Comparando o Desempenho de Implementações de Tabelas *Hash* Concorrentes em Haskell

Rodrigo M. Duarte[1], André R. Du Bois, Maurício L. Pilla, Renata Hax Sander Reiser

[1]Laboratory of Ubiquitous and Parallel Systems – CDTec
Universidade Federal de Pelotas
Pelotas – RS – Brasil

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

Abstract. *An algorithm which explores performance on concurrent hash tables is far from being trivial. This paper presents seven hash table Haskell implementations, which include low level synchronism models to high level ones such as transactional memories. The result of the comparison between the algorithms showed that the implementation using the STM Haskell transactional memory library presented the best performance.*

Resumo. *Implementar um algoritmo de tabela hash concorrente que extraia desempenho está longe de ser trivial. Neste artigo apresentamos sete diferentes implementações de tabelas hash em Haskell, explorando desde modelos de sincronização de baixo nível até os de mais alta abstração como memórias transacionais. Nos testes realizados a implementação usando a biblioteca STM Haskell de memória transacional foi a que apresentou melhor desempenho.*

1. Introdução

Estrutura de dados do tipo tabela *hash* possuem concorrência natural já que o acesso aos dados da tabela são suscetíveis de serem independentes. Porém, implementar uma tabela *hash* concorrente que possua ganho de desempenho não é algo trivial [Herlihy and Shavit 2012]. Problemas como o tratamento de conflitos (duas ou mais *threads* acessando uma mesma posição da tabela) e principalmente o crescimento da tabela são complexos e de difícil solução. Por exemplo, sincronizar a tabela com *lock* global é simples porém ineficiente, enquanto proteger cada posição da *hash* com um *lock* é eficiente porém possui alta complexidade.

A linguagem de programação Haskell é uma linguagem funcional pura de alto nível que apresenta várias abstrações de sincronização para programação concorrente, e.g., variáveis de sincronização (`MVars` [Marlow 2013]), memórias transacionais [Harris et al. 2008] e acesso a instruções de baixo nível para sincronização – CAS (*compare and swap*). Mesmo com todos estes métodos de sincronização disponíveis, a inexistência de tabelas *hash* concorrentes para Haskell ainda é um problema [Newton et al. 2011].

Este trabalho apresenta a implementação em Haskell de sete tipos de tabela concorrente com métodos de sincronismo distintos, usando como base quatro diferentes algoritmos de *hash*. Também é feita a comparação do desempenho entre elas e os resultados

mostram que a implementação utilizando memórias transacionais foi a que apresentou melhores resultados.

O artigo está organizado da seguinte forma: a Seção 2 apresenta os diferentes métodos de sincronismo presentes em Haskell. A Seção 3 apresenta os diferentes algoritmos utilizados e suas implementações. Na Seção 4 são descritos os resultados alcançados e finalmente nas Seções 5 e 6, são apresentados os trabalhos relacionados e as conclusões.

2. Métodos de sincronização em Haskell

A linguagem funcional Haskell fornece alguns modelos para sincronismo entre *threads*, sendo que os utilizados neste trabalho são `MVar`, `IORef + atomicModifyIORef` e STM Haskell [Sulzmann et al. 2009]. Os mecanismos são brevemente descritos a seguir.

2.1. 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. `MVars` são criadas a partir da função `newMVar :: a -> IO(MVar a)` onde cria uma `MVar` já com um valor inicial. As funções que manipulam `MVars` são a `takeMVar :: MVar a -> IO a` e `putMVar :: MVar a -> a -> IO()`. `takeMVar` retorna o valor da `MVar` se esta estiver cheia, ou espera (bloqueia) se estiver vazia. Já `putMVar` opera de forma contrária, bloqueia se a `MVar` estiver cheia e escreve se estiver vazia [Marlow 2013]. Assim, `MVars` operam como os tradicionais *mutex*. Neste trabalho usamos uma `MVar` como um *lock* para realizar a sincronização das *threads* no acesso à *hash*.

