Investigando STM em Haskell como Alternativa para a Programação de Sistemas com Memória Compartilhada

Ernâni T. Liberali[1]
Juliana K. Vizzotto[1]

Resumo: Neste trabalho investiga-se a utilização de Software Transactional Memory (STM) como alternativa de programação paralela para arquiteturas com memória compartilhada. Primeiramente, faz-se um estudo do conceito de programação paralela e suas principais características. Posteriormente, apresenta-se a abordagem do modelo STM para programação paralela, bem como a sua aplicação no contexto de linguagens de programação funcional, mais especificamente em Haskell. Como estudo de caso para ilustração das qualidades do modelo STM, apresenta-se uma implementação do problema clássico de sincronização do jantar dos filósofos em Haskell. Por fim, com o objetivo de demonstrar a simplicidade e elegância dos códigos com STM em Haskell, compara-se esta implementação com outra do mesmo problema utilizando-se o mecanismo de sincronização monitores em Java.

Palavras-chave: Programação paralela. STM. Haskell.

Abstract: *This work proposes to investigate the use of Software Transactional Memory (STM) as a programming alternative for parallel architectures with shared memory. First, we study the concept of parallel programming and its main characteristics. Then we describe the STM approach for parallel programming, as well its application in the context of functional programming languages, in special Haskell. As a case study to illustrate the advantages of using STM, we show the implementation in Haskell of the classic problem of synchronization, the dining philosophers problem. Finally, aiming to demonstrate the simplicity and elegance of codes with STM in Haskell, it compares this implementation with another implementation of the same problem using the mechanism of synchronization monitors in Java.*

Keywords: *Parallel programming. STM. Haskell.*

1 Introdução

Este trabalho propõe investigar uma alternativa de programação paralela para sistemas *multi-core*, com memória compartilhada, utilizando o paradigma de programação funcional. Hoje a solução que está sendo encontrada pelos fabricantes de microprocessadores para obter melhor desempenho é a utilização de mais de um núcleo em cada processador. Essa nova arquitetura é chamada de "multicore" e sua principal característica é que a carga de trabalho pode ser dividida entre os núcleos de processamento.

Nesse contexto, o principal desafio atualmente está na área de programação de sistemas, para que possam executar eficientemente nesta nova arquitetura. A programação paralela é uma área que nos permite ter mais poder computacional em nível de programação do sistema. A ideia é dividir os programas em subprocessos que poderão ser executados simultaneamente por núcleos ou processadores diferentes [1].

Tradicionalmente, em programação paralela para sistemas com memória compartilhada utilizam-se mecanismos de lock e variáveis condicionais. Programas paralelos escritos com locks são difíceis de programar e bastante suscetíveis a erros, como, por exemplo, deadlock [2]. Assim, este projeto visa ao estudo de Software

[1] Curso de Sistemas de Informação, Centro Universitário Franciscano (UNIFRA) Conjunto 1 - Rua dos Andradas 1614 - Bairro Centro - Santa Maria (RS) – Brasil.
{e.liberalli@gmail.com} {juvizzotto@gmail.com}

doi: 10.5335/rbca.2010.009

Transactional Memory (STM) [3], é uma alternativa para programação paralela utilizando abstrações simples e de alto nível.

STM é um mecanismo de controle de acesso a memórias compartilhadas que pode ser implementado sem o uso de locks. Seu funcionamento é análogo ao de transações em um Banco de Dados, onde não se podem fazer transações simultâneas na mesma região crítica [3]. Este modelo garante sincronismo entre threads concorrentes utilizando o conceito de transação, em que uma transação é execução de uma sequência de instruções que garante atomicidade e isolamento.

Além disso, objetivando agregar mais simplicidade aos códigos, propõe-se investigar a utilização de STM no paradigma de programação funcional, o qual é um modelo simples e de alto nível de computação [4]. Dentre as linguagens funcionais modernas destaca-se a linguagem funcional pura Haskell [5], a qual também se apresenta como uma interessante opção para o desenvolvimento de aplicações paralelas, pois não apresenta efeitos colaterais.

