Universidade Federal de Alagoas
Instituto de Computação

Trabalho de Conclusão de Curso

Detecção de paralelismo através de grafos de dependências em filtros convolucionais

Bruno Lopes Vieira
blv@tci.ufal.br

Orientadores:
Eliana Silva de Almeida
Alejandro C. Frery

Maceió, janeiro de 2007

Bruno Lopes Vieira

Detecção de paralelismo através de grafos de dependências em filtros convolucionais

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Eliana Silva de Almeida

Alejandro C. Frery

Maceió, janeiro de 2007

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Eliana Silva de Almeida - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Alejandro C. Frery - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Márcio N. Miranda - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, janeiro de 2007

Resumo

A computação paralela vem se desenvolvendo cada vez mais como uma forma de suprir a constante necessidade de aumentar o desempenho do *software*. Este trabalho propõe realizar uma análise de viabilidade de paralelização em filtros de imagens utilizando o formalismo de grafos de dependências. Para realizar esta análise define-se inicialmente um algoritmo de conversão de código seqüencial para um grafo de dependências. Uma vez efetuada essa definição, apresenta-se o código de um filtro convolucional e seu respectivo grafo mapeado através desse algoritmo. Na seqüência, é proposta a análise de viabilidade da paralelização, visto que há problemas temporais na comunicação dentre os nós de processamento, e discutem-se técnicas que podem ajudar a reduzir esse atraso na comunicação. Finalmente, observa-se a redução em 2 graus na ordem de operações seqüenciais no estudo de caso definido neste trabalho.

Abstract

The constantly increasing need of software performance is tackled, among other techniques, by the use of parallel computing. In this work we analyze the use of parallel computing by means of dependence graphs in the implementation of convolutional image filters. The first stage consists of defining an algorithm for mapping a sequential code into a dependence graph; we show an application of this mapping to a Gaussian filter. The feasiblility of this parallelization is performed, since there are temporal issues in the communication among nodes; we discuss techniques for alleviating possible delays. Finally, we note that a reduction of two orders of magnitude in the number of sequential operations is achieved in a case study.

Agradecimentos

Primeiramente tenho a obrigação de agradecer a meus pais, que, apesar de todas as dificuldades, proporcionaram-me a oportunidade de vivenciar este ambiente acadêmico.

A Profa (vulga Prof. Dra. Eliana Almeida) pelo carinho. Por ter acreditado em meu trabalho. Por todas as incontáveis reuniões para discutir o andamento do trabalho. Por todas as agradabilíssimas saídas ao Mr. Joca. Por todo o companheirismo que ela me proporcionou.

Ao Profe (vulgo Prof. Dr. Alejandro Frery) por toda a amizade. Por toda a ajuda. Por toda a paciência. Por todos os dias/tardes/noites em que abusei de sua boa vontade em reuniões intermináveis a discutir os resultados. Por todas as caronas que me foram oferecidas. Por tudo o que aprendi sobre canetas.

A Carlos Bazílio pela presteza em ajudar quando possível.

A Björk Guðmunsdóttir pelas horas a fio cantando enquanto eu redigia esse trabalho, mantendo-me calmo nas horas de problemas maiores.

A Herli e Val por toda a disposição em ajudar. Por toda a amizade. Por todo o apoio incondicional.

A Lorena por todas as horas em que me ouviu. Por toda a amizade. Por todo o apoio incondicional.

A Everton por ter fornecido a base para que eu pudesse escrever esse trabalho. Material este do qual proveio a maior parte de minha inspiração.

A todos aqueles que direta ou indiretamente me ajudaram. . .

Meus mais sinceros agradecimentos a todos.

Sumário

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

Este trabalho propõe uma ferramenta destinada a automatizar a paralelização de filtros de imagens gerados em código R. A proposta discorre em, dado um código R de um filtro de imagem convolucional como entrada, mapeá-lo em um grafo de dependências e, em seguida, produzir um filtro que funcione em paralelo, em código R. Vale ressaltar que os procedimentos definidos são correlatos, independentemente da linguagem de programação adotada.

O intuito é o de facilitar o processo de tradução de código seqüencial para a programação paralela, sem exigir conhecimentos de computação paralela.

