IMPLEMENTAÇÃO DE ESTRUTURAS HASH CONCORRENTES EM HASKELL

RODRIGO M. DUARTE; MAURÍCIO L. PILLA; RENATA H. S. REISER;
ANDRÉ R. DU BOIS

UFPEL – UNIVERSIDADE FEDERAL DE PELOTAS
LUPS – Laboratory of Ubiquitous and Parallel Systems
{rmduarte, pilla, reiser, dubois}@inf.ufpel.edu.br

1.INTRODUÇÃO

Algoritmos de tabela *hash* são naturalmente paralelizáveis, o motivo disso se dá ao fato que o acesso aos dados da tabela são suscetíveis de serem disjuntos. Porém a implementação de um algoritmo que explore de forma otimizada esta característica esta longe de ser trivial (HERLIHY, 2012). Implementar um algoritmo usando um *lock* global é simples porém ineficiente, e proteger cada posição da tabela com um *lock* é eficiente porém torna o crescimento da tabela complexo.

A linguagem de programação Haskell é uma linguagem funcional pura de alto nível que apresenta várias abstrações de sincronização para a programação concorrente, e.g., variáveis de sincronização (*MVars*), memórias transacionais e acesso a instruções de baixo nível para sincronização – CAS (*compare and swap*) (MARLOW, 2013).

Este trabalho apresenta a implementação de três diferentes algoritmos de tabela *hash* concorrente em Haskell, utilizando diferentes métodos de sincronismo. Também é feita a comparação do desempenho entre elas e os resultados mostram que, a implementação utilizando a técnica de granularidade fina usando memórias transacionais, apresentou os melhores resultados.

2. METODOLOGIA

Tipos de algoritmos de hash concorrente

Os algoritmos estudados para implementação (HERLIHY, 2012), possuem três operações básicas de acesso a tabela *hash* que são inserção, consulta e remoção. Todos os algoritmos utilizam endereçamento fechado, i.e., cada posição da tabela pode conter um conjunto de itens geralmente implementado usando uma lista encadeada. O crescimento da tabela *hash* depende do número de inserções, se o número de inserções alcança certo limite pré-estabelecido, o tamanho da tabela deverá dobrar (LEISERSON, 2012). Cada algoritmo possui uma forma diferente de tratar o crescimento da tabela e esta é a operação de maior complexidade na implementação dos algoritmos.

Hash com *lock* global:

Neste algoritmo um único *lock* protege a tabela *hash* inteira. Cada função adquire o *lock*, realiza sua operação e depois o libera. Na necessidade de crescimento da tabela, como a função de inserção já possui o *lock*, ela pode simplesmente aumentar a tabela, sem se preocupar com conflitos. Este algoritmo é extremamente simples de ser implementado, porém não explora nenhum paralelismo devido ao gargalo gerado pelo *lock*.

***Hash* de blocos:**

A implementação deste algoritmo utiliza dois *arrays* distintos, um para as entradas da tabela *hash* e outro de *locks*. Quando a tabela *hash* é iniciada, ambos os *arrays* possuem o mesmo tamanho. Ao realizar um crescimento na tabela, dobra-se o *array* de entradas da *hash* mas se mantem o tamanho do *array* de *locks*, assim cada *lock* passa a proteger duas posições da tabela. Como crescimentos na tabela *hash* são raros, existem dois motivos para evitar o crescimento do array de *locks* que são (HERLIHY,2012):

- Associar um *lock* para cada *slot* da tabela ocupa muito espaço, especialmente quando as tabelas são grandes e a contenção é baixa.
- Aumentar o array de *slots* é simples, porém aumentar o array de *locks* (quando estão em uso), é mais complicado.

