Verificando o desempenho e a eficiência energética das memórias transacionais.*

Rodrigo M. Duarte, André R. Du Bois, Gerson G. H. Cavalheiro

[1]Universidade Federal de Pelotas(UFPel)
Bacharelado em Engenharia da Computação
LUPS - Laboratory of Ubiquitous and Parallel Systems

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

Resumo. *O presente artigo tem por objetivo mostrar os problemas existentes na programação paralela tradicional. Demostrar como alternativa, a aplicação de memórias transacionais como um método para facilitar a programação multithread, bem como sua aplicação na linguagem STM-Haskell. Também propõem a utilização de um Benchmark e equipamentos para medir quão eficiente é este novo modelo de programação e qual seu impacto no consumo de energia, comparado ao método tradicional de programação.*

1. Introdução

Para se conseguir o máximo proveito de processadores *multi-core*, é necessário que estes tenham várias atividades concorrentes que possam ser alocadas nos *cores* disponíveis.

Tradicionalmente os programas desenvolvidos para esta arquitetura são escritos implementando *threads*, que se comunicam usando uma área de memória compartilhada (região crítica) [Silberschatz. Abraham and Gagne 2001]. Para evitar que threads interfiram de forma errada na execução uma das outras, são utilizados métodos de sincronismo, também conhecidos como *locks* ou *mutexs*. Esse modelo de programação é complexo e propenso a erros [Rajwar and Goodman 2003, Lee 2006, Jones 2007, Rigo S. and A. 2007].

Em virtude das dificuldades e dos problemas que o modelo tradicional de programação concorrente apresenta, novas abstrações de programação são desenvolvidas. Uma delas é a de memória transacional, que tende a ser mais fácil de programar e aumenta o nível de paralelismo.

Pesquisas demonstram também que uma melhor distribuição de tarefas entre os *cores* disponíveis, aumenta o desempenho e reduz o consumo de energia, levado a programação paralela a uma ligação muito forte com a filosofia de *green computing* [Vykoukal J. and R. 2009].

Este artigo esta organizado da seguinte forma: na seção 2, mostramos os problemas da utilização de bloqueios, bem como seu reflexo no desempenho e complexidade. As seções 3 e 4 dão uma introdução sobre memórias transacionais, que é um novo modelo de abstração para a programação paralela e apresenta a linguagem STM-Haskell, que é uma extensão da linguagem funcional Haskell, implementando memórias transacionais. A seção 5 aborda *green computing* e sua ligação com a programação paralela. E no final,

*Este trabalho é parte do projeto "Green Grid", financiado pelo Programa PRONEX FAPERGS/CNPq.

na seção 6, demonstra a proposta de pesquisa, onde será feita uma verificação tanto do desempenho, quanto da eficiência energética das memórias transacionais.

2. Os problemas da utilização de bloqueios

A utilização de *locks*, como método de sincronismo entre *threads*, torna a programação difícil e propensa a erros [Rajwar and Goodman 2003, Lee 2006, Jones 2007, Rigo S. and A. 2007]. Entre os problemas apresentados neste modelo de programação, são mais comuns a ocorrência de gargalos seriais, *deadlocks* e difícil composabilidade de códigos.

Um erro comum de programação é proteger grandes áreas de código por *locks*. Uma região crítica protegida por um bloqueio permite o acesso à somente uma *thread* por vez o que acaba gerando a ocorrência de gargalos seriais. Este tipo de problema acaba reduzindo o nível de paralelismo.

Para evitar o problema de gargalo serial, tende-se a utilizar um maior número de bloqueios. Porém esta técnica deixa o código ainda mais complexo e propenso a erros de instabilidade como *deadlocks*. Pode também ocorrer do programador adquirir um menor número de *locks* que o necessário, o que pode levar a erros de condição de corrida, que são de ocorrência imprevisível e de difícil depuração [Tanenbaum and Woodhull 2006].

Outro problema está no reuso de código. Códigos bem implementados e já validados usando *locks*, quando combinados, podem gerar erros. Não há garantias que estes códigos não vão apresentar problemas de sincronismo em uma nova aplicação, o que pode levar novamente a condições de corrida e ou *deadlocks*. Torna-se necessário que o programador revalide o código novamente para evitar estes tipos de problemas.

Por estes motivos, programas escritos usando *locks* apresentam baixo desempenho, elevada complexidade e difícil composabilidade [Rigo S. and A. 2007].

3. Memória Transacional

Memória Transacional é um modelo de programação concorrente que usa como base o conceito de transações para garantir sincronismo entre *threads* concorrentes [Rigo S. and A. 2007].

Transações de memórias é uma sequência de operações que modificam a memória e que podem ser executadas completamente (atomicidade), ou não geram nenhum efeito (podem ser abortadas). Essas operações são atômicas, ou seja, são executadas por completo ou não são executadas. Este tipo de transações de memórias são parecidas com as transações realizadas em bancos de dados. Nessas transações, os acessos a memória são armazenados em um *log* de transação. Quando o bloco da transação terminar de executar, seu *log* é validado para verificar se sua visão da memória é consistente, se esta for, então a transação passa seus valores para a memória.

Outra característica das memórias transacionais é o isolamento. Onde os resultados intermediários produzidos por uma transação não podem ser acessados por outras transações concorrentes. Assim, o resultado da execução das transações concorrentes equivale ao resultado da execução dessas mesmas transações em alguma ordem serial [Rigo S. and A. 2007].

