Análise de Desempenho de Estruturas de Dados Concorrentes Implementadas na Linguagem Java

Luiz Gustavo C. Xavier[1], **Odorico M. Mendizabal**[2]

[1]Centro de Ciências Computacionais
Universidade Federal do Rio Grande (FURG) – Rio Grande – RS – Brasil

[2]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

luiz.gustavo@furg.br, odorico.mendizabal@ufsc.br

Resumo. *O uso de estruturas de dados capazes de garantir exclusão mútua no acesso a seus elementos é visto como uma necessidade a desenvolvedores de aplicações paralelas, devido à grande abstração quanto ao controle de concorrência. Este trabalho tem por objetivo mensurar o custo destas estruturas no desempenho de aplicações com diferentes níveis de concorrência.*

1. Introdução

O desenvolvimento de aplicações paralelas tem por prerrogativa contornar gargalos de desempenho presentes em uma aplicação sequencial. Ao fragmentar um processo em diferentes linhas de execução, se faz necessário assegurar garantias de exclusão mútua e consistência dos dados utilizados pela aplicação. Diversos autores enunciam problemas referentes a possíveis comportamentos indeterminísticos ocasionados pelo não cumprimento de tais garantias [Courtois et al. 1971][Hoare 1978].

Há diferentes técnicas conhecidas na literatura para satisfazer a coordenação em um ambiente concorrente [Dijkstra 1968][Hoare 1974], onde são implementados meios para barrar o acesso simultâneo de diferentes processos ou *threads* a uma mesma área de endereçamento, garantindo assim a condição de exclusão mútua. Uma alternativa às técnicas é a implementação de estruturas de dados concorrentes. As mesmas têm por característica principal permitir a execução concorrente de operações próprias da estrutura (e.x. *add*, *get* e *remove*), de modo totalmente transparente ao programador.

2. Estratégias de Sincronização em Java

Nesta seção são apresentadas duas estruturas de dados contempladas neste estudo, bem como estruturas de bloqueio que são utilizadas como critério de comparação.

ArrayList: Sendo uma implementação da interface *List*, a estrutura *ArrayList* armazena internamente seus elementos em um vetor dinâmico, o qual é continuamente incrementado em capacidade na adição de novos elementos. Devido a esta necessidade eventual de realocação interna, a classe também possui meios de minimizar a frequência com que essa operação custosa é executada[1]. É importante ressaltar que esta estrutura não possui internamente nenhuma implementação de garantia de sincronismo, isto é, não

[1]https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

é assegurada consistência entre operações concorrentes que venham a ser executadas sobre a estrutura, ficando a cargo do programador a utilização de estruturas externas de sincronismo para garantir a atomicidade nas operações sobre este conjunto de dados.

CopyOnWrite ArrayList: Essa estrutura implementa internamente a técnica de cópia-na-escrita, que garante que os dados capturados por operações de leitura não sejam modificados por operações simultâneas de inserção, modificação ou remoção dos dados, sem a necessidade de coordenação entre estes métodos. A técnica de cópia-na-escrita consiste na alocação de cópias temporárias de regiões do espaço de endereçamento para assegurar a leitura consistente de um bloco protegido, mesmo com o acontecimento de diversas operações simultâneas de modificação destes dados.

Monitor Synchronized: O monitor se caracteriza como uma estrutura de sincronização de alto nível que armazena internamente uma fila de *threads* ou processos que requisitam acesso da região ao qual o mesmo está preservando. Com o uso de monitores, se torna dispensável ao programador a necessidade de bloqueios e desbloqueios de acesso a uma região crítica, uma vez que o mesmo após inicializado se encarrega de assegurar a exclusão mútua. A estrutura *synchronized* implementada em Java permite ao desenvolvedor definir métodos ou trechos de código (através da simples adição da *keyword* na região desejada) que ao serem executados realizam internamente a aquisição de uma trava de acesso exclusivo à região protegida.

Semaphore: A implementação de um semáforo utiliza variáveis internas para controlar o acesso a um recurso físico de endereçamento. A verificação quanto ao estados desta estrutura é realizada por operações atômicas de *acquire()* e *release()*, garantindo a condição de exclusão mútua do recurso protegido. A utilização de um semáforo não se restringe a um comportamento binário, quando se deseja bloquear ou liberar o acesso a uma região protegida. É possível utilizá-lo para assegurar que um número finito de processos acessem a região protegida por vez, ou, então, para delimitar diferentes regiões de um mesmo espaço de endereçamento.