Este artigo está organizado como segue: na seção 2 apresenta-se brevemente o conceito de programação paralela; a seguir, na seção 3, é apresentado o modelo memórias transacionais e suas principais características; na seção 4 apresenta-se a linguagem funcional Haskell e suas propriedades relacionadas à programação paralela; a seguir, na seção 5, é apresentada a implementação do problema clássico de sincronismo do jantar dos filósofos com monitores em Java e STM em Haskell; por fim, na seção 6 são apresentadas as conclusões sobre a utilização de transações ao invés de bloqueios para programação paralela em sistemas com memória compartilhada. Dessa forma, pode-se demonstrar o quão mais simples fica programar paralelamente utilizando o modelo de memórias transacionais em Haskell.

2 Programação paralela

O conceito de programação paralela data da década de 1940, quando Von Neumann estudava a possibilidade de utilizar algoritmos paralelos para a solução de equações diferenciais [6]. Já os primeiros trabalhos envolvendo o paralelismo, com implementação de hardware, apontam para o período 1944-1947, quando Stibitz e Williams, nos laboratórios da Bell Telephone, desenvolveram o sistema Model V [6]. Formado por dois processadores e três canais de entrada e saída, este multiprocessador primitivo já constituía um exemplo típico de arquitetura paralela. Nele poderiam ser executados dois programas distintos, bem como seus dois processadores poderiam ser alocados para uma única execução.

A programação paralela vem se mostrando uma solução interessante para a solução de problemas complexos, como, por exemplo, previsão do tempo, modelo de movimentação de corpos celestes e no aumento de desempenho de aplicações, reduzindo o tempo de execução (wall clock time) [7]. Seu objetivo é explorar o paralelismo existente em hardware, fazendo a divisão de tarefas complexas em subtarefas para executá-las em processadores diferentes no mesmo instante de tempo, facilitando e agilizando o cumprimento dessas tarefas.

Além disso, ajuda a reduzir custos em algumas arquiteturas, pois permite a utilização de vários processadores menores e mais baratos ao invés de se usar um de grande capacidade. Hoje, o desenvolvimento de aplicações paralelas é em geral mais demorado e trabalhoso do que o desenvolvimento de aplicações sequenciais, em virtude da dificuldade de se controlar o não determinismo das aplicações paralelas [8].

3 Memórias Transacionais

Na mesma época em que surgiu o conceito de transações em banco de dados, um conceito similar foi introduzido na programação concorrente para facilitar a estruturação e a sincronização de processos [9]. O conceito de transações evoluiu consideravelmente na área de banco de dados, tornando-se a principal forma de abstração para se resolver problemas em sistemas distribuídos [10]. O nome “Memória Transacional” (Transactional Memory, ou TM) foi dado por Herlihy e Mos em 1993 [8]. Neste trabalho foi descrita como uma nova arquitetura para multiprocessadores que objetiva tornar a sincronização livre de bloqueios tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua [3].

O modelo utiliza o conceito de transação para garantir sincronismo entre threads concorrentes. Uma transação é a execução de uma sequência de instruções que garantem atomicidade e isolamento. Durante a transação as ações de leitura e escrita nos dados compartilhados são feitas de forma local em um log; ao final da

transação, se nenhuma espécie de conflito foi detectada, essas alterações passam a ser visíveis ao resto do sistema, fazendo-se um *commit*. No caso de alguma falha, a transação é cancelada, suas alterações locais são descartadas e a transação é reiniciada. Este comportamento otimista permite que sistemas transacionais explorem mais o paralelismo, aumentando seu desempenho e sua escalabilidade [8].

No sistema de Memória Transacional todo o acesso à memória compartilhada é feito automaticamente pelo sistema que a implementa. Dessa forma, a programação multi-threaded também é beneficiada, pois a sincronização deixa de ser feita por bloqueios [11]. A aplicação do modelo de Memórias Transacionais pode ser feita por software (Software Transactional Memory, ou STM) ou por hardware (Hardware Transactional Memory, ou HTM). Existem, ainda, alguns estudos de uma alternativa híbrida dos dois modelos.

3.1 Memória Transacional e as Propriedades ACI

As transações são uma alternativa no controle de tarefas concorrentes e caracterizam-se pelas seguintes propriedades: atomicidade, consistência e isolamento, que permitem abstrair os detalhes das operações simultâneas de leitura e escritas de dados compartilhados em sistemas com multithreads ou multiprocessados. Em uma transação a atomicidade só permite que os dados na memória sejam alterados se ao término da transação não existirem erros; em caso contrário, será abortada. A atomicidade impede que uma transação que está sendo executada com erro se conclua e contamine as outras com resultados errados, o que poderia deixá-las em um estado inconsistente e imprevisível. O mecanismo que reverte uma transação com erro para o estado anterior torna-se importante para a implementação de mecanismos de controle de concorrência [8]. O isolamento garante que a transação produza o mesmo resultado, independentemente de outras estarem sendo executadas concorrentemente [8]. A consistência é importante para termos um código correto já que durante as transações as chamadas de execução são aleatórias.

