departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concorrência e Paralelismo – 2013-14

Projecto 1: *Book Repository*

Luis Silva (n 34535º), Ricardo Gaspar (nº 42038)
Departamento de Informática – FCT – UNL
lmt.silva@campus.fct.unl.pt, rf.gaspar@campus.fct.unl.pt

Resumo — Controlar o acesso concorrente a múltiplas tabelas de dispersão garantindo a consistência das operações realizadas. O desafio é conseguir que acessos simultâneos, por diferentes intervenientes, às tabelas ocorram sem que entrem em conflito. Para conseguir atingir este objectivo, a estratégia utilizada passa por cada interveniente solicitar o bloqueio dos recursos pretendidos às várias tabelas antes de realizar operações sobre estes.

iv) Quais os resultados obtidos?

Palavras Chave — Controlo; acessos concorrentes; recursos partilhados.

1. Introdução (Heading 1)

O trabalho consiste em gerir os acessos concorrentes a um repositório de artigos científicos. Este guarda informações acerca dos artigos científicos, dos seus autores e palavras-chave relacionadas em três *hashtables* (ver *hashtable*).

O objectivo é permitir que várias *threads* (ver *thread*) possam realizar um conjunto de operações sobre o repositório sem que este fique num estado inconsistente nem o programa bloqueado. As operações possíveis sobre o repositório são a inserção e remoção de um artigo e a listagem de artigos por autores ou palavras-chave. As duas primeiras operações requerem o acesso a todas as tabelas, enquanto que as restantes só necessitam de consultar uma tabela. Assim, para evitar que diferentes *threads* não acedem simultaneamente aos mesmos dados nas tabelas é necessário garantir tenham um acesso exclusivo sobre estes.

A forma utilizada para evitar os conflitos de acessos baseia-se na disponibilização, pela parte das tabelas, de um mecanismo de bloqueio dos recursos pretendidos.

2. Abordagem

Inicialmente, antes de qualquer tentativa de implementação de uma solução, foi necessário a utilização de uma ferramenta (*RoadRunner*) especializada na análise de programas concorrentes em Java. Esta ferramenta foi executada com uma série de parâmetros que indicaram que o programa continha erros ao nível da concorrência, nomeadamente d*ata races* (ver *data race* ou *race condition*).

A partir daqui, a primeira abordagem foi determinar uma solução que permitisse o acesso às estruturas de dados de uma forma concorrente, mas que não comprometesse a consistência dos dados nelas armazenados. Começou-se por utilizar uma estratégia mais simples que consistia em bloquear a tabela(1) para cada acesso realizado, fazendo com que cada *thread* acedesse em modo exclusivo à tabela. Para tal foram utilizados *locks* (ver *lock*) de grão grosso (ver *coarse grain locks*) ao nível das operações de inserção, remoção e de consulta da *hashtable*.

No entanto, esta solução não tira partido da concorrência, traduzindo-se num baixo desempenho devido ao facto de se estar a bloquear toda a tabela quando, na realidade, só se acede a um dado nó da mesma. Este facto motivou o desenvolvimento de uma solução alternativa que consistia no bloqueio de linhas da tabela para cada acesso. Isto é, ao invés de bloquear toda a tabela, apenas se bloqueia a linha na qual se encontram os nós a aceder. Desta forma é possível que acessos concorrentes a diferentes linhas da tabela ocorram. Para a solução utilizaram-se *locks* de grão médio ao nível das operações referidas anteriormente. A implementação desta solução exigiu, a par da *hashtable*, a utilização de uma estrutura de dados, no caso um *array*, para armazenar os *locks* correspondentes a cada linha da tabela. Com esta estrutura auxiliar, é possível controlar o bloqueio das linhas da *hashtable*. Estas alterações foram efectuadas ao nível da *hashtable*, o que significa que, até então, só se tinha garantido o controlo de acesso a uma tabela. Mas o que se pretendia controlar era o acesso simultâneo às três estruturas de dados do repositório. O modo de conseguir tal controlo foi fazer com que as *hashtables* disponibilizassem métodos de bloqueio dos seus recursos. Com isto, foi possível fazer com que as instruções do repositório de artigos (a inserção, remoção e consulta) pudessem, à priori, ter o acesso exclusivo sobre dados existentes em cada uma das tabelas, realizar operações com estes e, no fim, solicitar a tabela o desbloqueio destes. Além disso, para as operações de listagem de artigos dado uma lista de autores ou de palavras-chave, como argumento, foi necessário controlar o acesso às respectivas tabelas de forma especial. Isto porque, quando diferentes *threads*

