UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

VAGNER FRANCO PEREIRA

Paralelismo na linguagem Haskell

Trabalho de Conclusão apresentado como
requisito parcial para a obtenção do grau de
Bacharel em Engenharia de Computação

Prof. Dr. Rodrigo Machado
Orientador

Prof. Dr. Lucas Mello Schnorr
Co-orientador

Porto Alegre, 12 de Dezembro de 2014

A persistência é o menor caminho do êxito.

— Charles Chaplin

AGRADECIMENTOS

Gostaria de agradecer a todos que contribuíram para realização desse trabalho. Em especial, gostaria de agradecer a minha família pelo apoio em todos os momentos, e aos meus orientadores pela dedicação e ensinamentos que foram muito importantes durante o desenvolvimento desse trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

ACRÔNIMOS

GHC Glasgow Haskell Compiler.

PThreads Posix Threads.

OpenMP Open Multi-Processing.

MPI Message Passing Interface.

WHNF Weak Head Normal Form.

Repa Regular Parallel Arrays.

DPH Data Parallel Haskell.

STM Software Transactional Memory.

GPU Graphics processing unit.

ABSTRACT

The development of parallel algorithms is a complex and error-prone task, requiring a considerable effort from the point of view of the developers. The aim of this work is to evaluate if some of the difficulties regarding the development of parallel algoritms can be eased by using the purely functional programming language Haskell. For such, serial and parallel versions of the same algorithm in Haskell are compared with respect to performance and some considerations about readability. Additionally, implementations of the same algorithm in Haskell and in an imperative programming language are compared. Finally, results are presented.

Keywords: Haskell, Parallelism, Functional Language, Mandelbrot, nBody.

Paralelismo na linguagem Haskell

RESUMO

O desenvolvimento de algoritmos paralelos é uma tarefa complexa e requer grande esforço por parte dos desenvolvedores. Esse trabalho avalia se algumas das dificuldades presentes no desenvolvimento de algoritmos paralelos são reduzidas utilizando-se a linguagem de programação funcional pura Haskell. Para tal, propõe-se comparar implementações sequenciais e paralelas de algoritmos em Haskell considerando desempenho e algumas ponderações sobre legibilidade. Adicionalmente, propõe-se comparar implementações de um mesmo algoritmo em Haskell e em uma linguagem imperativa. Por último, resultados são apresentados.

Palavras-chave: Haskell, Paralelismo, Linguagens Funcionais, Mandelbrot, n-Corpos.

1 INTRODUÇÃO

O aumento de desempenho é um dos principais fatores na maioria dos projetos de sistemas computacionais. Tanto no nível de *hardware* quanto no nível de *software* tem se buscado diversas formas de atingir esse objetivo. Aumento da frequência dos circuitos, desenvolvimento de pipelines, compiladores que otimizem os códigos, e diversas outras técnicas tem sido estudadas ao longo dos anos para esse fim. Uma alternativa para ganho de desempenho, que vem ganhando bastante destaque atualmente, é o uso do paralelismo.

Embora o conceito de paralelismo em computação seja antigo, observa-se mais recentemente o início de um período de transição da computação sequencial para computação paralela: processadores multi-cores já são uma realidade no mercado. A tendência mostra que o paralelismo é o futuro da computação (ASANOVIC et al., 2009). Apesar de se conhecer as vantagens da computação paralela, como executar mais de uma instrução no mesmo ciclo de *clock*, o principal fator dessa mudança de paradigma é que se está chegando ao limite da computação sequencial em questão de desempenho. A lei de Moore (BROCK; MOORE, 2006) descreve a duplicação do número de transistores em circuitos integrados a cada dois anos. Alguns fatores como memória e potência fazem com que a lei de Moore não possa mais ser satisfeita pela computação sequencial. O maior desafio e motivo pelo qual a computação paralela não é dominante atualmente é a sua dificuldade. Questões como o não determinismo das aplicações tornam o desenvolvimento de programas de computadores (*softwares*) uma tarefa bastante complexa.

Linguagens imperativas foram preferencialmente utilizadas ao longo dos anos possivelmente por terem mais semelhanças com as arquiteturas de computadores baseadas no modelo de John von Neumann (NEUMANN, 1981). Porém, nesse novo contexto de utilização de computações paralelas, as linguagens funcionais apresentam algumas características mais convenientes em relação as linguagens imperativas, tal como a transparência referencial. Transparência referencial é uma propriedade que permite que uma função ou expressão seja substituída por outra de igual valor. Percebe-se que as linguagens funcionais ganham bastante espaço atualmente e muitas linguagens imperativas incorporam aspectos das linguagens funcionais como, por exemplo, funções de alta ordem. Funções de alta ordem são funções que recebem outras funções como parâmetros e retornam funções como resultado.

Haskell é uma linguagem que representa bem o paradigma funcional. Por ser uma linguagem pura, têm caraterísticas como *lazy evaluation* (avaliação preguiçosa) e mônadas, que apresentam soluções que diferem bastante das linguagens imperativas. Linguagens funcionais puras se caracterizam por utilizarem apenas funções que não produzem efeito colateral. Por essa razão, apresenta novas soluções no desenvolvimento de aplicações pa-

ralelas, entretanto, o suporte ao paralelismo em Haskell é relativamente recente e ainda esta em evolução. Esse trabalho se propõe a analisar soluções em Haskell e verificar se elas simplificam o desenvolvimento de aplicações paralelas.

1.1 Objetivos

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. Fatores como desempenho e implementação são verificados, e em alguns casos comparados com uma aplicação semelhante desenvolvida em uma linguagem imperativa.

Como forma de atingir o objetivo do trabalho é realizado um grande número de análises experimentais. Através dos experimentos realizados busca-se descobrir os principais fatores que afetam o desempenho de programas paralelos em Haskell. Com o conhecimento desses fatores novas implementações podem ser realizadas como forma de otimizar o desempenho.

1.2 Estrutura do Trabalho

O capítulo 2 apresenta um pequeno resumo sobre as linguagens funcionais e especificamente a linguagem Haskell. O capítulo 3 apresenta alguns conceitos sobre paralelismo e seu uso em Haskell através de bibliotecas da linguagem. O capítulo 4 apresenta os algoritmos que serão analisados no projeto e a métodologia que será utilizada nas análise. Os capítulos 5 e 6 apresentam o cálculo do fractal de Mandelbrot e a simulação n-Corpos gravitacional e suas análises. E, por fim, o capítulo 7 traz uma conclusão sobre a experiência da utilização do Haskell para obtenção desses resultados. Os apêndices no fim do trabalho trazem as implementações dos códigos escritos na linguagem Haskell e Ansi C.

2 LINGUAGENS FUNCIONAIS

2.1 Linguagens Funcionais em Geral

Linguagens Funcionais têm as suas origens nas funções matemáticas e no cálculo Lambda (CHURCH, 1932), um modelo de computação baseado na definição e aplicação de funções.

Pode-se classificar as linguagens funcionais de duas formas: as linguagens funcionais puras e as linguagens funcionais impuras. As linguagens funcionais puras se caracterizam por utilizarem funções que não produzem efeitos colaterais, funções puras, e, portanto, não carregam nenhum estado internamente. Em uma função pura sempre se obtêm o mesmo resultado da função na saída ao se utilizar os mesmos parâmetros de entrada. Isso é uma vantagem, pois, devido a essa ausência de efeitos colaterais as funções não produzem nenhuma dependência implícita entre elas, sendo possível executá-las em qualquer ordem sem alterar seu resultado. Outra vantagem das funções puras é a transparência referencial. Transparência referencial é a propriedade de substituir uma expressão por outra que produz o mesmo resultado, isso é possível devido ao fato da expressão sempre produzir o mesmo resultado. Em linguagens funcionais puras não existem variáveis, o que caracteriza os dados como sendo imutáveis. Uma consequência dessa imutabilidade é a inexistência de laços, a forma de lidar com a iteração é através da recursão. Alguns exemplos de linguagens funcionais puras são Haskell (ERISSON, 2010) e Miranda (TURNER, 1987).

Linguagens funcionais impuras se diferenciam por usarem, além das funções puras, funções que produzem efeitos colaterais (WADLER, 1992). Características como transparência referencial e independência da ordem de avaliação não se aplicam a essas funções. Porém, permite-se o uso de atribuições e variáveis. A vantagem de se utilizar linguagens funcionais impuras está relacionada à eficiência: durante muito tempo as linguagens funcionais impuras foram consideradas como tendo um desempenho superior. Outro fator que pode ter alguma influência é aproximar um pouco a linguagem funcional da imperativa. Exemplos de linguagens funcionais impuras são ML (MILNER; TOFTE; MACQUEEN, 1997) e Scheme (SUSSMAN; STEELE JR., 1998).

Expressões e funções são os principais componentes de uma linguagem funcional. Cada linguagem funcional implementa uma estratégia que determina como os parâmetros dentro de uma chamada de função serão avaliados. Estratégias podem ser estritas ou não-estritas. Em uma avaliação não-estrita os parâmetros são avaliados apenas quando o valor do resultado da função é realmente utilizado na aplicação, podendo eventualmente nunca serem avaliados. Em uma avaliação estrita os parâmetros da função são sempre avaliados

antes da chamada da função.

Uma última característica a ser ressaltada em linguagens funcionais é possibilidade de se definir funções de alta ordem. Essas funções recebem como argumentos outras funções deixando dessa forma o código mais simples e modular.

2.2 Haskell

Haskell é uma linguagem de programação funcional pura e de código aberto, cujo 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 que atualmente está na versão 7.8.2. Entre suas diversas funções o GHC oferece suporte a paralelismo e concorrência.

A linguagem Haskell é conhecida por ser uma linguagem preguiçosa (*lazy*) (HASKELLWIKI, 2012). Avaliação preguiçosa é uma técnica de avaliação não estrita, que permite ao Haskell, por exemplo, postergar a avaliação de uma expressão para ser mais eficiente ou construir estruturas como listas infinitas. Embora essa técnica traga algumas vantagens, como não avaliar valores que não são necessários, ela requer alguns cuidados, visto que, a linguagem pode acabar alocando mais memória do que o necessário na execução de um programa. Como consequência dos dados serem imutáveis em Haskell, muitos valores temporários são gerados e armazenados. A memória em Haskell é gerenciada automaticamente, sendo responsabilidade do coletor de lixo a liberação de espaços de memória que não serão mais utilizados. Haskell permite a avaliação estrita também embora esta estratégia não seja a padrão da linguagem, é possível graças a algumas bibliotecas. O GHC também oferece uma técnica de profiling (ERISSON, 2014) que tem como propósito oferecer uma análise de uso da memória.

Haskell é uma linguagem fortemente tipada (LIPOVACA, 2011): todas as funções e expressões têm um tipo. O compilador pode inferir em algumas situações o tipo se esse não for declarado. Todos os tipos são estáticos, ou seja definido durante a compilação. Essas características tornam os programas menos suscetíveis a erros. Haskell oferece os tipos de dados básicos que a maiorias das outras linguagens oferecem como inteiros, booleanos, caracteres, entre outros, e oferece tipos compostos, sendo os principais as listas e tuplas.

Pode-se destacar na linguagem o casamento de padrões (*pattern matching*) que é uma construção sintática que auxilia nas construções de funções e recursões. O casamento de padrões torna as funções mais simples e flexíveis, evitando o uso demasiado do construtor *if then else*. Sobre as funções, Haskell permite o uso de funções anônimas utilizando o operador "\". Funções anônimas tem o propósito de deixar o código mais limpo. Funções com mais de um parâmetro em Haskell passam por um processo de currificação[1] (*currying*) (LIPOVACA, 2011). Esse processo consiste em aplicar a função a um argumento e obter seu resultado, criando uma nova função que é aplicada ao próximo parâmetro. Isso permite a construção de funções parcialmente aplicadas. A forma como Haskell lida com funções com mais de um parâmetro mostra outra característica das funções em Has-

[1] Possível tradução para o termo currying, visto que não foi encontrado na literatura uma tradução mais clara.

kell: elas aceitam outras funções como parâmetros e retornam funções como resultado, gerando assim funções de alta ordem. Uma função de alta ordem bastante utilizada em Haskell é a função `map`. A declaração do tipo da função `map` é

```
map :: (a -> b) -> [a] -> [b]
```

(LIPOVACA, 2011). Pela declaração da função observa-se como primeiro argumento uma função que recebe um tipo `a` e retorna um tipo `b`. A função `map` aplica uma função aos elementos de uma lista, que é o seu segundo parâmetro, retornando uma nova lista com o resultado da função aplicada. Os parâmetros das funções em Haskell não necessitam ter um tipo definido, como no caso da função `map`. Pode-se então criar funções genéricas que podem ser reutilizadas. Essa forma de polimorfismo é outro ponto forte da linguagem.

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 (MOGGI, 1991). Mônadas são as construções que Haskell utiliza para lidar com funções que tem efeito colateral. As mônadas em Haskell obedecem algumas regras e simulam a parte impura, isolando-as do resto da linguagem. As operações dentro da mônada obedecem uma ordem de execução. O que ocorre na prática é que toda parte impura ocorre dentro da estrutura da mônada e depois seu resultado é ligado à parte pura do código. Pode-se então utilizar esse resultado sem qualquer tipo de efeito colateral, mantendo o código puro.

Abaixo é apresentado um programa em Haskell que calcula e imprime o décimo número da sequência de Fibonacci.

```
import System.Environment

fib::Int->Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main :: IO ()
main = do
 let f1 = fib 5
 let f2 = fib 10
 print  f2
```

O código apresenta duas funções: a função de cálculo da sequência de Fibonacci e a função `main`. A função `main` é a função principal do programa e é encapsulada pela mônada de `IO`. A mônada é necessária pelo fato da função lidar com eventos de entrada e saída, no caso a função `print`. A atribuição `let f1 = fib 5` não é avaliada pelo programa devido à avaliação preguiçosa da linguagem. O corpo da função `fib` é declarado através de três equações, sendo que, o uso do casamento de padrões elimina a necessidade de expressões `if-then-else` na definição.

3 PARALELISMO

3.1 Conceitos

Paralelismo na computação significa executar mais de uma tarefa ao mesmo tempo para resolução de um problema, com o objetivo de se obter um melhor desempenho ou melhor utilizar os recursos. Existem fatores que não permitem que ao se duplicar o número de processadores, consiga-se dividir o tempo de processamento na mesma proporção. Ao se realizar uma computação em paralelo existem questões como o gerenciamento das tarefas, a dependência entre elas, e outros fatores que acabam consumindo tempo de processamento. Algumas métricas como aceleração (*speedup*) e eficiência (WILKINSON; ALLEN, 1999) são utilizadas para se analisar o ganho de desempenho que uma aplicação paralela tem em relação a uma computação sequencial. A Lei de Amdahl (WILKINSON; ALLEN, 1999) fala sobre o ganho de desempenho máximo que é possível de uma aplicação paralela obter em relação a uma computação sequencial.

Aumento de desempenho é um dos principais objetivos da computação paralela. Nesse trabalho o paralelismo será explorado com foco em desempenho e utilizando uma arquitetura de memória compartilhada, que utiliza um espaço de endereçamento único para todos os processadores. Como o espaço de endereçamento é único, é necessário em algumas situações controlar o acesso a memória, em situações conhecidas como condições de corrida, para evitar inconsistência nos dados no caso de escritas/leituras concorrentes.

Computação distribuída é um conceito muito semelhante ao paralelismo, entretanto seus objetivos são diferentes. Em uma computação distribuída as tarefas são executadas em diferentes máquinas para resolver um problema com o objetivo de compartilharem recursos (COULOURIS; DOLLIMORE, 1988). Em problemas impossíveis de serem resolvidos em apenas um computador devido a sua limitação de memória (BARNEY; LABORATORY, 2014) a computação distribuída é uma solução. Concorrência é outro conceito que pode ser usado em conjunto com paralelismo. Concorrência é uma técnica que executa simultaneamente diferentes tarefas para resolução de um problema com o objetivo de estruturar um programa, tornando ele modular (MARLOW, 2013).

Aumentar o número de instruções executadas por segundo, utilizando uma computação sequencial, implica em aumentar a frequência do processador. Com o aumento da frequência maior é o consumo de potência no circuito (FRANK, 2002), e isso acaba levando a um aquecimento maior do circuito devido a potência dissipada em forma de calor. Atualmente está se chegando ao limite de temperatura em um circuito, e consequentemente aumentar a frequência não é mais uma alternativa para ganho de desempenho. O uso de paralelismo acaba sendo uma das soluções nesse cenário. Em uma aplicação pa-

ralela é possível se executar mais de uma instrução no mesmo ciclo de *clock*, acarretando em um melhor desempenho utilizando a mesma frequência.

O desenvolvimento de compiladores que automatizem o paralelismo em *softwares* ainda é uma realidade distante, cabendo ao desenvolvedor esta tarefa. Os sistemas operacionais têm como um de seus principais objetivos gerenciar os recursos de um sistema (TANENBAUM, 2007). Como forma de realizar esse gerenciamento são utilizadas abstrações como *threads* e processos, que variam em cada sistema operacional. *Threads* e processos (TANENBAUM, 2007) são algumas das formas que as aplicações tem de dividir o programa em subtarefas e executá-las em paralelo em múltiplos processadores. Uma das diferenças das *threads* em relação aos processos é o fato de *threads* compartilharem recursos como a memória. É natural se pensar em threads no uso de computação paralela de memória compartilhada, enquanto processos podem ser usados numa computação paralela de memória distribuída. Um padrão para programação com *threads* conhecido como *POSIX threads (pthreads)* (BARNEY; LABORATORY, 2013) foi desenvolvido com o objetivo de facilitar a portabilidade dos softwares em sistemas Unix. Esse padrão é utilizado nesse trabalho nos experimentos utilizando linguagem imperativa. Além das *pthreads* outros padrões como OpenMP (QUINN, 2003) e MPI (QUINN, 2003) são bastante utilizados no desenvolvimento de programas paralelos utilizando linguagens de programação imperativas.

O uso de *pthreads* no desenvolvimento de *softwares* acaba muitas vezes sendo uma tarefa bastante árdua e um dos principais motivos esta numa característica intrínseca das linguagens imperativas: as variáveis. Duas ou mais *threads* podem compartilhar recursos, no caso uma variável, e o resultado depende da forma de escalonamento dessas *threads*. Porém, esse comportamento pode ser indesejado em um programa e pode levar a erros difíceis de serem detectados, pois podem ocorrer raramente.

Para se obter o melhor desempenho em uma aplicação paralela o ideal é que se particione os dados em partes que levem o mesmo tempo para serem processadas, para que nenhum processador fique ocioso durante a execução do programa. Porém, esse particionamento nem sempre é simples de ser alcançado. Fatores como a velocidade de cada processador podem tornar o particionamento ideal impossível de ser conhecido antes da execução da aplicação. Esse particionamento pode ser estático ou dinâmico (WILKINSON; ALLEN, 1999). Outra questão importante é a granularidade das tarefas: todas as tarefas têm custos de processamento associados a sua criação e gerenciamento. Tarefas com uma granularidade muito pequena acabam não compensando seu custo, e podem levar a programas paralelos com desempenho até inferior a programas sequenciais. Tarefas com granularidade muito grande podem não ter uma boa distribuição das cargas de trabalho.

3.2 Paralelismo em Haskell

O paralelismo em Haskell é relativamente recente e ainda é um tópico de pesquisa. Por essa razão Haskell disponibiliza diversas bibliotecas para o uso de paralelismo, não existindo uma unanimidade na escolha entre elas. Cada uma dessas bibliotecas apresenta caracteristícas que podem ser mais adequadas dependendo do problema a ser resolvido. Abaixo serão apresentadas as bibliotecas mais utilizadas.

3.2.1 `Control.Parallel.Strategies`

Hakell oferece o módulo `Control.Parallel.Strategies` (HASKELL, 2010), que tem o propósito de 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. As principais estratégias oferecidas pelo módulo são: `rpar` e `rseq`.

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. O custo associado à criação e gerenciamento dos `sparks` é menor que os das `threads`, tornando o módulo ideal para paralelismo mesmo quando a granularidade das tarefas é pequena. A estratégia `rseq` avalia o seu argumento de forma sequencial e aguarda o seu retorno. O uso mais comum de `rseq` é o de criar uma sincronização para os *sparks*, entretanto seu uso não é obrigatório pois a dependência de dados no módulo é totalmente implícita. Nem sempre um `spark` cria uma *thread*. 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`.

A linguagem Haskell permite que o programador desenvolva suas próprias estratégias. Essa é uma flexibilidade muito boa da linguagem, pois permite que a parte paralela seja totalmente isolada da parte sequencial do programa, podendo ser reutilizável. As estratégias podem ser parametrizáveis, isto é, receber outras estratégias como parâmetros. Assim é possível combinar as estratégias oferecidas pela biblioteca com as desenvolvidas pelo programador.

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`.

Para compilação de um programa paralelo no GHC deve-se utilizar como parâmetro a opção `-threaded`. Para sua execução utiliza-se o parâmetro `-Nx`, sendo x o número de processadores.

3.2.2 `Control.Monad.Par`

O módulo `Control.Monad.Par` difere do anterior em vários aspectos. Nele, não existe a criação de `sparks` e sim de `threads`. Como consequência o custo de criação e gerênciamento é maior que o do módulo `Control.Parallel.Strategies`. O módulo utiliza avaliação estrita para avaliar seu argumento e obriga o programador à explicitamente gerenciar a dependência de dados. Como vantagens, o programador não necessita ter conhecimento de como ocorre a avaliação preguiçosa nem conhecer o funcionamento do coletor de lixo para gerar paralelismo, visto que, sempre será gerada uma `thread` onde for indicado no programa. Todavia, não é possível isolar a parte sequencial da parte paralela como ocorria com as estratégias. O módulo também mantêm a linguagem pura, garantindo um resultado determinístico.

A biblioteca oferece a função `fork` que cria um novo fluxo de execução para ser computado paralelamente um argumento encapsulado pela mônada `Par`. No entanto `fork` não retorna nenhum valor para `thread` que o criou, por isso foi criada a estrutura `IVar`. A estrutura tem a função de realizar a comunicação entre as `threads` e implementa diversas funções, sendo as principais `new`, `put` e `get`. A função `new` cria a estrutura, `put` coloca um argumento dentro da estrutura e força a avaliação do mesmo, sendo esse passado ao `fork`, por fim, `get` lê o valor da estrutura e dessa forma realiza a comunicação. O resultado da mônada `Par` é computado e ligado a parte pura da linguagem através da função `runPar`.

Uma característica interessante do módulo é que ele permite que o desenvolvedor modifique a política de escalonamento utilizada pelo GHC para as `threads`, permitindo assim que se otimize o escalonador de acordo com a aplicação.

3.2.3 `Repa`

Regular Parallel Arrays (Repa) é uma biblioteca que permite o uso de arrays eficientes e oferece paralelismo para sua manipulação. A biblioteca simplifica o uso do paralelismo, isentando do programador essa tarefa. Todas as funções oferecidas pela biblioteca realizam paralelismo de forma automática e implícita. A abstração utilizada para criação de paralelismo são `threads` e não `sparks`. Assim como os módulos anteriores `Repa` é uma biblioteca pura.

`Repa` é ideal para armazenamento de grande quantidade de dados. Listas e `arrays` convencionais em Haskell são imutáveis, consequentemente adição de novos elementos ou aplicação de funções a essas estruturas geram estruturas intermediárias. Como resultado, maior é a ação do coletor de lixo e pior o desempenho. `Repa` elimina a criação de novas estruturas através de um processo conhecido como transformação do espaço de índices (KELLER et al., 2010) de forma transparente ao programado. Transformação do espaço de índices é uma funcionalidade que permite que um `array` seja representado por uma função indexada, com isso é possível alterar os valores desses índices, que representam as posições de memória dos dados.