2.2. IORef + atomicModifyIORef:

Este método pode ser comparado ao CAS (*Compare And Swap*). Um `IORef` é uma referência a uma posição de memória que possui as seguintes operações: `newIORef :: a -> IORef a`, `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 (`atomicModifyIORefCAS`) que utiliza uma instrução em *hardware* atômica para modificação da referência [Data.CAS 2015].

2.3. STM Haskell (Software Transactional Memory):

Software transactional Memory (STM) é um modelo recente de sincronização entre *threads* que simplifica a programação concorrente, permitindo que operações possam ser compostas em uma simples operação atômica. A ideia principal 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 :: a -> STM(TVar a)`, e modificada pelas primitivas `readTVar :: TVar a -> STM a` e `writeTVar :: TVar a -> a -> STM ()`. Estas primitivas só podem ser executadas dentro de uma chamada a `atomically :: STM a -> IO a`. 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. Tipos de algoritmos de hash concorrente

Os algoritmos utilizados neste trabalho fornecem três operações básicas de acesso a tabela que são inserção(*insert*), consulta(*contains*) e remoção(*delete*). 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 depende do número de inserções, se estas alcançam um certo limite pré estabelecido, o tamanho da tabela deverá dobrar [Leiserson et al. 2001]. Cada algoritmo possui uma forma diferente de tratar o crescimento da tabela e esta é a operação de maior complexidade nas implementações.

3.1. 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 *insert* 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 serial gerado pelo *lock*. Para a implementação deste algoritmo usaram-se duas abordagens. Uma com `MVar` como *lock* e outra usando `TVar`. A estrutura da *hash* usando `MVar` foi implementada da seguinte forma:

```
1  type Buckets = Array Int [Int]
2  data HTable = TH
3  {
4    buckets :: MVar Buckets,
5    n_elements :: IORef Int,
6    len_tab :: IORef Int
7  }
```

Quando o limite é alcançado, o *array* de entradas (*buckets*) é duplicado, e armazenado dentro da `MVar` que sincroniza o acesso a tabela. Os outros atributos da tabela, i.e. número de elementos(`n_elements`) e tamanho(`len_tab`), podem ser armazenados em posições de memória simples (`IORef`), pois só são modificados por *threads* que possuem o *lock* da tabela (`MVar buckets`).

Na implementação usando STM Haskell, todos os atributos da tabela *hash* devem ser guardados em `TVars` para garantir a consistência desses dados quando acessados por transações concorrentes:

```
1  data HTable = TH
2  {
3    buckets :: TVar Buckets,
4    n_elements :: TVar Int,
5    len_tab :: TVar Int
6  }
```

O uso de `TVars` faz com que o sistema de tipos garanta que estes dados serão acessados apenas em blocos atômicos (primitiva `atomically`).

3.2. Hash de blocos

A implementação deste algoritmo utiliza dois *arrays* distintos, um para os *buckets* 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 *buckets* mas se mantém o tamanho do de *locks*, assim cada *lock* passa a proteger 2^n posições da tabela, sendo n o número de vezes que a tabela dobrou. Como crescimentos na tabela *hash* são raros, existem dois motivos para evitar o crescimento do *array* de *locks* que são [Herlihy and Shavit 2012]:

- Associar um *lock* para cada *bucket* da tabela ocupa muito espaço, especialmente quando as tabelas são grandes e a contenção é baixa;
- Aumentar o *array* de *buckets* é simples, porém aumentar o *array* de *locks* (quando estão em uso), é mais complicado pois é difícil de sincronizar as várias *threads* que estão acessando o *array* de *locks* ao mesmo tempo.

Nesta implementação os *locks* são representados por `MVars` contendo valores booleanos.

```
1  type Buckets = Array Int (IORef [Int])
2  type Locks = Array Int (MVar Bool)
3  data HTable = TH
4  {
5    buckets :: IORef Buckets,
6    lock :: Locks,
7    n_elemens :: IORef Int,
8    len_tab :: IORef Int,
9    len_lock :: Int
10 }
```

Quando surge a necessidade de duplicar o tamanho da tabela, a função de inserção começa a adquirir todos os *locks* a partir do inicio do *array*, seguindo sempre esta sequencia para evitar *deadlocks*. Após a duplicação da tabela todos os *locks* são liberados. Como o atributo `len_tab` só é modificado no processo de duplicação, este pode ficar armazenado em uma `IORef`. O atributo `n_elements` é alterado usando CAS, uma vez que uma *thread* consegue realizar uma alteração na tabela. Como o *array* de *locks* tem seu tamanho fixo, o atributo `len_lock` pode ser uma constante.

