Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

USO DE PARALELISMO DE DADOS PARA MAIOR EFICIÊNCIA DE ALGORITMOS DE PROCESSAMENTO DE IMAGENS

Aluno: Pedro Ribeiro Mendes Júnior
Matrícula: 08.2.4114

Orientador: Lucília Camarão de Figueiredo

Ouro Preto
8 de dezembro de 2011

Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

USO DE PARALELISMO DE DADOS PARA MAIOR EFICIÊNCIA DE ALGORITMOS DE PROCESSAMENTO DE IMAGENS

Relatório de atividades desenvolvidas apresentado ao curso de Bacharelado em Ciência da Computação, Universidade Federal de Ouro Preto, como requisito parcial para a conclusão da disciplina Monografia I (BCC390).

Aluno: Pedro Ribeiro Mendes Júnior
Matricula: 08.2.4114

Orientador: Lucília Camarão de Figueiredo

Ouro Preto
8 de dezembro de 2011

Resumo

O projeto busca explorar oportunidades de uso de paralelismo de dados em algoritmos voltados para processamento de imagens, com o objetivo de obter algoritmos mais eficientes para essas aplicações. Para isso, propomos desenvolver algoritmos de processamento de imagem usando a linguagem funcional Haskell, em particular Data Parallel Haskell, uma extensão do compilador GHC e de suas bibliotecas, que oferece suporte para paralelismo aninhado (*nested data parallelism*) com foco na utilização de CPUs multicore. Como trabalho futuro, os dados de tempo de execução dos algoritmos desenvolvidos serão medidos e comparados com o de outras implementações existentes. No caso dessas comparações indicarem bons resultados, planeja-se, como trabalho futuro, a implementação de uma biblioteca de funções para processamento de imagens com enfoque em paralelismo, a qual deverá ser útil para implementação eficiente de várias aplicações na área.

Palavras-chave: Data Parallel Haskell. Paralelismo de dados aninhados. Processamento digital de imagens.

Sumário

Lista de Tabelas

Lista de Códigos

1 Introdução

Este trabalho visa estudar oportunidades de explorar a capacidade de processamento paralelo dos computadores modernos, com o objetivo de obter algoritmos mais eficientes para aplicações de processamento digital de imagens (PDI).

A arquitetura dos computadores modernos, inclusive PCs, é baseada em múltiplos processadores, sendo um exemplo particularmente importante o de unidades de processamento gráfico (GPUs) [14], constituídas de dezenas de processadores, e altamente eficientes para a manipulação de imagens gráficas. A estrutura altamente paralela de GPUs as torna também mais efetivas do que as CPUs de propósito geral para algoritmos nos quais pode ser explorado o processamento de grandes blocos de dados em paralelo.

Várias formas de paralelismo podem ser exploradas na execução de programas. O mais comum é o paralelismo de controle, no qual se criam explicitamente *threads* que executam paralelamente e cujo processamento é sincronizado por meio de mensagens, *locks* ou outros mecanismos de sincronização [21]. O paralelismo de dados consiste na execução simultânea de uma mesma instrução em diferentes porções de um conjunto de dados, sendo esta abordagem particularmente promissora para o emprego de grande número de processadores [5].

O trabalho pioneiro de Blelloch em NESL [3] mostrou que é possível combinar um modelo de programação bastante flexível - paralelismo de dados aninhados (NDP - *nested data parallelism*) - com um modelo de execução eficiente e de grande escalabilidade - *flat data parallelism* (FDP) -, ampliando o leque de possibilidades para explorar paralelismo em programas. Essa abordagem é implementada em Data Parallel Haskell (DPH) [6, 13], o *framework* de programação que propomos utilizar para o desenvolvimento deste projeto, e que consiste em uma extensão do compilador GHC [10] da linguagem Haskell [12] e de suas bibliotecas, de maneira e embutir na linguagem o NDP.

Uma área na qual parece ser particularmente fértil explorar essa forma de paralelismo para obter implementações mais eficientes é a área de PDI. Alguns exemplos de sucesso na implementação de algoritmos paralelos para PDI são reportados, por exemplo, em [16, 23].

Na Seção 2 são apresentadas motivações para a realização deste trabalho. Na Seção 3 são expostos o objetivo geral deste projeto, assim como os objetivos específicos alcançados. Na Seção 4 é exposta a metodologia utilizada na realização deste trabalho. Na Seção 5 são apresentadas a descrição do problema de PDI implementado com o DPH, a descrição do modelo NDP e as atividades desenvolvidas durante a realização deste trabalho. Por fim, na Seção 6 são apresentados os trabalhos futuros que servirão para concluir o trabalho atual e na Seção seguinte (Seção 7) é apresentado o cronograma das atividades planejadas para serem realizadas na conclusão da monografia na disciplina Monografia II (BCC391).