O módulo disponibiliza uma grande quantidade de funções para manipulação de seus `arrays`, podendo-se citar: `append`, que concatena dois `arrays`, `slice`, que tem como função extrair uma parte do `array` e `map`, que já foi citado anteriormente. Todas essas funções são naturalmente paralelas, entretanto, é deixado a cargo do GHC realizar esse paralelismo. Portanto, oferece pouca flexibilidade ao programador e nem sempre o

compilador obtem o melhor desempenho, visto que, não pode-se modificar a granularidade ou a distribuição das tarefas.

Haskell ainda oferece outros módulos para paralelismo como `Accelerate`, sendo este mais voltado para o paralelismo em `GPUs` e o módulo `Data Parallel Haskell` (DPH) focado em estrutura de dados irregulares como árvores, por exemplo. Existem módulos não determinísticos também sendo esses mais relacionados à concorrência e computação distribuída como `forkIO`, `Cloud Haskell` e `Software transactional memory` (STM). Uma análise minunciosa desses outros módulos, contudo, se encontra fora do escopo deste trabalho.

A escolha da biblioteca a ser utilizada depende do problema a ser resolvido. A `Strategies` se mostra adequada em problemas onde a granularidade das tarefas é pequena, pois utiliza `sparks` para gerar paralelismo. O módulo `Repa` é o mais simples de ser utilizado, pois o paralelismo é gerenciado pelo compilador, entretanto seu uso é limitado aos `arrays` oferecidos pelo módulo. Por fim, a biblioteca `Control.Monad.Par` é indicada quando o programador deseja utilizar uma forma de avaliação estrita da funções, sendo em alguns casos mais eficiente. As bibliotecas são complementares, pode-se utilizar uma ou mais bibliotecas para paralelismo, dentre as citadas, no mesmo código.

4 METODOLOGIA

Como forma de atingir o objetivo do trabalho escolhemos inicialmente dois problemas: o cálculo do fractal de Mandelbrot e a simulação n-Corpos gravitacional. O próximo passo foi a implementação de uma versão sequencial e paralela em Haskell para cada problema. Também foi implementada uma versão em Ansi C para conjunto Mandelbrot por apresentar uma complexidade de implementação menor. Por fim, foram realizados experimentos e seus resultados analisados.

A escolha do cálculo do conjunto Mandelbrot deve-se a características como a simplicidade de paralelização, e não apresentar dependência dos dados. Essa escolha teve como meta verificar a complexidade de implementação dos mecanismos básicos do módulo, para paralelismo, `Control.Parallel.Strategies`, e o ganho de desempenho da aplicação paralela. No cálculo do conjunto Mandelbrot também foi possível comparar o desempenho com uma linguagem imperativa.

O segundo problema o n-Corpos foi escolhido por ter como características uma complexidade maior que o primeiro problema, tendo dependências de dados no algoritmo. O objetivo foi analisar aspectos da linguagem que não foram vistos no primeiro problema e como a dependência de dados afeta o desempenho. Verificamos a implementação dessas dependências na linguagem e a comunicação entre as subtarefas no algoritmo paralelo.

4.1 Ferramentas e ambiente de execução

O ambiente de execução dos experimentos é a máquina beagle do Instituto de Informática da UFRGS. A máquina beagle têm dois processadores com oito núcleos em cada um deles. Em cada núcleo pode-se executar duas *threads* ao mesmo tempo. A memória total da máquina é de 32 GigaBytes e a frequência é de 2000 MHz. O sistema operacional instalado é o Linux versão 3.8.13.13+. A versão do GHC é a 7.4.1 e a do GCC é 4.6.3.

Na análise dos algoritmos a ferramenta R (STATISTICS; WU, 2010) foi utilizada para geração dos gráficos e estatísticas. Foram realizados 50 experimentos para cada configuração dos algoritmos. Quanto maior o número de réplicas melhor é as estimativas das variações dos tempos de execução. Nesse trabalho escolheu-se utilizar 50 réplicas com o propósito de se obter a melhor estimativa possível em um tempo aceitável. Em todos os gráficos gerados os pontos nos representam individualmente cada experimento. Também é mostrado uma curva estimada de comportamento para os tempos médios dos experimentos, e uma área sombreada que representa o erro médio dessa curva com um intervalo de confiança de 0.95.

4.2 Cálculo do fractal de Mandelbrot

O algoritmo procurou representar o fractal de Mandelbrot para um conjunto finito de números complexos representando as coordenadas de um plano cartesiano bidimensional. O algoritmo consiste em testar se, após um número finito de iterações/recursões, o resultado tende ao infinito. No caso do resultado divergir para o infinito o número complexo não pertence ao conjunto, do contrário o número pertence. Adicionalmente também foi possível definir valores representando cores para os pontos que não pertencem ao conjunto através da quantidade de iterações/recursões realizadas.

A implementação do problema teve versões sequenciais e paralelas codificadas em Haskell e Ansi C. As versões paralelas em Haskell utilizaram os módulos `Control.Parallel.Strategies` e `Repa`. A biblioteca `Control.Parallel.Strategies` foi escolhida por utilizar características da linguagem como avaliação preguiçosa e utilizar `sparks` para paralelismo. A escolha do módulo `Repa` foi escolhida por ser adequada para grandes quantidades de dados. A versão paralela em Ansi C utilizou a biblioteca `PThreads`.

Para o algoritmo sequencial em Haskell foram realizados experimentos utilizando como fator a quantidade de dados. Para o algoritmo paralelo em Haskell foram realizados experimentos variando-se a quantidade de dados, o número de núcleos utilizados e a forma de particionamento dos dados. Os algoritmos em Ansi C, tiveram um número fixo de dados para a aplicação sequencial e um número fixo de dados, núcleos e uma forma de particionamento para aplicação paralela.

A análise foi feita executando-se diversas 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, como por exemplo a execução de tarefas do sistema operacional. Foram comparados os tempos de execuções dos experimentos do algoritmo sequencial com os do algoritmo paralelo em Haskell. A partir desses tempos foi determinado o ganho de desempenho do algoritmo paralelo, e gerados gráficos em alguns casos. Também foi feita uma observação sobre a distribuição das cargas de trabalhos nas diferentes formas de particionamento do algoritmo paralelo. Adicionalmente, realizamos uma comparação dos tempos de execução em Haskell com os tempos de execução em Ansi C. A idéia é comparar a linguagem funcional com a imperativa. Outro fator que foi verificado é a complexidade de implementação.

Os códigos utilizados nos experimentos do cálculo do fractal de Mandelbrot podem ser encontrados em *https://github.com/vfpereira/Mandelbrot.git*.

4.3 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. Só é possível calcular novamente as forças após todos os corpos terem realizado seu movimento anterior.

Implementou-se duas versões para o n-Corpos gravitacional, ambas bidimensionais, a primeira apresentando uma complexidade quadrática e a segunda uma complexidade

`O(nlogn)`, também conhecida como Barnes-Hut (Barnes; Hut, 1986). Para ambos utilizou-se a mesma implementação para as análises sequenciais e paralelas: o algoritmo sequencial é a versão paralela com parâmetro de particionamento 1. A biblioteca utilizada para paralelismo nos algoritmos foi a `Control.Parallel.Strategies`.

Os experimentos em ambos os algoritmos utilizaram como fatores o número de corpos, o número de passos realizados representando o movimento dos corpos, a quantidade de núcleos utilizados e o tempo que representa cada um dos passos. Apenas uma forma de particionamento é utilizada no algoritmo de complexidade quadrática. O Barnes-Hut utiliza duas estruturas de dados na sua implementação e o particionamento é estático em uma delas e dinâmico na outra.

A análise compara a implementação e o desempenho de ambos os algoritmos de forma semelhante à seção anterior. Também é realizada uma análise do efeito da dependência de dados comparado com o Mandelbrot.

Os códigos utilizados nos experimentos da simulação n-Corpos Gravitacional podem ser encontrados em *https://github.com/vfpereira/nCorpos.git*.

5 CÁLCULO DO FRACTAL DE MANDELBROT

O primeiro algoritmo utilizado é o cálculo do fractal de Mandelbrot. Fractais são uma área da geometria matemática que estuda figuras que são formadas a partir da repetição de um padrão. Um dos objetivos dos fractais é explicar fenômenos da natureza que não podem ser representados pela geometria euclidiana (MANDELBROT, 1983).

O conjunto de Mandelbrot é definido por:

$$z_0 = 0, \quad z_{n+1} = z_n^2 + c$$

para todos os pontos c no qual o resultado da recursão não tende ao infinito. O ponto c é um número complexo. Já foi entretanto provado que se $|z_n| > 2$ o resultado da recursão tende ao infinito, esse teste é realizado nos algoritmos deste trabalho para determinar se o ponto pertence ao conjunto Mandelbrot.

O fractal de Mandelbrot é um conjunto matemático de pontos que apresenta algumas características apropriadas para o uso de paralelismo. Em particular o conjunto pode ser definido recursivamente e o cálculo de um ponto é independente do cálculo dos outros pontos. O conjunto de Mandelbrot é um conjunto infinito de pontos limitados em uma área finita. Essa característica é impossível de ser reproduzida computacionalmente em máquinas que tem memórias finitas. Porém, é possível discretizar esses pontos para análise, e a quantidade de pontos discretizados pode variar. Isso é bastante favorável para os experimentos que serão realizados, e para alcançar o objetivo desse trabalho.

5.1 Implementações

Um descrição das implementações do cálculo do fractal de Mandelbrot é apresentada nas subseções abaixo.

5.1.1 Implementação sequencial em Haskell

O algoritmo sequencial do cálculo do fractal de Mandelbrot (`Mandelbrot_sequencial.hs`) baseia-se na construção de uma lista de números complexos. Estes representam coordenadas num plano cartesiano bidimensional com o fractal centralizado nos pontos (0,0). Os pontos testados estão dentro dos limites (-2,-2) e (2,2), pelo fato de todos os pontos que convergem do conjunto Mandelbrot estarem nesses limites. Em seguida utiliza-se uma função recursiva para testar se um número complexo pertence ao conjunto Mandelbot, com o número máximo de recursões igual a 50, visto que não é possível realizar um número infinito de recursões. Através do valor absoluto testa-se se o ponto

pertence ao conjunto Mandelbrot, em caso positivo a função retorna a posição do ponto no plano cartesiano e sua cor com o valor 0, caso contrário é retornado a posição com o valor da cor sendo igual ao da recursão onde o teste foi negativo. Concluindo, se utiliza uma função `map` para testar todos os pontos da lista. A cor é utilizada para se desenhar o fractal, uma versão que desenha o fractal de Mandelbrot também é encontrada no repositório (`Mandelbrot_OpenGL.hs`) , sendo reproduzida na Figura 5.1 a imagem gerada.

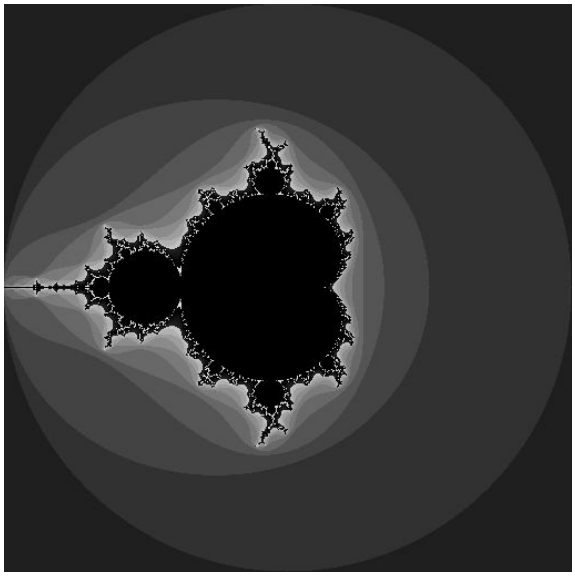

Figura 5.1: Imagem do conjunto Mandelbrot gerada em Haskell utilizando OpenGL.

5.1.2 Implementação paralela em Haskell

O algoritmo paralelo em Haskell (`Mandelbrot_paralelo.hs`) cria a lista de pontos complexos da mesma forma que o algoritmo sequencial. Porém, após são particionados os pontos de acordo com os parâmetros passados na execução do programa, criando uma lista de listas desses pontos que serão as subtarefas do programa. Foram implementadas 4 formas de particionamentos, para exemplificar os particionamentos supõe-se uma lista representando uma imagem 4x4 particionada em duas listas com os elementos [11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44] , o conjunto de pontos:

- particionado em linhas.

  Exemplo [[11,12,13,14,21,22,23,24],[31,32,33,34,41,42,43,44]]

- particionado em colunas.

  Exemplo [[11,21,31,41,12,22,32,42],[13,23,33,43,14,24,34,44]]

- particionado em linhas e com uma distância entre eles igual ao número de tarefas.

  Exemplo [[11,13,21,23,31,33,41,43],[12,14,22,24,32,34,42,44]]

- particionado em colunas e com uma distância entre eles igual ao número de tarefas.

  Exemplo [[11,31,12,32,13,33,14,34],[21,41,22,42,23,43,24,44]]

O número de tarefas criadas é igual ao primeiro parâmetro de execução que é o numero de `sparks` que se deseja criar. A mesma função do algoritmo sequencial foi utilizada para testar se o ponto pertence ao conjunto Mandelbrot. Contudo, a função `map` foi substituída por uma função recursiva que cria um *spark* para cada lista particionada ser executada em paralelo e assim avaliar todos os pontos. Pode-se notar na implementação do código poucas alterações em relação ao sequencial, demonstrando uma simplicidade na implementação. Encontrar a melhor forma de particionar os dados, contudo, não é necessariamente fácil. Outras implementações paralelas em Haskell foram implementadas, visando um melhor desempenho, e serão posteriormente explicadas.

5.1.3 Implementação sequencial em Ansi C

O algoritmo sequencial em Ansi C (`Mandelbrot_sequencial.c`) teve como objetivo criar uma implementação que fosse semelhante à da linguagem Haskell. Contudo, procurou-se manter as características mais comuns da linguagem, como o de iterações no lugar de recursões.

Na função principal do programa foi criado um laço que cria e armazena os números complexos em uma matriz. Após, essa matriz é passada a função `verificaPlanoComplexo` que itera sobre todos os elementos dessa matriz e para cada elemento chama a função `IteracaoMandelbrot`. Essa função verifica se o número complexo pertence ao conjunto Mandelbrot e retorna um `struct` com as coordenadas do número complexo e um valor representado a cor. Uma característica que difere bastante da linguagem Haskell é o retorno da função `IteracaoMandelbrot` que é um ponteiro para uma matriz de `structs`. Ansi C exige que o programador realize a alocação de memória explicitamente, o que é feito através da função `malloc` no algoritmo.

5.1.4 Implementação paralela em Ansi C

A implementação paralela em Ansi C (`Mandelbrot_Paralelo.c`) também cria e armazena os números complexos em uma matriz. Entretanto, foi utilizada a biblioteca `PThreads` para realizar o paralelismo. Foi criada um vetor para armazenar uma `struct` de `threads`, onde cada elemento desse vetor armazenava o identificador da `thread`, uma partição da matriz de números complexos e o tamanho dessa matriz para as alocações de memória. A forma de particionamento utilizada foi dividindo-se os pontos em linhas e com uma distância entre eles igual ao número de tarefas. Para cada elemento do vetor utilizou-se a função `pthread_create` para criar uma `thread` que realizavam os métodos para calcular se um número complexo pertence ao conjunto Mandelbrot. Esses métodos são semelhante aos da implementação sequencial, pórem no final do método a chamada da função `pthread_exit` para indicar que o resultado está pronto e disponível para ser usado na comunicação entre as tarefas. Para obter o resultado correto foi criado um laço para todas as `threads` com a chamada da função `pthread_join` que serve para sincronização, e, por fim, a junção das matrizes particionadas em uma unica matriz.

5.2 Análises dos resultados obtidos

Nos resultados em Haskell são obtidos dois tempos: um de cálculo dos pontos do fractal de Mandelbrot e outro do tempo de alocação e operações de criação das estruturas que contêm os pontos. Foram utilizados diversos códigos fontes em Haskell para melhor uma melhor análise da linguagem, e em todos os experimentos os códigos foram compilados por motivos de desempenho. Em Ansi C apenas é considerado o tempo de cálculo dos pontos, uma vez que o tempo de alocação da estrutura é muito pequeno e pode ser considerado desprezível.

5.3 Influência das otimizações do compilador na execução

Inicialmente foram realizados alguns experimentos com a aplicação sequencial em Haskell sem serem realizadas otimizações, que é a forma padrão de compilação do GHC. Porém, os tempos obtidos foram muito altos chegando próximo a 50 segundos no pior caso. Ao se utilizar as otimizações do compilador, nesse experimento o parâmetro -O3 na compilação, conseguiu-se reduzir consideravelmente os tempos de cálculo e de alocação da estrutura.

A Tabela 5.1 apresenta um comparativo entre os tempos médios do cálculo dos pontos utilizando otimizações do compilador (-O3) com os tempos sem otimizações (-O0), variando-se a quantidade de pontos. O algoritmo utilizado para para realização desse experimento foi o `Mandelbrot_sequencial.hs`.

Tempos Médios	Quantidade de Pontos				
	16641	66049	263169	1050625	4198401
Cálculo -O0	0.18s	0.73s	2.90s	11.54s	46.11s
Cálculo -O3	0.04s	0.15s	0.58s	2.33s	9.30s

Tabela 5.1: Tempo médio Haskell sequencial x Haskell sequencial otimizado.

Percebe-se pelos resultado um ganho de desempenho que varia de 450% a 500% no tempo de cálculo dos pontos do fractal de Mandelbrot. Esse fato demonstra o potencial do compilador em realizar otimização de forma transparente ao programador. Ao se utilizar essas otimizações alguns requisitos do sistema são afetados, como tempo de compilação e adicionar uma complexidade no uso de técnicas de profiling. Entretanto, o tempo de execução é significativamente menor. As técnicas de otimizações utilizadas pelo compilador produzem o mesmo resultado, mas o código final é bastante modificado em relação ao código original.

Outros parâmetros além do -O3 podem tornar o programa mais eficiente. Entre os parâmetros estão os de controle do coletor de lixo. No algoritmos de Mandelbrot utiliza-se uma grande quantidade de dados como parâmetros de entrada, causando assim uma grande quantidade de dados temporários em memória. Como consequência uma parte significativa do tempo é gasto pelo coletor de lixo. O parâmetro -A pode ser utilizado para definir o tamanho de área alocada para o coletor de lixo, e aumentando esse tamanho consequentemente se tem uma quantidade menor de chamadas do coletor. Contudo, obesrvamos que esses parâmetros isoladamente não tem um impacto muito significativo e grande parte deles estão relacionados com o *hardware* da máquina, principalmente com

o tamanho da memória principal e das caches. Dependendo do tamanho da mémoria pode-se dar espaços maiores para o coletor de lixo, por exemplo.

O compilador oferece opções para o uso de técnicas de *profiling*, e entre elas o de análise do uso da memória. Através destas técnicas consegue-se identificar os trechos de códigos que mais utilizam a memória, chamados de *space leaks*. Por intermédio dessa análise pode-se diminuir o uso da memória e consequentemente o tempo gasto pelo coletor de lixo.

5.4 Tempo de Alocação e operações em listas em Haskell

Em Haskell, por ser uma linguagem de avaliação preguiçosa, ter dados imutáveis e o coletor de lixo, os tempos de alocação de uma estrutura acabam sendo muito maiores que em linguagens imperativas. Existem formas de alocação estritas também, porém essa não é a forma padrão da linguagem. Ao se utilizar as otimizações do compilador muito desse tempo é diminuido, entretanto, o grande problema consiste nas operações realizadas sobre as listas em Haskell.

As listas são uma das principais estruturas de dados em Haskell e oferecem um grande número de funções para sua manipulação. Porém, quando o objetivo é eficiência listas não são as estruturas mais adequadas. Acessar um elemento de uma lista acaba tendo um tempo muito maior que em um vetor, visto que é necessário realizar uma busca sequencial a partir do inicio da lista até chegar ao elemento que se deseja. Como efeito, as operações sobre esses elementos também acabam tendo um tempo bastante elevado.

Para criação das estruturas de armazenamento dos números complexos, que posteriormente serão utilizados para o cálculo de Mandelbrot, foram realizadas quatro implementações em Haskell: `Mandelbrot_sequencial.hs`, `Mandelbrot_paralelo.hs`, `Mandelbrot_paralelo_repa.hs` e `Mandelbrot_e_alocacao_paralelo-.hs`. No algoritmo `Mandelbrot_sequencial.hs` utilizou-se uma lista de números complexos representando coordenadas em um sistema de coordenadas bidimensional, como esse algoritmo é sequencial não existiu a necessidade de outras estruturas.

O algortimo `Mandelbrot_paralelo.hs` foi criado baseando-se no algoritmo sequencial, e tentou-se realizar o menor número de alterações possível. Dessa forma manteve-se uma a lista de números complexos e sobre essa estrutura foram realizados particionamentos gerando sublistas para futuramente realizar o cálculo de Mandelbrot de forma paralela. Foram utilizadas quatro formas de particionamentos com o objetivo de verificar diferentes formas de distribuição das cargas de trabalho entre os `sparks`. As formas de particionamento escolhidas para essas sublistas foram em linha, coluna e para cada uma dessas também se particionou os pontos com uma distância igual ao número de sublistas. No entanto todo esse processo de geração das sublistas foi realizado de forma sequencial, pois não existiu inicialmente uma preocupação com as operações realizadas na lista original. Devido a ineficiência dessa implementação criou-se outras duas implementações, dessa vez modificando-se bastante o algoritmo sequencial.

Uma implementação utilizando vetores foi utilizada para se obter uma comparação com os tempos de listas. O algoritmo `Mandelbrot_paralelo_repa.hs` alterou a estrutura de dados para vetores fornecidos pelo módulo Repa. Vetores Repa são indicados quando se tem uma grande quantidade de dados, esses vetores geram paralelismo de forma

automática em todas as operações realizadas sobre eles. Todavia, não é possível ter um controle sobre o paralelismo quando esse vetores são utilizados, apenas a quantidade de núcleos que serão utilizados pode ser indicado como parâmetro. Todo o paralelismo é realizado internamente pelo compilador. Como a função de cálculo de Mandelbrot recebia como parâmetro de entrada uma lista de listas e não se desejava alterá-la foi necessário transformar esses subvetores em sublistas o que acabou elevando o tempo total.

Por fim se criou o algoritmo `Mandelbrot_e_alocacao_paralelo.hs` que também utilizava listas, porém ao contrário dos anteriores onde se criava uma estrutura e se dividia em subestruturas, foi criada uma função que gerava diretamente as sublistas. Com essa construção conseguiu-se eliminar as operações sobre as listas e obteve-se um tempo muito pequeno pois a função que gerava as sublistas também foi executada em paralelo.

A Tabela 5.2 mostra o tempo de geração das estruturas para os algoritmos citados nessa seção. O parâmetro -O3 mostrado na tabela representa as otimizações do compilador. O parâmetro CRR16 representa que escolheu-se utilizar 16 `sparks` para rodar em paralelo e que os pontos foram divididos em colunas separados por uma distância igual a 16, utilizando a função `colunaRoundRobin` nos códigos. A configuração CRR16 foi escolhida por realizar um grande número de operações para criação das estruturas no particionamento dos dados.