3.3. Hash de granularidade fina

Este algoritmo também utiliza dois *arrays*, um de *locks* e outro de *buckets*. Porém aqui, durante o processo de duplicação da tabela, ambos os *arrays* são duplicados. A tabela possui uma *flag* que serve para indicar se está em processo de crescimento ou não. Esta *flag* indica as outras *threads* se os *locks* são válidos ou não.

A estrutura da implementação é como o descrito abaixo:

```
1  type Buckets = Array Int (MVar [Int])
2  type Locks = IORef Buckets
3  data ThId = ThId ThreadId | Null
4
5  data HTable = TH
6      {
7      buckets   :: Locks,
8      n_elemens :: IORef Int,
9      len_tab   :: IORef Int,
10     onGrow    :: IORef (ThId, Bool)
11     }
```

Nesta implementação foi usado um único *array* de `MVars` para representar os *buckets*. Isto porque aqui ambos os *arrays* devem possuir o mesmo tamanho, não havendo a necessidade de *arrays* distintos.

Como o *array* de *locks* pode ser modificado, nesta implementação ele deve ser armazenado em uma referência `IORef`. A modificação dessa referência é controlada pela *flag* `onGrow` que é setada quando o processo de duplicação começar, informando para as outras *threads* para não usarem o *array* de *locks*. Esta *flag* possui dois campos (`ThId`,`Bool`), O campo `ThId` serve para que a *thread* que vai realizar o crescimento saiba se foi ela ou não que conseguiu setar o valor booleano desta *flag*.

Esta operação de alteração da *flag* fica a cargo da função `resize`:

```
1   resize :: IORef Buckets -> Int -> IORef Int -> IORef (ThId, Bool) -> ThreadId
        -> IO Bool
2   resize buckets old_lenTab len_tab flag tId = do
3       (t,i) <- readIORef flag
4       if i then do
5               return False
6           else do
7               old_array <- readIORef buckets
8               ok <- atomCAS flag (t,i) (ThId tId, True)
9               if ok then do
10                  lenTab <- readIORef len_tab
11                  if(old_lenTam == lenTab) then do
12                      (...) -- operação de duplicação
13                      writeIORef flag (Null,False)
14                      return True
15                      else do
16                          writeIORef flag (Null,False)
17                          return False
18                  else do
19                      return False
```

Para a duplicação é criado um novo *array* de `MVars` que recebe os dados do *array* antigo e depois é armazenado no `IORef` *buckets*. Apos esta operação a *flag* é tornada falsa e informa às outras *threads* que podem prosseguir.

Na implementação usando STM, não é necessário a *flag* para controle do crescimento da tabela. O *array* de *buckets* é armazenado em uma `TVar` e a modificação desta variável causará um conflito em outras transações que leram esta variável, e.g., para acessar um *bucket*.

```
1   type Buckets = Array Int (TVar [Int])
2   data HTable = TH
3       {
4       buckets   :: TVar Buckets,
5       n_elemens :: TVar Int,
6       len_tab   :: TVar Int
7       }
```

3.4. Tabela hash não bloqueante

A principal dificuldade na implementação de uma tabela *hash* não bloqueante é a realização do crescimento. Não basta fazer com que o *array* de *buckets* da tabela seja *lock-free*, pois na necessidade de realizar o crescimento da tabela, a mudança dos dados da tabela antiga para a nova deve ser realizada de forma atômica para evitar perda ou inconsistência de dados. Infelizmente as instruções para a realização de CAS, 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 [Shalev and Shavit 2006] utiliza uma técnica onde ao invés de mover os itens através dos *buckets*, movem-se os *buckets* através dos itens. Neste algoritmo, todos os dados são mantidos em uma lista encadeada *lock-free* e cada *bucket* é simplesmente uma referência para uma determinada posição na lista, como pode ser visto na Figura 1.

