UMA LINGUAGEM DE DOMÍNIO ESPECÍFICO COM SUPORTE A MEMÓRIA TRANSACIONAL DISTRIBUÍDA EMBUTIDA EM JAVA

JERÔNIMO DA CUNHA RAMOS[1]; MAURÍCIO LIMA PILLA[1]; ANDRÉ RAUBER DU BOIS[1]

[1]*Universidade Federal de Pelotas (UFPel) – {jdcramos, pilla, dubois}@inf.ufpel.edu.br*

1. INTRODUÇÃO

Para tirar proveito do paralelismo disponibilizado pelos computadores atuais, os programas devem conter atividades que possam executar concorrentemente. Normalmente, em arquiteturas de memória compartilhada, isto é feito com o uso de threads, variáveis compartilhadas e *locks*. Porém, este modelo de programação impõe quase toda a complexidade de exploração do paralelismo e de tratamento das condições de corrida ao programador, o que torna-o propenso a erros, como *deadlocks*. A comunidade científica vem propondo alternativas a este modelo para facilitar e popularizar a programação concorrente, como é o caso das memórias transacionais (RIGO; CENTODUCATTE; BALDASSIN, 2007).

Memória transacional é uma abstração para programação concorrente baseada na ideia de transações, similares às de banco de dados. Uma transação de memória é uma sequência atômica de operações que modificam a memória e que podem ser executadas completamente ou podem ser abortadas. Desta maneira, transações de memória executam como se estivessem modificando uma área de memória isolada do acesso de outras transações. Estas só conseguem ver o resultado após o *commit* (DU BOIS, 2008).

Este modelo possui várias vantagens sobre o modelo clássico de programação, sendo as principais: ausência de *deadlock*, possibilidade de abortar e retornar ao estado original do programa, maior possibilidade de exploração do paralelismo, facilidade de programação, escalabilidade e composabilidade (RIGO; CENTODUCATTE; BALDASSIN, 2007; DU BOIS, 2008).

Nos últimos anos as pesquisas sobre Memórias Transacionais têm sido bastante focadas em máquinas multicore, deixando outros tipos de arquiteturas, como *clusters* e máquinas NUMA (*Non-Uniform Memory Access*) quase inexploradas (KOTSELIDIS et al., 2008; BOCCHINO; ADVE; CHAMBERLAIN, 2008). Estas arquiteturas têm como principal diferencial o tempo variado de acesso à memória local e remota. No caso dos *clusters* e outros sistemas distribuídos (SD), cada nodo possui seu próprio espaço de endereçamento de memória, fazendo com que a comunicação tenha que ser feita por troca de mensagens e, geralmente, não haja coerência de *cache*.

Sendo assim, este trabalho propõe uma linguagem de domínio específico embutida em Java para implementação de transações de memória que podem envolver objetos locais e distribuídos em uma rede. As contribuições deste trabalho são:

- **Linguagem DCMTJava:** Permite que programadores possam definir ações transacionais que são objetos em Java, dessa forma métodos podem receber transações como argumentos e retornar transações como resposta de sua execução. Essas transações podem ser compostas usando uma notação especial, gerando novas ações transacionais. DCMTJava estende a linguagem CMTJava (DU BOIS; ECHEVARRIA, 2009; BANDEIRA, 2013) permitindo que transações envolvam tanto objetos locais quanto remotos.

- **Sistema transacional distribuído:** Para dar suporte às abstrações da linguagem é necessário um sistema de tempo de execução que permita a composição de transações envolvendo objetos locais e remotos. O sistema aqui proposto combina dois algoritmos para transações; para objetos locais um algoritmo baseado no swissTM (DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009), onde conflitos de escrita-escrita são detectados de forma adiantada, evitando que essas transações executem até o final. Para objetos distribuídos os conflitos são detectados tardiamente, evitando comunicação excessiva, durante a execução das transações.
- **Prototipação:** O algoritmo proposto foi implementado, em Java, estendendo a implementação da linguagem CMTJava. Para validação, foram feitos casos de teste envolvendo diferentes números de transações com objetos locais e distribuídos.