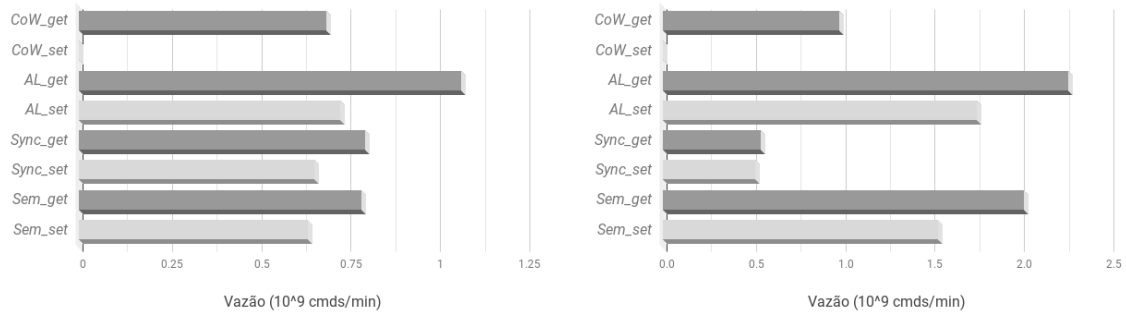
3. Avaliação Experimental

Para avaliar o desempenho das estruturas discutidas na Seção 2, foi desenvolvido um simulador, que executa um número pré-definido de *worker threads* responsáveis por executar operações *get* e *set* sobre estruturas de dados *CopyOnWriteArrayList* e *ArrayList*. Parâmetros de configuração como número de *threads* concorrentes, tamanho do espaço de endereçamento, porcentagem de incidência de operações *get* e *set* e tempo de experimentação, são configuráveis.

Os experimentos foram realizados em uma máquina com sistema operacional Ubuntu 18.04LTS 64bits, processador Intel Core i5 *quad-core* de 2.5GHz e 8GB de memória DDR4. O simulador executa sobre uma JVM (*Java Virtual Machine*) versão 1.8.0_162. São populados na estrutura dez milhões de elementos de 256 bytes, totalizando um uso de memória *heap* pelos dados alocados de aproximadamente 2,5GB.

Para avaliar o desempenho dos mecanismos de sincronização citados, foram analisados o número de operações independentes *get* e *set* executadas pelo simulador em intervalos de um minuto, em cenários com 1 e 8 *threads* em execução. A Figura 1 apresenta a vazão em comandos/minuto obtida com as estruturas de dados analisadas.

Operações sobre *ArrayList* apresentam os melhores resultados nas duas operações, pois esta estrutura não implementa controle de concorrência. Observa-se que *Syncronized* tem um desempenho inferior em cenários com maior concorrência de acesso, em razão da sua implementação interna de enfileiramento de processos. *Semaphore* e *CopyOnWriteArrayList (CoWAL_get)* apresentam maior vazão no cenário *multithread*, não somente por explorar a arquitetura *multi-core* utilizada nos experimentos, mas também por apresentar custo de sincronização mais escalável com o aumento de concorrência.



(a) Cenário 1 thread

(b) Cenário 8 threads

Figura 1. Vazão individual de operações *get* e *set*.

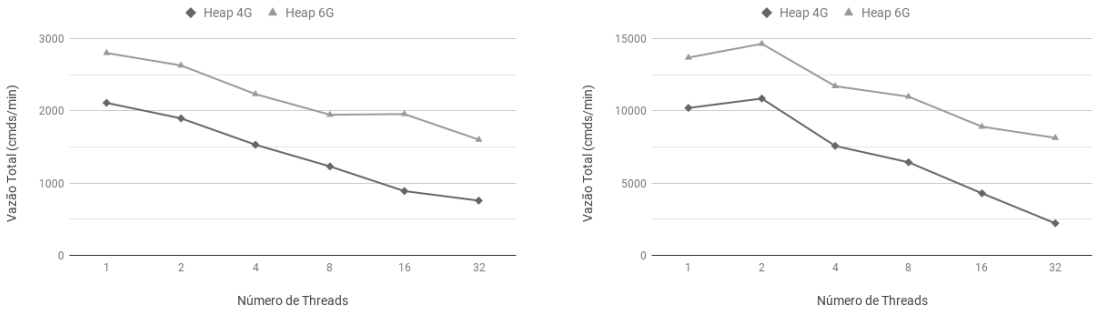
Os experimentos acessam uma única instância de cada estrutura de dados analisada. Portanto, é possível que alguns acessos a regiões específicas da estrutura em avaliação sejam beneficiados pelo gerenciador de memória da JVM e do sistema operacional. Trabalhos futuros devem levar em consideração o efeito do princípio da localidade.