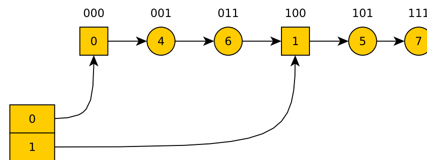


Figura 1. Esquema da *hash lock-free*

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 *buckets* diferentes, não havendo a necessidade de movê-los para uma nova tabela. Porém aqui, a capacidade da tabela *hash* deve ser sempre uma potência de dois, para que se possa dividir a tabela de forma que a distribuição dos dados seja coerente. Para cada *bucket* 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 *bucket*, o mesmo não tenha que apontar para uma posição inválida e ou para um elemento de um outro *bucket*.

Para a implementação da lista encadeada usou-se um esquema como visto em [Sulzmann et al. 2009]. Nesta lista encadeada, o maior problema se encontra na remoção de elementos da lista sem que afete a estrutura global da mesma. Remover os elementos diretamente da lista pode causar uma inconsistência em sua estrutura, levando a quebra do encadeamento. Para evitar este problema usa-se um esquema de *lazy deletion*. onde um elemento nunca é deletado diretamente da lista, primeiramente o elemento é removido da lista logicamente ficando a remoção física postergada para a próxima operação sobre a lista, como visto em [Sulzmann et al. 2009].

A estrutura da lista encadeada é como segue:

```
1  data List = Node { val::Word, next::IORef (List)}
2                   | DelNode {next::IORef (List)}
3                   | Null
4                   | Guard {val::Word, next::IORef (List)}
5  type  ListHandle = IORef List
```

E a estrutura para a tabela hash:

```
1     data Slot = Lista {list::ListHandle} | Nil
2         deriving(Eq)
3     type Buckets = Array Int (IORef Slot)
4     data HTable = TH
5         {
6         buckets :: IORef Buckets,
7         n_elemens :: IORef Int,
```

```
8         len_tab :: IORef Int
9         }
```

Quando uma tabela *hash* é criada, ela tem somente o *bucket* zero inicializado, ou seja, o *bucket* zero aponta para a guarda zero que é o *head* da lista.

```
1  newHash :: Int -> IO (HTable)
2  newHash tam = do
3      x <- replicateM tam (newIORef Nil)
4      let arrayHash = listArray (0,tam-1) x
5      let first = arrayHash ! 0
6      headList <- newList
7      writeIORef first Lista {list=headList}
8      buckets <- newIORef arrayHash
9      n_elemens <- newIORef 0
10     len_tab <- newIORef tam
11     return (TH buckets n_elemens len_tab)
```

Os *buckets* restantes vão sendo inicializados conforme a requisição da utilização dos mesmos. Para cada *bucket* inicializado, uma nova guarda é inserida na lista encadeada e o *bucket* correspondente passa a apontar para a mesma. Esta inicialização é realizada através de um método chamado *split-ordered keys*[Shalev and Shavit 2006], onde cada *bucket* é inicializado através de chamadas recursivas até encontrar um *bucket* que já esteja inicializado. Ao encontrar tal *bucket*, este insere a guarda referente na lista encadeada e passa esta guarda como referência para a inserção do próximo *bucket*.

```
1  initializeGuard :: Buckets -> Int -> Int -> IO ListHandle
2  initializeGuard slots lenTab guardVal = do
3      x <- readIORef $ slots ! searchParent lenTab guardVal
4      case x of
5          Lista {list = lista} -> do
6              inverseVal <- reverseBits guardVal
7              a <- addGuardToList lista   inverseVal
8              atomicWriteIORef (slots ! guardVal) (Lista {list=a})
9              return a
10         Nil -> do
11             newGuard <- initializeGuard slots lenTab $ searchParent lenTab
                   guardVal
12             inverseVal <- reverseBits guardVal
13             a <- addGuardToList newGuard inverseVal
14             atomicWriteIORef (slots ! guardVal) (Lista {list=a})
15             return a
```

Outro item importante da implementação é a utilização de FFI (Haskell Foreing Function Interface) [O'Sullivan et al. 2008], para a função reverseBits.

```
1  foreign import ccall safe reverseBits :: Int -> IO Word
```

Esta chamada a uma função de uma linguagem de mais baixo nível tornou-se necessária devido a não se ter achado uma implementação de reversão de bits em Haskell que fosse eficiente o suficiente. Através de análise do tempo de execução, observou-se que algumas das funções chegavam a ocupar 66% do tempo total de execução do código, reduzindo em muito o desempenho da tabela.