1.1 Motivação

No decorrer de toda a história da computação, desempenho de *software* sempre foi um fator crucial. De forma a prover uma solução a essa questão, o *hardware* evoluiu consideravelmente nos últimos anos. Porém, limitações físicas, como a crescente dificuldade de criar componentes menores, frearam essa melhoria de desempenho. Com o uso de algoritmos que exigem cada vez mais recursos computacionais, busca-se uma alternativa para suprir as necessidades de otimização. A computação paralela apresenta-se como alternativa concreta de acréscimo de desempenho (ver Pancake, 1996).

Não obstante a estagnação no nível de processamento disponível, têm-se as limitações de dependência internas às estruturas dos programas. Ao se dividir um programa entre unidades de processamento, pode-se comprometer o seu fluxo de execução, ou seja, uma unidade pode ter de esperar que outra finalize alguma tarefa para que possa dar continuidade ao fluxo de sua execução. Para ilustrar, observe-se o código 1.1.

Mapeando este código para uma estrutura que permita visualizar o fluxo de controle (figura 1.1) em uma execução, pode-se observar que algumas das

Código 1.1: Trecho de código com dependência

```
1 for ( y in 0 : 24 )
2     for ( x in 0 : 80 ) {
3         settext [x,y] ( "#" )
4     }
```

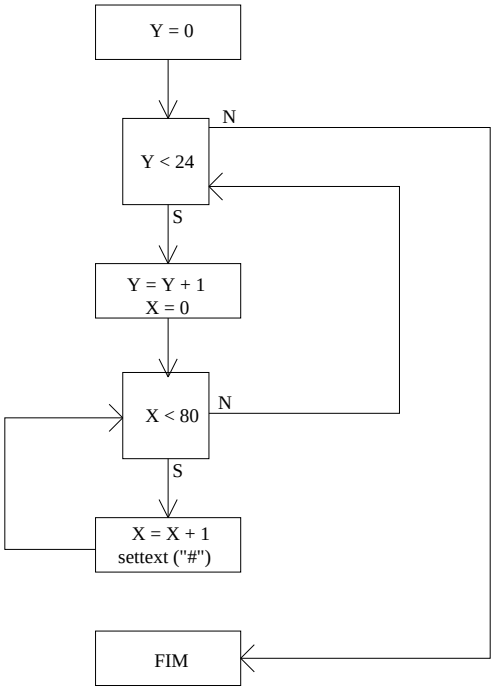

Figura 1.1: Fluxo de controle de código com dependência

estruturas não podem ser executadas paralelamente, visto que possuem dependência de dados. Ou seja, para que algumas operações sejam efetuadas, é necessário o resultado da execução de outras. Assim sendo, pode-se observar que o desenvolvimento de algoritmos paralelos exige uma análise de fluxo e de dependências. Outros problemas também serão encontrados, conforme pode-se observar no trabalho de Bergmark & Pancake (1990).

1.2 Justificativa

Ao analisar um programa através de sua representação via um grafo de dependências algumas vantagens podem ser identificadas, tais como:

- as dependências de um programa são claramente visíveis;
- a fácil conversão do programa seqüencial para o grafo de dependências e do grafo para seu código equivalente paralelo;

- a independência da linguagem de programação utilizada na codificação do algoritmo.

Para que um usuário possa, de forma simples, efetuar uma análise de paralelização, o uso de grafos de dependências apresenta-se como alternativa considerável. Outras fontes teriam como resultado final também uma "caixa preta", porém as até então propostas, assim como Martins (2000), não conseguem administrar a independência da linguagem de programação em que o código inicial foi gerado.

1.3 Objetivos

Como objetivo geral, este trabalho visa realizar um estudo dos algoritmos de filtragem de imagens monoespectrais quanto a geração do código paralelo, utilizando a plataforma R.

Especificamente, pretende-se:

- Estudar o formalismo grafo de dependências e utilizá-lo como linguagem de representação intermediária.
- Estudar os algoritmos para filtragem convolucional de imagens.
- Representar estes algoritmos no formalismo de grafo de dependências.
- Efetuar análise de paralelização.