3.2 Bloco Atômico

A principal vantagem de se utilizar transações é que as operações no bloco atômico ocultam os recursos compartilhados, dados e mecanismos de sincronização. Esta característica a distingue dos outros construtores, tais como monitores, onde o programador define os nomes dos dados protegidos pela seção crítica [12].

Já as transações especificam o resultado desejado da execução (falha, atomicidade e isolamento), o que depende de um sistema de TM para implementá-la. Como resultado, um bloco atômico pode ter oculta sua implementação em função das abstrações e pode ser composto em função de suas propriedades [8].

Da perspectiva do programador, um bloco atômico pode ter três possíveis resultados: commit, abortar e indefinido. Se a transação foi bem sucedida é dada a confirmação de que foi executada de forma correta, é feito o commit e seus resultados tornam-se parte do estado do programa, sendo visíveis para o código executado fora do bloco atômico. Ao contrário, se a operação abortar, o programa permanecerá em um estado inalterado. Ainda, se a operação não terminar, o programa ficará em um estado indefinido. Nos dois últimos casos haverá reexecução do código no bloco atômico [8].

A Memória Transacional pode abortar a transação durante a execução da aplicação. Primeiro é chamado um mecanismo de execução se houver conflito entre operações no acesso a recursos ou se uma operação não está conseguindo acessar determinado recurso. Nestes dois casos o sistema abortará a(s) operação(ões) com problema e irá reexecutá-la, na esperança de que o conflito não volte a ocorrer. Todo esse processo não é visível na aplicação, a não ser pela queda de desempenho causada pelas paradas e reexecuções.

Todas as operações de escrita e leitura são registradas em um log. Ao término de cada bloco, esse log é avaliado, de modo a se checar a consistência da memória e, logo depois, o seu conjunto de operações sofre uma espécie de commit. Caso a validação falhe, o bloco é reexecutado [5]. Essas características garantem, em situações como as de transferência em banco, que nenhuma thread observe o estado intermediário no qual o fundo saiu de uma conta, mas ainda não foi depositado em outra [5].

De uma forma prática, o programador define o bloco atômico e o código, que pode ter chamadas aninhadas e será executado atomicamente dentro dele, condição que é garantida pelo sistema transacional. O sistema garantirá a execução de forma atômica, pois será garantida a consistência dos dados e a coordenação na execução do mesmo bloco por várias threads. Este modelo dispensa o uso de variáveis externas e bloqueios. No

entanto, o uso de blocos atômicos não garante o término do processo ou a conformidade dos resultados [8]. Logo, abaixo apresenta-se um exemplo de declaração de um bloco atômico em um pseudocódigo.

```
exemploSTM (x, y, z){
   atomically {
               x=x+y;
               y =y+z;
              }
}
```

Como pode ser observado, dentro do bloco atômico são declaradas as operações críticas do código. Dessa forma, estas operações serão executadas de forma isolada.

3.3 Uso do *Retry* (Repetir)

O estudo sobre a semântica não indica nenhuma forma de controle sobre como duas transações sendo executadas concorrentemente devem modificar o estado do programa. Isso traz a necessidade de se controlar como e quando serão executadas. Como solução para este problema Harris [5] propôs a ideia da declaração retry (repetir) para coordenar transações.

O mecanismo de repetir uma transação permite que esta, ao entrar em um estado arbitrário, abandone sua execução para ser reexecutada posteriormente. Isso evita os conflitos da execução anterior, produzindo outros resultados. A reexecução é adiada até que sejam detectadas pelo sistema alterações nos valores que a transação havia acessado na primeira tentativa, aumentando consideravelmente a chance de sucesso na sua reexecução.

Em uma implementação real há necessidade de um mecanismo para limitar o número de tentativas, porém a transação não tem como saber quantas vezes terá de ser repetida, limite controlado pelo sistema que implementa a TM. Essas características tornam o mecanismo de reexecução eficaz na coordenação de transações [8].