A implementação deste algoritmo usando STM foi a simples troca do tipo das variáveis `IORef` para `TVar`, e alguma redução no algoritmo de sincronização, visto que esta é garantida pelo sistema transacional.

```
1  data List = Node { val::Word, next::TVar (List)}
2                 | DelNode {next::TVar (List)}
```

```
3                         | Null
4                         | Guard {val::Word, next::TVar (List)}
5  type   ListHandle = TVar List
```

```
1  data Slot = Lista {list::ListHandle} | Nil
2  type Buckets = Array Int (TVar Slot)
3
4  data HTable = TH
5      {
6      buckets   :: TVar Buckets,
7      n_elemens :: TVar Int,
8      len_tab   :: TVar Int
9      }
```

4. Resultados

Para a realização dos testes foi utilizado um computador com processador *core* i7 de 8 *cores* (4 físicos + 4 *Hyperthreading*), com 8Gb Ram e Sistema Operacional Ubuntu 14.04 64Bits. O compilador Haskell utilizado foi o ghc 7.6.3 com STM 2.4.2. Foram realizados 30 testes com cada implementação, variando o número de *threads* de 1 a 16 em potências de 2, sendo que até 8, uma *thread* por *core*. Foram realizadas 1 milhão de operações na tabela, sendo 10% inserções, 10% deleções e 80% consultas, conforme estatísticas de uso [Herlihy and Shavit 2012]. Também foram executados testes realizando 80% inserções, 10% deleções e 10% consultas para testar a sobrecarga de alterações na tabela. Os testes realizados foram sobre as seguintes implementações:

- *Global lock MVar*: implementação de *hash* usando *lock* global, Seção 3.1;
- *Global using STM*: mesma implementação anterior usando memórias transacionais. Seção 3.1;
- *Block MVar*: tabela usando algoritmo de *lock* em bloco. Seção 3.2;
- *Fine using MVar*: algoritmo de lock fino. Seção 3.3;
- *Fine using STM*: implementação de grão fino usando memórias transacionais. Seção 3.3;
- *CAS*: implementação da *hash lock-free* usando `IORef`. Seção 3.4;
- *CAS using STM*: implementação onde todas as `IORef` foram substituídas por `TVAr`. Seção 3.4.

A Figura 2 apresenta os resultados dos tempos de execução das aplicações, onde o eixo das abscissas representa o número de cores utilizado e o eixo das ordenadas o tempo de execução em segundos em escala logarítmica, sendo as Figuras 2(a) os tempos de execução realizando 80% consultas e a Figura 2(b) considerando 80% de inserções. As Figuras 3(a) e 3(b) representam os gráficos de escalabilidade das implementações.

