Verificando a Interferência do Escalonador do Glasgow Haskell Compiler em Aplicações Usando STM Haskell

Rodrigo M. Duarte[1], **André R. Du Bois**[1]
Gerson Geraldo H. Cavalheiro[1], **Mauricío Lima Pilla**[1]

[1]Universidade Federal de Pelotas (UFPel)
CDTec - Centro de Desenvolvimento Tecnológico
PPGC - Programa de Pós-Graduação em Computação
LUPS - Laboratory of Ubiquitous and Parallel Systems

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

Resumo. *Apesar do Glasgow Haskell Compiler (GHC) ser constante atualizado, o escalonador interno de seu runtime system ainda não faz um tratamento distinto para aplicações usando o STM-Haskell. Este trabalho faz a verificação do impacto do escalonador do GHC sobre aplicações usando STM-Haskell em diferentes cenários. Os resultados demostram que o escalonador apresenta comportamentos distintos para diferentes aplicações e que, desativa-lo pode ser uma opção viável para não se perder desempenho.*

1. Introdução

Haskell é uma linguagem de programação funcional que apresenta varias características que facilitam a programação concorrente. Entre elas esta o STM-Haskell, que é uma extensão da linguagem que fornece a abstração de memórias transacionais.

Memória transacional (MT) é um modelo de sincronização entre *threads* que utiliza o conceito de transações, parecidas com as presentes em bancos de dados, para garantir a correta sincronização entre *threads*. Neste modelo o programador somente define o bloco que deve ser sincronizado e o sistema transacional fica a cargo de realizar a sincronização, isentando o programador desta tarefa.

O compilador e máquina virtual estado da arte para Haskell é o *Glasgow Haskell Compiler* (GHC). Este possui em seu *runtime* (RTS) uma implementação completa para programação paralela, incluindo um sistema de escalonamento de *threads* e o STM-Haskell.

Apesar do GHC ser constantemente atualizado, o mesmo ainda não possui um escalonador especifico para MT, o que dependendo do tipo de aplicação, pode levar a uma perda de desempenho [Nicácio et al. 2013]. O objetivo deste trabalho é averiguar qual o impacto que o escalonador presente no GHC tem sobre aplicações rodando MT, verificando qual a sua interferência no desempenho e na quantidade de cancelamentos. Como pode ser visto nos resultados, o escalonador interfere de formas distintas para diferentes aplicações de MT e que, desativa-lo em alguns casos pode fazer o sistema manter o desempenho.

2. Glasgow Haskell Compiler

GHC (*Glasgow Haskell Compiler* [Peyton-Jones et al. 2019] é um compilador, interpretador e máquina virtual para Haskell. Este possui a implementação de uma biblioteca

de MT (implementada em sua máquina virtual) que é o STM-Haskell. Nesta, um novo tipo de variável transacional é definida (`TVar`) [Marlow 2013]. Uma `TVar` só pode ser acessada por duas funções que são a `readTVar` (leitura) e `writeTVar` (escrita).

Ações de STM, que são combinações das primitivas básicas, só podem ser executadas dentro de uma chamada atômica (`atomically`). O sistema de tipos do Haskell garante que operações transacionais não serão executadas de forma incorreta (e.g. programador tenta modificar uma `TVar` fora de uma transação). Isso ajuda o programador a corrigir erros já no momento da compilação.

Porém, apesar das qualidades citadas anteriormente, o RTS do GHC não possui um escalonador especifico para transações. Outro item a salientar, é que o escalonador interno do GHC implementa o modelo *round robin*, que segundo trabalhos relacionados [Bai et al. 2007, Maldonado et al. 2010, Zhou et al. 2016], pode não ser o melhor escalonador para MT.