Tempos Médios	Quantidade de Pontos				
	16641	66049	263169	1050625	4198401
Mandelbrot_sequencial.hs	0.0047s	0.017s	0.06s	0.25s	0.95s
Mandelbrot_sequencial.hs -O3	0.0011s	0.003s	0.01s	0.04s	0.15s
Mandelbrot_paralelo.hs CRR16 -O3	0.0310s	0.118s	0.56s	2.36s	9.67s
Mandelbrot_paralelo_repa.hs CRR16 -O3	0.0130s	0.035s	0.20s	0.74s	2.77s
Mandelbrot_e_alocacao_paralelo.hs CRR16 -O3	0.0013s	0.022s	0.08s	0.13s	0.53s

Tabela 5.2: Tempos médios de Alocação e operações em listas em Haskell.

Para o programa sequencial o tempo representa a alocação isoladamente, uma vez que não é realizada nenhuma operação sobre a lista. Verifica-se que sem otimizações o tempo de alocação chega próximo a um segundo, porém com as otimização do compilador esse tempo fica muito menor, indicando que esse tempo é proximo ao de uma linguagem imperativa. No código `Mandelbrot_paralelo.hs` pode-se observar que a criação de uma lista com uma quantidade grande elementos e a realização de operações sobre ela mostrou-se extremamente ineficiente. Verifica-se na tabela 5.2 um tempo próximo a 10 segundos para uma entrada de 4198401 pontos, sendo aproximadamente 64 vezes maior que o algoritmo sequencial (-O3) para mesma entrada. Observando-se o código da algoritmo `Mandelbrot_paralelo.hs` para a configuração escolhida, mais especificamente a função `colunaRoundRobin`, verifica-se a utilização das operações de `take`, `drop`, `transpose` e `concat`. O uso de apenas quatro operações em uma lista mostra o alto custo associado a eles, e o tempo aumenta considerávelmente a medida que se aumenta o número de elementos da lista. Apartir desse experimento foi possível perceber que em Haskell deve-se ter cuidado com a utilização de listas quando o objetivo é desempenho.

O algoritmo utilizando operações em vetores `Mandelbrot_paralelo_repa.hs` mostrou-se muito melhor em questão de desempenho quando comparado ao algoritmo

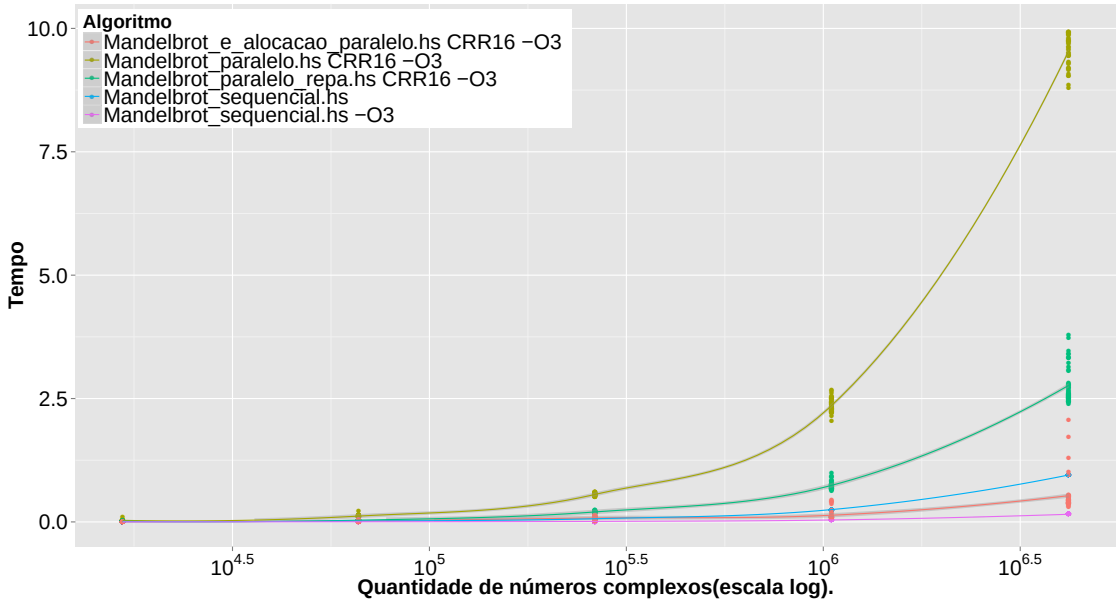

Figura 5.2: Tempos de criação das estruturas em Haskell.

`Mandelbrot_paralelo.hs`. As operações utilizadas na configuração escolhida são: `transpose`, `slice`, `reshape` e `toList`. Para esse algoritmo são usadas quatro operações também, porém o maior tempo obtido foi de 2,77 segundos, um tempo aproximadamente 3,5 vezes menor que os 9.67 segundos obtidos usando listas. Como ja foi citado anteriormente, parte do tempo foi gasto na transformação de vetor para lista, a função `toList`, sem essa função o tempo seria menor. O algoritmo `Mandelbrot_e_alocacao_paralelo.hs` mesmo utilizando lista foi o que obteve o menor tempo para os algoritmos paralelos. Esse resultado foi possível justamente pelo fato de não serem realizadas operações sobre a lista, para configuração escolhida a função `planoColunaRoundRobin` gera diretamente as sublistas. Para uma entrada de 4198401 pontos obteve-se um tempo de 0,53 segundos, um tempo muito pequeno indicando que o custo associado a criação da estrutura compensa a sua utilização posterior no cálculo de Mandelbrot em paralelo.

Os resultados também são mostrados no gráfico da figura 5.2. Pelo gráfico é possível perceber uma grande variação nos tempos das réplicas, principalmente quando se aumenta o tempo de execução. Esse fato é devido a ação do coletor de lixo, pois ele acaba sendo não determinístico e tem um certo impacto no tempo de execução conforme foi discutido anteriormente. Pelos resultados obtidos, os próximos experimentos apresentados serão realizados utilizando as otimizações do compilador -O3 e o código paralelo será o `Mandelbrot_e_alocacao_paralelo.hs`.

5.5 Efeito das diferente formas de particionamento no algoritmo de Mandelbrot paralelo em Haskell

Embora os tempos de criação das estruturas, discutidos na seção anterior, sejam importantes no tempo final de execução, o tempo principal a ser considerado é o da função que realiza o cálculo de Mandelbrot. O objetivo de se criar estruturas intermediarias é

justamente para que elas possam ser executadas em paralelo na função de cálculo. Um enfoque maior será dado ao tempo de cálculo no restante do capítulo.

No algoritmo paralelo a função `mandelbrot` recebe como entrada uma lista de listas de numeros complexo, o parâmetro `iteracao` e retorna uma lista de `Pixel` encapsulada pela mônada `Eval`. `Pixel` é um `type` representando as coordenadas x e y do plano cartesiano com um valor associado a cor. Para cada lista no primeiro argurmento da função é gerado um `spark` e para cada ponto pertencente a essa lista é chamada a função `iteracaoPontoMandelbrot` que efetivamente verifica se um número complexo pertence ao conjunto. Essa verificação é realizada recursivamente, conforme explicado na seção 5.1, e o número de recursões é o parâmetro `iteracao`. A cada recursão o parâmetro é decrementado e novos testes são realizados para verificar se o número pertence ao conjunto. Se o parâmetro chegar a 0 significa que número pertence ao conjunto Mandelbrot do contrário a recursão termina e o valor do parâmetro representa a cor. Nem todos os pontos complexos pertencem ao conjunto Mandelbrot. Uma consequência é que alguns pontos realizam mais recursões que outros, levando a tempos diferentes para seu cálculo. Essa é uma característica indesejável quando se utiliza paralelismo, pois nem todas as `threads` têm a mesma carga de trabalho.

O objetivo de se ter diversas formas de particionamento é observar como as diferentes cargas de trabalho nas `threads` afetam o desempenho. O fractal de mandelbrot apresenta uma certa simetria com isso algumas formas de particionamento conseguem chegar próximo ao ideal.

Os experimentos realizados utilizaram as quatro formas de particionamento citadas anteriormente. Para cada um deles se variou a quantidade de `sparks` pois esses representam também a quantidade de listas usadas. A quantidade de número complexos foi fixada em 4198401.

Tempos Médios	Número de sparks			
	2	4	8	16
linhaNormal	6.51s	5.25s	4.47s	3.76s
linhaRoundRobin	6.39s	3.26s	2.09s	1.90s
colunaNormal	8.01s	5.33s	4.44s	3.80s
colunaRoundRobin	6.51s	3.29s	2.04s	1.92s

Tabela 5.3: Tempos médios de cálculo de Mandelbrot em Haskell utilizando diversos particionamentos.

Pela Tabela 5.3 os melhores resultados obtidos foram usando as funções `linhaRoundRobin` e `colunaRoundRobin`, que representam o particionamento em linha e coluna com os pontos separados a uma distância igual ao número de `sparks`. Esse era um resultado esperado, pois ao se obter uma melhor distribuição dos pontos consegue-se uma melhor divisão das cargas de trabalho. Pode-se dizer que ambos têm o mesmo desempenho, o que fica melhor visualizado no gráfico 5.3. As funções `linhaNormal` e `colunaNormal` também tiveram resultados semelhantes entre si, exceto quando se usa dois `sparks`, isso pode ser explicado pela simetria horizontal que a Figura 5.1 apresenta.

Por fim, a Figura 5.4 mostra os tempos de criação das listas que foram utilizadas para o cálculo em cada particionamento. Em todos os particionamentos os tempos são muito próximos, o que é coerente pois é o único tempo existente é o de alocação e todos apresentam a mesma quantidade de pontos. Esse tempo também é paralelo pois esta se usando

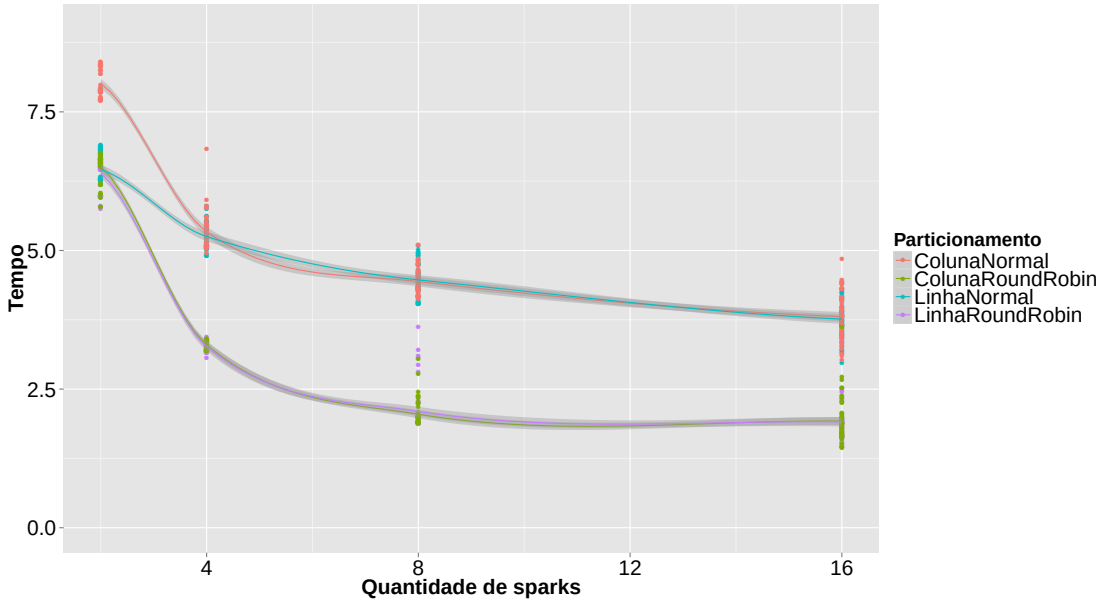

Figura 5.3: Tempos de cálculo do conjunto Mandelbrot para diferentes formas de particionamento.

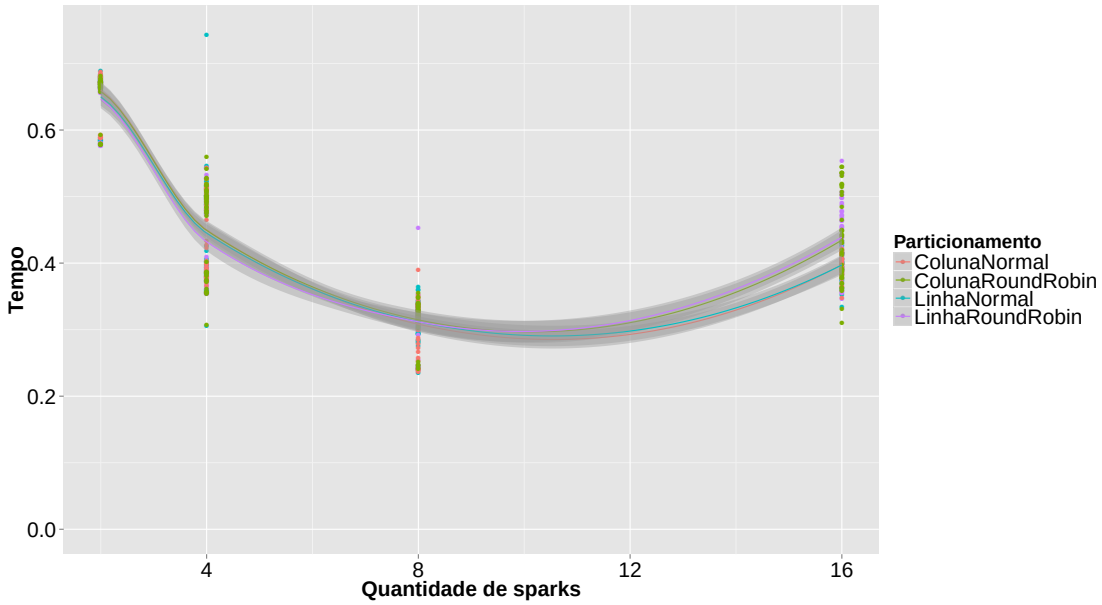

Figura 5.4: Tempos de criação das estruturas do conjunto Mandelbrot para diferentes formas de particionamento.

o algoritmo `Mandelbrot_e_alocacao_paralelo.hs`. Esse resultado ressalta a vantagem de se usar os particionamento em linha e coluna com os pontos separados a uma distância igual ao número de `sparks`.

5.6 Análise da escalabilidade fraca do algoritmo de Mandelbrot em Haskell

A escalabilidade fraca deriva da lei de Gustafson (GUSTAFSON, 1988), e trata do aumento do tamanho de um problema e sua relação com o aumento do número de processadores. Em muitos problemas o paralelismo cresce de acordo com o tamanho do problema. A escalabilidade fraca apresenta uma solução para as restrições impostas pela lei de Ahmdal. Nesse experimento a proporção de crescimento da quantidade de pontos e processadores utilizados não é exatamente a mesma, pois os resultados foram reutilizados dos experimentos anteriores. O tamanho do problema que utilizamos acaba crescendo numa proporção próximo a quatro vezes maior que o número de processadores, com isso procuramos verificar se o paralelismo aumenta. Utilizou-se apenas a forma de particionamento em linha com os pontos separados a uma distância igual ao número de `sparks`, pois este foi o que obteve melhor resultado. A Tabela 5.4 apresenta os tempos médios de cálculo dos experimentos variando-se o número de `sparks` e a quantidade de pontos de entrada. O gráfico 5.5 apresenta as acelerações para mesma configuração.

Tempos Médios de cálculo	Quantidade de Pontos				
	16641	66049	263169	1050625	4198401
sequencial	0.04s	0.15s	0.58s	2.33s	9.30s
2 `sparks`	0.023s	0.093s	0.37s	1.61s	6.39s
4 `sparks`	0.017s	0.059s	0.21s	0.83s	3.26s
8 `sparks`	0.012s	0.033s	0.16s	0.50s	2.09s
16 `sparks`	0.018s	0.046s	0.14s	0.55s	1.90s

Tabela 5.4: Tempos médios de cálculo do conjunto Mandelbrot variando-se a quantidade de pontos e `sparks`.

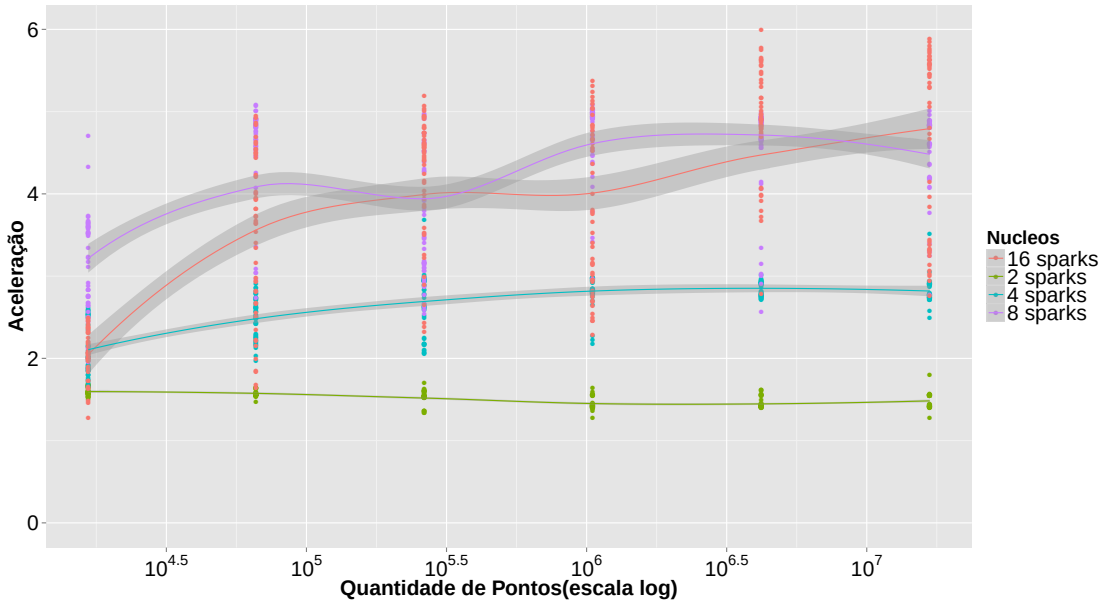

Figura 5.5: Acelerações do conjunto Mandelbrot variando-se a quantidade de pontos e `sparks`.

O algoritmo de Mandelbrot não apresenta dependência de dados dessa forma todo

o algoritmo pode ser paralelizável. Como resultado não existe uma parte sequencial no algoritmo e o paralelismo em teoria deveria aumentar indefinidamente com o número de processadores. Entretanto existem fatores que não permitem que isso ocorra. Entre os fatores pode-se citar o coletor de lixo, e a forma como o sitema operacional gerencia a distribuição das `threads` entre os processadores. Com uma análise da escalabilidade fraca procurou-se ver se com o aumento do tamanho do problema o efeito desses fatores acabam se reduzindo.

Pela Figura 5.5 é possível notar que quando se utiliza 2 `sparks` a aceleração se mantém praticamente constante independente do tamanho da entrada. Porém é bem visível que quando se utiliza mais `sparks` a aceleração, mesmo com algumas oscilações, cresce conforme o tamanho do problema. O que ocorre é que esse crescimento acaba atingindo um limite onde aumentar o tamanho do problema para essa quantidade de processadores acaba não tendo mais efeito. Pode-se concluir que para uma quantidade maior de processadores esse comportamento acaba se potencializando, mas, devido as limitações da máquina utilizada não foi possível reproduzir esse fato.

5.7 Comparação entre os algoritmos sequencial e paralelo em Haskell

Nas seções anteriores buscou-se analisar quais as melhores configurações do algoritmo paralelo para obter-se melhor desempenho. Porém, o principal objetivo é analisar a viabilidade da implementação paralela em Haskell. Para isso é necessário uma comparação entre as implementações em paralelo e sequencial.

A mônada `Eval` escolhida como forma de paralelismo para o algoritmo de Mandelbrot é bastante simples de ser utilizada. Observando os algoritmos em paralelo basicamente se usa a função `rdeepPar`, que é o uso das funções nativas `rpar` e `force`, dentro da mônada para lançar `sparks`, e depois se extrai o resultado da mônada através da função `runEval`. A complexidade se encontra em particionar os dados de entrada do algoritmo, e encontrar a melhor forma de fazê-lo. Analisar a melhor forma de realizar o particionamento para obter melhor desempenho é algo que depende de cada algoritmo e nem sempre é uma tarefa trivial. Para o algoritmo de Mandelbrot se utilizou quatro formas de particionamento. Formas de particionamento adicionais não foram consideradas por questão de escopo. Uma análise prévia foi feita antes e conseguiu-se resultados próximos ao ideal com dois dos particionamentos utilizados. Quando comparado com o algoritmo sequencial esse é um custo adicional em um projeto que deve ser levado em conta. Outra preocupação que deve ser levada em conta quando o objetivo é desempenho é a estrutura que se está utilizando e as funções. Por exemplo, operações em listas têm um alto custo, mesmo sendo uma das principais estruturas da linguagem. Diversos outros fatores devem ser pesados quando se busca desempenho como o coletor de lixo e otimizações. Foi possível chegar a conclusão de que embora o uso da mônada `Eval` seja bastante simples para paralelismo, quando se deseja o melhor desempenho possível é necessário um conhecimento mais aprofundado do ambiente de execução e uma análise prévia do algoritmo utilizado.

Em relação ao desempenho, a melhor aceleração obtida em todos os experimentos foi próximo 5 conforme a Figura 5.5. Reduzir o tempo em cinco vezes é um resultado que justifica a utilização de paralelismo, no entanto para obter essa aceleração se utilizou

16 núcleos. Por esse resultado a eficiência acaba não sendo muito expressiva. O tabela mostra a eficiência variando-se a quantidade de `sparks`.

Eficiência LinhaRoundRobin	Quantidade de Pontos
	4198401
2 `sparks`	72%
4 `sparks`	71%
8 `sparks`	55%
16 `sparks`	30%

Tabela 5.5: Tempos médios de alocação e operações em listas em Haskell.

Como resultado dessa eficiência e pelos resultados dos experimentos anteriores fica claro que sem aumentar o tamanho do problema, não se fundamenta aumentar a quantidade de processadores. Essa acaba sendo uma limitação, visto que muitas vezes não tem sentido aumentar o tamanho do problema. Porém, no caso específico do algoritmo de Mandelbrot, esse comportamento pode não ser um problema, visto que existe um número infinito de pontos no plano cartesiano, podendo-se aumentar indefinidamente o tamanho da entrada.

5.8 Comparação de Haskell com Ansi C

Ansi C é uma linguagem imperativa e bastante utilizada para computação de alto desempenho. Um padrão muito utilizado na linguagem para computação paralela são as `POSIX Threads`. Por essa razão acabou-se optando por realizar uma comparação com o algoritmo escrito em Haskell. Através dessa comparação busca-se analisar se os desempenhos e implementações em Haskell apresentam vantagens em relação a uma linguagem imperativa.

`Pthreads` apresentam uma certa complexidade de implementação porém é muito subjetivo comparar a dificuldade de implementação com a mônada `Eval` em Haskell. Contudo, assim como em Haskell o programador tem que ter uma preocupação com o particionamento dos dados de entrada do algoritmo. Embora seja possível o uso de recursão na linguagem Ansi C, a forma mais comum de percorrer uma estrutura é através de iteração. No algoritmo de Mandelbrot em Ansi C essa foi a forma utilizada. Outra diferença em relação linguagem Haskell é o uso de matrizes ao contrário de listas. Por usar matrizes os problemas relacionados ao desempenho de operação em listas acabam não ocorrendo. Ansi C não apresenta coletor de lixo o que de certa forma é positivo para o desempenho, entretanto uma complexidade é adicionada ao programador, pois esse acaba ficando responsável pela alocação e liberação de valores da memória. Enfim, ambas as implementações apresentam vantagens e desvantagens, porém a idéia não é quantificar como uma linguagem é melhor que outra. Propõe-se apresentar uma comparação quantitativa entre as implementações paralelas do algoritmo em ambas as linguagens.