solicitam simultaneamente a uma *hastable* o acesso exclusivo a várias linhas, não há nada que as impeça de obter os mesmos recursos por ordem trocada. O que pode originar um bloqueio geral do programa, *deadlock* (ver *deadlock*). Deste modo, houve a necessidade de garantir que o bloqueio e desbloqueio de inúmeras linhas de uma tabela fossem operações atómicas (ver *atomicity*). Por conseguinte, uma *hashtable* só aceita um pedido de bloqueio de várias linhas de cada vez. O mesmo sucede para o método de desbloqueio. Assim, a tabela possibilita que bloqueios e desbloqueios de múltiplas linhas ocorram simultaneamente por diferentes *threads*.

3. Solução

Tal como mencionado anteriormente, a solução consistiu na implementação de *locks* de grão médio na *hastable*, ou seja, os acessos concorrentes são controlados ao nível das suas linhas. Para o conseguir foi necessário, no código fonte da *hashtable*, um *array* de *ReentrantReadWriteLocks* de tamanho igual ao da *hashtable* e outras duas variáveis do tipo *Lock*. Uma das vantagens de utilizar *ReentrantReadWriteLocks* é que, pelo facto de serem reentrantes, não é necessário verificar se uma dada linha da tabela já se encontra bloqueada. Pois deste modo, mesmo que uma dada *thread* já tenha adquirido o *lock* da linha, pode voltar a bloquear a linha não ficando ela própria bloqueada. Além disto, ao serem *ReadWriteLocks* permitem que sejam várias *threads* possam ler simultaneamente os dados partilhados sem se bloquearem. Apenas quando há uma *thread* que necessita de realizar operações de escrita é que todas as outras se bloqueiam. A utilização deste tipo de *locks* é ideal para o problema abordado uma vez que a ideia é que hajam mais operações de leitura que de escrita. Para utilizar este tipo de *locks* foi necessário importar a biblioteca *java.util.concurrent.locks.ReentrantReadWriteLock*.

As variáveis do tipo *Lock* serviram para poder controlar o acesso de threads aos métodos de bloqueio de múltiplas linhas da *hashtable*. Uma delas foi utilizada para controlar o acesso ao método de aquisição de *locks* sobre várias linhas da tabela. Já a outra foi utilizada para controlar o acesso ao método de realiza o *unlock* de várias linhas. Ambas as variáveis foram inicializadas através do construtor *ReentrantLock()*.

De modo a evitar ter de iterar do *array* todo e inicializá-lo com *ReentrantReadWriteLocks* foi criado um método designado de *getLock*. Este recebe como argumento um número inteiro que representa o índice da *hashtable* ao qual é necessário obter o *lock*. O que este faz é devolver o objecto *lock* associado ao índice dado. Caso este não exista, é criado, guardado no *array* de *locks* e devolvido como resultado.