2 Justificativa

O projeto pretende apresentar contribuições relativas a uma questão extremante atual e importante, que consiste em estudar, desenvolver e avaliar técnicas para explorar a capacidade de processamento dos computadores modernos, com arquitetura baseada em múltiplos processadores, no sentido de obter algoritmos e implementações mais eficientes.

Em muitos campos as palavras paralelismo e concorrência são sinônimas; mas não é tão assim quando se fala de programação em Haskell, onde são usados para descrever conceitos fundamentalmente diferentes. Enquanto a programação paralela está preocupada apenas com a eficiência, programação concorrente está preocupada com a estruturação de um programa que precisa interagir com múltiplos e independentes agentes externos (por exemplo, o usuário, um servidor de banco de dados e alguns clientes externos). Concorrência permite que tais programas sejam modulares; a *thread* que interage com o usuário é diferente da *thread* que fala com o banco de dados. Embora seja possível programar paralelamente usando concorrência, isso é muitas vezes uma má escolha, pois a concorrência sacrifica o determinismo. Em Haskell, os modelos de programação paralela são deterministas.

Com a realização deste trabalho pretende-se estudar a eficiência do NDP, implementado no DPH [6], uma extensão do GHC [12], em algoritmos de PDI, uma vez que essa é uma abordagem recente na linguagem Haskell e são necessários experimentos em diversas aplicações para encontrar os possíveis pontos de melhoramento neste *framework* que se encontra em desenvolvimento.

3 Objetivos

3.1 Objetivo geral

O objetivo geral do projeto é explorar oportunidades de uso de NDP para a implementação eficiente de algoritmos de PDI.

3.2 Objetivos específicos

- Identificar algoritmos de PDI para os quais possam ser obtidas implementações mais eficientes, por meio do uso de paralelismo.
- Identificar a possibilidade do uso do NDP.
- Obter implementações paralelas eficientes desses algoritmos, que possam ser usadas em aplicações reais.
- Obter evidências a favor ou contra o argumento de que linguagens funcionais são particularmente apropriadas para a implementação de paralelismo.

4 Metodologia

1. Estudos dos diversos modelos de paralelismo, incluindo paralelismo de controle, paralelismo de dados, e NDP.

2. Seleção de aplicações interessantes de PDI, que possam apresentar melhoria de eficiência se implementadas usando paralelismo.

3. Instalação e configuração de plataformas de desenvolvimento, em particular, DPH.

4. Estudo da linguagem e bibliotecas de DPH.

5. Implementação de alguns algoritmos selecionados no item 2.

6. Comparação entre a implementação desenvolvida neste projeto e implementações que não exploram paralelismo, e também com outras implementações pararelas eventualmente existentes.

7. Avaliação dos resultados obtidos dos algoritmos implementados com NDP, em termos de eficiência, novas oportunidades de paralelismo, etc.

5 Desenvolvimento

O objetivo deste trabalho é a implementação de algoritmos de PDI utilizando DPH, uma extensão da linguagem Haskell que utiliza NDP.

Nesta primeira etapa (semestre) do projeto foi escolhido o problema de gerar árvore de componentes conectados (CCT - *Connected Component Tree*) [17].

5.1 Descição do problema

Em aplicações em que é utilizada a CCT, sua computação tipicamente demanda a maior parte tempo, chegando a utilizar 80% do tempo de execução somente na parte da computação da CCT, entre as partes de transformações sobre a CCT gerada e restituição da imagem. A vantagem da utilização da CCT é que, uma vez computada, o processamento é realizado na árvore com transformações sobre grafo e somente uma estrutura de dados é utilizada no processamento de baixo e alto nível [17].

A CCT pode ser utilizada em diversas aplicações [17], entre elas, filtragem [22], extração de movimentos [22], segmentação de bacias hidrográficas [8], segmentação de imagens astronômicas [1] e visualização de dados [7].

Existe três principais classes de algoritmos sequenciais para computar a CCT:

- *Flooding-based algorithms* - O processamento começa em um *pixel* com a menor intensidade e uma busca em profundidade é utilizada para a construção da CCT. Normalmente esses algoritmos utilizam estruturas de ordenação de dados como, por exemplo, filas de prioridade [24].