Para avaliar o desempenho de ambas as implementações se realizou experimentos em Ansi C sequencial e comparou-se com o mesmo em Haskell, e o mesmo para o algoritmo paralelo. Porém, para o algoritmo paralelo em Ansi C criou-se apenas uma forma de particionamento que foi o que utiliza a função `linhaRoundRobin` pois esse foi o que obteve o melhor desempenho em Haskell. Nas implementações tentou-se criar os algoritmos da mesma forma dada as características de cada linguagem. Os tempos considerados

são de criação das estrutura somados ao de cálculo do conjunto Mandelbrot diferente dos experimentos anteriores onde eles analisados isoladamente.

Tempos Médios	Número de sparks				
	sequencial	2	4	8	16
Haskell	9.30s + 0.15s	6.39s + 0.53s	3.26s +0.53s	2.09s +0.53s	1.90s+0.53s
Ansi	9.91s	5.05s	2.62s	1.41s	0.84s

Tabela 5.6: Haskell x Ansi C.

A tabela 5.6 mostra o tempo médio dos algoritmos com o número de pontos fixos em 4198401. A Figura 5.6 oferece uma melhor visualização onde é possivel perceber que o algoritmo sequencial em Haskell é ligeiramente mais rápido, contudo, o inverso ocorre quando em paralelo. Também é possível perceber que em Haskell o tempo a partir de 8 `sparks` praticamente se mantém igual enquanto 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 pode-se se citar o coletor de lixo que é bastante punitivo para desempenho na linguagem Haskell. Outro fator que têm influência 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 GHC armazena esses `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* (MARLOW, 2013). Pelo motivo citado, não existe um garantia que o número de `sparks` criados seja igual ao número de processadores que estarão executando o cálculo de Mandelbrot paralelamente. Quanto mais próximo a quantidade de processadores da máquina menor é a probabilidade de processadores ociosos e dessa forma explica o comportamento da Figura 5.6.

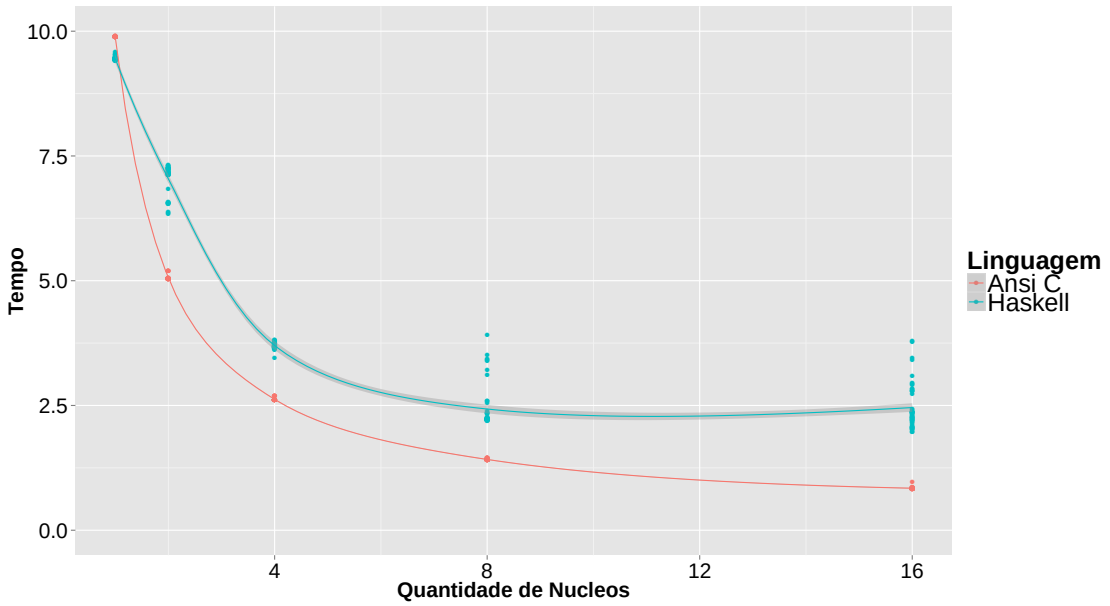

Figura 5.6: Tempo CxHaskell.

Em desempenho Ansi C apresenta melhores resultados porém a diferença não é muito acentuada. Esse fato representa que o módulo `Control.Parallel.Strategies` da

linguagem Haskell é satisfatório para a computação paralela, podendo ser uma alternativa às `Pthreads` em Ansi C. A linguagem Haskell acaba eximindo o programador de lidar com ponteiros e gerência de memória com um custo em desempenho, esse é um fator que deve ser mensurado, pois elimina muitos dos erros de programação.

6 SIMULAÇÃO N-CORPOS GRAVITACIONAL

O problema do n-Corpos tem como objetivos simular a iteração entre partículas de acordo com as leis da física, sendo a gravidade um caso específico. A força gravitacional entre dois corpos é representada pela fórmula:

$$F = G\frac{m_1 m_2}{r^2}$$

onde G é a constante universal de gravitação, $m1$ e $m2$ as massas dos corpos, e r a distância entre eles.

O somatório das forças exercidas pelos corpos em cada um resulta na aceleração e consequentemente no movimento dos corpos no espaço. O algoritmo como já citado anteriormente apresenta dependência de dados a cada movimento.

6.1 Implementações

Um descrição das implementações da simulação n-Corpos gravitacional é apresentada nas subseções abaixo.

6.1.1 Implementação n-Corpos de complexidade quadrática

O algoritmo `ncorpos.hs` apresenta um complexidade computacional $O(n^2)$. Os corpos são representados por uma tripla cujas componentes são a posição a massa e a velocidade. A função `randomBodies` gera randomicamente esses corpos que são armazenados em uma lista. São aplicadas as funções `forçaIndividual` e `forcaTotalUmCorpo`, que calculam a força que os outros corpos exercem em um corpo e somam essas forças respectivamente. Esse processo é aplicado a todos os corpos através de um `map` na função `forcaTotalTodosCorpos`. Com o resultado obtido a função `movimento` calcula a nova posição e velocidade de cada corpo.

O particionamento da lista é feito por meio da função nativa `chunksOf` em `linhaNormal` que cria uma lista de listas de corpos. Por fim a função `forcaTotalParalelo` cria um `spark` para cada lista e a função `rseq` aguarda que o resultado esteja pronto para realizar o próximo passo. Nesse caso o particionamento dos dados é trivial, pois o cálculo das forças nos corpos individualmente levam teoricamente o mesmo tempo, logo se as partições tiverem o mesmo número de corpos, se obtém o particionamento ideal para o paralelismo. O algoritmo `ncorposOpengl.hs` representa graficamente o comportamento dos corpos no espaço.

6.1.2 Implementação Barnes-Hut

A utilização de paralelismo, e mesmo otimizações modificando a estrutura de armazenamento reduzem o tempo final de execução em um algoritmo quadrático. No entanto, o algoritmo acaba se tornando inviável em questão de desempenho para algumas centenas de corpos. Por essa razão optou-se por utilizar um algoritmo mais eficiente, o Barnes-Hut, por este apresentar uma complexidade $O(nlogn)$.

Para conseguir essa complexidade o algoritmo utiliza duas estruturas, uma *quadtree* e uma lista. A *quadtree* é uma generalização de uma árvore binária cujas folhas armazenam os corpos e os nodos armazenam o centro de massa de seus filhos. Com isso, não existe a necessidade de percorrer toda a árvore para calcular as forças, pode-se calcular diretamente pelos nodos, exceto quando o corpo pertence ao nodo. A criação da árvore é representada pela função `populaArvore` que recebe uma lista de corpos e recursivamente divide em quatro quadrantes essa lista. A recursão termina quando cada quadrante tiver apenas um corpo, ou quando esses quadrante tiver um tamanho muito pequeno para evitar que quando existam corpos na mesma posição o algoritmo entre em *loop*, pois nessa versão não tratamos colisões entre corpos.

O cálculo das forças é realizado pela função `somaForcas` que recebe como parâmetro um corpo e a árvore criada. A função verifica se o corpo pertence a algum dos filhos do nodo, em caso negativo a função calcula a força com o centro de massa desse filho descrito na fórmula da força gravitacional. Se o corpo pertence ao nodo a função é chamada recursivamente para cada um dos quatro filhos até que chegue a folha, se a folha é o próprio corpo então é retornado 0 como força. Por fim as forças obtidas são somadas. A função `forcaTotalTodosCorpos` realiza um `map` para que esse processo em todos os corpos da lista.

O paralelismo é realizado em duas etapas, primeiro na criação da árvore e depois no cálculo das forças. Para o cálculo das forças o processo é idêntico ao do algoritmo de complexidade quadrática: particiona-se a lista e utiliza-se função `forcaTotalParalelo`. A árvore pode ser desbalanceada: não se sabe em tempo de compilação quantas recursões cada filho da árvore realizará. Por essa razão, optou-se por utilizar um particionamento dinâmico, a função `populaArvore` verifica a quantidade de corpos um quadrante possui, se possuir uma quantidade relativamente grande é lançado um `spark` para realizar a próxima recursão, do contrário é realizada uma computação sequencial. Esse particionamento dinâmico não produz uma distribuição ideal das tarefas e a granularidade também varia dependendo do número total de corpos. Pelo fato de se ter criado um tipo de dados, a `quadtree`, que não é nativo da linguagem, o compilador não sabe como forçar uma avaliação dessa estrutura com o uso da função `force`. É necessário criar uma instância desse tipo de dado, onde se redefine a função base `rnf`, que availa cada um dos contrutores, utilizada pela função `force`.

6.2 Análise do resultados obtidos

Como os tempos de alocação e operações em listas já foram previamente analisados, não será realizada novamente essas análises. Os tempos apresentados nesse capítulo representam o tempo total de execução do algoritmo. Também os códigos foram compilados com as otimizações `-O3`.

6.3 Comparação entre os algoritmos sequencial e paralelo

Para essa análise fixou-se a quantidade de corpos e o passo, com isso buscou-se avaliar o desempenho dos dois algoritmos e realizar uma comparação entre eles. Inicialmente utilizou-se entradas iguais para ambos o algoritmos, com a quantidade de corpos em 600 e o número de passos realizados igual a 100. Cada passo corresponde a 60 minutos. O único parâmetro que foi variado é a quantidade de `sparks` lançados para a parte de particionamento estático. No Barnes-Hut a parte de particionamento dinâmico depende da quantide de corpos, exceto quando se utiliza 1 que representa o algortimo sequencial e não é lançando nenhum `spark`.

Tempos Médios	Núcleos				
	sequencial	2	4	8	16
Quadrático	26.35s	15.92s	9.61s	5.94s	5.22s
Barnes-Hut	0.85s	0.55s	0.39s	0.31s	0.30s

Tabela 6.1: Tempos médios da simulação n-Corpos variando o número de `sparks`.

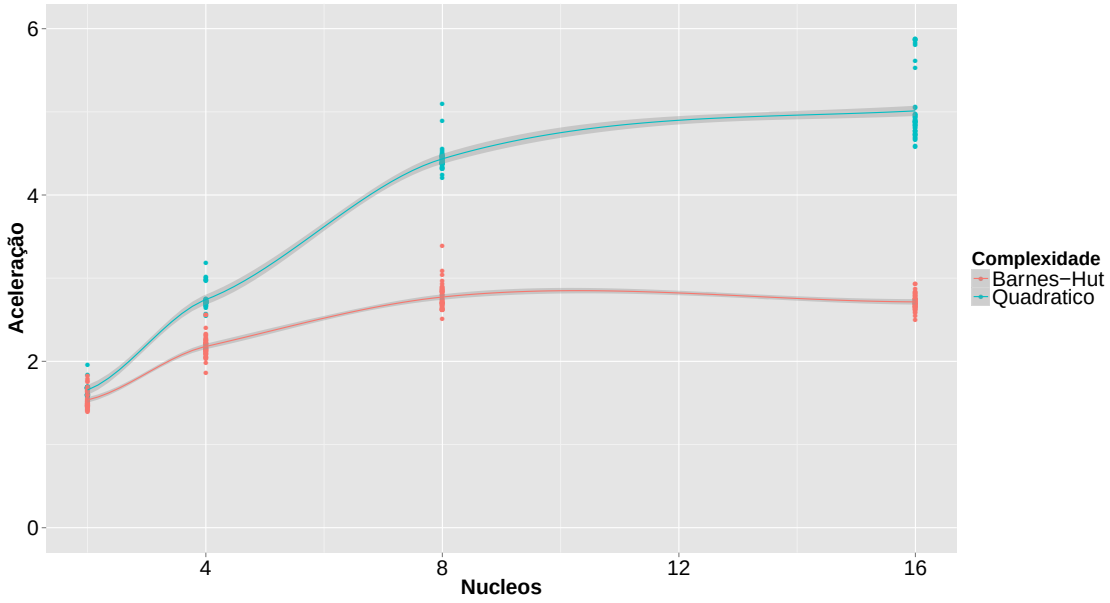

Figura 6.1: Aceleração da simulação n-Corpos.

A Tabela 6.1 mostra a grande diferença nos tempos de execução entre os algoritmos Barnes-Hut e o quadrático. A execução sequencial é 31 vezes mais lenta no algoritmo quadrático apresentando um tempo bastante elevado para apenas 600 corpos. Mesmo com paralelismo o tempo ainda é bastante elevado com o melhor desempenho chegando a 5.22 segundos. Isso mostra que para algoritmos com complexidade quadrática o uso de paralelismo apenas não é suficiente.

A Figura 6.1 mostra as acelerações resultantes. O algoritmo de complexidade quadrática obtém um valor próximo a 5 no melhor caso, semelhante a obtida no cálculo de Mandelbrot. Esse resultado pode ser explicado pelo fato de se obter um particionamento próximo do ideal, com isso a dependência de dados acaba não degradando o desempenho. Esse fator acaba ficando mais evidente pelo resultado do Barnes-Hut, nele o particionamento não é idealmente balanceado, com isso o tempo entre um passo e outro é

determinado pela tarefa que tiver uma carga maior. Outro fator que influencia é o particionamento dinâmico, nele é mais ressaltado o efeito do *work stealing*. Pelo comportamento do algoritmo é possível uma criação maior de `sparks` do que processadores disponíveis, por isso a diferença entre as acelerações aumenta conforme o aumento de utilização de processadores.

6.4 Análise da escalabilidade fraca da simulação n-Corpos gravitacional

Com essa análise deseja-se saber o efeito do aumento do tamanho do problema em um algoritmo com dependência de dados. Foram realizados experimentos para ambos os algoritmos o quadrático e o Barnes-Hut. Para os experimentos variou-se a quantidade de núcleos utilizados e a quantidade de corpos. A quantidade de passos foi fixada em 100 onde cada um representa 60 minutos.

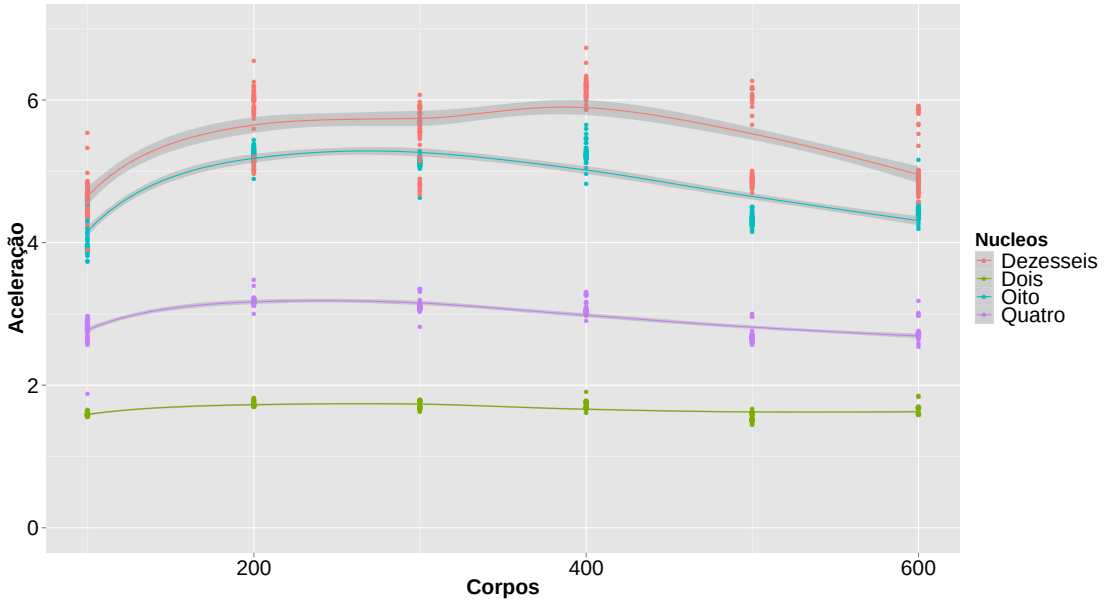

Figura 6.2: Escalabilidade Fraca no algoritmo quadrático.

A Figura 6.2 apresenta a aceleração em relação ao aumento de corpos no algoritmo quadrático. É possível perceber que inicialmente ocorre um crescimento da aceleração, entretanto após 400 corpos essa aceleração diminui. Esse resultado mostra que o algoritmo acaba não escalando para quantidades grandes de dados. Como causa pode-se citar a dependência de dados que começa a degradar o algoritmo a partir de determinada quantidade de dados, juntamente com o coletor de lixo.

A Figura 6.3 apresenta o mesmo experimento no algoritmo Barnes-Hut. Diferente do anterior, existe um crescimento da aceleração inicialmente. Entretanto, após um determinada quantidade de corpos a aceleração se mantém constante. Esse fator pode ser explicado pelo fato do algoritmo criar os sparks dinamicamente para criação da árvore, o que acaba limitando o crescimento da aceleração, contudo ele se mantém praticamente o mesmo independente da quantidade corpos.

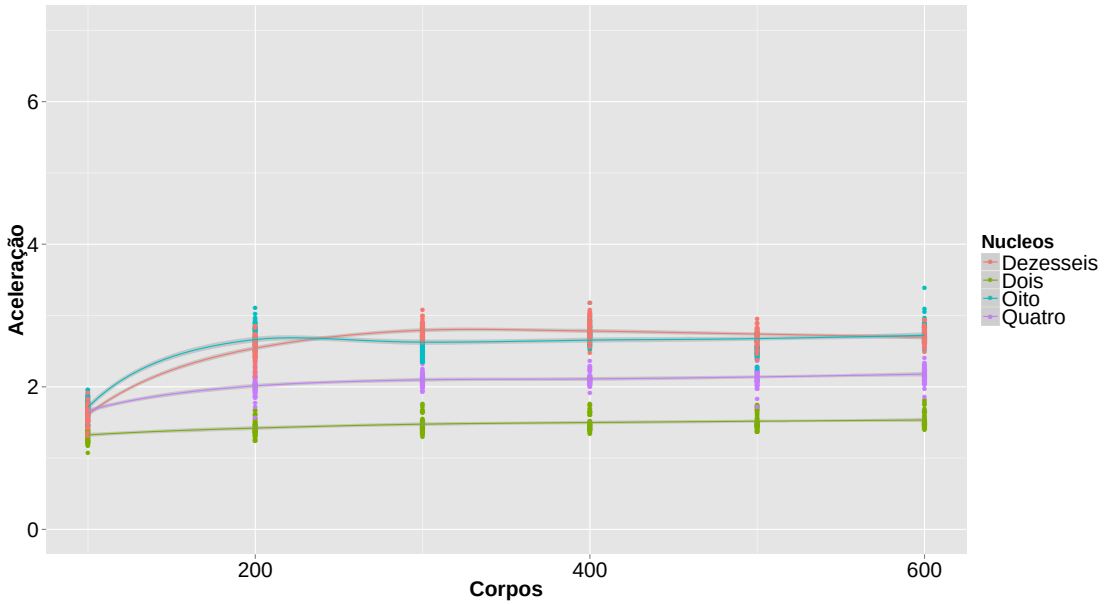

Figura 6.3: Escalabilidade Fraca no Barnes-Hut.

7 CONCLUSÃO

Desenvolver programas paralelos ainda é um grande desafio. Existem diversos padrões e linguagens que oferecem recursos para para criação de aplicações paralelas. Entretanto, não existe uma unanimidade na escolha da melhor forma de desenvolvimento. A linguagem Haskell surge como uma alternativa menos convencional para esse problema. Por ser uma linguagem funcional pura, ela pode ser melhor explorada nesse novo contexto, visto que apresenta novas soluções para problemas que persistem até o momento em linguagens imperativas. Pensar algoritmos de forma paralela não é natural para muitos desenvolvedores. Da mesma forma algoritmos baseados em linguagens funcionais encontram rejeição por alguns programadores. Contudo, problemas resolvidos através de linguagens imperativas de forma sequencial nem sempre produzem as melhores soluções.

Durante o desenvolvimento desse trabalho e dos algoritmos, foi possível perceber que a linguagem Haskell tem uma quantidade de material muito pequena quando comparado a linguagens mais tradicionais como Java e Ansi C. Esse trabalho buscou contribuir investigando se a linguagem é adequada para o desenvolvimento de programas paralelos. Os experimentos realizados nesse trabalho tiveram como objetivo avaliar aspectos da linguagem Haskell nesse contexto.

Haskell produz um código bastante modular, oferecendo funções bastante práticas. Consegue-se resolver o problema do cálculo do conjunto de Mandelbrot com poucas linhas de código. Esse comportamento parece ser uma característica da linguagem o que torna os programas fáceis de serem compreendidos e de fácil manutenção. Um ponto negativo da linguagem é a dificuldade de depuração, com falta de ferramentas eficientes para essa tarefa no desenvolvimento dos programas.

Através dos experiementos foi possível concluir que Haskell oferece abstrações de alto nível para o programador e simplifica a criação de códigos paralelos. Haskell oferece uma grande diversidade de bibliotecas para paralelismo, sendo que algumas delegam a maior parte da tarefa ao compilador enquanto outras deixam a cargo do desenvolvedor, sendo possível utilizar-se em conjunto essas bibliotecas. Os módulos `Repa` e `Control.Parallel.Strategies`, utilizados nos experimentos, representam bem essa flexibilidade que a linguagem oferece. Embora as bibliotecas apresentem um facilidade de utilização, para se obter melhores desempenhos é necessário um conhecimento mais aprofundado da linguagem e uma análise prévia do algoritmo. Questões como a estrutura a ser utilizada e a melhor forma de particionar os dados acabam tendo um impacto significativo no tempo de execução.

Os tempos de execução da linguagem Haskell quando comparados com Ansi C são

superiores, entretanto não é uma diferença muito acentuada. Um fator relevante na linguagem Haskell é o coletor de lixo. Em programas paralelos o coletor de lixo tem uma influência maior no desempenho da linguagem, pois ele é paralelizado implícitamente pelo compilador e o ganho não é ideal. Pode-se considerar Haskell aplicável para programação paralela pelos resultados obtidos. A maioria das bibliotecas para paralelismo da linguagem são relativamente recentes, por essa razão é possível que exista uma evolução das mesmas.

