IMPLEMENTAÇÃO DE TABELA HASH NÃO BLOQUEANTE EM HASKELL

RODRIGO MEDEIROS DUARTE[1];
A. R. DU BOIS. M. L. PILLA, R. H. S. REISER

[1]*Universidade Federal de Pelotas – {rmduarte, dubois, pilla, reiser}@inf.ufpel.edu.br*

1. INTRODUÇÃO

Estruturas 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 realizados 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,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, 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,2005). 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 apresentamos a implementação em Haskell de um algoritmo de tabela *hash* não bloqueante usando duas abordagens diferentes de sincronização (CAS e STM). Comparamos o desempenho das duas implementações e verificamos que a implementação usando CAS obteve o melhor resultado. Porém a implementação usando STM foi a mais simples de implementar e obteve resultados a serem considerados. Outro objetivo deste trabalho é o de fornecer uma implementação de tabela *hash* eficiente e segura como uma biblioteca para Haskell, que não possui uma biblioteca deste tipo (NEWTON,2011).

2. METODOLOGIA

2.1 – MÉTODOS DE SINCRONIZAÇÂO EM HASKELL

Esta seção descreve primeiramente a metodologia para consolidação das duas abordagens para a implementação de *hash* não bloqueante: CAS e STM. Na sequencia, apresenta-se estes métodos de sincronização e a implementação da tabela *hash*.

2.1.1 - IORef + atomicModifyIORef

Este método pode ser comparado ao uso da instrução CAS 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.

2.1.2 - 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,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,2005). 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 função *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.

2.2 – TABELA HASH NÂO BLOQUEANTE

As principais dificuldades na implementação de um algoritmo não bloqueante esta na duplicação da tabela, pois as instruções CAS, somente operam com uma única posição de memória, o que impossibilita o deslocamento dos dados de uma tabela antiga para a nova de tamanho duplicado.

Para evitar este problema, baseamos nossa implementação em um algoritmo proposto por (SHALEV,2006), onde utiliza uma técnica que ao invés de movermos os itens através dos *slots* da *hash*, movemos os *slots* (posições da tabela) 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.

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 movê-los para uma nova tabela. Para cada *slot* da tabela é criado um nodo sentinela na lista encadeada que nunca será removido. Neste contexto, 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.*

Em nossa implementação para a lista encadeada utilizamos como base o algoritmo proposto em (SULZMANN,2009) com modificações para se adequar a nossa implementação. Dentre estas modificações estão a inclusão de guardas e ordenação dos dados na lista pelo valor dos bits reversos, como previsto no algoritmo original da *hash* não bloqueante.

A segunda parte da implementação foi a *hash* em si, que consta de um vetor onde cada *slot* faz referência a uma posição especifica da lista encadeada, sendo esta a parte de maior complexidade da implementação.

3. RESULTADOS E DISCUSSÃO

Para os teste foi utilizado um computador com processador core i7 de oito núcleos de processamento, sendo quatro núcleos físicos e quatro lógicos (*Hyper Thread*), 8Gb de memória RAM, rodando Ubuntu 14.04 de 64bits e Compilador Haskell GHC 7.6.3 com a biblioteca de STM 2.4.2. Os resultados obtidos da média do tempo para 30 execuções, com 10 milhões de operações sobre a tabela para cada execução, podem ser vistos nos gráficos da Figura 1, que estão organizados em tempo de execução e *speedup* respectivamente. Os gráficos de cima para execução com 10% de inserções e 80% consultas sobre as 10 milhoes de operações e os abaixo para 80% inserções e 10% consulta. Estas segunda métrica foi utilizada para sobrecarregar alterações na tabela, visto que consultas não geram modificações na mesma.