3. Testes Realizados

Os testes realizados foram a comparação da execução de duas aplicações do *STM-Haskell Benchmark* [Perfumo et al. 2007], sendo uma de alta e outra de média contenção, com e sem a utilização do escalonador interno do GHC. Para isso uma *flag* foi passada para o RTS, informando que o escalonador deveria ser desabilitado. Também foi realizado outro teste, limitando a quantidade de *cores* que eram disponíveis para rodar a aplicação. O objetivo destes testes foi verificar qual o comportamento das aplicações, quando estas possuem mais *threads* do que *cores*, em conjunto com o desligamento do escalonador.

As aplicações testadas foram a SI, que consta de um inteiro compartilhando que é incrementado dez milhões de vezes e a LL, que é uma lista encadeada onde são realizadas operações de inserção e remoção de valores aleatórios, em um total de 100 mil operações.

A plataforma de testes foi um i9-7900X de 10 *cores* com 32Gb RAM, sistema operacional Debian 10 64Bits, Foi utilizado o GHC 8.6.3 com a biblioteca de STM-Haskell 2.5.0.0. Os testes constaram de 30 execuções de cada aplicação com 1, 2, 4, 8, 16 e 32 threads nas seguintes configurações:

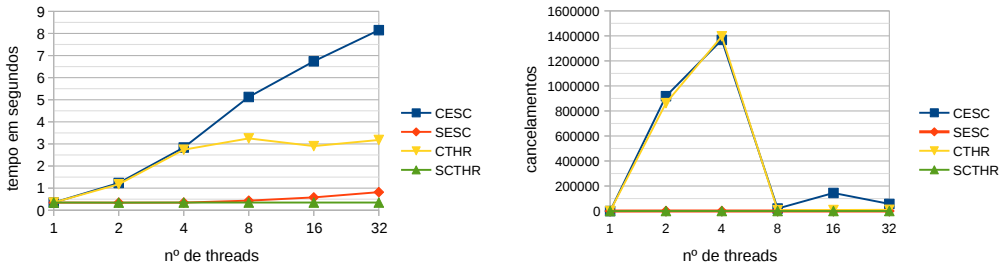
- CESC: Execução normal das aplicações com o escalonador ativo.
- SESC: Execução sem o escalonador.
- CTHR: Execução limitando em 4 *cores* com o escalonador ativo.
- SCTHR: Execução limitando em 4 *cores* sem o escalonador.

4. Resultados

Os resultados obtidos na aplicação SI demostram que desligar o escalonador impactou positivamente no tempo de execução, como pode ser visto na Figura 1a. Como esperado os tempos de execução não foram melhores que os da execução sequencial (devido a alta contenção da aplicação). Porém, ao se desligar o escalonador (casos SESC e SCTHR), obteve-se um tempo de execução quase constante para todas as quantidades de *threads* usadas, sendo o melhor resultado com o controle de *cores* sem o escalonador (SCTHR).

Outro item a se observar é que a quantidade de cancelamentos não esta relacionada diretamente ao desempenho do sistema transacional. Isso pode ser observado nos casos

CESC e CTHR na Figura 1b (onde o escalonador está ativo) em que a quantidade de cancelamentos diminui a partir de 8 *threads*. A explicação para este comportamento esta relacionado ao fato de que, como as transações são pequenas (incremento de um inteiro), usar uma quantidade maior de *threads* que *cores*, faz o sistema começa a ter mais *overhead* na gerência das *threads* (trocas de contexto pelo escalonador) do que nos reinícios das transações. Além disso, assim como indicado em outros trabalhos [Zhou et al. 2016], a serialização das transações acaba levando a redução dos cancelamentos.