7.1 Trabalhos Futuros

Alguns tópicos de pesquisa que apareceram durante o desenvolvimento desse trabalho não foram mais detalhados por uma questão de escopo. Entre eles podemos citar:

- Um estudo sobre o compilador GHC. O compilador acaba tendo um papel fundamental no paralelismo, em especial o coletor de lixo. Como resultado, uma compreensão maior do GHC pode levar a construção de programas com melhor desempenho.
- Um análise das técnicas de *profiling*. As técnicas de *profiling* podem levar a uma análise mais profunda dos fatores que influênciam no desempenho das aplicações. Através dessas técnicas é possível a identificação dos *space leaks*, ou de funções que são ineficentes, entre outros.
- A exploração de outros módulos para paralelismo não destacados nesse trabalho. A biblioteca `Accelerate` é uma delas e explora o uso de GPUs para obtenção de alto desempenho em aplicações.

7.2 Artigos submetidos

Um artigo científico contendo os principais resultados dessa monografia foi submetido para o fórum de iniciação científica da 15º Escola Regional de Alto Desempenho (ERAD 2015).

REFERÊNCIAS

ASANOVIC, K. et al. A view of the parallel computing landscape. **Communications of the ACM - A View of Parallel Computing**, [S.l.], v.52, p.56–67, 2009.

Barnes, J.; Hut, P. A hierarchical O(N log N) force-calculation algorithm. **nat**, [S.l.], v.324, p.446–449, Dec. 1986.

BARNEY, B.; LABORATORY, L. L. N. **POSIX Threads Programming**. [Online; acessado em Mai 26, 2014], `https://computing.llnl.gov/tutorials/pthreads/`.

BARNEY, B.; LABORATORY, L. L. N. **Introduction to Parallel Computing**. [Online; acessado em Mai 25, 2014], `https://computing.llnl.gov/tutorials/parallel_comp/`.

BROCK, D.; MOORE, G. **Understanding Moore's Law**: four decades of innovation. [S.l.]: Chemical Heritage Foundation, 2006.

CHURCH, A. A Set of Postulates for the Foundation of Logic Part I. **Annals of Mathematics**, [S.l.], v.33, n.2, p.346–366, 1932. http://www.jstor.org/stable/1968702Electronic Edition.

COULOURIS, G. F.; DOLLIMORE, J. **Distributed Systems**: concepts and design. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

ERISSON, S. **The Haskell Programming Language**. [Online; acessado em Jun 6, 2014], `http://www.haskell.org/`.

ERISSON, S. **The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.2**. [S.l.: s.n.], 2014.

FRANK, D. J. Power-constrained CMOS Scaling Limits. **IBM J. Res. Dev.**, [S.l.], v.46, n.2-3, p.235–244, Mar. 2002.

GUSTAFSON, J. L. Reevaluating Amdahl's Law. **Commun. ACM**, New York, NY, USA, v.31, n.5, p.532–533, May 1988.

HASKELL. **Control.Parallel.Strategies**. [Online; acessado em Mai 26, 2014], `http://hackage.haskell.org/package/parallel-parallel-3.1.0.1/docs/Control-Parallel-Strategies.html`.

HASKELLWIKI. **Lazy vs. non-strict**. [Online; acessado em Mai 25, 2014], `http://www.haskell.org/haskellwiki/Lazy_vs._non-strict`.

KELLER, G. et al. Regular, Shape-polymorphic, Parallel Arrays in Haskell. **SIGPLAN Not.**, New York, NY, USA, v.45, n.9, p.261–272, Sept. 2010.

LIPOVACA, M. **Learn You a Haskell for Great Good! A Beginner's Guide**. 1.ed. [S.l.]: No Starch Press, 2011.

MANDELBROT, B. B. **The Fractal Geometry of Nature**. New York: W. H. Freedman and Co., 1983.

MARLOW, S. **Parallel and Concurrent Programming in Haskell**: techniques for multicore and multithreaded programming. [S.l.]: "O'Reilly Media, Inc.", 2013.

MARLOW, S. **The Glasgow Haskell Compiler**. [Online; acessado Mai 25, 2014], `http://www.haskell.org/ghc/`.

MILNER, R.; TOFTE, M.; MACQUEEN, D. **The Definition of Standard ML**. Cambridge, MA, USA: MIT Press, 1997.

MOGGI, E. Notions of Computation and Monads. **Inf. Comput.**, Duluth, MN, USA, v.93, n.1, p.55–92, July 1991.

NEUMANN, J. von. First Draft of a Report on the EDVAC. In: STERN, N. B. (Ed.). . [S.l.]: Digital Press, 1981.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003.

STATISTICS, I. for; WU, M. of. **The R Project for Statistical Computing**. [S.l.: s.n.], 2010.

SUSSMAN, G. J.; STEELE JR., G. L. Scheme: a interpreter for extended lambda calculus. **Higher Order Symbol. Comput.**, Hingham, MA, USA, v.11, n.4, p.405–439, Dec. 1998.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd.ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

TURNER, D. An overview of Miranda. **Bulletin of the EATCS**, [S.l.], v.33, p.103–114, 1987.

WADLER, P. The Essence of Functional Programming. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 19. **Proceedings...** ACM, 1992. p.1–14. (POPL '92).

WILKINSON, B.; ALLEN, M. **Parallel Programming**: techniques and applications using networked workstations and parallel computers. [S.l.]: Prentice-Hall, Inc., 1999.

APÊNDICE A CÁLCULO DO FRACTAL DE MANDELBROT

A.1 Mandelbrot_sequencial.hs

```
1  import Graphics.UI.GLUT
2  import System.Environment
3  import Data.Complex
4  import Data.Time.Clock
5  import Control.DeepSeq
6  import Foreign.C.Types
7
8  imageHeight' = 4.0:: GLdouble
9  imageWidth' =  imageHeight'
10 tela = 512
11 numeroIteracaoes = 50
12
13 plano passo = [(width,height) | width<-[-imageWidth'/2,-imageWidth'/2+passo..
       imageWidth'/2],height<-[-imageHeight'/2,-imageHeight'/2+passo..imageHeight
       '/2]]
14 planoComplexo passo= map(\(real,imaginario)->real:+imaginario) (plano passo)
15
16 type Iteracao           = Int
17 type Width'             = GLdouble
18 type Height'            = GLdouble
19 type Cor                = Int
20 type Pixel              = (Height',Width',Cor)
21 type Pontos             = [Pixel]
22 type NroComplexo        = Complex GLdouble
23 type PlanoComplexo      = [Complex GLdouble]
24
25 instance NFData CDouble
26
27 iteracaoMandelbrot::Iteracao->NroComplexo->NroComplexo->Pixel
28 iteracaoMandelbrot  0 complexo complexoIterado=  (realPart (complexo),imagPart
       (complexo),0)
29 iteracaoMandelbrot iterador complexo complexoIterado
30  | magnitude complexoIterado > 2 =(realPart (complexo),imagPart (complexo),
        iterador)
31  | otherwise = iteracaoMandelbrot(iterador-1) complexo  zn
32  where
33   zn = (complexoIterado^2)+(complexo)
34
35 mandelbrot l = map(\pixel->iteracaoMandelbrot numeroIteracaoes pixel pixel ) l
36
37 main :: IO ()
38 main = do
39  args<- getArgs
40  let (tamanho) = processaArgs args
41  let telaArg   =  tamanho
```

```
42   let passoArg  =  imageHeight'/telaArg
43   t0 <- getCurrentTime
44   t1 <- (planoComplexo passoArg)  `deepseq` getCurrentTime
45   t2 <- mandelbrot  (planoComplexo passoArg) `deepseq` getCurrentTime
46   let totalPontos = length(planoComplexo passoArg)
47
48   putStrLn $ (show $ (totalPontos))  ++ ",␣" ++  (show $ diffUTCTime t1 t0) ++ "
         ,␣" ++ (show $ diffUTCTime t2 t1) ++ "\n"
49
50 processaArgs args = case args of
51       (a:_) -> (read a)
52       _           -> (512)
```

A.2 Mandelbrot_paralelo.hs

```
1  import Graphics.UI.GLUT
2  import System.Environment
3  import Data.Complex
4  import Data.Time.Clock
5  import Control.DeepSeq
6  import Foreign.C.Types
7  import Control.Parallel.Strategies
8  import Data.List
9  import Data.List.Split
10
11 imageHeight' = 4.0:: Double
12 imageWidth' =  imageHeight'
13
14 numeroIteracaoes = 50
15 tela = 512
16
17 plano passo = [(height,width) | width<-[-imageWidth'/2,-imageWidth'/2+passo..
       imageWidth'/2],height<-[-imageHeight'/2,-imageHeight'/2+passo..imageHeight
       '/2]]
18 planoComplexo passo= map(\(real,imaginario)->real:+imaginario) (plano passo)
19
20 type Iteracao           = Int
21 type Width'             = Double
22 type Height'            = Double
23 type Cor                = Int
24 type Pixel              = (Height',Width',Cor)
25 type Pontos             = [Pixel]
26 type NroComplexo        = Complex Double
27 type PlanoComplexo      = [Complex Double]
28 type NumChunks          = Int
29 type TamanhoChunks      = Int
30 type TamanhoLinha       = Int
31 type NumBlocos          = Int
32
33 instance NFData CDouble
34
35 rdeepPar :: NFData a => Strategy a
36 rdeepPar pontos = rpar(force pontos)
37
38 mandelbrot :: [[NroComplexo]]->Iteracao->Eval [Pixel]
39 mandelbrot [] _  =return []
40 mandelbrot (listaPonto:plano) numeroIteracaoes = do
41  p1 <-rdeepPar(map(\pixel->iteracaoPontoMandelbrot numeroIteracaoes pixel pixel
        ) listaPonto)
42  p2 <- mandelbrot plano numeroIteracaoes
43  return (p1++p2)
44
45 iteracaoPontoMandelbrot::Iteracao->NroComplexo->NroComplexo->Pixel
```

```
46 iteracaoPontoMandelbrot  0 complexo complexoIterado=  (realPart (complexo),
       imagPart (complexo),0)
47 iteracaoPontoMandelbrot iterador complexo complexoIterado
48  | magnitude complexoIterado > 2 =(realPart (complexo),imagPart (complexo),
        iterador)
49  | otherwise = iteracaoPontoMandelbrot(iterador-1) complexo  zn
50  where
51   zn = (complexoIterado^2)+(complexo)
52
53 linhaNormal :: PlanoComplexo->TamanhoChunks->[[NroComplexo]]
54 linhaNormal plano tamanhoChunk= chunksOf tamanhoChunk plano
55
56 linhaRoundRobin :: PlanoComplexo->NumChunks->Bool->[[NroComplexo]]
57 linhaRoundRobin [] _ _ =  []
58 linhaRoundRobin plano numSubLists ultimaRecursao =
59  if ultimaRecursao == False then ([take numSubLists plano] ++ linhaRoundRobin (
        drop numSubLists plano) numSubLists False)
60  else transpose ([take numSubLists plano] ++ linhaRoundRobin (drop numSubLists
        plano) numSubLists False)
61
62 colunaNormal :: PlanoComplexo->TamanhoLinha->TamanhoChunks->[[NroComplexo]]
63 colunaNormal plano tamanhoLinha tamanhoChunk = do
64  let planoColuna =concat(transpose (chunksOf tamanhoLinha plano))
65  chunksOf tamanhoChunk planoColuna
66
67 colunaRoundRobin :: PlanoComplexo->TamanhoLinha->NumChunks->Bool->[[NroComplexo
       ]]
68 colunaRoundRobin [] _ _ _ =  []
69 colunaRoundRobin plano tamanhoLinha numSubLists ultimaRecursao =
70    if ultimaRecursao == False then ([take numSubLists plano] ++
          colunaRoundRobin (drop numSubLists plano) tamanhoLinha numSubLists False
          )
71    else do  let planoColuna =concat(transpose (chunksOf tamanhoLinha plano))
72    transpose ([take numSubLists planoColuna] ++ colunaRoundRobin (drop
          numSubLists planoColuna) tamanhoLinha numSubLists False)
73
74 main :: IO ()
75 main = do
76  (_, args) <- getArgsAndInitialize
77  let (cores,totalPontos,particao) = processaArgs args
78  let telaArg =  totalPontos
79  let passoArg =imageHeight'/telaArg
80  let numIteracoes = 50
81  let numeroPontos = length(planoComplexo passoArg)
82  let tamanhoChunks = (length(planoComplexo passoArg) `div` (cores))+1
83  t0 <- getCurrentTime
84  let planoParticao particao = case particao of
85   0 -> linhaNormal (planoComplexo passoArg) tamanhoChunks
86   1 -> linhaRoundRobin (planoComplexo passoArg) (cores) True
87   2 -> colunaNormal (planoComplexo passoArg)  (floor telaArg) tamanhoChunks
88   _ -> colunaRoundRobin (planoComplexo passoArg)  (floor telaArg) (cores) True
89
90  t1 <- (planoParticao particao)  `deepseq` getCurrentTime
91  let avaliacao = mandelbrot (planoParticao particao)  numIteracoes
92     t2 <-runEval(avaliacao)  `deepseq` getCurrentTime
93
94  let tipoParticao particao = case particao of
95   0 -> "Linha␣Normal"
96   1 -> "Linha␣RoundRobin"
97   2 -> "Coluna␣Normal"
98   _ -> "Coluna␣RoundRobin"
99
```

```
100   putStrLn $ (show $ (cores))  ++ ",␣" ++ (show $ (numeroPontos)) ++ ",␣" ++(
          show $ tipoParticao particao)  ++ ",␣" ++ (show $ diffUTCTime t1 t0) ++ ",
          ␣" ++ (show $ diffUTCTime t2 t1) ++ "\n"
101
102   where
103     processaArgs args = case args of
104       (a:b:c:_) -> (read a,read b,read c)
105       _         -> (4,512,0)
```

A.3 Mandelbrot_paralelo_repa.hs

```
1   import System.Environment
2   import Data.Complex
3   import Data.Time.Clock
4   import Control.DeepSeq
5   import Foreign.C.Types
6   import Control.Parallel.Strategies
7   import Data.List as List
8   import Data.List.Split as Split
9   import qualified Data.Array.Repa as Repa
10
11  imageHeight' = 4.0:: Double
12  imageWidth' =  imageHeight'
13
14  numeroIteracaoes = 50
15  tela = 512
16
17  plano passo = [(height,width) | width<-[-imageWidth'/2,-imageWidth'/2+passo..
        imageWidth'/2],height<-[-imageHeight'/2,-imageHeight'/2+passo..imageHeight
        '/2]]
18  planoComplexo passo= map(\(real,imaginario)->real:+imaginario) (plano passo)
19
20  type ArrayRepaU = Repa.Array Repa.U Repa.DIM2 (Complex Double)
21  type ArrayRepaD = Repa.Array Repa.D Repa.DIM2 (Complex Double)
22
23  planoComplexoRepa passo linha coluna = Repa.fromFunction (Repa.Z Repa.:. linha
        Repa.:.coluna ) (\(Repa.Z Repa.:. i Repa.:.j)-> (((-imageWidth'/2)+ (
        fromIntegral i)* passo):+ ((-imageHeight'/2)+ (fromIntegral j)*passo))   ::
         (Complex Double)):: Repa.Array Repa.D Repa.DIM2 (Complex Double)
24
25  type Iteracao           = Int
26  type Width'             = Double
27  type Height'            = Double
28  type Cor                = Int
29  type Pixel              = (Height',Width',Cor)
30  type Pontos             = [Pixel]
31  type NroComplexo        = Complex Double
32  type PlanoComplexo      = [Complex Double]
33  type NumChunks          = Int
34  type TamanhoChunks      = Int
35  type TamanhoLinha       = Int
36  type NumBlocos          = Int
37  type Tamanho            = Int
38  type Indice             = Int
39  type Linha              = Int
40  type Coluna             = Int
41  instance NFData CDouble
42
43
44  rdeepPar :: NFData a => Strategy a
45  rdeepPar pontos = rpar(force pontos)
46
47  mandelbrot :: [[NroComplexo]]->Iteracao->Eval [Pixel]
```

```
48 mandelbrot [] _  =return []
49 mandelbrot (listaPonto:plano) numeroIteracaoes = do
50  p1 <-rdeepPar(map(\pixel->iteracaoPontoMandelbrot numeroIteracaoes pixel pixel
       ) listaPonto)
51  p2 <- mandelbrot plano numeroIteracaoes
52  return (p1++p2)
53
54 iteracaoPontoMandelbrot::Iteracao->NroComplexo->NroComplexo->Pixel
55 iteracaoPontoMandelbrot  0 complexo complexoIterado=  (realPart (complexo),
     imagPart (complexo),0)
56 iteracaoPontoMandelbrot iterador complexo complexoIterado
57  | magnitude complexoIterado > 2 =(realPart (complexo),imagPart (complexo),
      iterador)
58  | otherwise = iteracaoPontoMandelbrot(iterador-1) complexo  zn
59  where
60   zn = (complexoIterado^2)+(complexo)
61
62 repa :: ArrayRepaU->NumChunks->NumChunks->Bool->[[NroComplexo]]-- [[NroComplexo
     ]]
63 repa _ 0 _ _ = []
64 repa array numChunksIterado numChunks False = do
65  let z = Repa.slice array (Repa.Any Repa.:. ((numChunks-numChunksIterado):: Int
      ) Repa.:. Repa.All )
66  [Repa.toList z]++ repa array (numChunksIterado-1) numChunks False
67
68 linhaNormal :: ArrayRepaD->Int->Int-> IO (ArrayRepaU)
69 linhaNormal array cores numeroPontos= do let linha = cores
70  let coluna = numeroPontos `div` cores
71  let arrayFinal  = (Repa.reshape (Repa.Z Repa.:. linha Repa.:.coluna ) array)
72  Repa.computeUnboxedP arrayFinal
73
74 linhaRoundRobin :: ArrayRepaD->Int->Int-> IO (ArrayRepaU)
75 linhaRoundRobin array cores numeroPontos= let linha = numeroPontos `div` cores
76  let coluna = cores
77  let arrayReshape  = (Repa.reshape (Repa.Z Repa.:. linha Repa.:.coluna ) array)
78  let arrayFinal = Repa.transpose arrayReshape
79  Repa.computeUnboxedP arrayFinal
80
81 colunaNormal :: ArrayRepaD->Int->Int-> IO (ArrayRepaU)
82 colunaNormal array cores numeroPontos =
83  let newArray = Repa.transpose array
84  linhaNormal newArray cores numeroPontos
85
86 colunaRoundRobin :: ArrayRepaD->Int->Int-> IO (ArrayRepaU)
87 colunaRoundRobin array cores numeroPontos =
88  let newArray = Repa.transpose array
89  linhaRoundRobin newArray cores numeroPontos
90
91 main :: IO ()
92 main = do
93  args<- getArgs
94  let (cores,totalPontos,particao) = processaArgs args
95  let telaArg =  totalPontos
96  let passoArg =imageHeight'/telaArg
97  let numIteracoes = 50
98  let numeroPontos = round ((totalPontos)*(totalPontos))
99  let tamanhoChunks = numeroPontos `div` cores
100  t0 <- getCurrentTime
101  let arrayPlanoComplexoRepa = planoComplexoRepa passoArg (round totalPontos) (
       round totalPontos)
102
103  arrayPlanoComplexoRepaParallel <- case particao of
104   0->  linhaNormal arrayPlanoComplexoRepa cores numeroPontos
```

```
105    1->  linhaRoundRobin arrayPlanoComplexoRepa cores numeroPontos
106    2->  colunaNormal arrayPlanoComplexoRepa cores numeroPontos
107    _->  colunaRoundRobin arrayPlanoComplexoRepa cores numeroPontos
108
109   let planoParticao particao = repa arrayPlanoComplexoRepaParallel  cores cores
          False
110   t1 <- ((planoParticao particao))  `deepseq` getCurrentTime
111   let avaliacao = mandelbrot (planoParticao particao)  numIteracoes
112      t2 <-runEval(avaliacao)  `deepseq` getCurrentTime
113   let tipoParticao particao = case particao of
114    0 -> "Linha␣Normal"
115    1 -> "Linha␣RoundRobin"
116    2 -> "Coluna␣Normal"
117    _ -> "Coluna␣RoundRobin"
118
119   putStrLn $ (show $ (cores))  ++ ",␣" ++ (show $ (numeroPontos)) ++ ",␣" ++(
          show $ tipoParticao particao)  ++ ",␣" ++ (show $ diffUTCTime t1 t0) ++ ",
          ␣" ++ (show $ diffUTCTime t2 t1) ++ "\n"
120
121   where
122     processaArgs args = case args of
123       (a:b:c:_) -> (read a,read b,read c)
124       _           -> (4,512,0)
```