Além disto, a *hashtable* foi modificada para que os métodos de bloqueio das linhas fossem públicos para serem utilizados por outras classes, mais concretamente, pela classe do repositório. Os métodos de bloqueio foram divididos em dois tipos: com *locks* de leitura (*readLocks*) e com *locks* de escrita (*writeLocks*). E, para cada um destes foram criados dois tipos de métodos: os de aquisição e de libertação dos *locks*. Para ambos os tipos, foram definidas dois tipos de argumentos: apenas uma chave ou uma lista de chaves. As chaves referem-se a valores na tabela. Por conseguinte os métodos de controlo de acesso aos dados da *hashtable* são: *acquireWriteLockKey*, *acquireReadLockKey*, *releaseWriteLockKey*, *releaseReadLockKey*. Nenhum deles retorna qualquer tipo de resultado. Para os métodos que recebem apenas uma chave, bastou calcular o indice na tabela associado à chave, obter o lock da linha e realizar a operação de *readLock().lock()* ou *writeLock().lock()* no caso dos métodos de *acquire* e *r*eadLock().unlock() ou *writeLock().unlock()* nos métodos *release*. Em relação aos métodos que recebem uma lista de chaves como argumento, foi preciso iterar a lista e ir bloqueando a linha correspondente, tal como realizado nos métodos com apenas uma chave como argumento. Ainda nestes métodos, houve a necessidade de fazer com que a iteração da lista fosse realizada em exclusão mútua. Garantindo desta forma que apenas uma *thread* executa um *acquire* de cada vez. O mesmo para os métodos *release*. Para a exclusão mútua foram usadas as variáveis do tipo *Lock* auxiliares, mencionadas anteriormente. Uma só para os métodos de *acquire* e outra para os métodos de *release*.

Até aqui as alterações efectuadas ainda não eram suficientes. Portanto, para resolver o problema do acesso às três *hashtables* foi necessário que os métodos que as utilizam, bloqueassem os recursos necessários antes de acederem a dados e no final efectuassem a sua libertação. Por conseguinte, , no código fonte do repositório foram criados dois métodos privados auxiliares: *lockAllResourcesToWrite* e *unLockAllResourcesToWrite*. Ambos recebem um artigo como argumento. O primeiro realiza o bloqueio de cada uma das estruturas de dados a que vai aceder para escrita, isto é, as *hashtables* de artigos, autores e palavras-chave consoante os dados do artigo recebido. O segundo tem a responsabilidade de realizar o oposto. Isto é, libertar os recursos previamente bloqueados associados ao artigo. Estes são utilizados pelos métodos de inserção e remoção existentes na classe do repositório. No repositório existem ainda os métodos de listagem de autores e palavras-chave. Estes necessitaram de sofrer alterações por forma a controlar o acesso à estruturas respectivas. Assim sendo, o bloqueio dos recursos para leitura foi realizado no início dos respectivos métodos utilizando a chamada ao método *acquireReadLockKey* da respectiva *hashtable*. Em ambos, antes do seu fim, a libertação de recursos foi efectuada através da chamada ao método release*ReadLockKey*.

4. Avaliação

A solução elaborada pretende garantir a correcção do programa. Isto é, que cada operação realizada sobre o repositório é reflectida nas estruturas de dados que o integram. Deste modo, um dos factores importantes em que se deve avaliar a solução é a variação do número de *threads* a executar concorrentemente. Com objectivo de perceber os efeitos desta variação sobre o desempenho do programa. Outro dos parâmetros que se pretende avaliar é o comportamento da

solução nas operações de listagem em função da variação do tamanho das listas. Quer-se verificar se o aumento do tamanho destas implica uma espera maior por parte de outras *threads*.

Outro dos cenários interessantes de avaliar é o impacto resultante do aumento do número de operações de consulta e inserção sobre o repositório. Partindo do pressuposto que estas operações são as mais frequentes, queremos averiguar o comportamento do programa quando existem muitas operações de escrita e leitura sobre varias estruturas de dados em simultâneo.

Todos os testes presentes neste relatório foram realizados utilizando o servidor di110.di.fct.unl.pt. Devido às restrições desta máquina, todos os testes tiveram a duração de 10 segundos. O programa tem ainda outros parâmetros. São eles:
nthreads – número de *threads* em execução concorrente;
nkeys – número de palavras a considerar de um dicionário de palavras guardado num ficheiro;
put(%) – percentagem de inserções realizadas;
del(%) – percentagem de remoções realizadas;
get(%) – percentagem de consultas realizadas;
nauthors – número médio de autores por artigo;
nkeywords – número médio de palavras-chave por artigo;
nfindlist – tamanho das listas para as pesquisas por autor e por palavras-chave.
De referir ainda que em cada teste foram realizadas obtidos quatro resultados. Sendo que para cada um foram realizadas cinco medições, descartando a melhor e a pior para se conseguir resultados mais consistentes. Sendo os resultados presentes neste relatório a média das restantes medições.