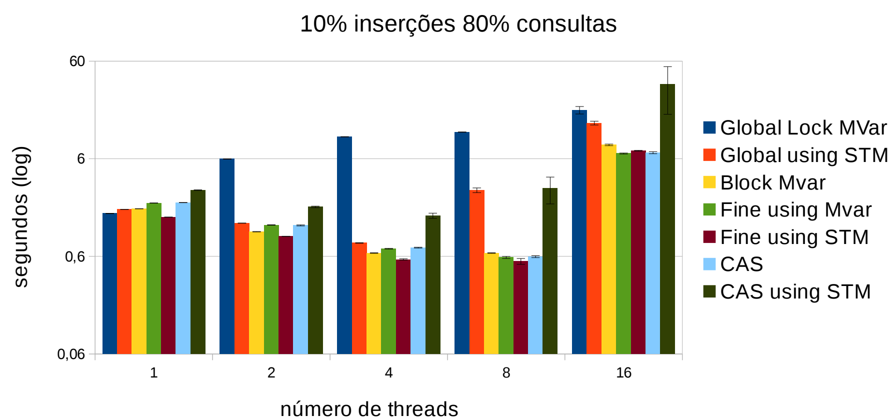
Como esperado a implementação *Global lock MVar* que usa a técnica de *lock global* com mais de uma *thread* apresentou perda de desempenho devido ao gargalo serial gerado pelo único *lock* que protege a tabela. Já a implementação *Global using STM*, quando a carga de trabalho foi pequena, caso em que 10% eram inserções, a aplicação chegou a ganhar certo desempenho até 4 *threads*. Esse desempenho está relacionado ao fato de que transações que não modificam o estado do programa não geram conflitos. Assim como 80% das operações neste caso eram consultas, as mesmas não alteram o estado da tabela e assim conseguiu-se um pouco de desempenho. Quando aumentamos a carga de trabalho para 80% de inserções o desempenho desta aplicação ficou comprometido,

isso devido ao número de conflitos ter aumentado. O pior caso foi com 16 *threads* em que os valores excederam os 60 segundos.

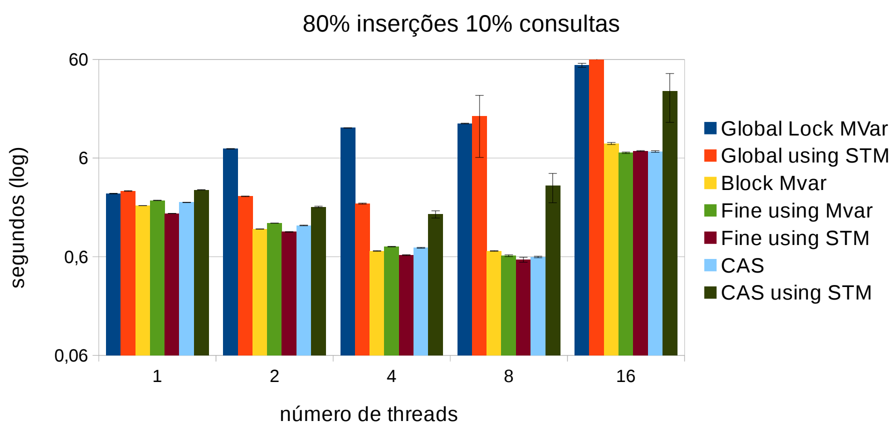
Pode-se observar pelos gráficos das Figuras 2(a) e (b) que, conforme aumentou-se o número de *cores*, com exceção da versão utilizando *lock* global(*Global Lock MVar*), todas as implementações apresentaram ganho de desempenho até quatro *cores*. Ao chegar em oito *cores* as únicas implementações que conseguiram manter um tempo de execução baixo foram *Block MVar*, *Fine using MVar*, *Fine using STM* e *CAS*. Nestas implementações ao aumentarmos o número de inserções na tabela, as mesmas não sofreram uma alteração que fosse a se considerar, ficando todas com o tempo de execução parecidos quando a carga de modificações na tabela era pequena.

Na implementação *Block MVar*, o tempo de execução da aplicação chegou a ser mais baixo que o da técnica usando *Fine using MVar*. Essa diferença de tempo se dá principalmente pelo fato de o algoritmo de aquisição dos *locks* e de duplicação da tabela serem mais simples do que os da *Fine using MVar*.

As implementações *Block MVar*, *Fine using MVar* e *CAS* apresentaram tempos de execução parecidos, isso se dá pelo fato de que o *overhead* gerado pela complexidade



(a) 10% inserções.



(b) 80% inserções.

Figura 2. Tempos de execução para todas as implementações de tabela *hash*

do algoritmo para a correta sincronização na aplicação usando *CAS* e o *overhead* para a duplicação da tabela nas aplicações *Block MVar* e *Fine using MVar* terem sido parecidos. Já na implementação *CAS using STM*, o baixo desempenho está principalmente na ocorrência de falsos conflitos que são gerados na lista encadeada, o que acabou levando ao baixo desempenho da aplicação, este problema pode ser visto em detalhes na Seção 3.3 em [Sönmez et al. 2007].