A.4 Mandelbrot_e_alocacao_paralelo.hs

```
1   import Graphics.UI.GLUT
2   import System.Environment
3   import Data.Complex
4   import Data.Time.Clock
5   import Control.DeepSeq
6   import Foreign.C.Types
7   import Control.Parallel.Strategies
8   import Data.List
9   import Data.List.Split
10
11  imageHeight' = 4.0:: Double
12  imageWidth' =  imageHeight'
13
14  numeroIteracaoes = 50
15  tela = 512
16
17  type Iteracao           = Int
18  type Width'             = Double
19  type Height'            = Double
20  type Cor                = Int
21  type Pixel              = (Height',Width',Cor)
22  type Pontos             = [Pixel]
23  type NroComplexo        = Complex Double
24  type PlanoComplexo      = [Complex Double]
25  type NumChunks          = Int
26  type TamanhoChunks      = Int
27  type TamanhoLinha       = Int
28  type NumBlocos          = Int
29  type P0Linha          = Double
30  type P0Coluna          = Double
31  type Contador          = Double
32  type Passo           = Double
33  type Total          = Double
34  type TamanhoColuna  = Double
35  instance NFData CDouble
36
37  rdeepPar :: NFData a => Strategy a
```

```
38 rdeepPar pontos = rpar(force pontos)
39
40 mandelbrot :: [[NroComplexo]]->Iteracao->Eval [Pixel]
41 mandelbrot [] _  =return []
42 mandelbrot (listaPonto:plano) numeroIteracaoes = do
43  p1 <-rdeepPar(map(\pixel->iteracaoPontoMandelbrot numeroIteracaoes pixel pixel
       ) listaPonto)
44  p2 <- mandelbrot plano numeroIteracaoes
45  return (p1++p2)
46
47 iteracaoPontoMandelbrot::Iteracao->NroComplexo->NroComplexo->Pixel
48 iteracaoPontoMandelbrot  0 complexo complexoIterado=  (realPart (complexo),
     imagPart (complexo),0)
49 iteracaoPontoMandelbrot iterador complexo complexoIterado
50  | magnitude complexoIterado > 2 =(realPart (complexo),imagPart (complexo),
       iterador)
51  | otherwise = iteracaoPontoMandelbrot(iterador-1) complexo  zn
52  where
53   zn = (complexoIterado^2)+(complexo)
54
55 planoLinhaNormal:: NumChunks->NumChunks->P0Linha->Passo->TamanhoLinha->Eval [[
     NroComplexo]]
56 planoLinhaNormal _ 0 _ _ _ = return []
57 planoLinhaNormal numSublistas numSublistasIterado linha passo tamanhoLinha= do
58  let contador =numSublistas - numSublistasIterado +1
59  let fimLinha = (-imageWidth'/2)+(passo*(fromIntegral(contador*tamanhoLinha `
       div` numSublistas)::Double)-passo)
60  let subLista = [(height,width) | width<-[linha,(linha+passo)..(fimLinha)],
       height<-[-imageHeight'/2,(-imageHeight'/2)+passo..imageHeight'/2]]
61   p1 <-rdeepPar([map(\(real,imaginario)->real:+imaginario) subLista])
62  p2 <- planoLinhaNormal numSublistas (numSublistasIterado-1) (fimLinha+passo)
       passo tamanhoLinha
63  return (p1 ++ p2)
64
65 planoLinhaRoundRobin :: NumChunks->NumChunks->P0Coluna->Passo->Eval [[
     NroComplexo]]
66 planoLinhaRoundRobin  _ 0 _ _ = return []
67 planoLinhaRoundRobin numSublistas numSublistasIterado coluna passo = do
68  let contador = fromIntegral (numSublistas - numSublistasIterado)
69  let newPasso = (fromIntegral numSublistas)*passo
70  let subLista =[(height,width) | width<-[-imageWidth'/2,(-imageWidth'/2)+passo
       ..(imageWidth'/2)],height<-[coluna+contador*passo,coluna+contador*passo+
       newPasso..imageHeight'/2+contador*passo - newPasso]]
71  p1 <-rdeepPar([map(\(real,imaginario)->real:+imaginario) subLista])
72  p2 <-planoLinhaRoundRobin numSublistas (numSublistasIterado-1) coluna passo
73  return (p1 ++ p2)
74
75 planoColunaNormal:: NumChunks->NumChunks->P0Linha->Passo->TamanhoLinha->Eval [[
     NroComplexo]]
76 planoColunaNormal _ 0 _ _ _ = return []
77 planoColunaNormal numSublistas numSublistasIterado linha passo tamanhoLinha= do
78  let contador =numSublistas - numSublistasIterado +1
79  let fimLinha = (-imageWidth'/2)+(passo*(fromIntegral(contador*tamanhoLinha `
       div` numSublistas)::Double)-passo)
80  let subLista = [(width,height) | width<-[linha,(linha+passo)..(fimLinha)],
       height<-[-imageHeight'/2,(-imageHeight'/2)+passo..imageHeight'/2]]
81   p1 <-rdeepPar([map(\(real,imaginario)->real:+imaginario) subLista])
82  p2 <- planoLinhaNormal numSublistas (numSublistasIterado-1) (fimLinha+passo)
       passo tamanhoLinha
83  return (p1 ++ p2)
84
85 planoColunaRoundRobin :: NumChunks->NumChunks->P0Coluna->Passo->Eval [[
     NroComplexo]]
```

```
86  planoColunaRoundRobin  _ 0 _ _ = return []
87  planoColunaRoundRobin numSublistas numSublistasIterado coluna passo = do
88   let contador = fromIntegral (numSublistas - numSublistasIterado)
89   let newPasso = (fromIntegral numSublistas)*passo
90   let subLista =[(width,height) | width<-[-imageWidth'/2,(-imageWidth'/2)+passo
         ..(imageWidth'/2)],height<-[coluna+contador*passo,coluna+contador*passo+
         newPasso..imageHeight'/2+contador*passo - newPasso]]
91   p1 <-rdeepPar([map(\(real,imaginario)->real:+imaginario) subLista])
92   p2 <-planoLinhaRoundRobin numSublistas (numSublistasIterado-1) coluna passo
93   return (p1 ++ p2)
94
95  main :: IO ()
96  main = do
97
98   (_, args) <- getArgsAndInitialize
99   let (cores,totalPontos,particao) = processaArgs args
100  let telaArg =totalPontos-1
101  let numeroPontos = totalPontos*totalPontos
102  let passoArg =imageHeight'/telaArg
103  let nroSubLinhas = cores
104  let numIteracoes = 50
105  t0 <- getCurrentTime
106
107  let planoParticao particao = case particao of
108   0 -> planoLinhaNormal cores cores (-imageWidth'/2) passoArg (round
         totalPontos)
109   1 -> planoLinhaRoundRobin  cores cores (-imageHeight'/2) passoArg
110   2 -> planoColunaNormal cores cores (-imageWidth'/2) passoArg (round
         totalPontos)
111   _ -> planoColunaRoundRobin  cores cores (-imageHeight'/2) passoArg
112
113  let alocacao = runEval (planoParticao particao)
114  t1 <- (alocacao)  `deepseq` getCurrentTime
115  let avaliacao = mandelbrot alocacao  numIteracoes
116        t2 <-runEval(avaliacao)  `deepseq` getCurrentTime
117
118  let tipoParticao particao = case particao of
119   0 -> "Linha␣Normal"
120   1 -> "Linha␣RoundRobin"
121   2 -> "Coluna␣Normal"
122   _ -> "Coluna␣RoundRobin"
123
124  putStrLn $ (show $ (cores))  ++ ",␣" ++ (show $ (numeroPontos)) ++ ",␣" ++(
       show $ tipoParticao particao)  ++ ",␣" ++ (show $ diffUTCTime t1 t0) ++ ",
       ␣" ++ (show $ diffUTCTime t2 t1) ++ "\n"
125  where
126    processaArgs args = case args of
127      (a:b:c:_) -> (read a,read b,read c)
128      _           -> (4,512,0)
```

A.5 Mandelbrot_sequencial.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <complex.h>
4  #include <GL/glut.h>
5  #include <sys/timeb.h>
6
7  struct Pixel
8  {
9   double width;
10  double height;
11  int cor;
```

```
12 } ;
13
14 struct Pixel IteracaoMandelbrot(double complex ponto,int numeroDeIteracoes)
15 {
16  struct Pixel pixel;
17  int iterador;
18  double complex pontoIterado;
19  pontoIterado = ponto;
20  for(iterador = numeroDeIteracoes;iterador>=0;iterador--)
21  {
22   if (cabs(pontoIterado)>2 )
23   {
24    pixel.width = creal(ponto);
25    pixel.height = cimag(ponto);
26    pixel.cor = iterador;
27    return pixel;
28   }
29   pontoIterado = cpow(pontoIterado,2)+ponto;
30
31  }
32 }
33
34 struct Pixel** verificaPlanoComplexo(int x,int y ,double complex planoComplexo[
       x][y],int numeroDeIteracoes)
35 {
36  int i,j,z;
37  struct timeb start, end;
38  long seconds;
39  int militm;
40
41  ftime(&start);
42
43  struct Pixel **pixeis= (struct Pixel **)malloc(x*sizeof(struct Pixel ));
44  for(i = 0; i < x; i++)
45   pixeis[i] = malloc(y*sizeof(struct Pixel));
46
47  ftime(&end);
48  seconds = (long)(end.time - start.time);
49  militm = end.millitm - start.millitm;
50  if (0 > militm) {militm += 1000;seconds--; }
51  printf("tempo␣de␣alocacao␣de␣memoria␣%ld.%03d␣seconds\n", seconds, militm);
52
53  ftime(&start);
54
55  for (j=0;j<y;j++)
56   for (i=0;i<x;i++)
57    pixeis[i][j]=IteracaoMandelbrot(planoComplexo[i][j],numeroDeIteracoes);
58  ftime(&end);
59
60  seconds = (long)(end.time - start.time);
61  militm = end.millitm - start.millitm;
62  if (0 > militm) {militm += 1000;seconds--; }
63  printf("tempo␣do␣calculo␣%ld.%03d␣seconds\n", seconds, militm);
64  return pixeis;
65 }
66
67 int main(int argc,char** argv)
68 {
69  double tela,imageHeigth,imageWidth,numeroDeIteracoes;
70  tela = 512;
71  imageHeigth = 4;
72  imageWidth = 4;
73  numeroDeIteracoes = 50;
```

```
74
75   double passo = imageHeigth/tela;
76   double valorx,valory;
77   double teste ;
78   int x,y,i,j;
79
80   x = floor((imageHeigth/passo)+1);
81   y = floor((imageWidth/passo)+1);
82
83   double complex planoComplexo[x][y];
84   struct Pixel** pixeis;
85
86   valory=-imageHeigth/2;
87
88   for (j=0;j<y;j++)
89   {
90
91    valorx=-imageWidth/2;
92    for (i=0;i<x;i++)
93    {
94     planoComplexo[i][j]=valorx+valory*I;
95     valorx=valorx+passo;
96    }
97    valory=valory+passo;
98   }
99
100  for (i=1;i<4;i++)
101   data[i] = i;
102
103  pixeis=verificaPlanoComplexo(x,y,planoComplexo,numeroDeIteracoes);
104
105  return 0;
106 }
```

A.6 Mandelbrot_Paralelo.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <complex.h>
4  #include <GL/glut.h>
5  #include <sys/timeb.h>
6  #include <pthread.h>
7
8  struct Pixel
9  {
10  double width;
11  double height;
12  int cor;
13 } ;
14
15 typedef struct str_thdata
16 {
17     int thread_no;
18     double complex *planoComplexoParticionado;
19     int size;
20
21 } thdata;
22
23 struct Pixel IteracaoPontoMandelbrot(double complex ponto,int numeroDeIteracoes
       )
24 {
25  struct Pixel pixel;
26  int iterador;
```

```
27   double complex pontoIterado;
28   pontoIterado = ponto;
29   for(iterador = numeroDeIteracoes;iterador>=0;iterador--)
30   {
31    if (cabs(pontoIterado)>2 )
32    {
33     pixel.width = creal(ponto);
34     pixel.height = cimag(ponto);
35     pixel.cor = iterador;
36     return pixel;
37    }
38    pontoIterado = cpow(pontoIterado,2)+ponto;
39
40   }
41  }
42
43  void Mandelbrot (void *ptr)
44  {
45      thdata *data;
46      data = (thdata *) ptr;
47      int i;
48      struct Pixel *pixeis= malloc(data->size*sizeof(struct Pixel ));
49      for (i=0;i< data->size;i++)
50   {
51   pixeis[i]=IteracaoPontoMandelbrot(data->planoComplexoParticionado[i],50);
52      }
53      pthread_exit((void*)pixeis);
54  }
55
56  int main(int argc,char** argv)
57  {
58   double imageHeigth,imageWidth,numeroDeIteracoes;
59   int numThreads,tela;
60   if(argc == 3)
61   {
62    tela = atoi(argv[2]);
63    numThreads = atoi(argv[1])*atoi(argv[1]);
64
65   }
66   else
67   {
68    tela = 512;
69    numThreads = 2;
70   }
71
72   imageHeigth = 4;
73   imageWidth = 4;
74   numeroDeIteracoes = 50;
75
76   double passo = imageHeigth/tela;
77   double valorx,valory;
78   double teste ;
79   int x,y,i,j,z,indice;
80   struct timeb start, end;
81
82   z=0;
83   x = floor((imageHeigth/passo)+1);
84   y = floor((imageWidth/passo)+1);
85   pthread_t threads[numThreads];
86       thdata data[numThreads];
87
88   double complex* planoComplexo =malloc(x*y*sizeof(double complex ));
89   double complex* planoComplexoReconstruido[numThreads];
```

```
90   struct Pixel* pixeis;
91   struct Pixel* pixeisReconstruidos=malloc(x*y*sizeof(struct Pixel ));
92   long seconds;
93   int militm;
94
95   valory=-imageHeigth/2;
96
97   ftime(&start);
98
99   for (j=0;j<y;j++)
100  {
101
102   valorx=-imageWidth/2;
103   for (i=0;i<x;i++)
104   {
105    planoComplexo[z]=valorx+valory*I;
106    valorx=valorx+passo;
107    z++;
108   }
109   valory=valory+passo;
110  }
111
112
113  for (i=0;i<numThreads;i++)
114  {
115   data[i].thread_no = i;
116   data[i].planoComplexoParticionado = malloc(((z/numThreads)+1)*sizeof(double
         complex));
117   indice = 0;
118
119   for (j=i;j<z;j=j+numThreads)
120   {
121    data[i].planoComplexoParticionado[indice] =planoComplexo[j];
122    indice++;
123   }
124
125   data[i].size = indice;
126   pthread_create (&threads[i], NULL, (void *) &Mandelbrot, (void *) &data[i]);
127  }
128  indice = 0;
129
130  for (i=0;i<numThreads;i++)
131  {
132   pthread_join(threads[i], (void**)&pixeis);
133   for (j=0;j<data[i].size;j++)
134   {
135    pixeisReconstruidos[indice]=pixeis[j];
136    indice++;
137   }
138  }
139
140  ftime(&end);
141
142  seconds = (long)(end.time - start.time);
143  militm = end.millitm - start.millitm;
144  if (0 > militm) {militm += 1000;seconds--; }
145
146  int totalThreads =numThreads*numThreads;
147  int numPontos = tela*tela;
148  printf("%d,␣%d␣,Linha␣RoundRobin␣,␣␣%ld.%03d␣s\n",totalThreads,numPontos,
         seconds, militm);
149  return 0;
150 }
```

APÊNDICE B SIMULAÇÃO N-CORPOS GRAVITACIONAL

B.1 ncorpos.hs

```
1  import System.Environment
2  import StateUtil
3  import Graphics.Rendering.OpenGL
4  import Data.IORef
5  import Graphics.UI.GLUT
6  import Graphics.Rendering.OpenGL.GLU
7  import Control.Parallel.Strategies
8  import Data.List
9  import Data.List.Split
10 import Foreign.C.Types
11 import System.Random
12 import Data.Time.Clock
13 import Control.DeepSeq
14
15 g = 6.674287*10**(-11)
16 instance NFData CDouble
17
18 forcaGravitacional m1 m2 r = if r == 0  then  0
19         else if r<0 then -g*m1*m2/r**2
20           else g*m1*m2/r**2
21
22 velocidadefinal velocidadeinicial dtempo forca massa =velocidadeinicial+dtempo*
       forca/massa
23
24 raiovetorial :: (X,Y)->GLdouble
25 raiovetorial v1  = sqrt (((fst v1 )**2)+(((snd v1)**2)))
26
27 angulovetorial :: (X,Y)->GLdouble
28 angulovetorial v1 =  atan((snd v1)/(fst v1 ))
29
30 subtracaovetorial :: (X,Y)->(X,Y)->(X,Y)
31 subtracaovetorial v1 v2 = (fst v1 - fst v2,snd v1 -snd v2)
32
33 type X  = GLdouble
34 type Y  = GLdouble
35 type Posicao = (X, Y)
36 type Velocidade = (X, Y)
37 type Massa = GLdouble
38 type Corpo  =  (Posicao,Massa,Velocidade)
39 type Corpo2 =  (Posicao,Massa,Velocidade,Forca)
40 type Forca  = (X,Y)
41 type Aceleracao = (X,Y)
42 type Tempo = GLdouble
43
44 forcaIndividual :: [Corpo] -> Corpo -> [Forca]
```

```
45  forcaIndividual f corpo =
46   map (\(posicao,massa,velocidade) -> do
47    let distancia = subtracaovetorial  posicao (fstTripla corpo)
48    let raio = raiovetorial distancia
49    let angulo = if (fst distancia) == 0 then pi/2
50       else abs (angulovetorial distancia)
51
52    let forca = forcaGravitacional massa (sndTripla corpo) raio
53    let s1 = if (fst distancia) < 0 then -1
54      else 1
55    let s2 = if (snd distancia) <0 then -1
56      else  1
57    (s1*forca*cos angulo,s2*forca*sin angulo)
58   ) f
59
60  forcaTotalUmCorpo :: [Forca] -> Forca
61  forcaTotalUmCorpo f  = foldl (\a (x,y)->(fst a +x, snd a+y)) (0,0) f
62
63  forcaTotalTodosCorpos :: [Corpo]->[Corpo]->[Forca]
64  forcaTotalTodosCorpos corpos corposIterados = do
65   let f1 = map(\c -> forcaIndividual corpos c) corposIterados
66   map (\forcas ->forcaTotalUmCorpo forcas   ) f1
67
68  numeroMovimentos :: Int->Int ->Tempo->[Forca]->[Corpo]->[Corpo]
69  numeroMovimentos _ 0 _ _ _ =[]
70  numeroMovimentos chunks n dtempo forca corpo = do
71   let particao = linhaNormal  corpo (chunks)
72   let forcaTotal = runEval( forcaTotalParalelo corpo particao)
73   let teste = movimento (60*60) forcaTotal  corpo
74   teste++numeroMovimentos chunks (n-1) dtempo forca (teste)
75
76  movimento :: Tempo->[Forca]->[Corpo]->[Corpo]
77  movimento _ [] [] = []
78  movimento dtempo (forca:forcaxs) (corposAntes:corposAntesxs) = do
79   let massa = sndTripla corposAntes
80   let newVelocidadex =( velocidadefinal (fst(thdTripla corposAntes)) dtempo (fst
         forca) massa)
81   let newVelocidadey = ( velocidadefinal (snd(thdTripla corposAntes)) dtempo (
        snd forca) massa)
82   let newVelocidade = (newVelocidadex,newVelocidadey)
83   let newPosicaox= fst(fstTripla(corposAntes))+ newVelocidadex*dtempo
84   let newPosicaoy =snd(fstTripla(corposAntes))+ newVelocidadey*dtempo
85   let newPosicao = (newPosicaox,newPosicaoy)
86   [(newPosicao,massa,newVelocidade)]++movimento dtempo  forcaxs  corposAntesxs
87
88  linhaNormal :: [Corpo]->Int-> [[Corpo]]
89  linhaNormal c tamanhoChunk= chunksOf tamanhoChunk c
90
91  forcaTotalParalelo :: [Corpo]->[[Corpo]]->Eval [ Forca]
92  forcaTotalParalelo  _ [] = return []
93  forcaTotalParalelo c0 (c:cs) = do
94   p1 <- rpar( force (forcaTotalTodosCorpos c0 c))
95   p2 <- forcaTotalParalelo c0 cs
96   rseq p1
97   return (p1++p2)
98
99  fstTripla (a,b,c) = a
100 sndTripla (a,b,c)  = b
101 thdTripla (a,b,c)  = c
102
103 randomBodies ::  Int -> IO [Corpo]
104 randomBodies len =
105     if len==0 then return []
```

```
106     else do
107         b <- do mass <- getStdRandom (randomR (1e15,8e30))
108             x    <- getStdRandom (randomR (-1.5*10^11,1.5*10^11))
109             y    <- getStdRandom (randomR (-1.5*10^11,1.5*10^11))
110             return ((x,y),(mass),(0,0))
111
112     l <- randomBodies (len-1)
113     return $ (b:l)
114
115 main :: IO ()
116 main = do
117   (_, args) <- getArgsAndInitialize
118   let (numeroCorpos,qtdMovimentos,cores) = processaArgs args
119   corpos <- randomBodies numeroCorpos
120
121   t0 <- getCurrentTime
122   let particao = linhaNormal  corpos (numeroCorpos `div` cores)
123   let forcaTotal = runEval( forcaTotalParalelo corpos particao)
124
125   t1 <- ( numeroMovimentos (numeroCorpos `div` cores) qtdMovimentos (60*60)
          forcaTotal corpos) `deepseq` getCurrentTime
126   putStrLn $ ( (show $ diffUTCTime t1 t0)++ ",␣" ++ (show $ (numeroCorpos))++ "
          ,␣" ++ (show $ (cores))++ ",␣" ++ (show $ (qtdMovimentos)))
127   where
128   processaArgs args = case args of
129     (a:b:c:_) -> (read a,read b,read c)
130     _           -> (100,50,2)
```