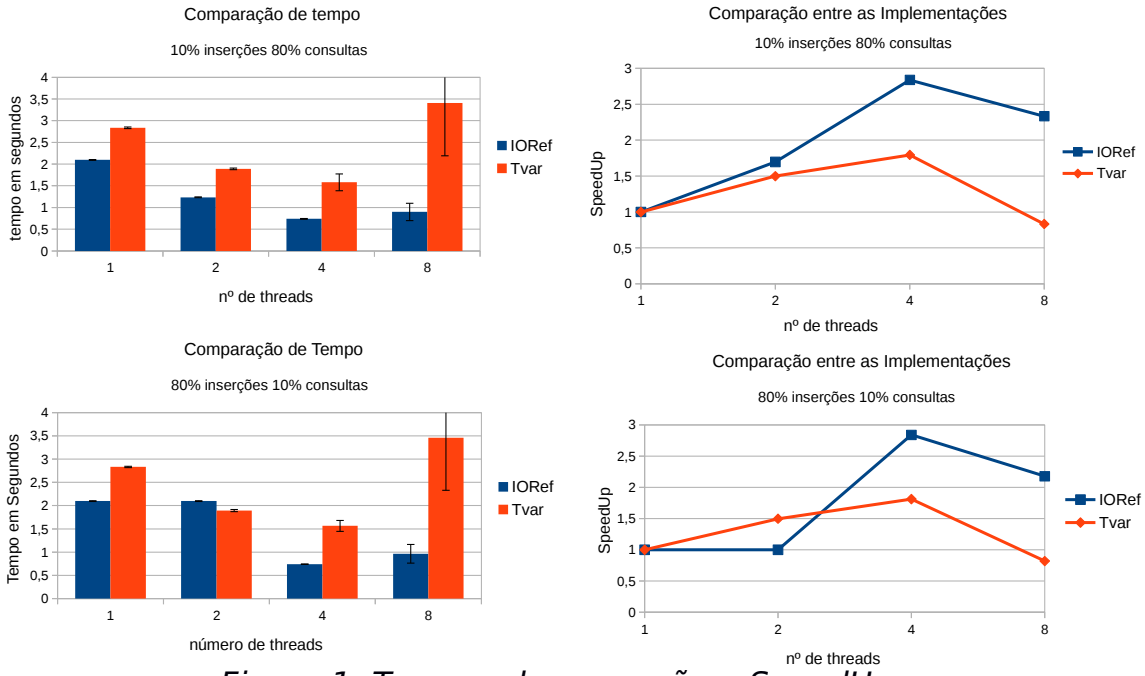

Figura 1: Tempos de execução e SpeedUp

Como se pode ver nos gráficos, tanto a implementação usando IORef como TVar obtiveram desempenho, conforme aumentou-se o número de *threads* (um *thread* por núcleo de processamento).

A implementação usando IORef apresentou menor tempo de execução devido ao baixo *overhead* imposto por este tipo de sincronização, o que não ocorreu com TVar. Isso devido aos custos da gerência de memória impostos pelas implementações de STM. Nos dois últimos gráficos vemos que ao se aumentar o número de inserções da tabela, a implementação usando TVar não ganha muito desempenho pois a contenção (alterações sobre a tabela) aumenta consideravelmente, aumentando assim o custo da STM para tratar conflitos sobre a tabela. O mesmo pode ser observado com a utilização de IORef que quase não apresenta desempenho ao se dobrar o número de threads. Quando ambas as implementações foram executadas com oito threads, ambas perdem desempenho, sendo que a implementação com Tvar foi mais evidente.

O desvio padrão de ambos os testes até quatro threads não passou de 0.1, o que indica que os valores observados durantes os testes não apresentaram uma variação muito grande, ficando dentro do intervalo mensurado.

4. CONCLUSÕES

O a tabela hash utilizando IORef foi a que obteve melhor desempenho geral, considerando assim que em uma aplicação onde se necessite desempenho seria a mais indicada, porém a complexidade de implementação da mesma se torna um item a ser considerado em sua implementação. A garantia de corretude de código (problemas de condição de corrida por acessos irregulares na tabela) fica a cargo do programador que desenvolve a aplicação.

Na utilização de TVar (STM), o desempenho não foi muito inferior ao utilizando IORef. Assim, para aplicações que não necessitem de muito desempenho, a utilização de STM deve ser considerada, pois a facilidade de implementação do código e a garantia de corretude de acesso a tabela fica todo a carago do sistema transacional, ou seja, abstrai do programador todo o processo de sincronização.

Como trabalhos futuros, pretende-se comparar os resultados de desempenho das implementações apresentadas neste trabalho com as de outro trabalho também envolvendo tabelas *hash* concorrentes utilizando outros métodos de sincronização (DUARTE,2014).

5. REFERÊNCIAS BIBLIOGRÁFICAS

HERLIHY, M. e Shavit, N. **The art of Multiprocessor Programming**. Elsevier. 2012.

MARLOW, S. **Parallel and Concurrent Programming in Haskell**: O'Relly, 2013.

SULZMANN, M., Lam, E. S. e Marlow, S. Comparing the Performance of concurrent linked-list implementations in haskell. **Proceedings of the 4th workshop on Declarative aspects of multicore programming**, ACM, 2009. 37-46.

DUARTE, R. M., Pilla, M. L., Reiser, R. H. S. e Du Bois, A. R. Implementacão de algoritmos de tabelas hash concorrentes em haskell. **Anais WSCAD**, 284-289. 2014.

HARRIS, T., Marlow, S., Jones, S. P. e Herlihy, M. Composable memory transactions. Commun. **ACM**, 91–100. 2008.

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

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

SHALEV, O. and Shavit, N. Split-ordered lists: Lock-free extensible hash tables. Journal of the **ACM** (JACM), 53(3):379–405, 2006.