***Hash* de granularidade fina:**

No algoritmo usando granularidade fina, a tabela *hash* possui uma *flag* que serve para indicar se está em um processo de crescimento ou não, e tanto o *array* de *locks* quanto o de entradas é duplicado. Esta *flag* possui um valor booleano e é alterada atomicamente pela *thread* que está realizando um crescimento da tabela. Quando o valor desta *flag* é verdadeiro, isso indica que alguma *thread* está realizando um crescimento da tabela, informando as outras *threads* que os *locks* não podem ser adquiridos porque não são mais validos, e que devem esperar até que a *flag* torne-se falsa para poderem prosseguir.

Apesar de parecer simples, este processo envolve certa complexidade, pois durante o processo de crescimento, temos de verificar se todas as *threads* terminaram suas operações, caso contrário não se pode prosseguir com a operação de crescimento.

Métodos de sincronização em Haskell

A linguagem funcional Haskell fornece alguns modelos para sincronismo entre threads, os estudados aqui são *MVar*, *IORef + atomicModifyIORef* e *STM Haskell*, que são modelos de sincronismo já utilizados em outros trabalhos envolvendo estrutura de dados em Haskell (SULZMANN2009).

MVar:

É o mecanismo básico de comunicação entre *threads* de Haskell. Uma *MVar* é uma espécie de variável que pode assumir duas situações: cheia ou vazia. (MARLOW, 2013). *MVars* operam como os tradicionais *mutex*. Neste trabalho usaremos uma *MVar* como um *lock* para realizar a sincronização das *threads* no acesso a *hash*.

***IORef + atomicModifyIORef*:**

Este método pode ser comparado ao CAS (*compare and swap*) de outras linguagens de programação. Um *IORef* é uma referência a uma posição de memória. 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.

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*. *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.

3. RESULTADOS E DISCUSSÃO

Foram implementadas tabelas *hash* usando as três primitivas apresentadas usando os algoritmos estudados. Entre as implementações foram duas de *lock* global e granularidade fina, usando *MVar* e *TVar* e uma em bloco usando *MVars*.

Os testes foram realizados em uma máquina core i7 de 3.0Ghz com 8 cores (4 físicos e 4 lógicos) e 8Gb de RAM. Foram feitas 30 execuções de cada implementação, realizando 1 milhão de operações sobre a tabela, sendo 10% inserções, 10% deleções e 80% consultas, conforme estatísticas em (HERLIHY, 2012). Os gráficos das Figura 1 e 2 mostra os resultados dos testes.