Pretende-se também definir um modelo de um sistema capaz de transpor um filtro de imagens convolucional elaborado monoliticamente, ou seja, com execução seqüencial (ver Tanembaum, 2001), para o sistema R, à um filtro paralelo.

1.4 Estado da arte

O uso de grafos de dependências para a otimização de programas foi proposto por Ferrante & Ottenstein (1987). No estudo citado, pela primeira vez utiliza-se essa subclasse de grafos, compreendida pelos autores como linguagem intermediária, consistindo numa ferramenta de otimização de *software*.

Dentre os trabalhos correlatos a essa proposta, tem-se o de Allen et al. (1987), onde, para sistemas paralelos que possuem memória local a cada nó

de execução e uma outra global, apresentam-se técnicas concentradas na paralelização de estruturas de repetição na forma `do–loop`. Essas são ditas instruções *forall*, que representam uma estrutura de repetição independente de outras iterações, ou seja, não possuem índices de controle alterados fora do laço. Esse é o caso ideal para paralelização, pois não há necessidade de modificação em controles externos à iteração.

Para determinar a quantidade de instruções seqüenciais a serem enviadas a cada nó de processamento, McGregor & Riehl (1989) analisam a agregação de cada módulo do programa. Dessa forma, o trabalho apresenta um algoritmo para definir cada bloco de instruções a ser enviado a cada nó de processamento. Isso pode resolver vários problemas de comunicação dentre os nós, proporcionando maior desempenho do código resultante.

Blume et al. (1994) apresentam um conjunto de técnicas para detecção de paralelismo voltado a sistemas de memória compartilhada com endereçamento global. Isto é, sistemas onde virtualmente haja uma única memória acessível por todos os nós. São utilizados conceitos como análise de dependência, privatização e reconhecimento de idiomas, de forma a trabalhar independente da linguagem de origem do código a ser analisado.

Para a analisar os resultados da paralelização, Eigenmann et al. (1998) expõem dados de *benchmark* em sistemas paralelizados manualmente, propondo a implantação de algumas das técnicas atualmente utilizadas em compiladores. Trata-se de um estudo com base empírica, de onde foram extraídos alguns conceitos para tentar melhorar o desempenho de sistemas paralelos.

No trabalho de Banerjee et al. (1993) tem-se a detecção de paralelismo em trechos de códigos, num patamar genérico a linguagens, gerando um baixo grau de paralelismo. Além disso, o trabalho apresenta uma série de técnicas de análise de viabilidade da paralelização, de forma a garantir que o código resultante possua desempenho satisfatório.

Martins (2000) apresenta um modelo de automação da paralelização através de grafos de dependências e semântica denotacional. Apresenta-se a implementação de um detector automático de paralelismo para a linguagem `C`.

Neste trabalho utiliza-se a base teórica supracitada de forma a gerar um algoritmo de detecção de paralelismo para filtros de imagens convolucionais. Diferente do trabalho de Martins (2000), não se trabalha com a semântica da linguagem e sim com um mapeamento direto do código a um grafo de dependências. Dessa forma o algoritmo independerá da linguagem de origem, assim como no trabalho de Banerjee et al. (1993), porém o presente oferece alto grau de paralelização. Ou seja, via um formalismo teórico, observar-se-á claramente as estruturas passíveis de paralelização. A partir daí, sua trans-

posição a um código paralelo pode ser definida algoritmicamente. Para que sejam evitados demasiados problemas de comunicação, os conceitos expostos por McGregor & Riehl (1989) serão utilizados.

Como base para a análise do grafo de dependências, de forma a compreender as estruturas que podem ser paralelizadas, utilizam-se os conceitos apresentados em Banerjee et al. (1993). No trabalho citado são descritas técnicas, numa quase que engenharia de paralelização. Assim, o supracitado descreve formas de analisar estruturas de repetição, possibilitando sua conversão em estruturas seqüenciais mais fáceis de compreender através de um grafo, sendo esse material aprovado pelo IEEE (ver IEEE, 2006).

1.5 Metodologia

Como metodologia que conduz o estudo têm-se:

1. Levantamento bibliográfico atualizado de forma a compreender o processo de filtragem convolucional de imagens, a estrutura de grafo de dependências, a linguagem de programação `R` e o paradigma de computação paralela.