- *Emerging-based algorithms* - Os *pixels* da imagem são processados em ordem decrescente de luminosidade, requerindo o ordenamento dos *pixels* a priori [20].

- *1-D algorithms* - Os algoritmos pertencentes a essa classe constroem a CCT em sinais de uma dimensão. Não é necessário a ordenação dos *pixels* e a CCT pode ser construída em tempo linear [19]. Mas se a junção entre duas CCT for utilizada, os algoritmos desta classe podem ser empregados para gerar a CCT para qualquer dimensão [18].

O algoritmo implementado neste trabalho pertence à terceira classe [17].

5.2 Descrição do modelo de paralelismo

Em [3] foi desenvolvida uma nova linguagem útil para ensinar e implementar algoritmos em paralelo. A linguagem foi criada para permitir descrições em alto nível dos algoritmos paralelos e ao mesmo tempo ter um mapeamento bem compreendido em um modelo de desempenho. De acordo com [3] as seguintes características são importantes de se conseguir:

- Um modelo de desempenho baseado na linguagem que utiliza *work* (o número total de operações executadas por uma computação; especifica o tempo de execução em um processador sequencial) e *depth* (a mais longa cadeia de dependências sequenciais na computação; representa o menor tempo de execução possível assumindo uma máquina ideal com um número ilimitado de processadores) ao invés de modelo baseado na máquina que utiliza do tempo de execução.

- Suporte para construções de NDP. Isto é, a capacidade de aplicar uma função em paralelo em cada elemento de um conjunto de dados e a capacidade de aninhar tais chamadas paralelas.

No trabalho [3] em NESL de Blelloch foi mostrado que é possível combinar um modelo de programação bastante flexível - NDP - com um modelo de execução eficiente e de grande escalabilidade - FDP. A distinção entre as linguagens que aceitam FDP e NDP pode ser descrita da seguinte maneira: em linguagens com FDP uma função pode ser aplicada em paralelo sobre um conjunto de valores, mas essa função deve ser sequencial. Em linguagens com NDP [2] qualquer função pode ser aplicada sobre um conjunto de valores, inclusive funções paralelas.

5.2.1 Exemplo

Nesta Seção é mostrado a diferença entre uma implementação sequencial do algoritmo Quicksort sobre listas de Haskell e sobre o vetor paralelo (VP) do *framework* DPH. No Código 1 encontra-se a implementação utilizando o tipo List de Haskell e no Código 2 encontra-se a implementação com a utilização do DPH.

Código 1: Quicksort utilizando o tipo List de Haskell

```
1 quicksort :: [Int] -> [Int]
2 quicksort a = if length a <= 1 then a
3               else sa !! 0 ++ eq ++ sa !! 1
4   where m  = a !! 0
5         lt = [e | e <- a, e < m]
6         eq = [e | e <- a, e == m]
7         gt = [e | e <- a, e > m]
8         sa = [quicksort e | e <- [lt, gt]]
```

Código 2: Quicksort utilizando DPH

```
1 quicksort :: [:Int:] -> [:Int:]
2 quicksort a = if lengthP a <= 1 then a
3               else sa !: 0 +:+ eq +:+ sa !: 1
4   where m  = a !: 0
5         lt = [:e | e <- a, e < m:]
6         eq = [:e | e <- a, e == m:]
7         gt = [:e | e <- a, e > m:]
8         sa = [:quicksort e | e <- [:lt, gt:]:]
```

Como pode ser visto, com a utilização do DPH a maneira com que o programador escreve um código em paralelo é muito semelhante à maneira com que se escreve um código sequencial na própria linguagem.

5.2.2 Problemas

Uma vez que o *framework* DPH ainda se encontra em desenvolvimento e somente alguns componentes são utilizáveis [9], ainda não se tem a possibilidade de programar sobre o VP do DPH com as mesmas facilidades que se programa sobre listas em Haskell.

5.3 Atividades

5.3.1 Instalação dos softwares

O DPH está disponível como uma extensão do GHC 7.2 [11] e está disponível separadamente na forma de pacote cabal [4]. Uma vez que o cabal está instalado, para a instalação do DPH é necessário a execução dos seguintes comandos:

Código 3: Instalação dos pacotes do DPH

```
$ cabal update
$ cabal install dph-examples
```

As principais limitações do DPH incluem a incapacidade de misturar código que utiliza o VP (código vetorizado) e código que não o utiliza em um mesmo módulo, a necessidade de um Prelude de propósito especial no código vetorizado e a ausência de otimizações (levando a um baixo desempenho em alguns casos) [9].