B.2 BarnesHut.hs

```
1  import System.Environment
2  import Data.IORef
3  import Graphics.UI.GLUT
4  import Graphics.Rendering.OpenGL.GLU
5  import Control.Parallel.Strategies
6  import Data.List
7  import Data.List.Split
8  import Foreign.C.Types
9  import System.Random
10 import Data.Time.Clock
11 import Control.DeepSeq
12 import Debug.Trace
13 import Control.Exception
14
15 type X  = GLdouble
16 type Y  = GLdouble
17 type Posicao = (X, Y)
18 type Velocidade = (X, Y)
19 type Massa = GLdouble
20 type Corpo  =  (Posicao,Massa,Velocidade)
21 type Corpo2 =  (Posicao,Massa,Velocidade,Forca)
22 type Forca  = (X,Y)
23 type Aceleracao = (X,Y)
24 type Tempo = GLdouble
25 type Quadrante = Int
26 type Coordenadas = ((X,X),(Y,Y),Quadrante)
27
28 rdeepPar :: NFData a => Strategy a
29 rdeepPar pontos = rpar(force pontos)
30
31 g = 6.674287*10**(-11)
32 instance NFData CDouble
33
```

```
34 forcaGravitacional m1 m2 r =
35  if r == 0  then  0
36  else if r<0 then -g*m1*m2/r**2
37       else g*m1*m2/r**2
38
39 velocidadefinal velocidadeinicial dtempo forca massa =velocidadeinicial+dtempo*
       forca/massa
40
41 raiovetorial :: (X,Y)->GLdouble
42 raiovetorial v1  = sqrt (((fst v1 )**2)+(((snd v1)**2)))
43
44 angulovetorial :: (X,Y)->GLdouble
45 angulovetorial v1 =  atan((snd v1)/(fst v1 ))
46
47 subtracaovetorial :: (X,Y)->(X,Y)->(X,Y)
48 subtracaovetorial v1 v2 = (fst v1 - fst v2,snd v1 -snd v2)
49
50 data QuadTree m coordenadas limites = Empty [Corpo]
51     | Nodo m coordenadas limites
52   (QuadTree  m coordenadas limites )
53   (QuadTree  m coordenadas limites )
54   (QuadTree  m coordenadas limites )
55   (QuadTree  m coordenadas limites )
56   deriving (Show, Eq)
57
58 instance (NFData m,NFData coordenadas,NFData limites) =>NFData (QuadTree m
       coordenadas limites) where
59  rnf (Empty z) =  ()
60  rnf (Nodo a1 a2 a3 a4 a5 a6 a7) = rnf a1 `seq ` rnf a2 `seq ` rnf a3 `seq `
       rnf a4 `seq ` rnf a5 `seq ` rnf a6 `seq ` rnf a7
61
62 populaArvore ::  Int-> Int->[Corpo]->Coordenadas->Eval (QuadTree Massa Posicao
       Coordenadas)
63 populaArvore _ _ [] _ =return $ Empty []
64 populaArvore _ 1 corpo c1  = do
65  let massaTotal =  foldl (\a (x,y,z)->(a +y)) (0) corpo
66  let centroMassa= fstTripla(head corpo)
67  return $ Nodo massaTotal centroMassa c1 (Empty []) (Empty []) (Empty []) (
       Empty [])
68 populaArvore nTotalCorpos _ corpo c1 = do
69     let xmin = fst(fstTripla c1)
70  let xmax = snd (fstTripla c1)
71  let ymin = fst (sndTripla c1)
72  let ymax = snd (sndTripla c1)
73  let coord1 = ((xmin,(xmin+xmax)/2),(((ymin+ymax)/2),ymax),1)
74  let coord2 = (((xmin+xmax)/2,xmax),((ymin+ymax)/2,ymax),2)
75  let coord3 = ((xmin,(xmin+xmax)/2),(ymin,(ymin+ymax)/2),3)
76  let coord4 = (((xmin+xmax)/2,xmax),(ymin,(ymin+ymax)/2),4)
77  let p1tupla = partition (\(x,y,z)-> fst x < snd( fstTripla coord1) && snd x>=
       fst (sndTripla coord1)) corpo
78
79  let p1 = fst p1tupla
80  let p2tupla = partition (\(x,y,z)-> fst x >= fst(fstTripla coord2) && snd x >=
        fst(sndTripla coord2)) (snd p1tupla)
81  let p2 = fst p2tupla
82  let p3tupla = partition (\(x,y,z)-> fst x< snd(fstTripla coord3) && snd x <
       snd(sndTripla coord3)) (snd p2tupla)
83  let p3 = fst p3tupla
84  let p4 = snd p3tupla
85  if (((xmax-xmin))<100 && ((ymax-ymin))<100 ) then do
86   let massaTotal =  foldl (\a (x,y,z)->(a +y)) (0) corpo
87   let centroMassaTotal= foldl (\a (x,y,z)->(fst a+(fst x)*y ,snd a+(snd x)*y ))
        (0,0) corpo
```

```
88    let centroMassa = (fst centroMassaTotal/massaTotal,snd centroMassaTotal/
          massaTotal)
89    return $ Nodo massaTotal centroMassa c1 (Empty p1) (Empty p2) (Empty p3) (
          Empty p4)
90   else do
91       a1' <-populaArvore nTotalCorpos (length p1) p1 coord1
92    a1 <- if length p1 > (nTotalCorpos `div` 2) then  rpar $  force (a1' ::
          QuadTree Massa Posicao Coordenadas)
93        else rseq $  a1'
94    a2' <-populaArvore nTotalCorpos (length p2) p2 coord2
95    a2 <- if length p2 > (nTotalCorpos `div` 2) then rpar $  force (a2' ::
          QuadTree Massa Posicao Coordenadas)
96       else rseq(a2')
97    a3'<-populaArvore nTotalCorpos (length p3) p3 coord3
98    a3 <- if length p3 > (nTotalCorpos `div` 2) then  rpar $  force (a3' ::
          QuadTree Massa Posicao Coordenadas)
99       else rseq(a3')
100   a4' <-populaArvore nTotalCorpos (length p4) p4 coord4
101   a4 <- if length p4 > (nTotalCorpos `div` 2) then  rpar $  force (a4' ::
          QuadTree Massa Posicao Coordenadas)
102      else rseq(a4')
103
104   let (Nodo massaTotal1 centroMassa1 a b c d e )=
105    if a1 == (Empty []) then (Nodo 0 (0,0) ((0,0), (0,0),0) (Empty []) (Empty
          []) (Empty []) (Empty []) )
106    else a1
107   let (Nodo massaTotal2 centroMassa2 a b c d e )=
108    if a2 == (Empty []) then (Nodo 0 (0,0) ((0,0), (0,0),0) (Empty []) (Empty
          []) (Empty []) (Empty []) )
109    else a2
110   let (Nodo massaTotal3 centroMassa3 a b c d e )=
111    if a3 == (Empty []) then (Nodo 0 (0,0) ((0,0), (0,0),0) (Empty []) (Empty
          []) (Empty []) (Empty []) )
112    else a3
113   let (Nodo massaTotal4 centroMassa4 a b c d e )=
114    if a4 == (Empty []) then (Nodo 0 (0,0) ((0,0), (0,0),0) (Empty []) (Empty
          []) (Empty []) (Empty []) )
115    else a4
116
117   let massaTotal = massaTotal1+massaTotal2+massaTotal3+massaTotal4
118   let centroMassaTotal = ((fst centroMassa1*massaTotal1)+(fst centroMassa2*
          massaTotal2)+(fst centroMassa3*massaTotal3)+(fst centroMassa4*massaTotal4
          ),(snd centroMassa1*massaTotal1)+(snd centroMassa2*massaTotal2)+(snd
          centroMassa3*massaTotal3)+(snd centroMassa4*massaTotal4))
119   let centroMassa = (fst centroMassaTotal/massaTotal,snd centroMassaTotal/
          massaTotal)
120   return $ Nodo massaTotal centroMassa c1 (  a1 ) ( a2) ( a3) (  a4)
121
122 forcaIndividual :: [Corpo] -> Corpo -> [Forca]
123 forcaIndividual f corpo =
124  map (\(posicao,massa,velocidade) -> do
125   let distancia = subtracaovetorial  posicao (fstTripla corpo)
126   let raio = raiovetorial distancia
127   let angulo = if (fst distancia) == 0 then pi/2
128       else abs (angulovetorial distancia)
129
130   let forca = forcaGravitacional massa (sndTripla corpo) raio
131   let s1 = if (fst distancia) < 0 then -1
132      else 1
133
134   let s2 = if (snd distancia) <0 then -1
135      else  1
136
```

```
137    (s1*forca*cos angulo,s2*forca*sin angulo)
138   ) f
139
140  forcaTotalUmCorpo :: [Forca] -> Forca
141  forcaTotalUmCorpo f  = foldl (\a (x,y)->(fst a +x, snd a+y)) (0,0) f
142
143  somaForcas::(QuadTree Massa Posicao Coordenadas)->Corpo ->Forca
144  somaForcas (Empty listacorpos) c = forcaTotalUmCorpo (forcaIndividual
         listacorpos c)
145
146  somaForcas (Nodo m coordenadas limites (Empty _) (Empty _) (Empty _) (Empty _))
         corpo =
147   if (pertence limites corpo)== True then (0,0)
148   else calculoForcas  m coordenadas corpo
149
150  somaForcas (Nodo m coordenadas limites a1 a2 a3 a4) corpo = do
151   if (pertence limites corpo)== True then
152    (fst (somaForcas a1 corpo) + fst (somaForcas a2 corpo) + fst (somaForcas a3
          corpo) + fst (somaForcas a4 corpo) ,  snd (somaForcas a1 corpo) + snd (
          somaForcas a2 corpo) + snd (somaForcas a3 corpo) + snd (somaForcas a4
          corpo))
153   else calculoForcas m coordenadas corpo
154
155  calculoForcas::Massa-> Posicao->Corpo->Forca
156  calculoForcas m coordenadas corpo = do
157   let distancia = subtracaovetorial  coordenadas (fstTripla corpo)
158   let raio = raiovetorial distancia
159   let angulo = if (fst distancia) == 0 then pi/2
160       else abs (angulovetorial distancia)
161
162   let forca = forcaGravitacional m (sndTripla corpo) raio
163   let s1 = if (fst distancia) < 0 then -1
164      else 1
165
166   let s2 = if (snd distancia) <0 then -1
167      else  1
168
169   (s1*forca*cos angulo,s2*forca*sin angulo)
170
171  pertence:: Coordenadas->Corpo->Bool
172  pertence limites corpo =
173   if thdTripla(limites) == 1 then
174    if fst(fstTripla corpo)>= fst(fstTripla limites) && fst(fstTripla corpo)< snd
          (fstTripla limites) && snd(fstTripla corpo)>= fst(sndTripla limites) &&
          snd(fstTripla corpo)<= snd(sndTripla limites) then
175     True
176    else False
177   else if thdTripla(limites) == 2 then
178    if fst(fstTripla corpo)>= fst(fstTripla limites) && fst(fstTripla corpo)<=
          snd(fstTripla limites) && snd(fstTripla corpo)>= fst(sndTripla limites)
          && snd(fstTripla corpo)<= snd(sndTripla limites) then
179     True
180    else False
181   else if thdTripla(limites) == 3 then
182    if fst(fstTripla corpo)>= fst(fstTripla limites) && fst(fstTripla corpo)< snd
          (fstTripla limites) && snd(fstTripla corpo)>= fst(sndTripla limites) &&
          snd(fstTripla corpo)< snd(sndTripla limites) then
183     True
184    else False
185   else
186    if fst(fstTripla corpo)>= fst(fstTripla limites)  && fst (fstTripla corpo)<=
          snd(fstTripla limites) && snd(fstTripla corpo)>= fst(sndTripla limites)
          && snd(fstTripla corpo)< snd(sndTripla limites) then
```

```
187     True
188    else False
189
190  eixoX:: [Corpo] ->[X]
191  eixoX [] = []
192  eixoX (c:cs) = [fst(fstTripla c)]++eixoX cs
193
194  eixoY:: [Corpo] ->[Y]
195  eixoY [] = []
196  eixoY (c:cs) = [snd(fstTripla c)]++eixoY cs
197
198  numeroMovimentos :: Int->Int ->Tempo->[Forca]->[Corpo]->[Corpo]
199  numeroMovimentos _ 0 _ _ _ =[]
200  numeroMovimentos chunks n dtempo forca corpo = do
201   let particao = linhaNormal  corpo chunks
202   let coord0 =   eixoX corpo
203   let coord1 =   eixoY corpo
204   let coord = ((-1.5*10^11,1.5*10^11),(-1.5*10^11,1.5*10^11),2)
205   let arvore = force $ runEval $populaArvore (length corpo) (length corpo) corpo
          coord
206   let a2 = force (arvore)
207   let forcaTotal = runEval( forcaTotalParalelo (force $ runEval $populaArvore (
          length corpo) (length corpo) corpo  coord) particao)
208   let m1 = runEval $  movimentoParalelo (60*60) (forcaNormal forcaTotal chunks)
          particao
209   m1++numeroMovimentos chunks (n-1) dtempo forca (m1)
210
211  movimentoParalelo :: Tempo->[[Forca]]->[[Corpo]]->Eval [Corpo]
212  movimentoParalelo _ [] _ = return []
213  movimentoParalelo dtempo (forca:forcaxs) (corposAntes:corposAntesxs) = do
214   p1 <- rpar(force $ movimento dtempo forca corposAntes)
215   p2 <- movimentoParalelo dtempo forcaxs corposAntesxs
216   rdeepseq p1
217   return (p1++p2)
218
219  movimento :: Tempo->[Forca]->[Corpo]->[Corpo]
220  movimento _ [] _ = []
221  movimento _ _ [] = []
222  movimento dtempo (forca:forcaxs) (corposAntes:corposAntesxs) = do
223   let massa = sndTripla corposAntes
224   let newVelocidadex =( velocidadefinal (fst(thdTripla corposAntes)) dtempo (fst
          forca) massa)
225   let newVelocidadey = ( velocidadefinal (snd(thdTripla corposAntes)) dtempo (
          snd forca) massa)
226   let newVelocidade = (newVelocidadex,newVelocidadey)
227   let newPosicaox= fst(fstTripla(corposAntes))+ newVelocidadex*dtempo
228   let newPosicaoy =snd(fstTripla(corposAntes))+ newVelocidadey*dtempo
229   let newPosicao = (newPosicaox,newPosicaoy)
230   [(newPosicao,massa,newVelocidade)]++movimento dtempo  forcaxs  corposAntesxs
231
232  forcaTotalTodosCorpos :: [Corpo]->(QuadTree Massa Posicao Coordenadas)->[Forca]
233  forcaTotalTodosCorpos corpos arvore = map (\c ->somaForcas arvore c) corpos
234
235  linhaNormal :: [Corpo]->Int-> [[Corpo]]
236  linhaNormal c tamanhoChunk= chunksOf tamanhoChunk c
237
238  forcaNormal :: [Forca]->Int-> [[Forca]]
239  forcaNormal c tamanhoChunk= chunksOf tamanhoChunk c
240
241  forcaTotalParalelo :: (QuadTree Massa Posicao Coordenadas)->[[Corpo]]->Eval [
       Forca]
242  forcaTotalParalelo  _ [] = return []
243  forcaTotalParalelo arvore (c:cs) = do
```

```
244   p1 <- rpar(force $ forcaTotalTodosCorpos c arvore)
245   p2 <- forcaTotalParalelo arvore cs
246   rseq p1
247   return (p1++p2)
248
249 fstTripla (a,b,c) = a
250 sndTripla (a,b,c)  = b
251 thdTripla (a,b,c)  = c
252
253 randomBodies ::  Int -> IO [Corpo]
254 randomBodies len =
255     if len==0   then return []
256     else do
257    b <- do mass <- getStdRandom (randomR (1e15,8e30))
258              x    <- getStdRandom (randomR (-1.5*10^11,1.5*10^11))
259              y    <- getStdRandom (randomR (-1.5*10^11,1.5*10^11))
260              return ((x,y),(mass),(0,0))
261
262            l <- randomBodies (len-1)
263            return $ (b:l)
264
265 main :: IO ()
266 main  =
267     do
268   args <- getArgs
269     let (numeroCorpos,qtdMovimentos,cores) = processaArgs args
270   let coord = ((-1.5*10^11,1.5*10^11),(-1.5*10^11,1.5*10^11),2)
271   let tempo = 60*60
272   t0 <- getCurrentTime
273   corpos <- randomBodies numeroCorpos
274   t2 <- getCurrentTime
275   let particao = linhaNormal  corpos (numeroCorpos `div` cores)
276   let chunks = numeroCorpos `div` cores
277   t1 <- ( numeroMovimentos chunks qtdMovimentos (tempo) [] corpos) `deepseq`
          getCurrentTime
278
279   putStrLn $ ((show $ diffUTCTime t1 t0) ++ ",␣" ++ (show $ (numeroCorpos))++ ",
          ␣" ++ (show $ (cores))++ ",␣" ++ (show $ (qtdMovimentos)))
280   where
281     processaArgs args = case args of
282       (a:b:c:_) -> (read a,read b,read c)
283       _           -> (100,50,2)
```

APÊNDICE C ARTIGO DA PRIMEIRA ETAPA DO TRABALHO DE GRADUAÇÃO

Abaixo o artigo entregue na primeira etapa de realização do trabalho de graduação de engenharia de computação, que teve como um de seus objetivos descrever o planejamento deste trabalho.

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

Abstract. *The development of parallel algorithms is a complex and error-prone task, requiring a considerable effort from the point of view of the developers. The aim of this work is to evaluate if some of the difficulties regarding the development of parallel algoritms can be eased by using the purely functional programming language Haskell. For such, serial and parallel versions of the same algorithm in Haskell are compared with respect to code readability and performance. Additionally, implementations of the same algorithm in Haskell and in an imperative programming language are compared. Finally, some preliminary results are presented.*

Resumo. *O desenvolvimento de algoritmos paralelos é uma tarefa complexa e requer grande esforço por parte dos desenvolvedores. Esse trabalho avalia se algumas das dificuldades presentes no desenvolvimento de algoritmos paralelos são reduzidas utilizando-se a linguagem de programação funcional pura Haskell. Para tal, propõe-se comparar implementações sequenciais e paralelas de algoritmos em Haskell considerando legibilidade e performance. Adicionalmente, propõe-se comparar implementações de um mesmo algoritmo em Haskell e em uma linguagem imperativa. Por último, alguns resultados preliminares são apresentados.*

1. Introdução

Embora o conceito de paralelismo em computação seja antigo, observa-se mais recentemente o início de um período de transição da computação sequencial para computação paralela: processadores multi-cores já são uma realidade no mercado. A tendência mostra que o paralelismo é o futuro da computação [Asanovic et al. 2009]. Apesar de se conhecer as vantagens da computação paralela a bastante tempo, como executar mais de uma instrução no mesmo ciclo de *clock*, o principal fator dessa mudança de paradigma é que se está chegando ao limite da computação sequencial em questão de desempenho. A lei de Moore [Brock and Moore 2006] descreve a duplicação do número de transistores em circuitos integrados a cada dois anos. Alguns fatores como memória e potência fazem com que a lei de Moore não possa mais ser satisfeita pela computação sequencial. O maior desafio e motivo pelo qual a computação paralela não é dominante atualmente é a sua dificuldade. Questões como o não determinismo das aplicações tornam o desenvolvimento de programas de computadores (*softwares*) uma tarefa bastante complexa.

Linguagens imperativas foram preferencialmente utilizadas ao longo dos anos possivelmente por terem mais semelhanças com as arquiteturas de computadores baseadas no modelo de John von Neumann [von Neumann 1981]. Porém, nesse novo contexto de utilização de computações paralelas, as linguagens funcionais apresentam algumas

características mais convenientes em relação as linguagens imperativas, tal como a transparência referencial. Percebe-se que as linguagens funcionais ganham bastante espaço atualmente e muitas linguagens imperativas incorporam aspectos das linguagens funcionais em suas implementações.

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. Fatores como desempenho e implementação são verificados, e em alguns casos comparados com uma aplicação semelhante desenvolvida em uma linguagem imperativa.

A estrutura do artigo é apresentada a seguir. A seção 2 apresenta um pequeno resumo sobre as linguagens funcionais e especificamente a linguagem Haskell. A seção 3 apresenta alguns conceitos sobre paralelismo e seu uso em Haskell. A seção 4 apresenta os algoritmos que serão analisados no projeto, o método que será utilizado para análise e o cronograma das atividades. A seção 5 apresenta os resultados preliminares obtidos até esse momento. E, por fim, a seção 6 traz uma conclusão sobre a experiência da utilização do Haskell para obtenção desses resultados.

2. Linguagens Funcionais

Linguagens Funcionais têm as suas origens nas funções matemáticas [Hughes 1989]. Características dessas funções juntamente com o cálculo Lambda [Jones 1987] foram incorporadas para criação dessas linguagens, tornando elas independentes da arquitetura da máquina e baseadas em um modelo bastante sólido.

Pode-se classificar as linguagens funcionais de duas formas: as linguagens funcionais puras e as linguagens funcionais impuras. As linguagens funcionais puras se caracterizam por utilizarem funções que não produzem efeitos colaterais, funções puras, e, portanto, não carregam nenhum estado internamente. Em uma função pura ao se utilizar os mesmos parâmetros de entrada sempre se obtêm o mesmo resultado da função na saída. Isso é uma vantagem, pois, devido a essa ausência de efeitos colaterais as funções não produzem nenhuma dependência implícita entre elas, sendo possível executá-las em qualquer ordem sem alterar seu resultado. Outra vantagem das funções puras é a transparência referencial. Transparência referencial é a propriedade de substituir uma expressão pelo seu resultado contanto que a expressão já tenha sido avaliada anteriormente, isso é possível devido ao fato da expressão sempre produzir o mesmo resultado. Em linguagens funcionais puras não existem variáveis, o que caracteriza os dados como sendo imutáveis. Uma consequência dessa imutabilidade é a inexistência de laços, a forma de lidar com a iteração é através da recursão. Alguns exemplos de linguagens funcionais puras são Haskell [Team 2014a] e Miranda [Turner 1987].

Linguagens Funcionais impuras se diferenciam por usarem, além das funções puras, funções que produzem efeitos colaterais [Wadler 1992]. Características como transparência referencial e independência da ordem de avaliação não se aplicam a essas funções. Porém, permite-se o uso de atribuições e variáveis. A vantagem de se utilizar linguagens funcionais impuras está relacionada à eficiência: durante muito tempo as linguagens funcionais impuras foram consideradas como tendo um desempenho superior. Outro fator que pode ter alguma influência é aproximar um pouco a linguagem funcional da imperativa. Exemplos de linguagens funcionais impuras são ML [Milner et al. 1997]

e Scheme [Sussman and Steele 1998].

Expressões e funções são os principais componentes de uma linguagem funcional. A avaliação dos parâmetros na chamada de uma função pode ser estrita ou não-estrita. Em uma avaliação estrita os parâmetros são avaliados apenas quando o valor do resultado da função é realmente utilizado na aplicação, podendo eventualmente nunca serem avaliados. Em uma avaliação não-estrita os parâmetros da função são sempre avaliados antes da chamada da função.

Uma última característica a ser ressaltada em linguagens funcionais é possibilidade de se definir funções de alta ordem. Essas funções recebem como argumentos outras funções deixando dessa forma o código mais simples e modular.

2.1. 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 é conhecida por ser uma linguagem preguiçosa (*lazy*) [HaskellWiki 2014]. Avaliação preguiçosa é uma técnica de avaliação não estrita, que permite ao Haskell, por exemplo, postergar a avaliação de uma expressão para ser mais eficiente ou construir estruturas como listas infinitas. Embora essa técnica traga algumas vantagens, como não avaliar valores que não são necessários, ela requer alguns cuidados. A linguagem pode acabar alocando mais memória do que o necessário na execução de um programa. A linguagem utiliza o coletor de lixo para gerenciamento automático da memória. O coletor de lixo tem como principal função liberar valores temporários da memória. Em Haskell os dados são imutáveis, como consequência muitos valores temporários são gerados e armazenados na memória, por esse motivo o coletor de lixo tem um papel fundamental na linguagem. Haskell permite a avaliação estrita também embora esta estratégia não seja a padrão da linguagem, é possível graças a algumas bibliotecas. O GHC também oferece uma técnica de profiling [Team 2014b] que tem como propósito oferecer uma análise de uso da memória.

Haskell é uma linguagem fortemente tipada [Lipovaca 2011]: todas as funções e expressões têm um tipo. Apesar de ser fortemente tipada, o compilador pode inferir em algumas situações o tipo se esse não for declarado. Todos os tipos são estáticos, ou seja definido durante a compilação. Essas características tornam os programas menos suscetíveis a erros. Haskell oferece os tipos de dados básicos que a maiorias das outras linguagens oferecem como inteiros, booleanos, caracteres, entre outros, e oferece tipos compostos sendo os principais as listas e tuplas.

Pode-se destacar na linguagem o casamento de padrões (*pattern matching*) que é uma construção sintática que auxilia nas construções de funções e recursões. O casamento de padrões torna as funções mais simples e flexíveis, evitando o uso demasiado do construtor *if then else*. Sobre as funções, Haskell permite o uso de funções anônimas utilizando o operador "\". Funções anônimas tem o propósito de deixar o código mais limpo.

Funções com mais de um parâmetro em Haskell passam por um processo de currificação[1] (*currying*) [Lipovaca 2011]. Esse processo consiste em aplicar a função a um argumento e obter seu resultado, criando uma nova função que é aplicada ao próximo parâmetro. Isso permite a construção de funções parcialmente aplicadas. A forma como Haskell lida com funções com mais de um parâmetro mostra outra característica das funções em Haskell: elas aceitam outras funções como parâmetros, gerando assim funções de alta ordem. Uma função de alta ordem bastante utilizada em Haskell é a função `map`. A declaração do tipo da função `map` é

```
map :: (a -> b) -> [a] -> [b]
```

[Lipovaca 2011]. Pela declaração da função observa-se como primeiro argumento uma função que recebe um tipo `a` e retorna um tipo `b`. A função `map` aplica uma função aos elementos de uma lista, que é o seu segundo parâmetro, retornando uma nova lista com o resultado da função aplicada. Pelo fato de Haskell conseguir inferir os tipos de dados, os parâmetros das funções em Haskell não necessitam ter um tipo definido, como no caso da função `map`. Pode-se então criar funções genéricas que podem ser reutilizadas. Essa forma de polimorfismo é outro ponto forte da linguagem.

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]. Mônadas são as construções que Haskell utiliza para lidar com funções que tem efeito colateral [Wadler 1995]. As mônadas em Haskell obedecem algumas regras e simulam a parte impura isolando-as do resto da linguagem. As operações dentro da mônada obedecem uma ordem de execução. O que ocorre na prática é que toda parte impura ocorre dentro da estrutura da mônada e depois seu resultado é ligado a parte pura do código. Pode-se então utilizar esse resultado sem qualquer tipo de efeito colateral, mantendo o código puro.

3. Paralelismo

Paralelismo na computação significa executar mais de uma tarefa ao mesmo tempo para resolução de um problema, com o objetivo de se obter um melhor desempenho ou melhor utilizar os recursos. Existem fatores que não permitem que ao se duplicar o número de processadores, consiga-se dividir o tempo de processamento na mesma proporção. Ao se realizar uma computação em paralelo existem questões como o gerenciamento das tarefas, a dependência entre elas, e outros fatores que acabam consumindo tempo de processamento. Algumas métricas como aceleração (*speedup*) e eficiência [Wilkinson and Allen 1999] são utilizadas para se analisar o ganho de desempenho que uma aplicação paralela tem em relação a uma computação sequencial. A Lei de Amdahl [Wilkinson and Allen 1999] fala sobre o ganho de desempenho máximo que é possível de uma aplicação paralela obter em relação a uma computação sequencial.

Aumento de desempenho é um dos principais objetivos da computação paralela. Nesse trabalho o paralelismo será explorado com foco em desempenho e utilizando uma arquitetura de memória compartilhada, que utiliza um espaço de endereçamento único para todos os processadores. Como o espaço de endereçamento é único, é necessário em algumas situações controlar o acesso a memória, em situações conhecidas como

[1] Possível tradução para o termo currying, visto que não foi encontrado na literatura uma tradução mais clara.

condições de corrida, para evitar inconsistência nos dados no caso de escritas/leituras concorrentes.