Memórias transacionais proporcionam que se explore mais paralelismo, aumentando o desempenho e escalabilidade. O uso deste também facilita a programação *multi-thread* porque o programador não precisa se preocupar com problemas de sincronização, e.g., *deadlocks*, pois todo o controle de acesso a memória compartilhada é feito automaticamente pelo sistema transacional.

4. STM-Haskell

STM-Haskell é uma extensão da linguagem Haskell, que fornece a abstração de memórias transacionais para a programação concorrente.

Haskell é ideal para implementar memórias transacionais por dois motivos [Harris et al. 2008]: o sistema de tipos consegue separar as ações que possuem efeitos colaterais das que não e grande parte das computações são puras, ou seja, não realizam efeitos colaterais. Como esse tipo de ação pura não modifica a memória, ela não precisa ser tratada pelo sistema transacional. Logo estas ações nunca precisam ser desfeitas, elas simplesmente podem ser repetidas caso a transação seja abortada.

O sistema de tipos do Haskell também garante que ações que realizam efeitos colaterais sejam associadas a tipos específicos, por exemplo, dentro de transações apenas operações que modificam a memória serão executadas. Assim, o sistema de tipos garante que operações de entrada e saída não serão executadas dentro de transações.

Por ser também uma linguagem funcional, apresenta várias vantagens para se desenvolver programas paralelos/concorrentes [Hammond and Michelson 2000], como facilidade em particionar programas paralelos, pois o resultado do programa independe do escalonamento das tarefas, ausência de *deadlock* e abstrações de alto nível.

5. Green Computing

Computação verde (*Green computing*) refere-se a um conjunto de práticas adotadas no setor tecnológico, para tornar o meio ambiente mais sustentável. A programação paralela tem forte influência nesta área [Vykoukal J. and R. 2009], pois uma distribuição mais otimizada das tarefas entre os *cores* disponíveis aumenta o desempenho e reduz o consumo de energia. A diminuição do tempo gasto na execução de tarefas reflete diretamente na quantidade de energia consumida em uma computação paralela.

Segundo [Stéfano D. K. 2010], o objetivo da programação paralela para a economia de energia é obter uma melhor forma de aproveitamento dos recursos disponíveis, utilizando de técnicas de escalonamento de tarefas nos *cores* e do controle da granularidade. Em seu artigo, sobre o papel da programação paralela no aproveitamento dos recursos energéticos, demostra que um algoritmo paralelo bem programado é capaz de utilizar os recursos à disposição de maneira mais eficiente, distribuindo melhor as tarefas entre os *cores* disponíveis, aumentando ao máximo o uso destes, diminuindo sua ociosidade.

6. Proposta

STM-Haskell tem se mostrado uma alternativa interessante para a programação concorrente. O objetivo deste trabalho é avaliar a eficiência, tanto no desempenho de processamento, quanto no consumo de energia, para um conjunto de aplicações reais escritas em STM Haskell e compará-las com aplicações escritas no modelo tradicional.

Serão utilizadas aplicações do *STM Haskell BenchMark* [Perfumo et al. 2007], desenvolvidas utilizando tanto o método de bloqueio tradicional, quanto o de memorias transacionais. Este conjunto de aplicações servirão para realizar os testes de desempenho e comparações entre os modelo transacionais e tradicional.

Também serão utilizados programas como o *Threadscope* [Donnie Jones 2011], para verificar como as tarefas estão sendo distribuídas e escalonadas entre os *cores* disponíveis para a avaliação dos testes, bem como a utilização de equipamentos, para a medição do consumo de energia.

7. Agradecimentos

O primeiro autor gostaria de agradecer ao CNPq pela bolsa PIBIC, concedida ao projeto. Este trabalho foi financiado pelos projetos FAPERGS/PRONEX/CNPq GREEN-GRID e FAPERGS/PESQUISADOR GAÚCHO CMTJava.

Referências

Donnie Jones, Simon Marlow, S. S. (2011). threadscope 0.1.1. Disponível em: <http://research.microsoft.com/threadscope/>. Acesso em: 16 nov. 2011.

Hammond, K. and Michelson, G., editors (2000). *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK.

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

Jones, S. P. (2007). Beautiful concurrency. *Beautiful Code*, (1):1–24.

Lee, E. A. (2006). The problem with threads. *Computer*, 39:33–42.

Perfumo, C., Sonmez, N., Cristal, A., Unsal, O. S., Valero, M., and Harris, T. (2007). Dissecting transactional executions in haskell. In *In The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*.

Rajwar, R. and Goodman, J. (2003). Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23:117–125.

Rigo S., C. P. and A., B. (2007). Memorias transacionais, uma nova alternativa para programação concorrente. *In Anais Eletrônicos do WSCAD*.

Silberschatz. Abraham, G. P. and Gagne, G., editors (2001). *Sistemas Operacionais - Conceitos e Aplicações*. Elsevier, São Paulo, BR.

Stéfano D. K., M. A. Z. A. e. J. V. L. (2010). Eficiência energética em computação de alto desempenho: Uma abordagem em arquitetura e programação para green computing. *SEMISH - XXXXVII Seminário Integrado de Software e Hardware.*

Tanenbaum, A. S. and Woodhull, A. S. (2006). *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall.

Vykoukal J., W. M. and R., B. (2009). Does green it matther? analysis of the relationship between green it and grid technology from a resource-based view perspective. *In PACIS*.