Para avaliar o desempenho das estruturas em cenários com diferentes perfis de execução, foram definidos comportamentos de utilização que variam a porcentagem de incidência de operações de leitura e escrita, sendo (I) 50% leitura, 50% escrita e (II) 90% leitura, 10% escrita. Para estes cenários, foram computadas a vazão máxima de operações executadas pelas estruturas de dados concorrentes em intervalos de um minuto.

Na Figura 2 é possível observar o impacto na vazão total de operações gerado pela limitação de memória sobre a estrutura *CopyOnWriteArrayList*. Em ambos os casos, (I) e (II), acontece uma redução na vazão ao diminuir a quantidade de memória disponível. Para estes experimentos, foram utilizados os parâmetros `-Xms` e `-Xmx` para definir as quantidades iniciais e máxima de memória *heap* utilizada pela JVM.

É possível constatar também que na execução do Perfil I, que possui grande incidência de escritas, resultou em uma menor vazão suportada pela estrutura, atingindo o máximo de 2.700 operações/min., o que comprova um grande custo computacional do método *set* devido ao mesmo realizar um cópia temporária de todo conteúdo da estrutura antes de efetivamente efetuar a operação de modificação neste conjunto de endereçamento protegido. Na Figura 2(b), que representa a execução com o Perfil II, é observada uma maior vazão de comandos executados pela estrutura, chegando a 14.000 operações/min. A maior vazão deve-se a maior incidência de métodos *get*, o que indica uma melhor aplicação do *CoWAL* em cenários de intensa leitura e raras operações de escrita.

Em [Pinto et al. 2016], os autores também apontam um alto consumo de memória demandado por *CoWAL*, o que acarreta em maior consumo energético. Por exemplo,



(a) Perfil I

(b) Perfil II

Figura 2. Impacto da indisponibilidade de memória no desempenho de *CoWAL*.

comparando com um *Vector*, o consumo ao usar *CoWAL* pode ser 46 vezes maior. Em [Goetz 2003], são comparadas as estruturas *ConcurrentHashMap* e *HashTable*, onde a primeira apresenta melhor escalabilidade devido à sincronização em *HashTable* ser feita por uma única trava de acesso a toda estrutura.

4. Considerações Finais

Os experimentos realizados apontam que o uso da estrutura *Semaphore* para assegurar exclusão mútua sobre dados operados em uma estrutura *ArrayList* resulta em uma menor sobrecarga na vazão de uma aplicação concorrente. Porém, o uso desta estrutura muitas vezes é descartado devido a mesma não possuir nenhuma abstração quanto ao controle de concorrência ao programador.

CoWAL demostrou um desempenho superior ao monitor *Synchronized* com a invocação de operações de leitura. Ao analisar o comportamento do primeiro em diferentes perfis de utilização em um cenário concorrente, foi observada a diminuição do número total de operações executadas em um cenário com maior incidência de escritas, o que demonstra um alto custo computacional de alocação das cópias temporárias da técnica de *copy-on-write*, e sua limitação quanto à disponibilidade de memória.

Referências

Courtois, P.-J., Heymans, F., and Parnas, D. L. (1971). Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668.

Dijkstra, E. W. (1968). Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer.

Goetz, B. (2003). Java theory and practice: Concurrent collections classes. *IBM Developer Works*.

Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. In *The origin of concurrent programming*, pages 272–294. Springer.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

Pinto, G., Liu, K., Castor, F., and Liu, Y. D. (2016). A comprehensive study on the energy efficiency of java’s thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31. IEEE.