Computação distribuída é um conceito muito semelhante ao paralelismo, entretanto seus objetivos são diferentes. Em uma computação distribuída as tarefas são executadas em diferentes máquinas para resolver um problema com o objetivo de compartilharem recursos [Coulouris and Dollimore 1988]. Em problemas impossíveis de serem resolvidos em apenas um computador devido a sua limitação de memória [Barney and Laboratory 2014a] a computação distribuída é uma solução. Concorrência é outro conceito que pode ser usado em conjunto com paralelismo. Concorrência é uma técnica que executa diferentes tarefas para resolução de um problema com o objetivo de estruturar um programa, tornando ele modular [Marlow 2013].

Aumentar o número de instruções executadas por segundo, utilizando uma computação sequencial, implica em aumentar a frequência do processador. Com o aumento da frequência maior é o consumo de potência no circuito [Frank 2002], e isso acaba levando a um aquecimento maior do circuito devido a potência dissipada em forma de calor. Atualmente está se chegando ao limite de temperatura em um circuito, e consequentemente aumentar a frequência não é mais uma alternativa para ganho de desempenho. O uso de paralelismo acaba sendo uma das soluções nesse cenário. Em uma aplicação paralela é possível se executar mais de uma instrução no mesmo ciclo de *clock*, acarretando em um melhor desempenho utilizando a mesma frequência.

O desenvolvimento de compiladores que automatizem o paralelismo em *softwares* ainda é uma realidade distante, cabendo ao desenvolvedor esta tarefa. Os sistemas operacionais têm como um de seus principais objetivos gerenciar os recursos de um sistema [Tanenbaum 2007]. Como forma de realizar esse gerenciamento são utilizadas abstrações como *threads* e processos, que variam em cada sistema operacional. *Threads* e processos [Tanenbaum 2007] são algumas das formas que as aplicações tem de dividir o programa em subtarefas e executá-las em paralelo em múltiplos processadores. Uma das diferenças das *threads* em relação aos processos é o fato de threads compartilharem recursos como a memória. É natural se pensar em threads no uso de computação paralela de memória compartilhada, enquanto processos podem ser usados numa computação paralela de memória distribuída. Um padrão para programação com *threads* conhecido como *POSIX threads(pthreads)* [Barney and Laboratory 2014b] foi desenvolvido com o objetivo de facilitar a portabilidade dos softwares em sistemas Unix. Esse padrão é utilizado nesse trabalho nos experimentos utilizando linguagem imperativa. Além das *pthreads* outros padrões como OpenMP [Quinn 2003] e MPI [Quinn 2003] são bastante utilizados no desenvolvimento de programas paralelos utilizando linguagens de programação imperativas.

O uso de *pthreads* no desenvolvimento de *softwares* acaba muitas vezes sendo uma tarefa bastante árdua e um dos principais motivos esta numa característica intrínseca das linguagens imperativas: as variáveis. O uso de variáveis em programas paralelos pode criar as condições de corrida [Netzer and Miller 1992]. Essas condições são erros que ocorrem quando não existe sincronização das subtarefas no acesso à memória. Quando duas ou mais subtarefas modificam a memória em diferentes ordens sem a devida sincronização o programa pode ter um comportamento não determinístico. As condições de corridas levam a erros que podem ser extremamente difíceis de serem detectados, pois

podem ocorrer raramente e não serem detectados pelos desenvolvedores.

Para se obter o melhor desempenho em uma aplicação paralela o ideal é que se divida a aplicação em partes que levem o mesmo tempo para serem processadas. Porém, esse particionamento nem sempre é simples de ser alcançado. Fatores como a velocidade de cada processador podem tornar o particionamento ideal impossível de ser conhecido antes da execução da aplicação. Logo esse particionamento pode ser estático ou dinâmico [Wilkinson and Allen 1999]. Outra questão importante é a granularidade das tarefas: todas as tarefas têm custos de processamento associados a sua criação e gerenciamento. Tarefas com uma granularidade muito pequena acabam não compensando seu custo, e podem levar a programas paralelos com desempenho até inferior a programas sequenciais.

3.1. Paralelismo em Haskell

Em princípio todas as funções em Haskell poderiam ser executadas em paralelo como consequência da execução fora de ordem. No entanto, nem sempre executar todas funções em paralelo produz um melhor desempenho, e nem sempre é possível. A razão disso é a granularidade das tarefas e a dependência dos dados [Jones and Singh 2008].

Hakell oferece o módulo `Control.Parallel.Strategies` [Haskell 2014], que tem o propósito de 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. As principais estratégias oferecidas pelo módulo são: `rpar` e `rseq`.

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*. Nem sempre um `spark` cria uma *thread*. 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 for 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`.

A linguagem Haskell permite que o programador desenvolva suas próprias estratégias. Essa é uma flexibilidade muito boa da linguagem, pois permite que parte paralela seja totalmente isolada da parte sequencial do programa, podendo ser reutilizável. As

estratégias podem ser parametrizáveis, isto é, receber outras estratégias como parâmetros. Assim é possível combinar as estratégias oferecidas pela biblioteca com as desenvolvidas pelo programador.

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`. Haskell oferece outros módulos para uso de paralelismo que podem ser mais adequados dependendo do problema a ser resolvido. Entre os módulos podem ser citados o `Control.Monad.Par` e o pacote `Repa` que utiliza paralelismo em vetores.

Para compilação de um programa paralelo no GHC deve-se utilizar como parâmetro a opção `-threaded`. Para sua execução utiliza-se o parâmetro `-Nx`, sendo x o número de processadores.

4. Metodologia

Como forma de atingir o objetivo do trabalho vamos incialmente escolher dois problemas. O próximo passo é implementar uma versão sequencial e paralela em Haskell para cada problema. Dependendo da complexidade de implementação também será implementada uma versão em Ansi C. Por fim, serão realizados experimentos e analisados seus resultados.

A escolha do primeiro problema terá como características a simplicidade de paralelização, e a não dependência dos dados. Essa escolha têm como meta verificar a complexidade de implementação dos mecanismos básicos do módulo `Control.Parallel.Strategies`, e o ganho de desempenho da aplicação paralela.

O segundo problema terá como características uma complexidade maior que o primeiro problema, tendo dependências de dados no algoritmo. O objetivo é analisar aspectos da linguagem que não forem vistos no primeiro problema. Será verificado a implementação dessas dependências na linguagem e a comunicação entre as subtarefas utilizadas no código paralelo.

A implementação de ambos os problemas terá uma versão sequencial e paralela em Haskell. Para a versão paralela, o primeiro problema utilizará o módulo `Control.Parallel.Strategies`. Para a resolução do segundo problema, serão analisados os módulos `Control.Parallel.Strategies` e `Control.Monad.Par`, sendo, após, escolhido o módulo que for mais adequado para solução do problema. Será implementada um versão sequencial e paralela em Ansi C para o primeiro problema devido a sua simplicidade. Para o segundo problema será analisada a sua complexidade de implementação em Ansi C e, se for aceitável, será implementado.

Para o algoritmo sequencial em Haskell serão realizados experimentos utilizando como fator a quantidade de dados. Para o algoritmo paralelo em Haskell se realizarão experimentos variando-se a quantidade de dados, o número de núcleos utilizados e a forma de particionamento dos dados. Os algoritmos em Ansi C, quando implementados, terão um número fixo de dados para a aplicação sequencial e um número fixo de dados, núcleos e uma forma de particionamento para aplicação paralela.

A análise será feita executando-se diversas 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, como por exemplo a execução de tarefas do sistema operacional. Serão comparados os tempos de execuções dos experimentos do algoritmo sequencial com os do algoritmo paralelo em Haskell. A partir desses tempos será determinado o ganho de desempenho do algoritmo paralelo, e quando necessário serão gerados gráficos. Também será feita uma observação sobre a distribuição das cargas de trabalhos nas diferentes formas de particionamento do algoritmo paralelo. Quando forem implementados algoritmos em Ansi C, será feita uma comparação dos tempos de execução em Haskell com os tempos de execução em Ansi C. A idéia é comparar a linguagem funcional com a imperativa. Outro fator que será verificado é a complexidade de implementação.

A Tabela 1 mostra o cronograma de realização das atividades. O símbolo ✓representa a conclusão de uma atividade. Os sombreamentos indicam o tempo em que as atividades foram realizadas ou o tempo estimado de realização das atividades. As atividades são:

1. Definição do trabalho
2. Estudo da linguagem Haskell
3. Definição do cálculo do fractal de Mandelbrot como primeiro algoritmo
4. Implementação do algoritmo de Mandelbrot sequencial em Haskell
5. Estudo do módulo `Control.Parallel.Strategies`
6. Implementação do algoritmo de Mandelbrot paralelo em Haskell
7. Implementação do algoritmo de Mandelbrot sequencial em Ansi C
8. Implementação do algoritmo de Mandelbrot paralelo em Ansi C
9. Execução parcial dos experimentos
10. Escrita do primeiro trabalho
11. Definição do segundo experimento a ser analisado
12. Estudo do módulo `Control.Monad.Par`
13. Implementação sequencial do segundo problema na linguagem Haskell
14. Implementação paralela do segundo problema na linguagem Haskell
15. Análise da viabilidade da implementação do segundo problema em Ansi C
16. Possível implementação do segundo problema na linguagem Ansi C
17. Execução dos experimentos
18. Escrita do trabalho final

5. Resultados Preliminares

O ambiente de execução dos experimentos é máquina beagle do Instituto de Informática da UFRGS. A máquina beagle têm dois processadores com oito núcleos em cada um deles. Em cada núcleo pode-se executar duas *threads* ao mesmo tempo. A memória total da máquina é de 32 GigaBytes e a frequência é de 2000 MHz. O sistema operacional instalado é o Linux versão 3.8.13.13+. A versão do ghc é a 7.4.1 e a do gcc é 4.6.3.

O primeiro algoritmo utilizado é o 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

Etapa	Meses									
	Fev 2014	Mar 2014	Abr 2014	Mai 2014	Jun 2014	Jul 2014	Ago 2014	Set 2014	Out 2014	Nov 2014
1	✓									
2		✓								
3		✓								
4			✓							
5			✓	✓						
6			✓	✓						
7				✓						
8				✓						
9				✓	✓					
10				✓	✓					
11										
12										
13										
14										
15										
16										
17										
18										

Tabela 1. Cronograma das atividades previstas

depende do cálculo dos outros pontos. O conjunto de Mandelbrot é um conjunto infinito de pontos limitados em uma área finita. Essa característica é impossível de ser reproduzida computacionalmente em máquinas que tem memórias finitas. Porém, é possível discretizar esses pontos para análise, e a quantidade de pontos discretizados pode variar. Isso é bastante favorável para os experimentos que serão realizados, e para alcançar o objetivo desse trabalho.

O conjunto de Mandelbrot é definido por:

$$z_0 = 0, \quad z_n + 1 = z_n^2 + c$$

para todos os pontos c no qual o resultado da recursão não tende ao infinito. O ponto c é um número complexo. Já foi entretanto provado que se $|z_n| > 2$ o resultado da recursão tende ao infinito, esse teste é realizado nos algoritmos deste trabalho para determinar se o ponto pertence ao conjunto Mandelbrot.

5.1. Implementações

As implementações do algoritmos utilizados nos resultados preliminares encontram-se no endereço *https://github.com/vfpereira/Mandelbrot.git*, com as instruções de compilação e execução no arquivo readme.

5.1.1. Haskell Sequencial

O algoritmo sequencial do cálculo do fractal de Mandelbrot (Mandelbrot_sequencial.hs) baseia-se na construção dos pontos no plano cartesiano com a imagem centralizada nos

pontos (0,0). Os pontos testados estão dentro dos limites (-2,-2) e (2,2), pelo fato de todos os pontos que convergem do conjunto Mandelbrot estarem nesses limites. Em seguida utiliza-se uma função recursiva para testar se um número complexo pertence ao conjunto Mandelbot, com o número máximo de recursões igual a 50, visto que não é possível realizar um número infinito de recursões. Através do valor absoluto testa-se se o ponto pertence ao conjunto Mandelbrot, em caso positivo a função retorna a posição do ponto no plano cartesiano e sua cor com o valor 0, caso contrário é retornado a posição com o valor da cor sendo igual ao da recursão onde o teste foi negativo. Concluindo, se utiliza uma função `map` para testar todos os pontos da lista. A cor é utilizada para se desenhar o fractal, uma versão que desenha o fractal de Mandelbrot também é encontrada no repositório (Mandelbrot_OpenGL.hs), sendo reproduzida na Figura 1 a imagem gerada.

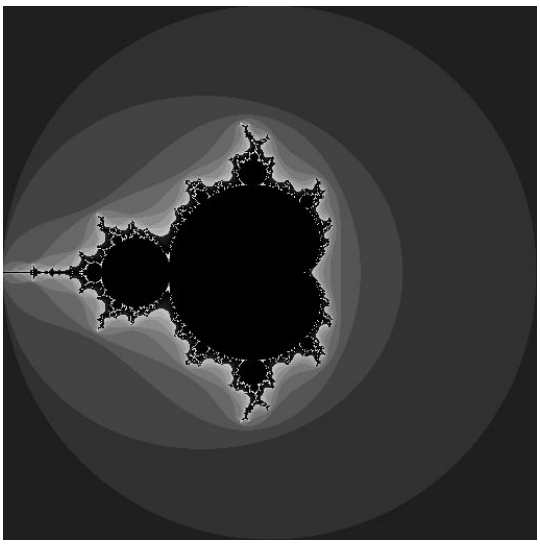

Figura 1. Imagem do conjunto Mandelbrot gerada em Haskell utilizando OpenGL

5.1.2. Haskell Paralelo

O algoritmo paralelo em Haskell (Mandelbrot_paralelo.hs) cria a lista de pontos complexos da mesma forma que o algoritmo sequencial. Porém, após são particionados os pontos de acordo com os parâmetros passados na execução do programa, criando uma lista de listas desses pontos que serão as subtarefas do programa. As formas de particionamento são: dividindo-se os pontos em linhas, dividindo-se os pontos em linhas e misturando os pontos com uma distância igual ao número de tarefas, dividindo-se os pontos em colunas, e dividindo-se os pontos em colunas e misturando os pontos com uma distância igual ao número de tarefas. O número de tarefas criadas é igual ao primeiro parâmetro de execução que é o numero de `sparks` que se deseja criar. A mesma função do algoritmo sequencial foi utilizada para testar se o ponto pertence ao conjunto Mandelbrot. Contudo, a função `map` foi substituída por uma função recursiva que cria um *spark* para cada lista particionada ser executada em paralelo e assim avaliar todos os pontos. Pode-se notar na implementação do código poucas alterações em relação ao sequencial, demonstrando

uma simplicidade na implementação, o que tem uma certa complexidade é encontrar a melhor forma de particionar os dados.

5.2. Análises

Na análise dos algoritmos a ferramenta R [for Statistics and of WU 2014] será utilizada para geração dos gráficos e estatísticas. Nesse primeiro trabalho serão realizados 50 experimentos para cada um dos dois algoritmos e gerados seus respectivos gráficos. Em ambos os algoritmos será considerado apenas o tempo da função que realiza o cálculo de todos os pontos do conjunto Mandelbrot. Futuramente serão analisado o tempo de construção estrutura também, por isso a execução dos algoritmos produzem dois tempos.

5.2.1. Haskell Sequencial x Haskell Paralelo

Para essa análise foram realizados 500 experimentos. Foi variado o número total de pontos que representam os números complexos utilizados para construção do conjunto Mandelbrot. No algoritmo sequencial foi utilizado 5 quantidades de pontos diferentes e no algoritmo paralelo também se utilizou 5 quantidades de pontos. Para cada um deles foram executados 50 experimentos para se obter o tempo médio e o desvio padrão. No algoritmo paralelo foi usado um número fixo, igual a 4, de `sparks` e uma forma de particionamento que foi a divisão em linhas dos pontos. As quantidades de pontos utilizados em ambos foram: 16641, 66049, 263169, 1050625 e 4198401. Os tempos médios obtidos dos experimentos com o algoritmo sequencial foram respectivamente: 0.182653s, 0.7282s, 2.895139s, 11.550549s e 46.176269s. O desvio padrão de cada um foi: 0.000920s, 0.004723s, 0.023441s, 0.105834s, 0.273253s. Os tempos médios do algoritmo paralelo foram: 0.098857s, 0.38797s, 1.53636s, 6.330761s e 24.856478s. O desvio padrão para cada um foi: 0.003046s, 0.01048s, 0.110253s, 0.19699s e 0.945715s. Percebe-se que a aplicação paralela reduziu o tempo do cálculo praticamente pela metade. Isso é bom indicativo, uma vez que o particionamento utilizado não é o ideal, nem todas as tarefas levam o mesmo tempo, deixando assim o processador ocioso por algum tempo. O desvio padrão dos experimentos foi pequeno quando comparados ao tempo médio da aplicação o que indica que os fatores externos não tiveram grande influência na execução. As figuras 2 e 3 mostram os gráficos dos tempos de execuções.

Por fim a aceleração de todos os pontos foram muito próximas o que indica que independente da quantidade de pontos se obtém praticamente o mesmo ganho da aplicação paralela. As acelerações calculadas em ordem de quantidade de pontos foram: 1.85, 1.88, 1.88, 1.82 e 1.85.

Percebe-se na implementação a compactação do código, através do uso de funções de alta ordem, funções anônimas e o casamento de padrões, tornando fácil sua compreensão. Para o próximo trabalho será apresentada uma análise mais detalhada variando-se outros fatores, e uma comparação com uma implementação sequencial na linguagem imperativa.

6. Conclusão

Desenvolver programas paralelos ainda é um grande desafio para os desenvolvedores. Existem diversos padrões e linguagens que oferecem recursos para para criação de

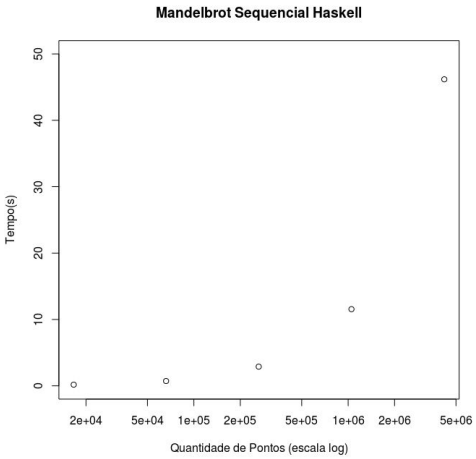

Figura 2. Gráfico do tempo médio de execuções do Mandelbrot sequencial em escala logaritmica.

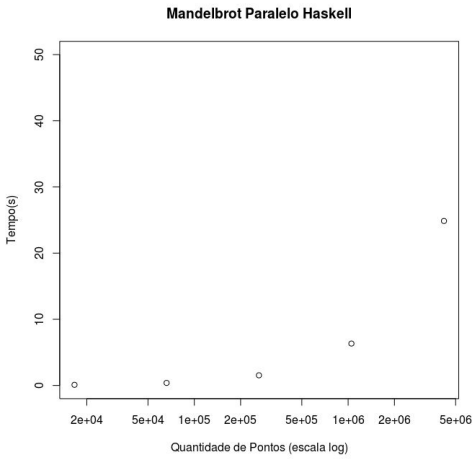

Figura 3. Gráfico do tempo médio de execuções do Mandelbrot Paralalo em escala logaritmica.

aplicações paralelas. Entretanto, não existe uma unanimidade na escolha da melhor forma de desenvolvimento. A linguagem Haskell surge como uma alternativa menos convencional para esse problema. Por ser uma linguagem funcional pura, ela pode ser melhor explorada nesse novo contexto, visto que presenta novas soluções para problemas que persistem até o momento em linguagens imperativas.

Pensar algoritmos de forma paralela não é natural para muitos desenvolvedores. Da mesma forma algoritmos baseados em linguagens funcionais encontram rejeição por alguns programadores. Contudo, problemas resolvidos através de linguagens imperativas de forma sequencial nem sempre produzem as melhores soluções.

Durante o desenvolvimento desse trabalho e dos algoritmos, foi possível perceber que a linguagem Haskell tem uma quantidade de material muito pequena quando comparado a linguagens mais tradicionais como Java e Ansi C. Esse trabalho busca contribuir investigando se a linguagem é adequada para o desenvolvimento de programas paralelos. Espera-se confirmar as vantagens da linguagem citadas nesse artigo através dos experimentos. Também será examinado se os desempenhos desses algoritmos são aceitáveis, bem como o custo/benefício de suas implementações.

Uma conclusão que foi possível chegar com o experimento feito até esse momento, é que Haskell produz um código bastante modular, oferecendo funções bastante práticas. Consegue-se resolver o problema do cálculo do conjunto de Mandelbrot com poucas linhas de código. Esse comportamento parece ser uma característica da linguagem o que torna os programas fáceis de serem compreendidos e de fácil manutenção. Um ponto negativo da linguagem é a dificuldade de depuração, com falta de ferramentas eficientes para essa tarefa no desenvolvimento dos programas.

Os experimentos parciais produziram um resultado encorajador até o momento, visto que foi possível reduzir o tempo de execução praticamente pela metade com a aplicação paralela. Espera-se resultados melhores no próximo trabalho, pois se utilizará um número maior de tarefas na resolução desse problema e outras formas de par-

ticionamento. Também será possível comparar com o tempo da linguagem Ansi C, que já é comprovadamente uma linguagem com bons desempenhos de algoritmos paralelos. Através dessa comparação poderá se analisar se os tempos das execuções em Haskell são aceitáveis. Um segundo problema será avaliado, assim como novas bibliotecas para paralelismo na linguagem. Para esse segundo algoritmo vai ser possível analisar também a comunicação entre as tarefas e seu impacto na linguagem.

Referências

Asanovic, K., Bodik, R., James Demmel, T. K., Keutzer, K., John Kubiatowicz, N. M., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM - A View of Parallel Computing*, 52:56–67.

Barney, B. and Laboratory, L. L. N. (2013 (acessado Mai 26, 2014)b). *POSIX Threads Programming*. `https://computing.llnl.gov/tutorials/pthreads/`.

Barney, B. and Laboratory, L. L. N. (2014 (acessado Mai 25, 2014)a). *Introduction to Parallel Computing*. `https://computing.llnl.gov/tutorials/parallel_comp/`.

Brock, D. and Moore, G. (2006). *Understanding Moore's Law: Four Decades of Innovation*. Chemical Heritage Foundation.

Coulouris, G. F. and Dollimore, J. (1988). *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

for Statistics, I. and of WU, M. (2010 (acessado Jun 11, 2014)). *The R Project for Statistical Computing*. `http://www.r-project.org/`.

Frank, D. J. (2002). Power-constrained cmos scaling limits. *IBM J. Res. Dev.*, 46(2-3):235–244.

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`.

Hughes, J. (1989). Why Functional Programming Matters. *Computer Journal*, 32(2):98–107.

Jones, P. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall International Ltd.

Jones, S. L. P. and Singh, S. (2008). A tutorial on parallel and concurrent programming in haskell. In *Advanced Functional Programming*, pages 267–305.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, first edition.

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/`.

Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

Sussman, G. J. and Steele, Jr., G. L. (1998). Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 11(4):405–439.

Tanenbaum, A. S. (2007). *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Team, H. (2010 (acessado Jun 6, 2014)a). *The Haskell Programming Language*. `http://www.haskell.org/`.

Team, T. G. (2014b). *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.2*.

Turner, D. (1987). An overview of miranda. *Bulletin of the EATCS*, 33:103–114.

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

Wadler, P. (1992). The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14. ACM.

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.

Wilkinson, B. and Allen, M. (1999). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc.