que as validações de consistência foram sempre bem sucedidas.

4. Avaliação

Para além da solução principal, o grupo decidiu criar uma outra alternativa, doravante referida como "Solução B", que utiliza uma abordagem de *locking* de grão médio na medida em que também faz uso de um *lock* por *bucket* em cada *HashMap* mas, ao realizar uma operação de escrita, a *thread* percorre todos os recursos do artigo (autores e *keywords*) que necessitará para a execução integral do método e realiza *acquire* de todos os *locks* que os protegem. No entanto existem algumas nuances: foi necessário adicionar 2 *locks* globais ao sistema para impedir situações de *dead-lock* devido a *circular-waits*, que são usados antes da *thread* percorrer todos os *locks* a dar *acquire* e quando pretende dar *release* a todos eles no final da operação. As métricas de execução desta alternativa são usadas nos ponto seguintes como forma de comparação à solução principal.

Os testes realizados[1] utilizaram os seguintes valores base (tentou-se que estes valores refletissem uma utilização real do sistema): *nkeys*=1000, *put*=30%, *del*=15%, *get*=55%, *nauthors*=5, *nkeywords*=50, *nfindlist*=10.

4.1 Avaliação com o aumento do número de *threads*

Este teste tem o objetivo de visualizar e acompanhar a evolução da performance (número de operações por segundo) das soluções à medida que o número de *threads* vai aumentando.

Antes da realização desta avaliação a equipa já previa o desfecho obtido: a solução principal é extremamente mais eficiente do que a solução B: na B existem dois enormes *bottlenecks* que ocorrem

[1] Testes realizados num computador com processador Intel Core i5 2,6GHz, Dual Core.

quando uma *thread* executa o *acquire* e o *release*, de uma só vez, dos *locks* associados a todos os recursos de um artigo no início e no fim, respetivamente, dos métodos de *insert* e *remove*. Já a solução principal apenas tem um *bottleneck*, de uma escala bastante inferior, quando o *lock* associado ao *id* do artigo é *acquired*, o que faz com que todas as outras *threads* que pretendam remover ou adicionar um artigo com o mesmo *id* tenham de esperar até que a *thread* atual finalize a execução da operação na sua totalidade.

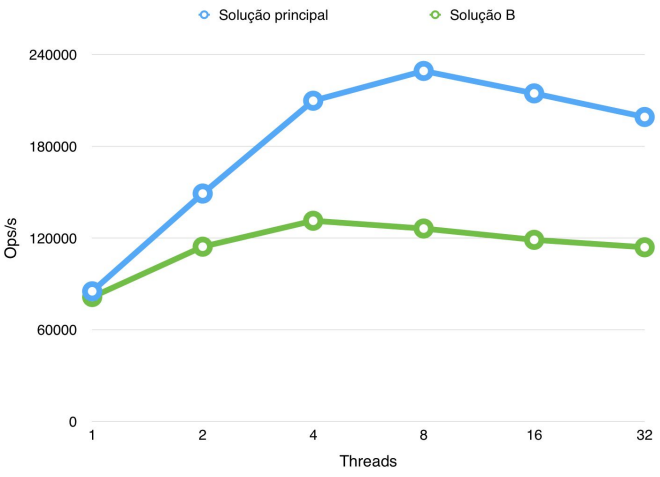

Gráfico 1: avaliação geral das soluções com o aumento do nº de threads

4.2 Avaliação semelhante à anterior com *nfindlist* = 100

A alteração do *nfindlist* de 10 para 100 tem um enorme, mas expectável, impacto no número de operações por segundo: o tempo da pesquisa conjunta aumenta. Isto acontece devido ao facto do número de recursos a obter ter aumentado e, como consequência disso, a *thread* poderá bloquear mais vezes pois a probabilidade de tentar aceder a um *lock* que já está *acquired* para escrita noutra *thread* também aumenta.

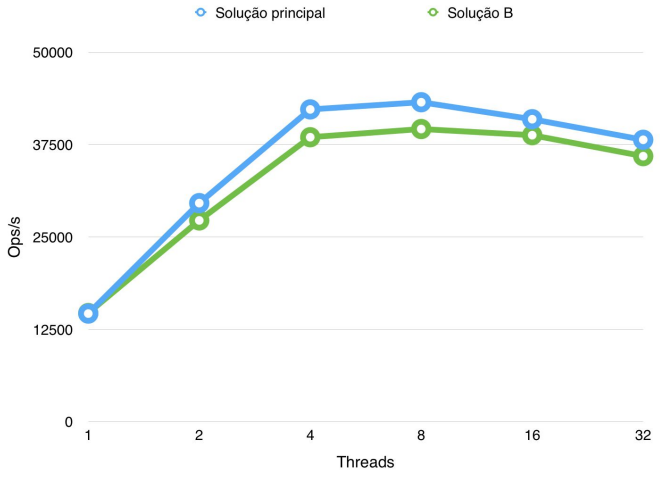

Gráfico 2: avaliação das soluções com o aumento do nº de thread usando nfindlist = 100