A aplicação que apresentou o menor tempo de execução em ambos os testes foi a *Fine using STM*. O algoritmo usado nesta aplicação (Seção 3.3), possuía características que permitiam que fosse implementado com características que favoreciam a utilização de STM, se mostrando o de mais simples implementação e de melhor desempenho.

Conforme pode-se observar pelos gráficos das Figuras 3(a) e 3(b), as implementações que se apresentaram mais escaláveis foram *Block MVar*, *Fine using MVar*, *Fine using STM* e *CAS*.

Com exceção das aplicações *Global lock MVar*,*Global using STM* e *CAS using STM*, os testes ficaram com desvio padrão igual ou abaixo de 0.03, o que demostra que a média dos valores de tempo se encontram dentro dos valores encontrados.

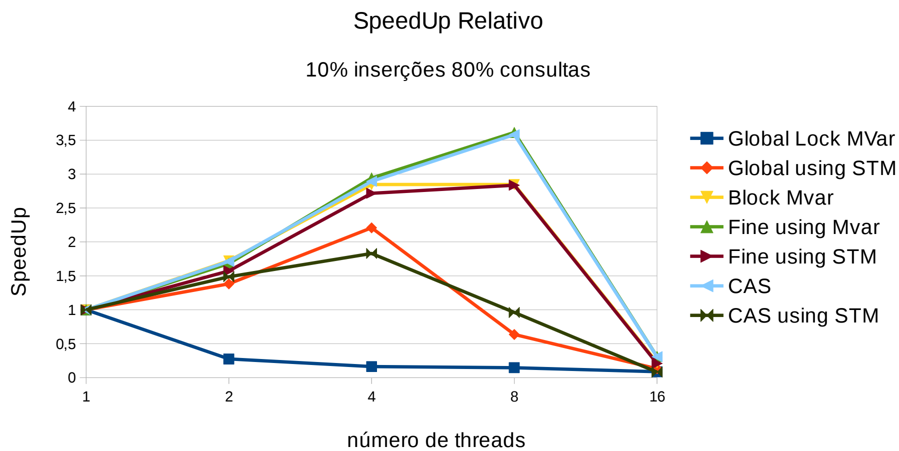
5. Trabalhos Relacionados

No trabalho desenvolvido em [Newton et al. 2011], foi exposta a inexistência de tabelas *hash* concorrentes para Haskell. A alternativa utilizada neste trabalho foi usar *locks* de granularidade grossa sobre estruturas tipo Data.Map, que são estruturas de dados presentes em Haskell mas que não são concorrente. Para o trabalho em questão o *overhead* imposto por este *lock* global não foi impactante. Isto porque a estrutura armazenava um quantidade mínima de dados. Neste trabalho, tentou-se reduzir ao máximo a granularidade dos *lock* ao ponto de explorar-se a não utilização deles, implementando uma tabela *lock free*.

