Paralelismo na linguagem Haskell

Vagner Franco Pereira[1], **Rodrigo Machado**[1], **Lucas Mello Schnorr**[1]
{vfpereira,rma,schnorr}@inf.ufrgs.br

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

Resumo. *Esse trabalho avalia se a linguagem funcional Haskell é interessante no contexto de programação paralela. Para tal, foram desenvolvidas implementações sequenciais e paralelas de algoritmos em Haskell e analisadas as duas abordagens de implementação. Adicionalmente, foi analisado o desempenho de um mesmo algoritmo implementado em Haskell e em ANSI C. Resultados apontam que Haskell é interessante para programação paralela.*

1. Introdução

Linguagens imperativas foram preferencialmente utilizadas ao longo dos anos, muito possivelmente por serem mais próximas às arquiteturas de computadores baseadas no modelo de John von Neumann [von Neumann 1981]. Porém, no novo contexto de utilização de máquinas paralelas, as linguagens funcionais apresentam algumas características mais convenientes em relação às linguagens imperativas. Por exemplo, a transparência referencial [Sondergaard and Sestoft 1990] evita a existência de condições de corrida na execução dos programas paralelos. Percebe-se que as linguagens funcionais ganham bastante espaço atualmente, sendo que muitas linguagens imperativas já incorporam algumas de suas características como, por exemplo, funções de alta ordem.

O objetivo deste trabalho é avaliar a aplicabilidade da linguagem de programação funcional Haskell para computação paralela. Para tal, são implementados algoritmos de forma sequencial e paralela na linguagem e é feita uma análise sobre os resultados obtidos. O desempenho das implementações é verificado, e, em alguns casos, comparado com uma aplicação semelhante desenvolvida em uma linguagem imperativa.

2. Haskell

Haskell é uma linguagem de programação funcional pura e de código aberto, do qual o nome é uma homenagem ao matemático Haskell B. Curry. A linguagem possui compiladores e interpretadores, tendo como principal o compilador Glasgow Hakell Compiler(GHC) [Marlow 2014]. O GHC, compilador que será usado nos experimentos do artigo, é um compilador otimizado escrito na própria linguagem Haskell e atualmente está na versão 7.8.2. Entre suas diversas funções, o GHC oferece suporte a paralelismo e concorrência.

A linguagem Haskell permite aplicação de estratégias diferenciadas como (*lazy evaluation*) [HaskellWiki 2014]. Esta consiste em uma técnica de avaliação de funções não estrita que permite ao Haskell, por exemplo, postergar a avaliação de uma expressão para ser mais eficiente (ao evitar cálculos desnecessários e condições de erro na avaliação de expressões compostas) ou construir estruturas como listas infinitas. Para Haskell lidar com eventos como Entrada/Saída, exceções, geração de números randômicos, e outros que produzem efeitos colaterais, um outro conceito matemático foi introduzido: as

mônadas [Wadler 1995]. As mônadas em Haskell permitem a simulação de características impuras, integrando-as com o restante da linguagem.

2.1. Paralelismo em Haskell

O paralelismo em Haskell é relativamente recente e ainda é um tópico de pesquisa, havendo diversas bibliotecas para tal. Cada uma dessas bibliotecas apresenta características que podem ser mais adequadas dependendo do problema a ser resolvido. As bibliotecas mais utilizadas são: `Control.Parallel.Strategies`, `Control.Monad.Par`, `Repa` e `Accelerate` [Marlow 2013]. A primeira foi a escolhida para os experimentos, por utilizar a *lazy evaluation* e gerenciar de forma implícita a dependência de dados.

O módulo `Control.Parallel.Strategies` [Haskell 2014] tem como propósito fornecer uma interface para o desenvolvedor expressar paralelismo. O tipo da estratégia é representado por: `type Strategy a = a -> Eval a`. Onde `type` representa a declaração de tipo, *strategy* é uma abstração da linguagem que tem como objetivo descrever como avaliar de forma paralela uma expressão de um determinado tipo e `Eval` é uma mônada para expressar ordem de avaliação. O parâmetro `a` representa um tipo polimórfico. O módulo oferece `rpar` e `rseq` como suas principais estratégias.