Segundo [9], a implementação atual do DPH funciona bem para código com NDP em que a profundidade de aninhamento é estaticamente fixo, assim como deve funcionar rasoavelmente bem quando o aninhamento é recursivo (caso da geração da CCT), uma vez que não tenha tipos de dados definidos com VP e certos idiomas são evitados.

5.3.2 Descrição da implementação

Como foi dito na Seção 5.1 o algoritmo que propomos implementar [17] pertence à classe de algoritmos que constroem a CCT para sinais de uma dimensão, mas que utiliza uma estratégia de junção entre as CCT para que possa ser construído para qualquer dimensão.

O Código 4 contém a implementação do módulo responsável por realizar a computação da CCT em uma dimensão.

Código 4: Módulo responsável por construir a CCT em uma dimensão

```
1
2  -- /
3  --
4  module Build1D
5          (
6            build1D,
7          )
8          where
9
10 import Prelude ((++))
11 import qualified Prelude
12
13 import Data.Array.Parallel.Prelude -- For boolean operations
14
15 import Data.Array.Parallel.Prelude.Int (Int)
16 import qualified Data.Array.Parallel.Prelude.Int as I
17 import Data.Array.Parallel.Prelude.Word8 (Word8)
18 import qualified Data.Array.Parallel.Prelude.Word8 as W
19
20 import Data.Array.Parallel.PArray
21
22 import Stack
23
```

```
24 -- /
25 type Point = Int
26
27 -- /
28 type Root = Point
29
30 -- | Parent Info of a Line (PIL).  The second 'Point' is the parent of
31 -- the first 'Point'.
32 type PIL = (Point, Point)
33
34 -- /
35 build1D :: PArray Word8 -> PArray Point
36 build1D imgl =
37   update (replicate imgl_length 0) (fromList buildPT1)
38   where
39     imgl_length = length imgl
40
41     buildPT1 :: [(Int, Int)]
42     buildPT1 = finish (build1 emptyStack 0 1 [])
43
44     build1 :: Stack Point -> Root -> Point -> [PIL]
45               -> (Stack Point, Point, [PIL])
46     build1 s xr xp t
47       | xp I.>= imgl_length = (s, xr, t)
48       | otherwise           = build1 s' xr' (xp I.+ 1) (pinfo ++ t)
49       where (s', xr', pinfo) = step s xr xp
50
51     step :: Stack Point -> Root -> Point -> (Stack Point, Root, [PIL])
52     step s xr xp = let
53       ir = imgl !: xr
54       ip = imgl !: xp
55       xq = stackLast s
56       iq = imgl !: xq
57       in if ir W.< ip
58          then (stackPush s xr, xp, []) else
59          if ir W.== ip
60          then (s, xr, [(xp, xr)]) else
61          if stackEmpty s || iq W.< ip
62          then (s, xp, [(xr, xp)]) else
63          if iq W.== ip
64          then (stackPop s, xq, [(xp, xq), (xr, xq)]) else let
65            (stack, root, parent_info) = step (stackPop s) xq xp
66            in (stack, root, parent_info ++ [(xr, xq)])
67
68     finish :: (Stack Point, Root, [PIL]) -> [PIL]
69     finish (s, xr, t)
70       | stackEmpty s = (xr, -1) : t
71       | otherwise    = finish (stackPop s, xq, (xr, xq) : t)
72       where xq = stackLast s
```

Uma vez que o algoritmo para uma dimensão não é passível de ser paralelizado, mas mesmo assim é necessário construí-lo sobre o VP do DPH, para compatibilidade com o módulo que irá computar a CCT para duas ou mais dimensões, foi construído sobre o tipo PArray.

A maneira que foi representada cada linha da imagem para ser utilizada pelo algoritmo consiste em um VP de Word8.

Para a utilização da função buildID definida neste módulo, o módulo que a utilizará deverá implementar uma função *unwrapper* (do tipo [:Word8:] -> [:Int:]) que converterá a entrada para o tipo PArray Word8 e utilizará a função buildID, convertendo novamente o retorno desta para ficar compatível com o retorno da função *unwrapper*. Esta função será definida no módulo que está sendo construído o algoritmo que faz a junção dos resultados parcias sobre uma dimensão, como mostrado mais adiante.