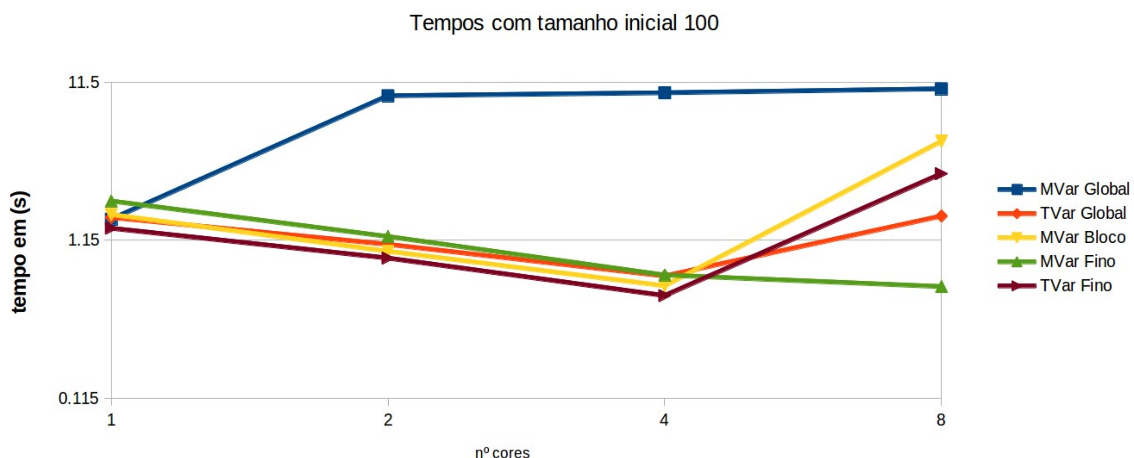

Figura 1

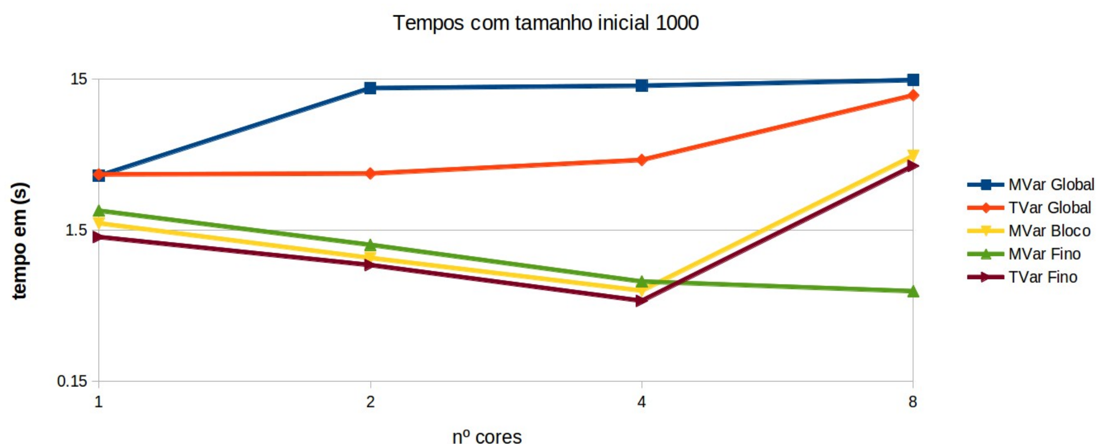

Figura 2

Podemos perceber que conforme o esperado, a tabela usando *lock* global com *MVar* teve perda de desempenho conforme se aumentou o número de *threads* utilizadas, porém o mesmo não se observa utilizando *TVars*, ao contrário do esperado, obtiveram desempenho, porém quando se aumentou o tamanho inicial da tabela de 10 para 10 mil elementos, ambas perderam desempenho devido ao número de operações sobre a tabela ter aumentado muito.

Nas implementações de *hash* em bloco e fina obteve-se desempenho conforme se aumentou o número de *threads.* Isso se deve ao fato de neste tipo de algoritmo podemos ter mais de uma *thread* acessando a tabela ao mesmo tempo, conseguindo assim maior desempenho. A diferença de tempo vista entre a implementação usando blocos da usando granularidade fina se da devido ao fato da maior complexidade para a aquisição dos locks na tabela com granularidade fina.

A implementação de granularidade fina usando *TVar* obteve maior desempenho se comparada a todas as outras, isso se dá devido a baixa quantidade de conflitos existentes na tabela.

4. CONCLUSÕES

A implementação usando granularidade fina com *TVar* foi a que obteve maior desempenho. Isso se deve ao fato de que em transações o nível de contenção é menor. Outra coisa importante ao se utilizar transações é que o sistema transacional garante todo o sincronismo de forma automática, eximindo o programador desta tarefa, tornando a programação paralela mais simples, evitando assim problemas comuns como a ocorrência de *deadlocks.*

Como trabalhos futuros, pretendesse implementar uma biblioteca para Haskell afim de fornecer a implementação para uso da comunidade em geral, visto que não existem implementações deste tipo de estrutura de dados em Haskell.

5. REFERÊNCIAS BIBLIOGRÁFICAS

HERLIHY, M. e Shavit, N. ***The Art of Multiprocessor Programming***, USA: Elsevier, 2012.

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

LEISERSON, C. E., Rivest, R. L., Stein, C., e Cormen, T.H. ***Introduction to algorithms***, USA: MIT Press, 2012.

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

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

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