A estratégia `rpar` cria um `spark` [Marlow 2013] para avaliar seu argumento em paralelo. O `spark` é um comportamento dinâmico que indica, em tempo de execução, que o seu argumento pode ser avaliado por uma *thread* em paralelo. A estratégia `rseq` avalia o seu argumento, e por estar dentro da mônada Eval ela avalia de forma sequencial. O uso mais comum de `rseq` é o de criar uma sincronização para as *threads*. Quando os *sparks* são criados eles são armazenados em uma piscina (*pool*) que tem um tamanho fixo. Se esse tamanho for extrapolado, os `sparks` excedentes são descartados. Os *sparks* podem também ser descartados pelo coletor de lixo, quando o argumento não estiver sendo utilizado no programa. Existem outras situações onde os `sparks` não criam paralelismo, algumas delas podem ser visualizadas no GHC utilizando-se o parâmetro `-rtsopts` na compilação e `-s` na execução. Ambas as estratégias avaliam seus argumentos apenas até o primeiro construtor. Essa forma de avaliação é chamada de *weak head normal form* (WHNF) [Marlow 2013]. Em muitas situações é necessário forçar a avaliação de uma expressão para se obter o máximo de paralelismo dela, o módulo oferece uma variação da estratégia `rseq` para esse caso a `rdeepseq`.

O módulo `Control.Parallel.Strategies` oferece paralelismo mantendo a linguagem pura, dessa forma situações de corrida não ocorrem. Essa característica elimina grande parte dos erros dos programas paralelos. A forma de ligar o resultado da mônada Eval com o resto do código é através do operador `runEval`.

3. Metodologia

Escolhemos inicialmente dois problemas e implementamos uma versão sequencial e paralela em Haskell para cada problema. O primeiro foi o cálculo de Mandelbrot, escolhido por ser fácil de paralelizar, e sem dependência dos dados. O segundo foi o n-Corpos gravitacional por ter uma complexidade de implementação maior que o primeiro problema e apresentar dependência de dados.

A análise foi feita executando-se 50 vezes o mesmo algoritmo para se obter uma média do tempo de execução e o desvio padrão. O objetivo disso é diminuir o efeito dos fatores externos do ambiente de execução. Serão comparados os tempos de execuções dos experimentos do algoritmo sequencial com os do algoritmo paralelo em Haskell. Também foram obtidos tempos de execução de versões sequenciais e paralelas do algoritmo de cálculo de Mandelbrot em Ansi C. A partir desses tempos será determinado o ganho de desempenho do algoritmo paralelo. Como forma de analisar o efeito do paralelismo, os algoritmos foram compilados com as otimizações do compilador (-O3) e procurou-se particionar os dados de forma a ter uma distribuição nas *threads* próximo ao ideal. Os códigos dos experimentos podem ser encontrados em `https://github.com/vfpereira/`.

4. Cálculo do fractal de Mandelbrot

O fractal de Mandelbrot é um conjunto matemático de pontos que apresenta algumas características apropriadas para o uso de linguagens funcionais e paralelismo. Algumas dessas características são: o conjunto pode ser definido recursivamente, o cálculo de um ponto não depende do cálculo dos outros pontos.

A Figura 1a mostra o tempo médio dos algoritmos com o número de pontos fixos em 4198401, onde é possível perceber que o algoritmo sequencial em Haskell é ligeiramente mais rápido. Contudo, o inverso ocorre quando consideramos o algoritmo paralelo. O tempo a partir de 8 `sparks` praticamente se mantém igual, ao passo que em Ansi C esse tempo continua decrescendo. Isso é um indicativo de que a linguagem Ansi C apresenta uma melhor eficiência nesse contexto. Como uma das razões para tal, pode-se se citar o coletor de lixo, que é bastante punitivo para desempenho na linguagem Haskell. Outro fator é o uso de `sparks` para paralelismo na biblioteca utilizada em Haskell. Os `sparks` diferem um pouco das `threads` utilizadas em Ansi C: durante o tempo de execução, o sistema de execução mantêm os `sparks` em um piscina, não sendo necessariamente executados imediatamente. Quando o GHC detecta algum processador ocioso então o `spark` vira uma `thread` num processo conhecido como *work stealing*. Pelo motivo citado, não existe uma garantia que o número de `sparks` criados seja igual ao número de processadores que estarão executando o cálculo de Mandelbrot paralelamente.