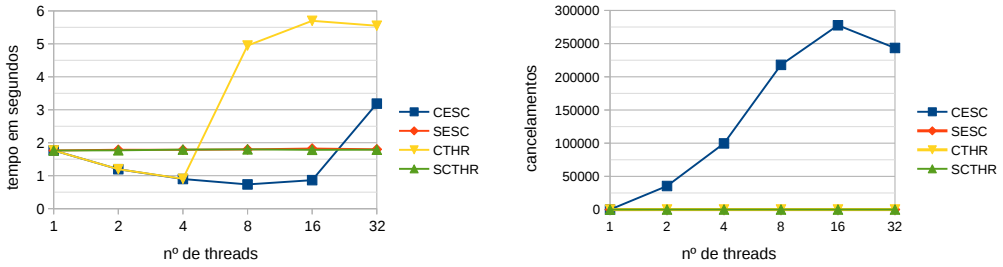


(a) SI Tempo **(b) SI Cancelamentos**

Figura 1. Resultados da aplicação SI

Já nos resultados obtidos com a aplicação LL, o sistema tem um comportamento diferente da aplicação anterior. Os melhores resultados foram obtidos com a configuração normal do RTS (sem controle de *threads* e com o uso do escalonador (CESC)), como pode ser observado na Figura 2a e, mesmo com a alta quantidade de cancelamentos, esta configuração foi a que obteve melhores resultados até 8 *threads*. Os piores resultados foram obtidos com o controle da quantidade de *cores* com o uso do escalonador (CTHR), onde novamente a perda de desempenho esta relacionada as trocas de contexto das *threads* pelo escalonador do RTS. Isto pode ser comprovado pela (Figura 2b), onde mesmo com uma baixa taxa de cancelamentos, há degradação do desempenho.

Também como esperado, o desligamento do escalonador fez com que as *threads* fossem serializadas pelo sistema, levando a um tempo quase constante independente do número de *theads*, como observado nos casos de SESC e SCTHR.



(a) LL Tempo **(b) LL Cancelamentos**

Figura 2. Resultados da aplicação LL

5. Conclusão e Trabalhos Futuros

Os testes demostraram que o STM-Haskell apresenta comportamento distinto para diferentes aplicações, o que já era esperado. Porém, a utilização ou não do escalonador interno

do RTS do GHC também altera de forma significativa o comportamento das aplicações usando STM. Utilizar o escalonador na aplicação SI levou a um perda de desempenho e a uma alta quantidade de cancelamentos. Já na aplicação LL, mesmo com a alta taxa de cacelamentos, o uso do escalonador apresentou os melhores resultados. Pode-se perceber assim que usar o escalonador do GHC não é sempre a melhor opção, desliga-lo quando a quantidade de *threads* é maior que o número de *cores* disponíveis e a aplicação possui alta contenção, também é uma estrategia viável para não degradar o desempenho.

Como trabalhos futuros pretende-se desenvolver um escalonador de transações para o RTS do GHC que, além da quantidade de cancelamentos das transações, leve em conta fatores como a desativação das trocas de contexto das *threads* pelo escalonador interno do RTS e o controle da quantidade de *threads* por recursos disponíveis (*cores*).

Agradecimentos

O presente trabalho foi realizado com apoio do Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES/Brasil 88882.151433/2017-01

Referências

Bai, T., Shen, X., Zhang, C., Scherer III, W. N., Ding, C., and Scott, M. L. (2007). A key-based adaptive transactional memory executor. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE.

Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J. L., and Muller, G. (2010). Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA. ACM.

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

Nicácio, D., Baldassin, A., and Araújo, G. (2013). Transaction scheduling using dynamic conflict avoidance. *International Journal of Parallel Programming*, 41(1):89–110.

Perfumo, C., Sonmez, N., Cristal, A., Unsal, O., Valero, M., and Harris, T. (2007). Dissecting transactional executions in haskell. In *TRANSACT 07: Second ACM SIGPLAN Workshop on Transactional Computing*, Portland, Oregon, USA. ACM.

Peyton-Jones, S., Marlow, S., et al. (2019). Glasgow haskell compiler. Disponivel em https://www.haskell.org/ghc/. Acesso em: Janeiro de 2019.

Zhou, N., Delaval, G., Robu, B., Rutten, E., and Méhaut, J.-F. (2016). Autonomic parallelism and thread mapping control on software transactional memory. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 189–198. IEEE.