A seguir é apresentado um exemplo de aplicação do *retry*, onde o valor x só pode ser decrementado se for maior que 0. Dessa forma, quando x for igual a 0 a transação será cancelada até que outra transação modifique o valor de x, tornando-o maior que 0.

```
atomically {
            if (x > 0)
              x=x-1;
             else
               retry;
           }
```

3.4 *Or Else* (Senão)

Trata-se de um mecanismo de coordenação entre transações. Quando uma transação commita, aborta ou repete, o OrElse muda o fluxo de execução para a outra tarefa. A seguir, apresenta-se exemplo da aplicação do mecanismo OrElse retirado do livro *Transactional Memory* [8].

No exemplo abaixo são mostradas duas transações, principal e secundária, onde a prioridade de execução é da principal. Caso não tenha elementos na lista principal, a secundária passa a ser executada. No caso de não haver elementos na lista secundária, a execução do bloco atômico fica suspensa até que exista algum elemento em, pelo menos, uma das listas.

```
atomically {
            { x = Lista1(); }  // lista principal
             orElse
            { x = Lista2(); } // lista secundária
           }
```

O OrElse deve ser declarado dentro do bloco atômico, pois as duas transações compõem uma operação que pode commitar, abortar ou repetir.

4 A Linguagem Funcional Haskell e STM

A linguagem Haskell [13] é uma linguagem de programação funcional pura, que surgiu em 1987 na universidade de Glasgow na Escócia. Haskell possui algumas características consideravelmente interessantes das linguagens funcionais, como lazy evaluation, pattern matching e modularidade. É uma das linguagens mais populares no meio acadêmico, além de ser utilizada em empresas tais como Microsoft Research Cambridge. O GPH (Glasgow Parallel Haskell*)* é uma variação de Haskell, com estruturas para diferenciar construções paralelas (par*)* de sequenciais (seq).

Haskell apresenta-se como uma interessante opção para o desenvolvimento de aplicações paralelas, pois, por ser uma linguagem funcional pura, não apresenta efeitos colaterais, oferecendo transparência referencial que permite avaliar subexpressões sem interferência. Isso assegura o resultado da função independente de quando ou como vai ser avaliada, pois não existe o conceito de referência à variável, onde qualquer ocorrência de uma variável em uma expressão pode sempre ser substituída pela sua definição [14].

Entretanto, considerando outro ponto de vista, o objetivo de um programa é produzir algum efeito, ainda que seja imprimir um resultado na tela do computador. A solução proposta para tratar essa "contradição" foi a utilização de Mônadas [15], permitindo se obterem efeitos colaterais em uma linguagem funcional, mase mesmo assim, manter as propriedades interessantes existentes nas linguagens funcionais puras.

4.1 Concorrências em Haskell

A concorrência em Haskell foi construída em cima da classe Monad (em português, mônada), que fornece um mecanismo para a criação de novos processos dinamicamente e é executada dentro do Glasgow Haskell [16]. Para se construir um programa concorrente em Haskell é preciso utilizar a primitiva, `forkIO()`. A thread é criada com a execução do argumento de `forkIO`(), que é uma operação do tipo I/O, a qual retorna o identificador da thread recém-criada. A seguir são apresentados os principais comandos para gerência de threads em Haskell.

- `forkIO :: IO () -> IO ThreadId`
- `killThread :: ThreadId -> IO ()`
- `threadDelay :: Int -> IO ()`

A declaração `forkIO` é responsável por criar uma thread nova e retorna seu identificador; a `killThread` finaliza a execução de uma determinada thread; já a declaração `threadDelay` suspende por um número determinado em microssegundos a execução de uma thread.

No exemplo abaixo é mostrada a declaração para se criar duas threads que irão imprimir concorrentemente as letras A e B:

```
main :: IO()
main = do { forkIO  (putStr "A")
          ; putStr  "B\n"
```

Neste caso, as duas threads são executadas simultaneamente, sendo suas ações de I/O intercaladas de forma aleatória. Dessa forma, tem-se a necessidade de se utilizar mecanismos de sincronização para se reduzir o não determinismo ao máximo em execuções concorrentes. Dentre os mecanismos de sincronização disponíveis em Haskell encontra-se a variável mutável, que é uma forma primitiva de bloqueio variável [16].

