Implementação de um algoritmo de tabela hash não bloqueante em Haskell.

Rodrigo M. Duarte[1]*, André R. Du Bois[1], Maurício L. Pilla[1], Renata H. S Reiser[1]

[1]CDTec – Universidade Federal de Pelotas (UFPEL)
Engenharia de Computação
Gomes Carneiro 1 – 91.501-970 – Pelotas – RS – Brasil

{rmduarte, dubois, pilla, reiser}@inf.ufpel.edu.br

Resumo. *Tabela Hash é um tipo de estrutura de dados amplamente usada por apresentar complexidade O(1). Neste artigo é proposta a implementação em Haskell de uma tabela hash concorrente usando um algoritmo não bloqueante. O objetivo é fornecer uma biblioteca de hash concorrente eficiente e também comparar este algoritmo com outros, que usam outras abstrações concorrentes de alto nível disponíveis em Haskell.*

1. Introdução

Estrutura de dados do tipo *hash* possuem um paralelismo natural, isso devido ao fato que o acesso aos dados em posições distintas da tabela podem ser realizado de forma independente. Essa característica permite que se explore diferentes técnicas de paralelismo neste tipo de estrutura, entre eles o de *lock global*, *lock em blocos* e de *granularidade fina* [Herlihy and Shavit 2012]. Todos estes algoritmos possuem em comum o fato de serem bloqueantes, ou seja, uma *thread* tem de ficar parada até que outra termine seu acesso à seção crítica. A fim de evitar este problema, pode-se usar um algoritmo de tabela *hash* não bloqueante para diminuir os efeitos do bloqueio no desempenho geral. Porém implementar um algoritmo de tabela *hash* deste tipo não é algo trivial, problemas como o crescimento da tabela, que exige a modificação atômica de mais de uma posição de memória, exigem uma abordagem diferente das convencionais.

Haskell é uma linguagem de programação funcional de alto nível que possui várias abstrações para a programação paralela, entre elas variáveis de sincronização `MVars`, acesso a instruções de baixo nível para sincronização – *compare and swap* (CAS)[Marlow 2013] e memórias transacionais [Harris et al. 2008]. Devido ao alto nível de abstração da linguagem e o forte tratamento de tipos, desenvolver programas paralelos se torna mais simples e seguro.

Neste artigo propomos a implementação em Haskell de um algoritmo de tabela *hash* não bloqueante a fim de comparar os resultados obtidos com os de outras implementações de tabela *hash* apresentados em [Duarte et al. 2014]. Dessa forma podemos avaliar desempenho, escalabilidade e facilidade de uso das diversas abordagens de implementação. Outro objetivo é fornecer uma implementação de tabela *hash* eficiente e segura como uma biblioteca para Haskell, que não possui uma biblioteca deste tipo [Newton et al. 2011].

*Bolsista AF-PBIP UFPel

O artigo esta organizado da seguinte forma. Na Seção 2 é apresentado o algoritmo estudado para a realização do trabalho. Na Seção 3 é feita uma pequena apresentação das primitivas para implementação de algoritmos não bloqueantes presentes em Haskell e finalizando, na Seção 4 é apresentada a proposta de trabalho.

2. Tabela hash não bloqueante

Um algoritmo de tabela *hash* não bloqueante não é algo trivial de ser implementado. A principal dificuldade neste tipo de algoritmo é na realização do crescimento da tabela. Não basta fazer com que os *slots* da tabela sejam *lock-free*, pois na necessidade de realizar o crescimento da tabela, a mudança dos dados da tabela antiga para a nova deve ser de forma atômica para evitar perda ou inconsistência de dados. Infelizmente as instruções presentes em hardware para a realização de *compare and swap*, são instruções que operam somente com uma única posição de memória, tornando difícil a mudança dos dados da tabela antiga para a nova de forma a evitar inconsistência.

