AVALIAÇÃO DE ÁRVORES RUBRO-NEGRAS UTILIZANDO DIFERENTES MÉTODOS DE SINCRONIZAÇÃO DA LINGUAGEM HASKELL

THAÍS S. HÜBNER; RODRIGO M. DUARTE; ANDRÉ RAUBER DU BOIS

Universidade Federal de Pelotas– {tshubner,rmduarte, dubois}@inf.ufpel.edu.br

LUPS - Laboratory of Ubiquitous and Parallel Systems

1. INTRODUÇÃO

Apesar do grande avanço nos processadores singlecore quando se trata da redução de tamanho, isso os levou a atingirem grande velocidade (frequência) e logo não haveria mais refrigeração eficiente o bastante para lidar com suas altas temperaturas.

Dessa forma, se tornou cada vez mais necessário o uso de processadores multicore, ou seja, que possuem dois ou mais núcleos de processamento no interior de um único chip.

Se antes os processadores multicore eram uma necessidade, hoje são uma realidade. Junto com as vantagens dos processadores multicore, veio também a exigência da programação paralela para que se possa tirar o máximo de proveito deste tipo de arquitetura. É preciso de mais de um fluxo de execução (thread) trabalhando ao mesmo tempo.

Todavia não é tão simples obter uma sincronização das threads no acesso à memória. A solução mais tradicional é o uso de bloqueios (locks), porém estes são de difícil utilização e propensos a erros de programação e.g. deadlocks [RAJWAR 2003].

Haskell é uma linguagem de programação funcional que oferece várias formas de sincronização além de locks, tais como memórias transacionais.

Este trabalho tem como objetivo implementar árvores rubro-negras em Haskell, utilizando suas diferentes formas de sincronização, a fim de analisar o desempenho de execução e facilidade de implementação de cada uma delas.

2. THREAD E O PROBLEMA NO USO DE BLOQUEIOS

Na programação sequencial temos uma única linha de execução. As threads estendem essa ideia, nos proporcionando executar linhas de execuções simultâneas em um programa.

Geralmente, neste tipo de programação, as threads se comunicam por uma área de memória chamada *região crítica*. É necessário métodos de sincronização para que estas threads não interfiram de forma errada na execução umas das outras.

Além do baixo desempenho no uso de bloqueios, também há a difícil composabilidade, ou seja, não há segurança em utilizar pequenos trechos de programa para criar trechos de programa mais complexos. Não há como garantir que dois trechos de códigos válidos, quando combinados, continuarão válidos.

3. MEMÓRIAS TRANSACIONAIS

A memória transacional é uma nova abstração para programação baseada na ideia de transações de banco de dados.

Anteriormente usada efetivamente no próprio contexto de banco de dados, foi introduzida em programação concorrente visando facilitar a programação concorrente.

Este método fornece maior facilidade na programação, ausência de deadlock e a composabilidade de código, muito importante na engenharia de software (RIGO, 2007).

4. MÉTODOS DE SINCRONIZAÇÃO EM HASKELL

Haskell possui diferentes abstrações para sincronização entre threads, como variáveis de sincronização (**MVars**), abstrações de alto nível como STM Haskell (**TVars**) e acesso a instruções de baixo nível (**IORef + AtomicModifyIORef**).

Estas abstrações em conjunto com a facilidade da linguagem funcional Haskell, fornecem uma excelente ferramenta para o desenvolvimento de programas paralelos e.g. [MARLOW 2013].

Mvar: É um mecanismo de comunicação entre threads em Haskell. São variáveis especiais de sincronização que podem assumir um estado: cheia ou vazia.

No momento da criação de uma **MVar**, pode ser definido seu estado inicial através das funções **newMVar** e **newEmptyMVar**.

Possui também funções de manipulação: **takeMVar** retorna o valor da **MVar** caso esteja cheia, ou bloqueia caso esteja vazia e **putMVar** que funciona de forma contrária, bloqueando caso esteja cheia e escrevendo-a caso esteja vazia.

STM Haskell: É uma extensão da linguagem Haskell que fornece a abstração de memórias transacionais e simplifica a programação concorrente.

STM Haskell fornece uma variável **TVar**, que conta com as funções **newTVar**, **readTVar** e **writeTVar** que permitem que uma **TVar** seja criada, lida e escrita, respectivamente.

Estas funções só podem ser utilizadas dentro de uma chamada à função atomically, garantindo que operações que modificam uma **TVar** sejam realizadas apenas dentro de uma transação. Desta forma, o programador precisa se preocupar apenas com a identificação das regiões críticas, sem que seja necessário se preocupar com problemas de sincronização, ordem de aquisição de bloqueios, deadlocks, etc.

IORef + AtomicModifyIORef: Um **IORef** é uma referência à uma posição de memória e possui as operações **newIORef**, **readIORef** e **writeIORef**, permitindo criação, leitura e escrita, respectivamente.

Apesar de por si só estas funções não garantirem segurança no acesso multithread das referências, Haskell fornece ainda a função **atomicModifyIORef** , que permite a modificação atômica da referência. Desta forma, uma **IORef** pode ser utilizada para implementar algoritmos não bloqueantes.