4.2 Variáveis Transacionais em Haskell

Apresentara-se neste tópico a utilização do modelo STM em paradigma de programação funcional, mais especificamente na linguagem Haskell. O suporte a transações é fornecido pelo Glasgow Haskell Compiler (GHC). A STM em Haskell é tratada como um tipo de dado transacional que tem suporte da classe Mônada [5]. A utilização de transações garante a atomicidade na execução entre threads concorrentes. A comunicação dentro da STM mônada utiliza declarações de controle e é feita por variáveis transacionais (TVar) [17].

- `atomically :: STM a -> IO a`
- `retry :: STM a`

- orElse :: STM a -> STM a -> STM a

Declaração `atomically` define região de execução das transações de forma atômica. O comando `retry` permite cancelar uma transação até que o estado de uma TVar mude. O comando `orElse` permite modificar o fluxo de execução das transações dentro do bloco atômico.

- `newTVar :: a -> STM (TVar a)`
- `readTVar :: TVar a -> STM a`
- `writeTVar :: TVar a -> a -> STM ()`

A primitiva `newTVar` permite que se crie uma nova variável transacional, a primitiva `readTVar` lê o valor de uma TVar e a primitiva `writeTVar` permite que se escreva um valor em uma TVar.

5 Implementando o Problema Clássico de Sincronismo Jantar dos Filósofos

Nesta seção discutem-se as vantagens de se utilizarem transações no controle de processos concorrentes ao invés de bloqueios. Como ilustração para motivar o uso de STM em Haskell apresentam-se implementações do problema clássico de sincronismo jantar dos filósofos.

5.1 O Problema do Jantar dos Filósofos

O problema do jantar dos filósofos é um problema clássico de sincronismo proposto por Dijkstra em 1965 [18]. Trata-se de uma representação simples da necessidade de alocar vários recursos entre vários processos. O cenário do problema é uma mesa com cinco filósofos sentados ao seu redor, cada um com um prato de espaguete e um garfo, tal que os filósofos só fazem duas coisas que é pensar e comer [18].

Neste problema, quando os filósofos sentem fome deverão pegar dois garfos para comer: primeiro, o filósofo pega o garfo da esquerda e, depois, o da direita. Como só há cinco garfos, os filósofos que estiverem ao lado e também sentirem fome não conseguirão comer até que o que está comendo largue os dois garfos e volte a pensar. Outro problema mais complicado é se todos os filósofos ficarem com fome ao mesmo tempo e pegarem um dos garfos, fazendo com que nenhum consiga pegar os dois necessários para comer causando um impasse permanente. Nesse problema os processos são os filósofos e os garfos são os semáforos, os quais são bloqueios que oferecem exclusão mútua com a finalidade de proteger a seção crítica do código. Dessa forma, os semáforos buscam garantir que a seção crítica seja executada por um processo de cada vez.

O problema do jantar dos filósofos é uma representação simples da necessidade de alocar vários recursos entre vários processos de uma maneira que não cause deadlock e ou starvation [19]. Para evitar que threads interfiram de maneira errada no trabalho de outras threads, podem ser utilizados mecanismos de sincronização, como, por exemplo, locks ou mutexes, características que tendem a tornar o código complicado e propenso a erros como deadlocks e ou starvation [19].

Fonte: Tanenbaum; Woodhull, 1997.

Figura1: O Jantar dos Filósofos

5.2 Implementando o Problema do Jantar dos Filósofos utilizando bloqueios em Java

A seguir apresenta-se uma solução para o problema clássico de sincronismo do jantar dos filósofos baseada na proposta de Silberschatz, Galvin e Gagne [19]. Neste exemplo o método será executado se a thread puder adquirir o monitor que pertence ao método. Caso contrário, a thread que invocou o método será suspensa até que possa adquirir o monitor. Os monitores em Java são definidos pela declaração synchronized na declaração do método.

A synchronized, ao contrário do atomically, limita a concorrência em sistemas paralelos, pois só permite que um processo execute a seção crítica por vez. Esse mecanismo de controle baseado em bloqueios expõe os detalhes da sincronização, tornando o código mais complexo e, consequentemente, mais suscetível a erros.

```
public class Filosofos extends Thread {

 int[] filosofos = new int [5] ;
 boolean[] garfos = new boolean [5];
 private static int pensando = 0;
 private int fome = 1;
 private int comendo = 2;
 private int filo=0;

 public void run (){

    while (true){
       try {

             Random rd = new Random();
             System.out.println("Filosofo"+filo+" pensando!");
             sleep(rd.nextInt(10000));
             filosofos[filo]=fome;
             System.out.println("Filosofo "+filo+" com fome!");
             pega(filo);
             System.out.println("Filosofo "+filo+" comendo!");
             sleep(rd.nextInt(10000));
             devolve(filo);
           }catch(InterruptedExceptione){e.printStackTrace();}
    }
 }
```