Em virtude disso, o algoritmo proposto por [Herlihy and Shavit 2012] utiliza uma técnica onde ao invés de movermos os itens através dos *slots*, movemos os *slots* através dos itens. Neste algoritmo, todos os dados são mantidos em uma lista encadeada *lock-free* e cada *slot* é simplesmente uma referência para uma determinada posição na lista, como pode ser visto na Figura 1.

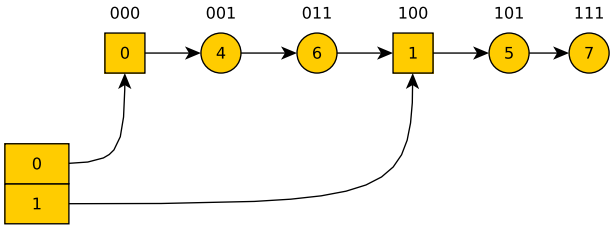

Figura 1. Esquema da *hash lock-free*

Os itens da tabela são ordenados através do valor dos bits reversos do código *hash* dos mesmos. Esse tipo de ordenação permite que, quando há a necessidade de crescimento da tabela, os dados da lista encadeada sejam divididos entre dois *slots* diferentes, não havendo a necessidade de move-los para uma nova tabela. Porém aqui, a capacidade da tabela *hash* deve ser sempre uma potência de 2, para que se possa dividir a tabela de forma que a distribuição dos dados seja coerente, ou seja, quando temos 2^1 *slots*, somente o primeiro bit mais significativo do item inserido é considerado para ordenação, se a tabela aumenta para 2^2, então agora consideramos os dois bits mais significativos do item. Figura 2.

Para cada *slot* da tabela é criado um nodo sentinela na lista encadeada que nunca será removido. Isso se deve ao fato de que se houver uma remoção completa dos dados de um determinado *slot*, o mesmo não tenha que apontar para uma posição invalida e ou para um elemento de um outro *slot*.

Quando a tabela *hash* é criada, todos os *slots*, com exceção do *slot* 0, são nulos e o *slot* 0 aponta para uma lista vazia. A medida que os dados são inseridos na tabela os *slots* vão sendo inicializados e vão dividindo os dados presentes na tabela entre os *slots*.

Um exemplo básico de como os dados são inseridos na tabela pode ser visto da Figura 2. considerando dados de 8 bits, que o item de valor 10 será inserido e uma tabela com os *slots* um e dois já inicializados.

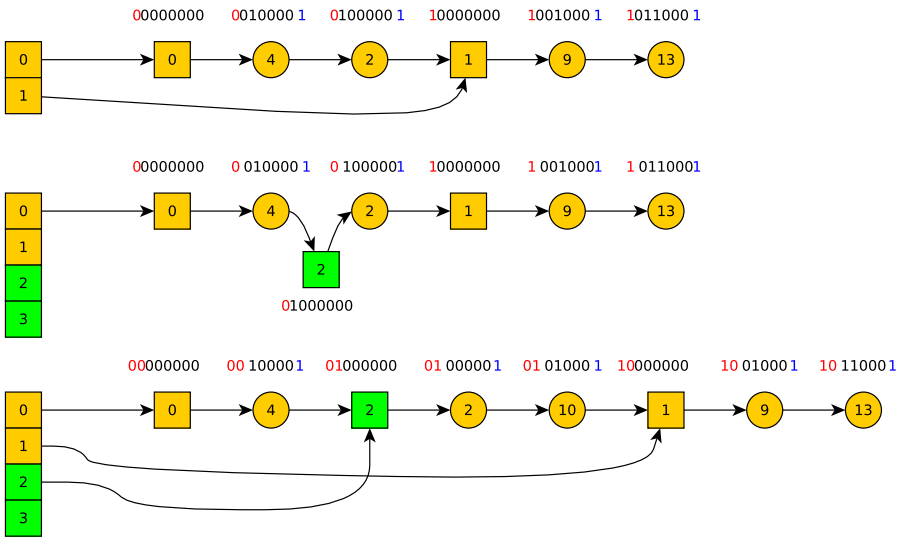

Figura 2. Inicializando o *slot* 2 e inserindo o valor 10.