2. Mapeamento do código de um filtro convolucional de imagens (gaussiano) em linguagem `R` para um grafo de dependências, de forma a analisar as possibilidades de paralelização.

3. Análise de agregação de comandos paralelos de forma a reduzir problemas de comunicação dentre os nós.

4. Proposta de codificação em linguagem `R` para o filtro paralelizado, com base na análise anterior.

De forma a contemplar a plena execução desse trabalho, utilizam-se como meios de pesquisa bibliográfica o acesso ao portal de periódicos da Capes (`www.periodicos.capes.gov.br`), ao portal *ISI Web of Science*, a ser acessado a partir do anterior, ao portal *Science Direct* (`www.sciencedirect.com`) e ao mecanismo de buscas acadêmicas *Google Scholar* (`scholar.google.com`). Essa é a fonte primária de referências bibliográficas.

Utiliza-se ao menos um computador com sistema operacional Linux, plataforma `R`, disponível a partir de `www.r-project.org` e o sistema `Rmpi` (Yu, 2006), que necessita da interface `MPI` (Burns et al., 1994). Como ferramenta para gerar grafos, o `Dot` (Gansner et al., 2002).

1.6 Estrutura

O trabalho segue estruturado em cinco capítulos, donde os dois primeiros são puramente teóricos, tratando das definições e bases necessárias à proposta, o terceiro teoriza sobre grafos e aplica sua usabilidade, na proposta de um sistema computacional e os demais apresentam os resultados do trabalho; a saber:

Capítulo 1: Introdução.

Capítulo 2: Fundamentos teóricos: definição de grafos de dependências, código intermediário e filtros de imagens convolucionais.

Capítulo 3: Definição da usabilidade no mapeamento do código de um filtro de imagem proposto em código `R`.

Capítulo 4: Análise de paralelização do grafo gerado.

Capítulo 5: Conclusão: discussão dos resultados, considerações finais e proposta de trabalhos futuros.

Capítulo 2

Fundamentos teóricos

Este capítulo tem como função prover o conhecimento necessário ao entendimento do trabalho. Seguem informações sobre os modelos de representação utilizados e sobre o objeto de estudo, explanando o estado da arte.

2.1 Grafos de dependências

Formalmente, um grafo pode ser definido como um conjunto $G = \{V, E\}$ finito e não vazio, onde V representa um conjunto de vértices e E um conjunto de arestas que interligam esses vértices; a partir daí, temos dois tipos de grafos:

Grafo Direcionado: apresenta as funções de mapeamento s e $t : E \to V$, que representam o mapeamento de ligações, onde $s(e)$ representa o vértice fonte e $t(e)$ o vértice alvo da aresta e;

Grafo Não Direcionado: também conhecido simplesmente como grafo, apresenta uma função $w : E \to P(V)$, onde P é o conjunto das partes, o que representa cada aresta como um par ordenado, conseguinte que o par $\{u, v\}$ representa a mesma aresta que $\{v, u\}$, ou seja, cada aresta conecta dois vértices em ambas as direções.

Um grafo de dependências representa unidades atômicas de execução de um programa, unidas por dependências de fluxo de dados e/ou controle, como pode ser visto na figura 2.1; dessa forma, pode-se representar todas as dependências dentre as estruturas de processamento de um algoritmo.

Sabendo-se que um grafo de dependências representa relações onde é imprescindível saber quem é o dependente, observa-se que ele pertence à categoria dos *grafos direcionados*. Uma forma de representá-lo, matematicamente, então, seria:
$V = \{a, b, c, d, e\}$

Código 2.1: Dependência de fluxo

```
1 X = 20
2 ...
3 Y = X * 2
```

Código 2.2: Anti-Dependência

```
1 Y = X * 2
2 ...
3 X = 20
```

$E = \{(a,b),(a,e),(b,c),(b,d),(c,d),(e,c)\}$
$G = \{V, E\}$

Um grafo de dependências é entendido como uma linguagem intermediária, ou seja, uma linguagem não implementável, porém, capaz de representar qualquer estrutura algorítmica computável, independente da linguagem na qual o código foi escrito, através de um mapeamento, como pode ser visto em Ferrante & Ottenstein (1987).

As dependências de código são divididas em três categorias. A saber:

Dependência de fluxo: Uma variável é definida e tem seu valor utilizado posteriormente (código 2.1).

Anti-dependência: Uma variável tem seu valor consultado e em seguida é redefinida (código 2.2).

Dependência de saída: Uma variável é definida e em seguida redefinida (código 2.3).

Tome-se como exemplo a figura 2.1, onde o grafo supracitado está representado graficamente. Observe-se a facilidade na análise dos fluxos, de controle e das dependências de dados. É possível observar que alguns nós dependem de outros para sua execução.

De modo a ilustrar, a figura 2.2 apresenta o grafo de dependências do código 1.1. No caso em questão, observa-se que `for_x` não pode ser executado

Código 2.3: Dependência de saída

```
1 X = 20
2 ...
3 X = Y
```

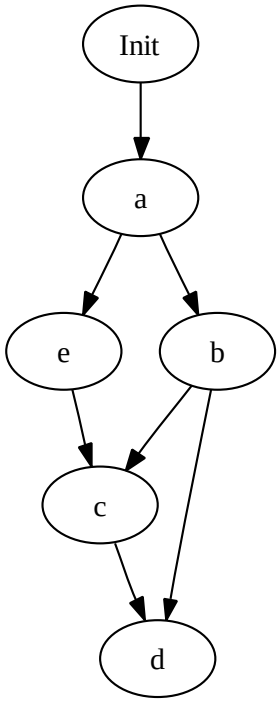

Figura 2.1: Representação gráfica de um grafo

antes de `for_y` devido a uma dependência de controle. Do mesmo modo, `inc_y` só pode ser processado após `for_y`, devido a uma dependência não só de controle, como de dados; visto que o valor da variável `y` seria alterado ao mesmo tempo em que comparado, gerando uma inconsistência.

2.2 Código intermediário

Como dito anteriormente, ao se representar um algoritmo através de um código intermediário, obtém-se a vantagem de independência no mapeamento, ou seja, o código mapeado não está atrelado à linguagem de origem. Dessa forma, aumentam-se as possibilidades de generalização de qualquer processo a ser efetuado no grafo então mapeado. Uma das vantagens de se trabalhar com esse tipo de linguagem é que se obtem uma maneira simples de paralelizar programas (ver Ferrante & Ottenstein, 1987; Martins, 2000).

Um grafo de dependências representa um sistema computacional independentemente da fonte anterior; ou seja, o grafo não possui resquícios da linguagem de programação utilizada para gerar o código que lhe serviu de base. A partir do advento desta independência, permite-se o mapeamento linguagem de programação $\rightarrow$ linguagem de especificação formal não atrelado ao paradigma do código de origem.

Exemplificando, observe-se o código 2.4 mapeado num grafo de dependên-

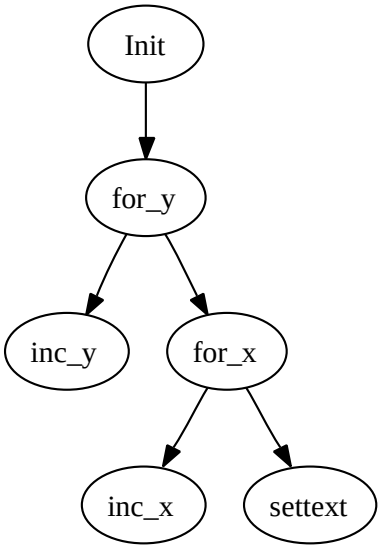

Figura 2.2: Grafo de dependências de laço com iterações externas

Código 2.4: Pseudocódigo para aplicação de dois filtros de imagens

```
for ( x in 1 : 2 )
    for ( y in 1 : 2 )
        aplicaFiltroNPixelAPixel ( x, y )
        aplicaFiltroMPixelAPixel ( x, y )
    end
end
```

cias na figura 2.3.