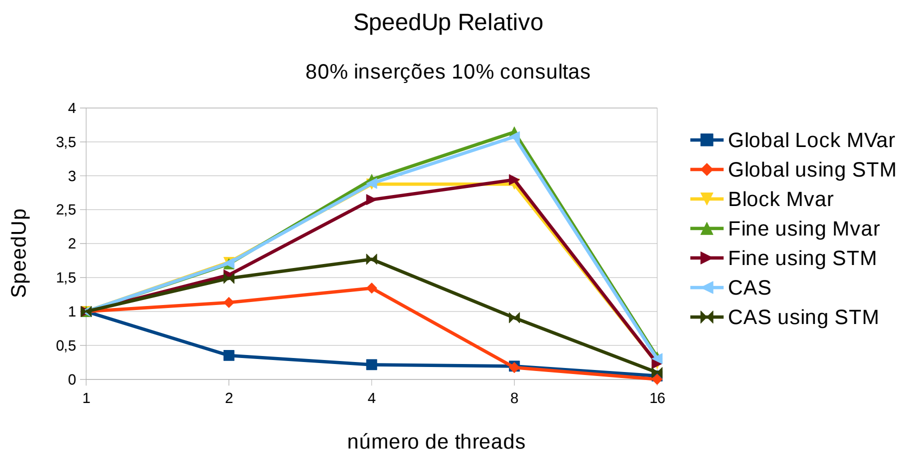
Em [Sulzmann et al. 2009] são implementadas listas encadeadas na abordagem de diferentes algoritmos concorrentes e usando as mesmas primitivas apresentadas neste trabalho. Neste é apresentado que a implementação usando `IORef` com `atomicModifyIORef` foi a que mostrou melhores resultados, mas que a utilizando STM é a mais atrativa pela facilidade de implementação e pela garantia de corretude, visto que as implementações usando `IORef` são complexas e dispendem de testes críticos para provar sua corretude. A implementação da lista encadeada não bloqueante da Seção 3.4 usou como base este algoritmo mas com modificações, pois era necessário uma lista ordenada e com a inclusão das guardas para as referências aos *buckets*, o que resultou em uma modificação considerável do algoritmo original.

6. Conclusões e trabalhos futuros

Neste trabalho, foram exploradas diferentes alternativas para a implementação de tabelas *hash* em Haskell. As implementações de tabela *hash Block MVar*, *Fine using MVar*, *Fine using STM* e *CAS* foram as que apresentaram melhor desempenho até 8 *threads* isso se dá pelo fato destes algoritmos explorarem melhor a concorrência dos recursos disponíveis. Na implementação *Block MVar* ganhamos desempenho porque o gargalo gerado pelo baixo número de *locks* protegendo a tabela foi compensado pela baixa complexidade na duplicação da tabela. A aplicação *Fine using MVar* apresentou comportamento



(a) 10% inserções.



(b) 80% inserções.

Figura 3. Speedup de todas as implementações.

distinto, o baixo gargalo devido a granularidade fina nos *locks* foi descompensada pela complexidade na duplicação da tabela, o que acabou levando a um tempo de execução parecido ao da implementação *Block MVar*.

A implementação *CAS*, devido ao custo imposto pelo algoritmo de sincronização levou a um desempenho parecido com as de *Block MVar* e *Fine using MVar*. Esta complexidade se torna um item a ser ressaltado, pois implementações mais simples como a de *Block MVar* acabaram apresentando menor tempo de execução. Cabe salientar que a aplicação *CAS* foi a que se apresentou melhor escalabilidade. Esta aplicação em um máquina com mais recursos poderá apresentar melhores resultados.

O resultado que mais se destacou foi o da implementação usando a técnica *Fine using STM* que até 8 *threads* apresentou ganho de desempenho e menor tempo de execução, apesar de não ter a melhor escalabilidade. Com este resultado aliado a facilidade de implementação desta aplicação, este algoritmo é um a ser relevado a sua utilização.

Como trabalhos futuros pretende-se expandir a implementação para tipos polimórficos, visto que em nossos testes só utilizamos tipos inteiros. Mais testes variando o tamanho inicial da tabela e também utilizando a mesma sobre diferentes condições serão considerados para os próximos trabalhos. Também pretende-se publicar as implementações que apresentarem melhor desempenho como uma biblioteca de *hash* concorrente para Haskell, visto que a linguagem não possui tabelas *hash* concorrentes [Newton et al. 2011].

Referências

Data.CAS (2015). http://hackage.haskell.org/package/IORefCAS-0.1.0.1/docs/src/Data-CAS.html.

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

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

Leiserson, C. E., Rivest, R. L., Stein, C., and Cormen, T. H. (2001). *Introduction to algorithms*. The MIT press.

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

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

O'Sullivan, B., Goerzen, J., and Stewart, D. B. (2008). *Real World Haskell*. O'Reilly.

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

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

Sönmez, N., Perfumo, C., Stipic, S., Cristal, A., Unsal, O. S., and Valero, M. (2007). unreadtvar: Extending haskell software transactional memory for performance. *Trends in Functional Programming*, 8:89–114.

Sulzmann, M., Lam, E. S., and 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.