Para diferenciar os nodos sentinelas dos valores, um esquema setando o último bit dos valores é utilizando.

3. Métodos de sincronização em Haskell

Entre os vários modelos para sincronização não bloqueante entre *threads* presentes em Haskell, dois modelos diferentes se destacam que são `IORef + atomicModifyIORef` e STM Haskell. STM é um modelo de sincronização não bloqueante, de alto nível de abstração que se destaca pela facilidade em se desenvolver programas concorrentes com sua utilização. Por apresentar resultados de desempenho considerável se comparada a outros modelos de sincronização [Duarte et al. 2014], STM Haskell é uma proposta interessante para se explorar a implementação de algoritmos não bloqueantes.

3.1. IORef + atomicModifyIORef:

Este método pode ser comparado ao uso da instrução CAS (*compare and swap*), disponível em alguns processadores. Um `IORef` é uma referência a uma posição de memória que possui as seguintes operações: `newIORef`, `readIORef` e `writeIORef`, correspondentes a criação, leitura e escrita respectivamente. Essas operações sozinhas não garantem segurança no acesso *multithread* das referências, porém Haskell fornece uma função `atomicModifyIORef`, que assim como a operação CAS, permite a modificação atômica da referência, podendo assim uma `IORef` ser utilizada para implementar algoritmos não bloqueantes.

3.2. STM Haskell (Software Transactional Memory):

Software transactional Memory (STM) é um novo modelo de sincronização entre *threads* que simplifica a programação concorrente, permitindo que operações possam ser compostas em uma simples operação atômica. A ideia é fazer com que as operações sejam realizadas como transações parecidas com as transações de bancos de dados [Rigo et al. 2007]. Neste modelo, todo o sincronismo é realizado pelo sistema transacional, evitando assim problemas como *deadlocks*.

STM Haskell é uma extensão da linguagem Haskell que fornece primitivas para a programação usando STM [Harris et al. 2008]. Nela é definida um tipo de variável transacional (`TVar`), que é criada pela primitiva `newTVar`, e modificada pelas primitivas `readTVar` e `writeTVar`. Estas primitivas só podem ser executadas dentro de uma chamada a `atomically`. STM Haskell garante que operações que modificam uma `TVar`, sejam somente realizadas dentro de uma transação. Assim, o programador não precisa se preocupar com o sincronismo, pois o sistema de tipos de Haskell garante que nenhuma variável `TVar` seja alterada fora de um bloco protegido, garantindo assim consistência e facilidade no desenvolvimento de programas paralelos.

4. Proposta

Este artigo traz como proposta a implementação do algoritmo de tabela *hash* apresentado na Seção 2, na linguagem funcional Haskell e compara-la a outras implementações existentes afim de verificar o desempenho e facilidade de programação. A ideia é se basear na implementação de uma lista encadeada não bloqueante já apresentada em outros trabalhos, como em [Sulzmann et al. 2009], realizando alterações para se adaptar ao algoritmo apresentado. Após verificado a validade da implementação, anexar a mesma a uma biblioteca para Haskell, afim de fornecer a implementação para outros usuários da linguagem, visto que não existe biblioteca de tabelas hash concorrentes em haskell [Newton et al. 2011].

Referências

Duarte, R. M., Pilla, M. L., Reiser, R. H. S. e Du Bois, A. R. (2014). Implementação de algoritmos de tabelas hash concorrentes em haskell. *Anais WSCAD*, pages 284–289.

Harris, T., Marlow, S., Jones, S. P. e Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51:91–100.

Herlihy, M. e Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier.

Marlow, S. (2013). *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. ”O’Reilly Media, Inc.”.

Newton, R., Chen, C.-P. e Marlow, S. (2011). Intel concurrent collections for haskell.

Rigo, S., Centoducatte, P. e Baldassin, A. (2007). Memórias transacionais: Uma nova alternativa para programação concorrente. In *Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho, WSCAD 2007*.

Sulzmann, M., Lam, E. S. e Marlow, S. (2009). Comparing the performance of concurrent linked-list implementations in haskell. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46. ACM.