O método `run` é responsável por coordenar as chamadas à seção crítica. Nele defini-se o comportamento dos filósofos que ficam permanentemente em loop. Nesta função é determinado um tempo randômico em que o filósofo ficará pensando e comendo. As chamadas para entrar e sair da seção crítica são feitas pela invocação dos métodos `pega()` e `devolve();`

```
public synchronized void pega (int filosofo){

       while(!(testa(filosofo))){
            try {
                  wait();
                } catch (InterruptedException e){}
            }
       garfos[(filosofo+1)%5]=false;
       garfos[(filosofo+4)%5]=false;
       filosofos[filosofo] = comendo;
 }
```

O método acima, com a declaração `synchronized`, permite que só uma thread o execute de cada vez. Este método é utilizado pelo filósofo para pegar seus dois garfos para comer. Caso o filósofo não consiga os dois garfos, a sua thread ficará esperando. Esta situação é determinada pela declaração `wait()`.

```
Public synchronized void devolve(int filosofo){

         garfos[(filosofo+1)%5]=true;
         garfos[(filosofo+4)%5]=true;
         filosofos[filosofo] = pensando;
         notifyAll();
 }
```

O método acima, com a declaração `synchronized`, permite que só uma thread o execute de cada vez. Este método é utilizado pelo filósofo para largar seus dois garfos. Depois de largar seus dois garfos, a thread que está executando este método (filósofo) avisará às outras threads que os largou. Este comportamento é definido pela chamada `notifyAll();`

```
private boolean testa(int i){

        if((filosofos[(i+4)%5] != comendo) && (filosofos[i] == fome) &&
(filosofos[(i+1)%5] != comendo) ){
            return true; }
        return false;
 }
```

O mecanismo de controle baseado em bloqueios, mesmo os de mais alto nível como monitores, não abstrai totalmente os detalhes da sincronização dos processos. No código acima foi necessária a implementação de um método (`testa`) para verificar se algum dos vizinhos estava comendo para, então, o filósofo solicitar os garfos. Neste modelo parte da tarefa de sincronizar os processos fica a cargo do programador. Isso torna a construção do código mais complexa e, consequentemente, vulnerável a uma quantidade maior de erros em situações de conflito. Uma situação de conflito seria quando todos os filósofos resolvessem pegar os seus garfos da esquerda. Essa situação, se não for prevista e tratada pelo programador, certamente colocará o sistema em um estado de *deadlock*.

5.3 Implementando o Problema do Jantar dos Filósofos utilizando STM em Haskell

O uso de memórias transacionais em Haskell traz vantagens, se comparado ao uso de bloqueios, no desenvolvimento de aplicações paralelas. Em virtude do sistema de tipos de linguagens funcionais como Haskell, não é permitido que se tenha ações impuras (I/O) em uma transação. Em conjunto com o modelo STM, que garante atomicidade e isolamento das transações, é possível desenvolver aplicações com um alto nível de abstração sem os problemas comuns encontrados quando utilizados sistemas baseados em bloqueios, como, por exemplo, os deadlocks. Dessa forma, podemos desenvolver aplicações de alto nível, nas quais o programador só terá de se preocupar em definir a região crítica, ficando o trabalho da sincronização e tratamento de eventuais conflitos por conta do sistema STM.

Pode-se perceber isso na solução baseada na proposta de implementação de Frank Huch e Frank Kupke do problema clássico do jantar dos filósofos utilizando STM em Haskell [20].

```
type Garfo = TVar Bool
```

É criado um sinônimo de tipo, com o nome garfo, que é uma variável transacional do tipo booleano.

```
criaGarfos :: Int -> STM [Garfo]
criaGarfos n = do garfos <- (mapM (const (newTVar True)) [1..n])
           return garfos
```

Na função `criaGarfos,` que recebe um inteiro (n) e retorna um vetor transacional de (n) garfos com o valor *True*.

```
pensa = do { waitTime <- getStdRandom (randomR (1, 1000000)) ; threadDelay
waitTime }
```

Esta função criará um valor aleatório para o delay da thread.

```
pegaGarfo :: Garfo -> STM ()
pegaGarfo g = do
                b <- readTVar g
                if b
                then (writeTVar g False)
                   else retry
```