5. Simulação n-Corpos gravitacional

O algoritmo n-Corpos gravitacional consiste de simular a movimentação de um conjunto de corpos no espaço sob influência da gravidade mútua. O algoritmo apresenta dependência de dados, uma vez que, a cada instante de tempo é necessário calcular a força sobre os corpos para então realizar um movimento, sendo possível calcular novamente as forças somente após todos os corpos terem realizado seu movimento anterior. A implementação realizada foi a que apresenta uma complexidade computacional quadrática.

Para implementação do algoritmo, fixou-se o número de corpos igual a 600 e o número de passos em 100, onde cada passo representa 60 minutos. A Figura 1b apresenta um comportamento semelhante ao observado no algoritmo de Mandelbrot. Como explicação desse resultado, pode-se citar distribuição das cargas de trabalho próximas ao ideal e a não existência de partes sequenciais no código.

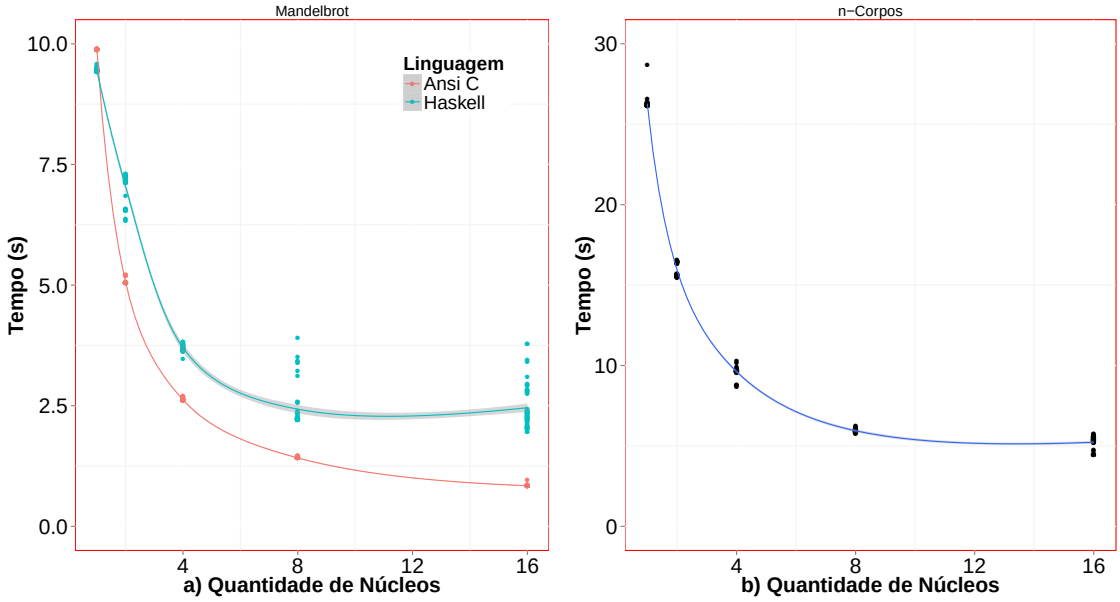

Figura 1. a) Cálculo do fractal de Mandelbrot.
Figura 1. b) Simulação n-Corpos gravitacional.

6. Conclusão

Conforme os experimentos realizados, percebe-se que Haskell consegue resultados muito bons ao se utilizar paralelismo. Entretanto, ao se aproximar do número de processadores da máquina, o ganho em desempenho diminui gradativamente. Quando comparado à linguagem Ansi C, que é bastante utilizada para paralelismo, Haskell obtém tempos razoavelmente próximos, fornecendo abstrações de mais alto nível ao programador. Pelos resultados alcançados, pode-se considerar Haskell uma linguagem interessante para o desenvolvimento de aplicações paralelas.

Referências

Haskell (2010 (acessado Mai 26, 2014)). *Control.Parallel.Strategies*. `http://hackage.haskell.org/package/parallel-parallel-3.1.0.1/docs/Control-Parallel-Strategies.html`.

HaskellWiki (2012 (acessado Mai 25, 2014)). *Lazy vs. non-strict*. `http://www.haskell.org/haskellwiki/Lazy_vs._non-strict`.

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

Marlow, S. (2014 (acessado Mai 25, 2014)). *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc/`.

Sondergaard, H. and Sestoft, P. (1990). Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517.

von Neumann, J. (1981). First draft of a report on the edvac. Digital Press.

Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52. Springer-Verlag.