Avaliação de árvores rubro negras em Haskell usando diferentes mecanismos de programação concorrente

Thaís S. Hübner[1], Rodrigo M. Duarte[1], André R. Du Bois[1]

{tshubner,rmduarte,dubois}@inf.ufpel.edu.br

LUPS – Laboratory of Ubiquitous and Parallel Systems

Computação - CDTec
[1]Universidade Federal de Pelotas (UFPEL)
Caixa Postal 354 · CEP 96001-970 · Pelotas, RS – Brasil

Resumo. *O objetivo deste artigo é propor a implementação de árvores rubro-negras em Haskell de forma concorrente, utilizando os diferentes mecanismos de sincronização disponíveis na linguagem. Serão analisados aspectos como escalabilidade, performance, dificuldade de implementação e uso. Além disso pretende-se disponibilizar as implementações como software livre para programadores Haskell.*

1. Introdução

Os processadores multicore estão cada vez mais presentes nos dias atuais. Apesar de antes serem usados somente em servidores de grande porte, hoje em dia eles estão presentes tanto em desktops como também em laptops, smartphones, etc.

Processadores multicore possuem dois ou mais núcleos completos de processamento no mesmo chip. A programação neste tipo de arquitetura deve ser de forma concorrente para que se possa usufruir de todo o desempenho que estas proporcionam. Desta forma, é preciso de mais de um fluxo de execução (thread) trabalhando para que se obtenha um bom resultado. Mesmo com todas as vantagens que este tipo de arquitetura traz, obter uma sincronização das threads no acesso à memória compartilhada do computador pode não ser tão simples. A solução usada mais comumente para a sincronização de threads são os bloqueios (locks), porém estes possuem dificuldade de utilização e são propensos a erros de programação e.g. deadlocks [RAJWAR 2003].

Haskell e.g. [MARLOW 2013] é uma linguagem de programação funcional que possui diferentes formas de sincronização de alto nível além de locks, como por exemplo memórias transacionais e.g. [RIGO 2007]. A proposta que este trabalho traz, é a implementação de árvores rubro-negras de forma concorrente em Haskell utilizando os diferentes métodos de sincronização existentes nesta linguagem para que seja analisado o desempenho em diferentes contextos de execução e a facilidade de implementação.

2. Memórias transacionais e Sincronização de Threads

2.1 Threads e Sincronização de Threads

Uma thread é um fluxo único de controle sequencial dentro de um programa que contém informações a respeito da execução do mesmo.

Uns dos problemas no uso de locks são: dificuldade de reuso de código (difícil composabilidade) e baixo desempenho.

O reuso de código é uma prática muito usada no desenvolvimente de software, já com o uso de locks isso se torna um problema.

Não há como garantir que dois trechos de códigos válidos, quando combinados, continuem válidos.

2.2 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).

3. Métodos de Sincronização em Haskell

Haskell possui diferentes abstrações para realizar o 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 aliadas a facilidade da linguagem funcional Haskell, fornecem uma excelente ferramenta para o desenvolvimento de programas paralelos e.g. [MARLOW 2013]. A seguir, apresentamos uma breve descrição dessas abstrações.

MVar: É um mecanismo de comunicação entre threads em Haskell. São variáveis especiais de sincronização que podem assumir um dos dois estados: cheia ou vazia. Na criação da variável pode ser definida o estado inicial através de duas funções: **`newMVar`** e **`newEmptyMvar`**. Possuem duas funções para manipulação: **`takeMVar`** e putMVar. **`takeMVar`** retorna o valor de **`MVar`** caso esteja cheia, ou bloqueia se estiver vazia. **`putMVar`** funciona de forma contrária, bloqueando no caso de a variável estar cheia e escrevendo se estiver vazia. **`MVars`** operam como os tradicionais mutex (locks).

IORef + atomicModifyIORef: Um **`IORef`** é uma referência a uma posição de memória e possui operações de criação, leitura e escrita (**`newIORef`**, **`readIORef`** e **`writeIORef`**). Apesar de por si só essas operações não garantirem segurança no acesso multithread das referências, Haskell fornece uma função que permite a modificação atômica da referência (**`atomicModifyIORef`**), desta forma uma **`IORef`** pode ser utilizada para implementar algoritmos não bloqueantes.

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

`STM Haskell` fornece um novo tipo de variável (**`TVar`**), que conta com as funções **`newTVar, readTVar e writeTVar`** que permitem que uma **`TVar`** seja criada, lida e escrita, respectivamente.

Essas 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. Dessa forma o programador se preocupa apenas em identificar as sessões críticas, sem ser necessário que se preocupe com problemas como sincronização, ordem de aquisição de locks, deadlocks, etc.

3. Árvores Rubro-Negras

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

O objetivo neste tipo de estrutura de dados é garantir que as operações básicas demorem O (lg n) no pior caso, como auxílio para tal, 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 acima é denominado equilibrado, caso contrário é dito desequilibrado. Em uma árvore rubro-negra todos os nós estão equilibrados.

A árvore rubro-negra necessita de no máximo uma rotação para balancear na inserção. 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 rápida, porém a reconstrução mais lenta quando comparadas com as árvores AVL e.g. [LEISERSON 2012]. Isso torna ela atraente para estruturas de dados que eventualmente podem construir a árvore uma vez e não precisar que ela seja reconstruída, assim como dicionários de linguagens (ou até mesmo 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.

4. Proposta de Implementação

Este trabalho propõe implementar as árvores rubro-negras na linguagem funcional Haskell utilizando os três métodos de sincronização citados no trabalho (**`MVar, IORef + atomicModifyIORef e STM Haskell`**). A implementação será feita

com o auxílio de um estudo sobre alguns algoritmos de árvores rubro-negras concorrentes como os apresentados em e.g.[KIM 2006], [PARK 2001].

Será feita a análise da escalabilidade, performance, dificuldade de implementação e uso com cada um dos mecanismos. Por fim disponibilizaremos as implementações como software livre.

5. Conclusões

Este texto apresenta uma proposta inicial de estudo das diferentes formas de sincronização existentes em Haskell e sua aplicação no desenvolvimento de uma estrutura de dados do tipo árvores rubro-negras. Os próximos passos do trabalho incluem um estudo mais aprofundado das abstrações de programação e dos algoritmos para árvores rubro-negras. Quando terminada a fase de estudo, serão desenvolvidos testes que serão executados nas máquinas disponíveis no laboratório LUPS (Laboratory of Ubiquitous and Parallel Systems/Laboratório de Sistemas Paralelos e Distribuídos) da Computação/CDTec/UFPEL.

6. Referências

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.