No Código acima, a função build1 é responsável por percorrer todos os pontos da linha da imagem recebida, utilizando da função step para avaliar os valores de cada iteração e definir os novos valores de parametros para a próxima iteração.

Após percorrida toda a linha, a função finish é responsável por terminar de construir uma nova lista de atualizações, com base na pilha (o algoritmo [17] utiliza uma pilha), o atual ponto raíz após o término do processamento anterior e a lista de atualizações, que será utilizada para construir a CCT para uma dimensão.

Uma vez conseguida a lista de atualizações com a função buildPT1, na linha 37 é construída a CCT com base nesta lista.

O Código 5 a seguir contém o que já foi implementado do módulo responsável por construir a CCT para duas dimensões. Observe que a linha 24 contém a definição da função *unwrapper* apontada anteriormente.

Código 5: Módulo responsável por construir a CCT

```
1
2  {-# LANGUAGE ParallelArrays #-}
3  {-# OPTIONS_GHC -fvectorise #-}
4
5  module Merge
6          (
7          )
8          where
9
10 import qualified Prelude
11
12 import Data.Array.Parallel
13
14 import Data.Array.Parallel.Prelude
15
16 import Data.Array.Parallel.Prelude.Int (Int)
17 import qualified Data.Array.Parallel.Prelude.Int as I
18 import Data.Array.Parallel.Prelude.Word8 (Word8)
19 import qualified Data.Array.Parallel.Prelude.Word8 as W
20
21 import qualified Build1D
22
23 -- | Unwrapper for the 'Build1D.build1D'.
24 build1D :: [:Word8:] -> [:Int:]
25 build1D imgl = fromPArrayP (Build1D.build1D (toPArrayP imgl))
26
27 type Image = [:[:Word8:]:]
28 type Line = Int
29 type Column = Int
30 type Point = (Line, Column)
31 type PTree1 = [:Point:]
32 type PTree2 = [:PTree1:]
33
```

```
34 type Temp0 = ([:PTree1:],[:PTree1:])
35 type Temp = [:Temp0:]
36
37 extract :: Int -> PTree2 -> [:Int:] -> Temp
38 extract n pt2 idxs = mapP f idxs
39   where x = (2 Prelude.^ n) I.- 1
40         f :: Int -> Temp0
41         f i = (sliceP (i I.- x) x pt2,
42                sliceP (i I.+ 1) x pt2)
43
44 mergeAll :: Image -> PTree2
45 mergeAll img = Prelude.undefined
46   where part :: [:PTree1:]
47         part = prepare 0 (mapP build1D img)
48           where prepare :: Line -> [:[:Column:]:] -> [:PTree1:]
49                 prepare = Prelude.undefined
50         h :: Int
51         h = lengthP img
52         w :: Int
53         w = lengthP (img !: 0)
54
55         f :: Point -> Word8
56         f (l, c) = img !: l !: c
57
58         merge :: Line -> (PTree1, PTree1)
59         merge l = merge2 lines 0
60           where lines :: (PTree1, PTree1)
61                 lines = (part !: l, part !: (l I.+ 1))
62                 merge2 :: (PTree1, PTree1) -> Int -> (PTree1, PTree1)
63                 merge2 = Prelude.undefined
64                 connect :: (Column, Column) -> (PTree1, PTree1)
65                 connect (pl1, pl2) = Prelude.undefined
66                   where p1 = (l, pl1)
67                         p2 = (l I.+ 1, pl2)
68                         xp = levroot p1
69                         yp = levroot p2
70                         (x, y) = if f yp W.> f xp
71                                  then (yp, xp) else (xp, yp)
72
73         levroot :: Point -> Point
74         levroot p@(_, -1) = p
75         levroot p         = let par_p = par p in
76           if f p W.== f par_p then levroot par_p else p
77         par :: Point -> Point
78         par (l, c) = part !: l !: c
79
80 -- | This function receive last line index of the image and return the
81 -- information of what lines will be merged in what order.
82 joinInfo :: Line -> [:[:Line:]:]
83 joinInfo n = Prelude.undefined
```

Como pode ser visto pelas linhas que contém a chamada Prelude.undefined, o módulo ainda contém partes que não foram implementadas, ficando o término desta implementação entre os trabalhos futuros, descritos na Seção 6.

6 Trabalhos futuros

O primeiro dos trabalhos futuros a ser realizado é o término da implementação do algoritmo de geração da CCT [17].

A partir daí, o próximo trabalho a ser realizado é a comparação com outras implementações, assim como uma análise de como o algoritmo está sendo paralelizado pelo DPH com o monitoramento do uso das CPUs durante a execução do mesmo.