A função `pegaGarfo` recebe um garfo (g) e faz uma ação STM, sendo utilizada pelas transações para gravar False no garfo. A TVar (g) é lida e, se seu valor for True (livre), será gravado False (ocupado) nela. Caso

a TVar esteja com o valor False (ocupada), a transação (filósofo) ficará aguardando sua mudança de estado para gravar False, ou seja, pegar o garfo. Esta ação é feita através da declaração retry. Neste estado de espera o filósofo não fica bloqueando nenhum garfo.

```
colocaGarfo :: Garfo -> STM ()
colocaGarfo g = writeTVar g True
```

A função `colocaGarfo` recebe um garfo (g) e faz uma ação STM. Ela gravará o valor *True* na TVar, liberando o garfo para o filósofo do lado.

```
filosofo :: Int -> Garfo -> Garfo -> IO ()
filosofo n e d = do
    putStr ("Filosofo esta com Fome!" ++show (t1));
    atomically ( do
                  pegaGarfo e
                  pegaGarfo d)
   putStrLn ("O filosofo "  ++ show n ++  " esta comendo!")
   atomically ( do
                  colocaGarfo e
                  colocaGarfo d)
   putStrLn ("Filosofo esta pensando!")
   pensa
   filosofo n e d
```

A função `filosofo` recebe o seu número (n) e os garfos da direita (d) e da esquerda (e) . A função `filosofo` possui dois blocos atômicos: no primeiro, ele adquire seus garfos de forma atômica e, no segundo, libera-os também de forma atômica.

A implementação deste código diferencia-se do apresentado no item 5.2 por estar implementado em uma linguagem funcional pura, que em função do seu sistema de tipos rigoroso, evita a leitura ou escrita numa TVar de fora de um bloco atômico e, principalmente, pelos acessos à região crítica do código estarem sendo feitos dentro de um bloco atômico. O uso do bloco atômico garante que o código dentro dele seja executado de forma atômica, ocultando os detalhes da sincronização dos processos. Dessa forma, o problema de todos os filósofos pegarem seus garfos da esquerda não existe para o programador, pois a propriedade da atomicidade garante que o filósofo pegará os dois garfos de que precisa ou não pegará nenhum. Caso um filósofo fique impossibilitado de pegar os dois garfos num dado instante, a sua transação será cancelada até que uma mudança de estado na TVar (garfo) compartilhada ocorra. Esta propriedade é controlada pela declaração retry. Dessa forma, o modelo STM garante que ou o filósofo pegará os dois garfos de que precisa para se alimentar ou não pegará nenhum. Com a aplicação deste modelo associado a uma linguagem funcional, a implementação de códigos paralelos torna-se uma tarefa menos dispendiosa, pelo fato de o sistema STM abstrair os detalhes da sincronização dos processos.

6 Conclusão

Neste trabalho fez-se um estudo de uma forma geral sobre os principais conceitos de programação paralela. A partir disso investigou-se detalhadamente a abordagem de Memórias Transacionais para programação paralela em arquiteturas com memória compartilhada. Essencialmente, estudou-se a programação com STM considerando o paradigma de programação funcional, por meio da linguagem Haskell. Assim, pôde-se verificar a utilização de STM para programação paralela, assim como a utilização do paradigma funcional. Essencialmente, a ideia deste trabalho é promover a discussão sobre a simplicidade, eficiência e aplicabilidade do modelo STM em paradigma funcional no contexto de programação paralela para as arquiteturas multicore.

Por meio de exemplos, foi possível mostrar a simplicidade de código que se obtém em problemas tradicionais de sincronização em programação paralela, através da utilização de STM em paradigma funcional. Mais especificamente, demonstrou-se através da implementação do problema clássico de sincronismo jantar dos filósofos as características e vantagens do uso de STM em Haskell, comparado a outras técnicas de sincronização de processo baseadas em bloqueios. Utilizar o modelo de Memórias Transacionais evita uma série de transtornos como deadlock e estarvations, que temos quando utilizamos bloqueios e variáveis de condição, pelo fato de estes mecanismos exporem os detalhes da sincronização. Parte desse resultado também se deve ao fato da linguagem funcional Haskell ter um sistema de tipos que evita a leitura ou escrita numa TVar de fora de um bloco atômico.