Observe-se na figura 2.3 que cada laço da repetição está explicitado. Considerando que não importa a ordem de execução de cada filtro (o resultado ao se aplicar o filtro M e o N em seguida é o mesmo de N e em seguida M), ou seja, não há dependência de dados. Pode-se concluir que os filtros podem ser aplicados em paralelo. Da mesma forma, conclui-se que o conteúdo do primeiro laço de repetição pode ser executado em vias paralelas (o segundo laço deve ser repetido 2 vezes, ver código 2.4, as quais podem ser executadas ao mesmo tempo).

2.3 Filtros de imagens

Filtros de imagens são algoritmos capazes de obter novas imagens a partir de transformações lógicas e/ou matemáticas dos dados de entrada. Podem se apresentar como um conjunto de instruções, tanto na linguagem de circuitos eletrônicos (ASIC, PLD) como em um programa de computador (processadores de uso geral).

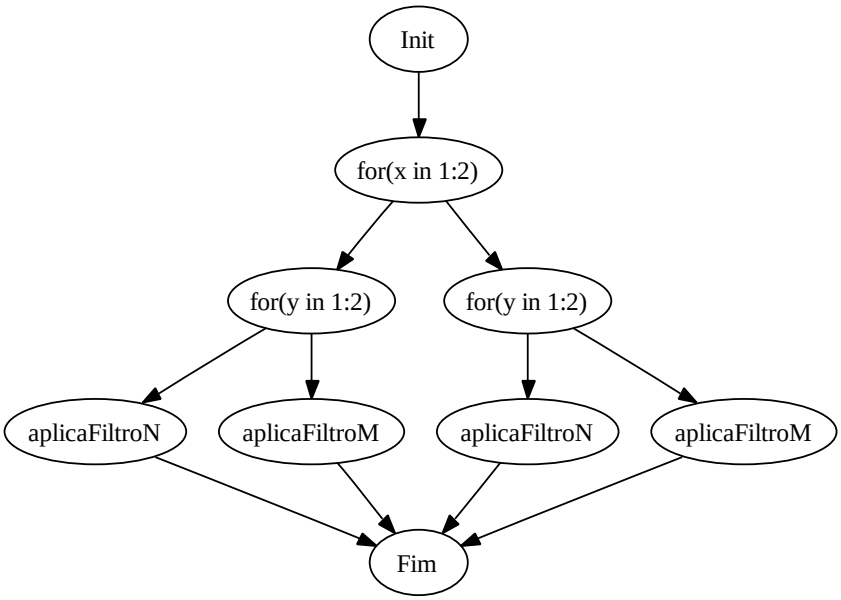

Figura 2.3: Grafo da aplicação de dois filtros

O uso de filtros é essencial na cadeia de processamento e análise de imagens, pois eles permitem combater elementos nocivos à compreensão das informações, como por exemplo ruídos e outros artefatos, e realçar aquelas características de interesse como, por exemplo, bordas.

2.3.1 Filtros de imagens convolucionais

Uma imagem monoespectral é uma função $f\colon S \to \mathbb{R}$, onde a grade euclidiana finita $S = \{0,\dots,m-1\}\times\{0,\dots,n-1\}$ é o suporte. O par $(s, f(s))$, com $s \in S$, chama-se pixel, e pode ser conveniente deixar em evidência as componentes da coordenada, isto é, usar (i,j) no lugar de $s \in S$.

Dada a matriz de convolução de lado ℓ ímpar

$$A = (a_{i,j})_{\frac{\ell-1}{2}\le i,j\le\frac{\ell+1}{2}},$$

com $a_{i,j} \in \mathbb{R}$, define-se a imagem $g\colon S \to \mathbb{R}$ resultante de filtrar a imagem f por convolução com a máscara A como sendo

$$g(k,\ell) = \begin{cases} \displaystyle\sum_{-\frac{\ell-1}{2}\le i,j\le\frac{\ell+1}{2}} a_{i,j}\, f(k-i,\ell-j) & \text{se } (k,\ell)\in S', \\ f(k,\ell) & \text{caso contrário,} \end{cases} \tag{2.1}$$

onde $S' = \{\frac{\ell-1}{2},\dots,m-\frac{\ell+1}{2}\}\times\{\frac{\ell-1}{2},\dots,n-\frac{\ell+1}{2}\}$. Freqüentemente tem-se que $\sum_{i,j} a_{i,j} = 1$.