Uma vez que o GHC compila o código Haskell ou diretamente para o código de máquina utilizando LLVM [15] ou utilizando código C como intermediário, um possível trabalho futuro é a análise das diferenças obtidas diante dessas duas abordagens.

A implementação de outros algoritmos de PDI e uma análise mais aprofundada da eficiência obtida com o DPH quando utilizado em algoritmos de PDI em geral.

No caso dessas análises indicarem bons resultados, planeja-se, a implementação de uma biblioteca de funções para PDI com enfoque em paralelismo, a qual deverá ser útil para implementação eficiente de várias aplicações na área.

Uma outra possibilidade de trabalho futuro é quanto à utilização de GPUs nas implementações dos algoritmos de PDI em Haskell.

7 Cronograma de atividades

As atividades estão planejadas para serem realizadas na conclusão da monografia na disciplina Monografia II (BCC391) de acordo com a Tabela 1.

Atividades	**Mar**	**Abr**	**Mai**	**Jun**	**Jul**
Terminar a implementação de [17]	X				
Seleção de aplicações		X	X	X	
Estudo das bibliotecas	X	X	X	X	
Implemetação de algoritmos	X	X	X	X	
Comparações entre implementações		X	X	X	
Avaliação dos resultados		X	X	X	
Redigir a monografia				X	X
Apresentação do trabalho					X

Tabela 1: Cronograma de Atividades.

Referências

[1] Ch. Berger, T. Geraud, R. Levillain, N. Widynski, A. Billard, and E. Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. In *IEEE International Conference on Image Processing*, pages IV–41–IV–44, San Antonio, TX, 2007.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, first edition, 1990.

[3] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] The Haskell Cabal, a common architeture for building applications and libraries. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.9361`.

[5] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proc. of Declarative Aspects of Multicore Programming*, 2011.

[6] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proc. of ACM Workshop on Declarative Aspects of Multicore Programming*, Nice, 2007.

[7] Yi-Jen Chiang, Tobias Lenz, Xiang Lu, and Güter Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. In *Computational Geometry: Theory and Applications - Special issue on the 19th European workshow on computational geometry*, pages 165–195, Amsterdam, The Netherlands, 2005.

[8] Michel Couprie, Laurent Najman, and Gilles Bertrand. Quasi-linear algorithms for the topological watershed. *Journal of Mathematical Imaging and Vision*, 22(2–3):231–249, 2005.

[9] Data Parallel Haskell project status. `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell#Project_status`.

[10] The Glasgow Haskell Compiler. `http://haskell.org/haskellwiki/GHC`.

[11] The Glasgow Haskell Compiler download. `http://www.haskell.org/ghc/`.

[12] Simon Peyton Jones. Haskell report. Technical report, Cambridge University Press, 2003.

[13] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 383–414, 2008.

[14] Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors. *Facing the Multicore-Challenge*. Aspects of New Paradigms and Technologies in Parallel Computing Series. Lecture Notes in Computer Science, first edition, 2010.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[16] Ben Lippmeier, Gabriele Keller, and Simon Peyton Jones. Efficient parallel stencil convolution in haskell. In *Proc. of International Conference on Functional Programming*, 2011.

[17] P. Matas, E. Dokladalova, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev. Parallel algorithm for concurrent computation of connected component tree. *Advanced Concepts for Intelligent Vision Systems*, 5259:230–241, 2008.

[18] Arnold Meijster. *Efficient Sequential and Parallel Algorithms for Morphological Image Processing.* Phd thesis, Faculty of Mathematics and Natural Sciences, Groningen, Feb. 2004. http://irs.ub.rug.nl/ppn/260518212.

[19] David Menotti, L. Najman, and A. Araújo. 1d component tree in linear time and space and its application to gray-level image multithresholding. In *Proceedings of the 8th International Symposium on Mathematical Morphology*, pages 437–448, Rio de Janeiro, RJ, Brazil, 2007.

[20] Laurent Najman and Michel Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11):3531–3539, 2006.

[21] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, first edition, 2011.

[22] P. Salembier, A. Oliveras, and L. Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.

[23] F. J. Seinstra and D. C. Koelma. User transparency: A fully sequential programming model for efficient data parallel image processing. *Concurrency and Computation: Practice and Experience*, 16(6):611–644, 2004.

[24] Michael H. F. Wilkinson. A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In *IEEE International Conference on Image Processing*, pages 1041–1044, 2011.