5. ÁRVORES RUBRO-NEGRAS

As árvores rubro-negras são estruturas de dados do tipo árvores binárias que inserem e removem de forma inteligente para assegurar que a árvore permaneça aproximadamente balanceada. Cada nó neste tipo de árvore possui: cor, valor, filho esquerdo, filho direito e pai e.g. [LEISERSON 2012].

O objetivo deste tipo de estrutura é garantir que as operações básicas demorem O(lg n) no pior caso. Como auxílio para que isto ocorra, existem cinco propriedades:

- Todo nó da árvore ou é vermelho ou é preto.
- A raiz é preta.
- Toda folha é preta.
- Se um nó é vermelho, então ambos os filhos são pretos.
- Para todo nó, todos os caminhos do nó até as folhas descendentes contêm o mesmo número de nós pretos.

Um nó que satisfaz as propriedades listadas acima é denominado equilibrado, caso contrário é dito desequilibrado. Em uma árvore rubro-negra, todos os nós são equilibrados.

A árvore rubro-negra necessita de no máximo uma rotação para balancear na inserção e pode ter log(n) trocas de cores. Na remoção, faz no máximo uma rotação simples ou dupla.

As árvores rubro-negras tem a inserção e remoção mais rápidas, porém a reconstrução mais lenta, quando comparada com as árvores AVL e.g. [LEISERSON 2012]. Isso a torna atraente para estruturas de dados que eventualmente podem construir a árvore uma vez e não precisar que ela seja reconstruída, assim como em dicionários de linguagens (ou até mesmo em dicionários de programas, como opcodes de um montador ou interpretador).

Algumas implementações que utilizam árvores rubro-negras: a classe std::map da biblioteca STL (Standard Template Library) de C++, o escalonador Completely Fair Scheduler do Linux (introduzido a partir da versão 2.6.23) e algumas estruturas de dados em geometria computacional.

6. IMPLEMENTAÇÃO UTILIZANDO STM HASKELL

Haskell utiliza uma sintaxe elegante para representar um tipo de estrutura de dados, os tipos algébricos. Estes, aumentam a expressividade da linguagem, permitindo a definição de tipos mais complexos.

Para definir a estrutura da árvore rubro-negra em Haskell, foi criado um novo tipo algébrico, como mostrado a seguir:

```
data TRedBlackTree = TRedBlackTree { root :: TVar TreeNode, empty :: TVar TreeNode}
deriving(Eq)
```

O tipo algébrico **TRedBlackTree** possui uma raiz do tipo TVar TreeNode e um nodo vazio, de mesmo tipo.

Em seguida foi criado o tipo algébrico **TreeNode**:

```
data TreeNode = TreeNode {value :: TVar Integer, left :: TVar TreeNode, right :: TVar
TreeNode, parent :: TVar TreeNode, color::TVar Color }
deriving(Eq)
```

TreeNode possui um **Integer** para o valor(**value**) do nodo, um **TreeNode** esquerdo(**left**), um **TreeNode** direito(**right**), um **TreeNode** pai(**parent**) e uma cor(**color**) do

tipo **Color** (que pode ser ou vermelha ou preta e é também um tipo algébrico), sendo todos **TVars**.

As funções da árvore foram definidas de acordo com estes três tipos. Sua implementação ainda não foi terminada e portanto, ainda não foram feitos testes.

7. CONCLUSÕES

Como próximos passos do trabalho, será finalizada a implementação utlizando **STM Haskell** e serão feitos testes para verificar a escalabilidade, performance, dificuldade de implementação e uso.

A mesma árvore será implementada utilizando **IORef + AtomicModifyIORef** e então **MVars**. Serão realizados os mesmos testes realizados no uso de **STM Haskell**, para comparar os métodos de sincronização.

9. REFERÊNCIAS BIBLIOGRÁFICAS

Rigo, S., Centroducatt, P. and Baldassin. (2007) "**A. Memórias transacionais uma nova alternativa para programação concorrente**", In Anais do WSCAD, Porto Alegre.

Rajwar, R. and Goodman. J. (2003) "**Transactional execution: Toward reliable, high-performance multithreading**", In IEEE Micro, 23:117–125.

Kim, J. H., Cameron, H. and Graham, P. (2006) "**Lock-Free Red-Black Trees Using CAS**", In Concurrency and Computation: Practice and Experience, 1—40.

Park, H. and Park, K. (2001) "**Parallel algorithms for red--black trees**", In Theoretical Computer Science, 415—435.

Leiserson, C. E., Rivest, R. L., Stein, C. and Cormen, T.H. (2012) "**Introduction to algorithms**", In MIT Press, USA.

Marlow, S. (2013) "**Parallel and Concurrent Programming in Haskell**", In O'Reilly, USA.

Herlihy, M. and Shavit, N. (2012) "**The Art of Multiprocessor Programming**", In Elsevier, USA.

9. AGRADECIMENTOS

A primeira autora gostaria de agradecer à UFPEL pela bolsa PBIP,

concedida ao projeto.