2. METODOLOGIA

Primeiramente foi realizada uma revisão bibliográfica, quando verificou-se que existem diversas implementações de sistemas de memórias transacionais para máquinas *multicore*, dentre elas a CMTJava (DU BOIS; ECHEVARRIA, 2009; BANDEIRA, 2013), porém o foco destas não está em arquiteturas distribuídas, sendo assim não preveem soluções para os diferenciais destas arquiteturas. As propostas focadas em arquiteturas distribuídas são pouco numerosas, sendo estas focadas em *clusters*, como é o caso de Cluster-STM (BOCCHINO; ADVE; CHAMBERLAIN, 2008), DiSTM (KOTSELIDIS et al., 2008) e TFA (SAAD; RAVINDRAN, 2012).

Em seguida, foi proposta uma extensão da linguagem CMTJava utilizando em seu sistema de tempos de execução a combinação de dois algoritmos para transações. Objetos locais são validados usando o algoritmo da swissTM, comprovadamente mais rápido que outros algoritmos clássicos de transações. Para objetos remotos, a validação é feita de forma similar ao algoritmo TFA, permitindo que objetos sejam validados somente no final da transação, evitando comunicação excessiva. Por fim, este modelo foi prototipado e testado, para validação de seu funcionamento.

3. RESULTADOS E DISCUSSÃO

Após a conclusão das implementações do protótipo, foram realizados testes para validação do funcionamento do sistema transacional da linguagem. Para estes testes foi implementada uma aplicação simples simulando contas bancárias, onde cada conta possui um atributo de saldo e, durante a execução, são realizados depósitos e saques, sendo que no final da execução o valor é checado para verificar sua corretude.

Na execução, foram utilizados dois nodos, onde cada um possui um objeto local e, alem disso, existe um objeto compartilhado entre todos os nodos. Esta configuração representa um caso extremo de conflitos, tanto por possuir apenas um objeto compartilhado quanto por suas transações serem muito pequenas. Ao se combinar estas duas características, cria-se um cenário onde todas as *threads*, de todos os nodos, devem adquirir os *locks* de um mesmo objeto e, como as transações são pequenas, o tempo em que a transação fica com *locks* adquiridos se torna relativamente elevado. Sendo assim, a contenção na aplicação que foi utilizada para os testes é extremamente elevada, influenciando o tempo de execução.

Como dito, o objetivo principal deste teste foi verificar a corretude do sistema transacional da linguagem. Além disso, para verificar o *overhead* adicionado por esta implementação, seus tempos de execução foram comparados aos tempos de execução da CMTJava. Foram realizados testes com ambas efetuando as mesmas operações, porém CMTJava utilizou dois objetos locais, ao invés de um local e um compartilhado. As comparações de tempo foram realizadas entre as execuções com o mesmo número total de operações, sendo que, na DCMTJava, foi utilizada a configuração de 50% das operações locais e 50% com o objeto compartilhado. A Figura 1 traz o resultado deste experimento.

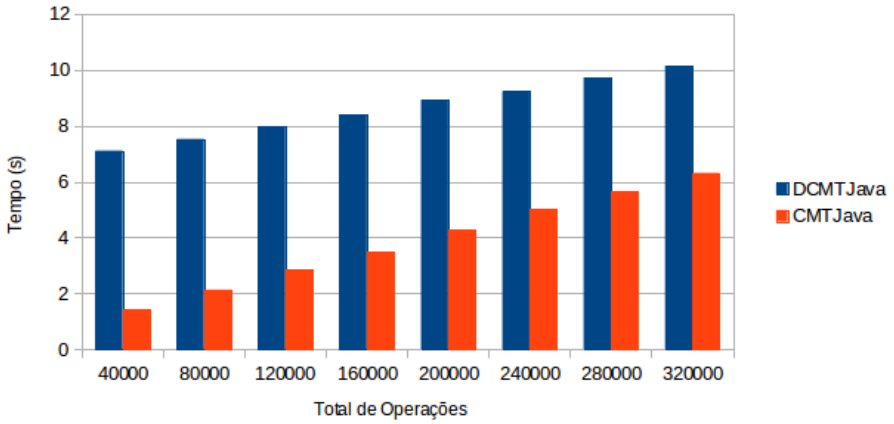

Figura 1 – Comparação dos tempos de execução de ambas as linguagens

Como esperado, o tempo de execução da DCMTJava foi maior, nos casos avaliados, pois esta apresenta o tempo adicional gasto nas comunicações pela rede. Este tempo se torna ainda mais significante devido às transações desta aplicação de teste serem pequenas. Já a CMTJava executa em um único nodo, não sendo afetada por este *overhead*. No entanto, percebe-se que a diferença entre ambas diminui conforme é aumentado o número de operações, como era esperado, já que a DCMTJava distribuiu o trabalho entre dois nodos e a CMTJava não, saturando sua capacidade de processamento antes da DCMTJava. Isto leva a crer que se forem realizados testes com ainda mais operações, em algum momento o tempo da DCMTJava será melhor que o da CMTJava, porém são necessários mais testes para que isto possa ser afirmado.