4.1. Valores referência

Foram considerados seguintes os valores como referência para os todos os testes presentes no relatório. A sua escolha deve-se ao facto de serem próximos da realidade. Os valores encontram-se descritos em baixo.
nthreads: 16; *nkeys*: 1000; *put(%)*: 25; *del(%)*: 25; *get(%)*: 50; *nauthors*: 6; *nkeywords*: 30; *nfindlist*: 10.

4.2. Avaliação da variação do número de threads

O objectivo deste teste é aumentar o número de *threads* e verificar se esse aumento se traduz num aumento de desempenho (medido em operações por segundo). Em seguida encontra-se uma tabela com os resultados obtidos.

Tabela 1 – Desempenho em função do número de threads

Nº de threads	*Ops/s*
4	*32730*
8	*29882*
16	*28562*
32	*27292*

4.3. Avaliação do aumento do tamanho das listas de procura

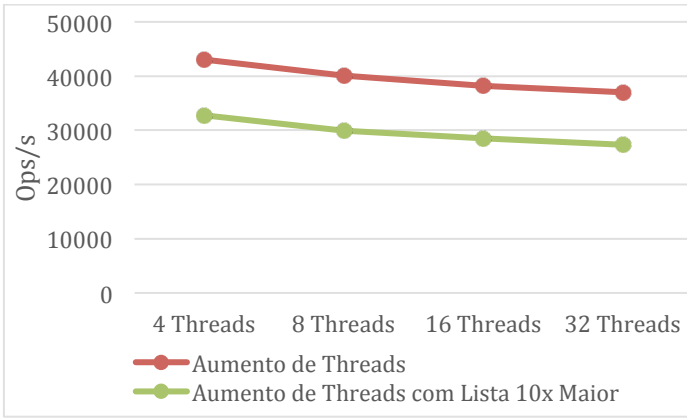
Este teste partiu das condições do teste anterior, o aumento de *theards*. Porém o parâmetro *nfindlist* foi aumentado num factor de dez em relação ao valor de referência. Com isto, pretende-se aferir que efeitos provoca o aumento das lista de procura.

Tabela 2 – Desempenho com listas de dimensão 10 vezes superior

Listas 10x maior	*Ops/s*
4 Threads	*10358*
8 Threads	*10220*
16 Threads	*9657*
32 Threads	*9699*

Para melhor avaliar os resultados obtidos foi elaborado um gráfico de comparação entre este teste e o anterior.

Gráfico 1 – Diferença entre listas de dimensão diferente



4.4. Avaliação da variação da percentagem de consultas

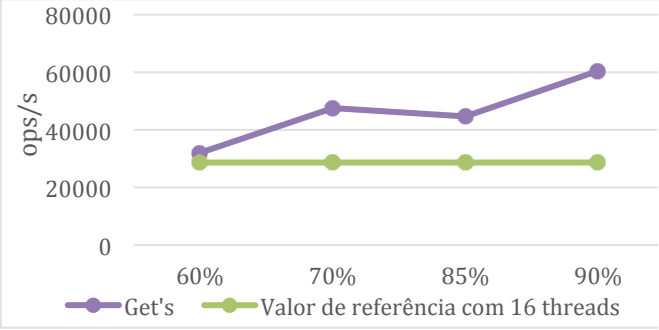
Neste teste pretendeu-se avaliar o impacto resultante do aumento da percentagem de consultas realizadas sobre o repositório. Para aumento da percentagem foram reduzidos as percentagens de inserções e remoções. Os resultados foram obtidos com a seguinte sequência de variação de percentagens:

Tabela 3 – Sequência de percentagens testadas

Put(%)	*Del(%)*	*Get(%)*
20	20	60
10	20	70
10	5	85
5	5	90

É possível comparar os resultados obtidos com o valor de referência.