4.3 Avaliação com o aumento da percentagem de *gets* (com 8 *threads*)

Nesta avaliação as soluções foram testadas à medida que a percentagem de pesquisas aumentava. Os resultados obtidos mostram claramente as vantagens de usar *ReentrantReadWriteLocks* pois, usando esta implementação de *locks*, a percentagem de pesquisas é inversamente proporcional à probabilidade de cada *thread* bloquear ao tentar aceder a um recurso (devido ao *lock* permitir leituras concorrentes).

Prova	put (%)	del (%)	get (%)
1	30	15	55
2	15	15	70
3	5	5	90

Tabela 1: parâmetros para as diferentes provas da avaliação 4.3

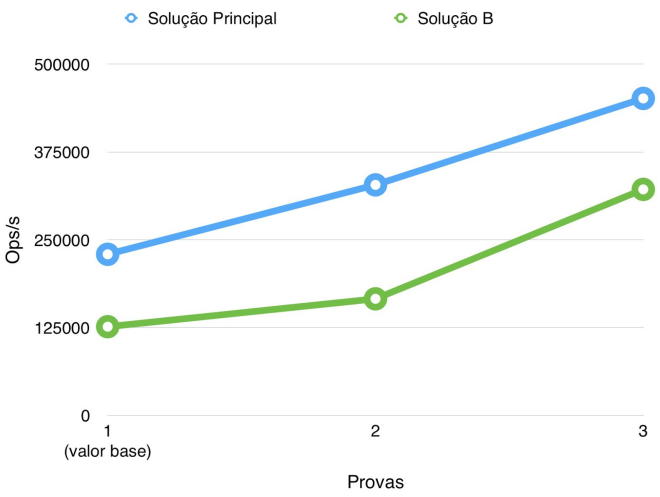

Gráfico 3: avaliação das soluções com o aumento do nº de gets

4.4 Avaliação com o aumento da percentagem de *puts* (com 8 *threads*)

Por fim, nesta avaliação foram testadas as soluções de forma a visualizar a *performance* à medida que o número de pedidos de criação de *papers* aumenta.

Prova	put (%)	del (%)	get (%)
1	30	15	55
2	60	15	25
3	90	5	5

Tabela 2: parâmetros para as diferentes provas da avaliação 4.4

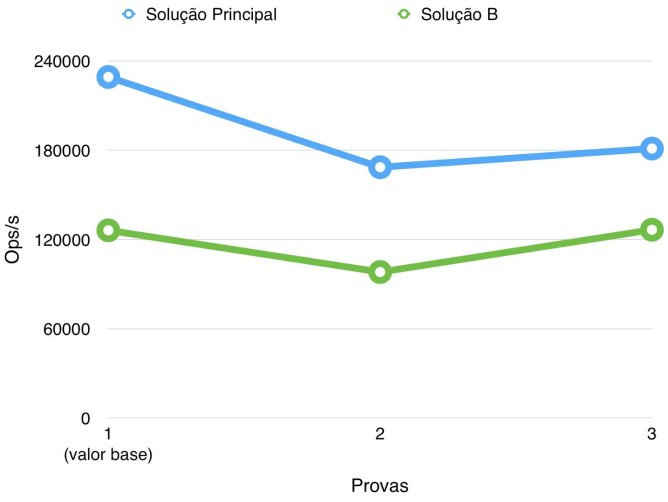

Gráfico 4: avaliação das soluções com o aumento do nº de puts

5. Conclusões

O grupo já tinha uma ideia formulada do que esperar em cada um dos testes devido ao facto da solução ter sido planeada, testada e provada conceptualmente, algo que é extremamente importante na construção de sistemas concorrentes, antes de se ter procedido à sua implementação. Por sua vez, a equipa de desenvolvimento acredita que a implementação apresentada é a solução que apresenta melhores resultados de forma correta com duas ou mais *threads*, como se pode verificar em todos os testes realizados.

Para além da Solução B, foram também testadas outras soluções que usaram *coarse-grained locking*: uma que fez uso da *keyword* "*synchronized*" na assinatura das operações e outra que implementou *locks* globais para controlar o acesso a cada um dos mapas. No entanto, como já era esperado (tendo em conta o *homework* 2), estas tiveram resultados extremamente aquém dos desejáveis. Infelizmente, devido às restrições do tamanho do relatório não nos foi possível aprofundar mais sobre elas.