4. CONCLUSÕES

Hoje em dia há uma tendência de cada vez mais dispositivos computacionais se comunicarem. Além disso, grande parte da computação de alto desempenho é composta por sistemas distribuídos. Estes fatos tornam o estudo destas arquiteturas cada vez mais importante. Elas são relativamente difíceis de programar, pois apresentam, além dos problemas de concorrência, dificuldades adicionais, como sincronização e comunicação. Tais dificuldades fazem com que o desenvolvimento de estratégias de programação mais simples seja importante.

O modelo de memórias transacionais tem se mostrado bastante promissor para facilitar a programação de arquiteturas paralelas, porém, devido às características diferenciadas das arquiteturas distribuídas, os modelos já propostos não podem ser utilizados diretamente, sendo assim o principal objetivo deste trabalho foi estender uma linguagem de programação para que esta suportasse transações envolvendo objetos distribuídos.

A principal contribuição deste trabalho foi a concepção da linguagem DCMTJava que estende a linguagem CMTJava, permitindo que transações

envolvam tanto objetos locais como distribuídos entre nodos de uma rede. Deste modo, o presente trabalho permite que o programador disponha das vantagens do uso de memória transacional, destacando-se a facilidade de programação, enquanto o sistema de tempo de execução encarrega-se de tratar, de maneira transparente ao programador, tanto os problemas de sincronização locais e distribuídos, quanto os problemas de serialização e comunicação.

São diversas as possibilidades de trabalhos futuros, sendo alguns deles: (i) realização de testes de escalabilidade, para analisar como a implementação se comportará quando se escala o número de nodos do sistema e o número de objetos; (ii) implementação de aplicações mais complexas para realização de testes; (iii) conclusão da implementação do compilador da linguagem, visto que, atualmente, algumas estruturas são geradas manualmente; (iv) otimização da implementação para ganhar desempenho, tanto nas transações locais quanto nas distribuídas; (v) utilização de outros modelos para as transações distribuídas, como algum modelo que mova as instruções ao invés dos objetos. Isto permitirá que sejam realizadas comparações entre as implementações para descobrir-se qual estratégia funciona melhor em cada cenário.

5. REFERÊNCIAS BIBLIOGRÁFICAS

BANDEIRA, R. L. **Um Sistema de Detecção de Conflitos com Invalidação Mista para a Linguagem CMTJava.** 2013. Dissertação (Mestrado em Computação) – Programa de Pós-graduação em Computação, UFPel.

BOCCHINO, R. L.; ADVE, V. S.; CHAMBERLAIN, B. L. Software Transactional Memory for Large Scale Clusters. **Proceedings of the 13th ACM SIGPLAN Symposium on PPoPP,** New York, p.247-258, 2008.

DRAGOJEVIĆ, A.; GUERRAOUI, R.; KAPALKA, M. Stretching Transactional Memory. **Proceedings of ACM SIGPLAN Conference on PLDI,** New York, p.155-165, 2009.

DU BOIS, A. R. Memórias Transacionais e Troca de Mensagens: Duas Alternativas para a Programação de Máquinas Multi-Core. **SBC, P. A. (Ed.). Escola Regional de Alto Desempenho,** p.43-76, 2008.

DU BOIS, A. R.; ECHEVARRIA, M. A. Domain Specific Language for Composable Memory Transactions in Java. **DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages,** p.170-186, 2009.

KOTSELIDIS, C.; ANSARI, M.; JARVIS, K.; LUJÁN, M.; KIRKHAM, C.; WATSON, I. DiSTM: A Software Transactional Memory Framework for Clusters. **Proceedings of 37th ICPP,** Washington, p.51-58, 2008.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. **Minicursos do VIII WSCAD,** 2007.

SAAD, M. M.; RAVINDRAN, B. Transactional Forwarding: Supporting Highly-Concurrent STM in Asynchronous Distributed Systems. **SBAC-PAD, 2012 IEEE 24th International Symposium,** p.219-226, 2012.