Gráfico 2 – Comparação entre o valor de referência com o aumento da percentagem de consultas



4.5. Avaliação da variação da percentagem de inserções

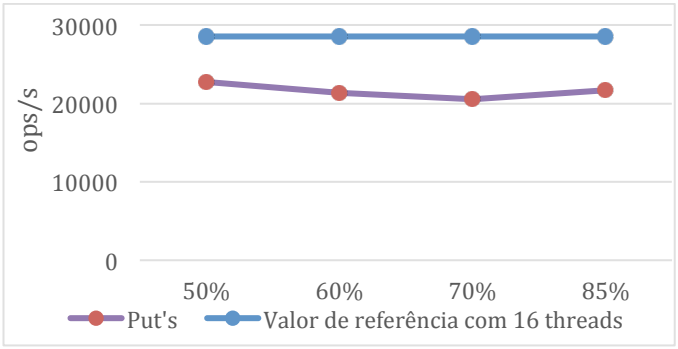
Neste teste pretendeu-se avaliar o impacto resultante do aumento da percentagem de inserções realizadas. Para aumento de cada percentagem foram reduzidos das restantes percentagens. Os resultados foram obtidos com a seguinte sequência de variação de percentagens:

Tabela 4 – Sequência de percentagens testadas

Put(%)	*Del(%)*	*Get(%)*
50	10	50
60	10	30
70	10	20
85	5	10

É possível comparar os resultados obtidos com o valor de referência.

Gráfico 3 – Comparação entre o valor de referência com o aumento da percentagem de inserções



5. Conclusões

Fazendo uma análise de cada conjunto de resultados obtidos consegue-se retirar várias conclusões.

Na primeira avaliação efectuada, depreende-se que com o aumento do número de *threads* não se verifica um ganho de desempenho. Isto significa que o acréscimo de concorrência traduz-se num maior tempo de espera das *threads* no acesso às estruturas de dados. Esta espera deve-se à utilização de *locks* para resolver o problema dos acessos concorrentes às tabelas.

No segundo teste quis-se analisar o efeito causado pelo aumento do tamanho das listas de procura. Verificou-se, em comparação com os resultados do primeiro teste, que há um agravamento do tempo de bloqueio das *threads* no acesso às *hashtables*.

Uma outra análise efectuada permite retirar conclusões acerca do impacto provocado pela utilização dos dois diferentes tipos de *locks*, de leitura e escrita. Através da análise conjunta entre o aumento da percentagem de consultas e o aumento da percentagem de inserções consegue-se inferir que as operações de consulta são menos pesadas que as de inserção. Essencialmente devido ao facto de as operações de leitura apenas têm a necessidade de bloquear uma das tabelas, ao passo que as de escrita têm que bloquear as três estruturas do repositório. E ainda, ao facto dos *locks* de leitura permitirem a leitura simultânea entre *threads* enquanto que, ao existir um *lock* de escrita obriga a que o recurso fique bloqueado para todas as *threads* até ser realizado o *unlock*.

Esta solução garante a correcção, apesar de não ser a mais eficiente nem a mais elegante. Uma vez que foram implementados *locks* de grão médio, ao nível das linhas das *hashtables*. No entanto, é possível construir uma solução com *locks* de grão fino ainda que seja uma tarefa de difícil implementação. A falta de elegância da solução concentra-se no facto de as *hashtables* terem de tornar visíveis os métodos de bloqueio de acesso dos dados armazenados. Ao invés de serem métodos privados às mesmas.

Referências

Deadlock ou race condition:
http://en.wikipedia.org/wiki/Race_condition#Software
Lock :
http://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html

Thread:
http://en.wikipedia.org/wiki/Thread_(computing)

Hashtable:
http://en.wikipedia.org/wiki/Hash_table

Atomicity:
http://en.wikipedia.org/wiki/Atomicity_(programming)

Outros Links consultados:

http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

http://ilkinbalkanay.blogspot.pt/2008/01/readwritelock-example-in-java.html

http://programmingexamples.wikidot.com/reentrantreadwritelock

http://tutorials.jenkov.com/java-concurrency/locks.html

http://www.javacodegeeks.com/2011/09/java-concurrency-tutorial-reentrant.html