Em virtude dessas características, não há necessidade de se utilizarem bloqueios, livrando o programador da árdua tarefa de definir a ordem e os bloqueios que terão de ser adquiridos durante a execução. Dessa forma, conclui-se que o uso de STM, especialmente em Haskell, possibilita construir códigos com um nível maior de abstração. Isso diminui, consideravelmente, a responsabilidade do programador referente à sincronização dos processos e ao tratamento da maioria dos conflitos que ocorrem durante a execução de uma aplicação paralela.

Referências

[1] GRAMA, A. et al. **Introduction to Parallel Computing.** 2. Ed. Addison-Wesley, 2003.

[2] JONES, S. P. Beautiful concurrency. In: **Beautiful Code**: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly)). O'Reilly Media, Inc., 2007.

[3] HERLIHY, M.; MOSS, J. Transactional memory: **Architectural support for lock-free data structures.** 1993. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5910>. Acesso em: 24 ago. 08.

[4] GOLDBERG, B. **Functional Programming Languages**, New York University 96. Disponível em: <http://www.cs.nyu.edu/goldberg/pubs/gold96.pdf>. Acessado em: 13/08/2008.

[5] HARRIS, T. M.; JONES, P. S. **Composable Memory Transactions.** In: PPoPP'05: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Chicago, Illinois 2005.

[6] YAMIN, C. A. **Um estudo das potencialidades e limites na exploração do paralelismo.** UCPEL 2008 Disponível em: <http://paginas.ucpel.tche.br/~adenauer/ppad/index.php?arquivo=programa>. Acesso em: 11 out. 2008.

[7] MARTINS, L. S. **Programação Paralela.** Universidade Federal do Mato Grosso do Sul, Agosto/2002. Disponível em: <http://www.lncc.br/~biologia/english/downloads/ProgramacaoParalela1.pdf>. Acesso em: 20 ago. 08.

[8] LAURUS, J. R.; RAJWAR, R. **Transactional Memory.** Editor: Mark D. Hill University of Wisconsin Morgan & Claypool 2007.

[9] LOMET, D. B. **Process Structuring, Synchronization, and Recovery Using Atomic Actions.** In Proc. ACM Conf. on Language Design for Reliable Software, Raleigh, NC, 1977.

[10] GRAY, J.; REUTER, A. (1993). **Transaction Processing**: Concepts and Techniques. Morgan Kaufmann Publishers, Inc.

[11] RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. **Memórias transacionais uma nova alternativa para programação concorrente.** Laboratório de Sistemas de Computação –Instituto de Computação – Unicamp, 2007.

[12] HOARE, C. A. R. Monitors: **An Operating System Structuring Concept.** Communications of the ACM, 1974: p. 549-557.

[13] WADLER, P**. The essence of functional programming.** In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principies of Programming Languages. Albequerque, New México 1992. Disponível em: <https://wiki.ittc.ku.edu/lambda/images/1/12/Wadler__The_essence_of_functional_progamming_%281992%29.pdf>>. Acesso em: 13 fev. 09.

[14] FIGUEIREDO, L. **Entrada e Saída em Haskell Tutorial.** Departamento de Computação Universidade Federal de Ouro Preto, 30 de junho de 2005.

[15] MOGGI, E. **Computational Lambda-Calculus and Monads.** In Proc. of 4th Ann. IEEE Symp. On Logic in Computer Science, LICS'1989 (Pacific Grove, CA, USA).

[16] SPILIOPOULOU, E. **Concurrent and Distributed Functional Systems.** University of Bristol, Department of computer science, CSPHD, September 1999.

[17] HASKELLWIKI. página oficial da linguagem Haskell. Disponível em: <http://www.haskell.org/haskellwiki/Pt/Introdu%C3%A7%C3%A3o>. Acesso em: 30 mai. 2009.

[18] TANENBAUM, S. A.; WOODHULL, A. S. **Sistemas Operacionais Projeto e Implementação**. 2. ed. Porto Alegre: Editora Bookman, 2000.

[19] SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Sistemas Operacionais com Java.** Rio de Janeiro: Elsevier, 2004.

[20] HUCH, F.; KUPKE, F**. Composable Memory Transactions in Concurrent Haskell.** In Implementation and Applications of Functional Programming Languages Springer-Verlag 2007.

[21] DE SÁ, C, C.; DA SILVA, F,M. **Haskell uma abordagem prática.** Novatec 2006, SP, Brasil.