Não pertencem à classe dos filtros convolucionais os filtros não lineares

(como, por exemplo, o da mediana) nem os adaptativos (como, por exemplo, o de Lee). Embora possa parecer restrita, o uso da classe de filtros convolucionais permite obter resultados muito importantes.

Uma representação conveniente de sinais em geral e de imagens em particular, é através da transformada de Fourier, ou representação no domínio da freqüência (Richards & Jia, 1999). Se $f\colon S \to \mathbb{R}$, com $S = \{0,\ldots,m-1\}\times\{0,\ldots,n-1\}$ é uma imagem, então a sua transformada de Fourier é a matriz complexa $F\colon S \to \mathbb{C}$ dada por

$$F(u,v) = \mathcal{F}(f)(u,v) = \frac{1}{2\pi}\sum_{k=0}^{m-1}\sum_{\ell=0}^{n-1} f(k,\ell)\exp\{-i2\pi uk/m\}\exp\{-i2\pi v\ell/n\},$$

onde $i = \sqrt{-1}$. De acordo com o *teorema da convolução*, pode-se expressar qualquer convolução no domínio da freqüência como

$$\mathcal{F}(f * g) = \mathcal{F}(f)\ \mathcal{F}(g) \tag{2.2}$$

(ver Barrett & Myers, 2004).

2.3.2 Paralelização em filtros de imagens

Tal como visto na equação (2.1), os filtros convolucionais são baseados em operações matriciais de multiplicação e soma. As operações matriciais, além de muitas outras, são classicamente paralelizáveis (ver Golub & Loan, 1996; Lätt & Chopard, 2004; Quinn, 2003) e, com elas, os filtros convolucionais.

A princípio, tentou-se utilizar uma capacidade dos processadores conhecida como *superescalar* (ver Grama et al., 2003), a qual permite que instruções sejam paralelizadas em *pipelines* (ver Tanembaum, 2001) internos. Entretanto, há um limite nessa capacidade, não suportando uma quantidade muito grande de operações simultâneas.

Atualmente existem diversas linguagens de programação capazes de gerar filtros paralelos, inclusive a R (ver Yu, 2006). Porém, não se tem conhecimento da paralelização automatizada de programas integrada a uma análise de viabilidade da paralelização; isto é, em alguns casos, devido à necessidade de troca de mensagens dentre os processadores, paralelizar um programa faz com que ele perca desempenho.

Basta compreender que em computação paralela utilizam-se diversos processadores (nós). Para que o processamento seja consistente, é necessário que os nós consigam trocar mensagens, isto é, transmitir o resultado de sua tarefa para um outro que espera esse resultado para poder continuar seu trabalho

(ver Quinn, 2003; Grama et al., 2003). Essa troca de mensagens, entretanto, é transparente ao usuário, sendo gerenciada, nesse estudo, pela interface `MPI` (ver Burns et al., 1994), que possibilita gerir essa troca de mensagens na linguagem `R` (através do trabalho de Yu, 2006), sem nenhuma necessidade de intervenção.

Capítulo 3

Mapeamento de código seqüencial em um grafo de dependências

Neste capítulo discutem-se os princípios de paralelização, bem como explicita-se o material utilizado no estudo de caso. A partir da discussão, define-se um algoritmo genérico de conversão a filtros convolucionais.

3.1 A linguagem R

Neste trabalho, faz-se necessário o entendimento da linguagem adotada. Optou-se por utilizar a linguagem R por ela estar sendo fortemente utilizada nas aplicações de filtros de imagens, visto que apresenta diversas ferramentas para computação científica, além de ter seu uso gratuito e código fonte disponível (utiliza a licença GPL – ver www.fsf.org).

A princípio, utiliza-se apenas a base estruturada da linguagem, tendo, portanto, como base léxica:

- funções
- procedimentos
- laços de repetição (for, repeat, while)
- estruturas condicionais (if)

Como ferramental capaz de prover suporte à programação paralela no R, tem-se a biblioteca Rmpi (ver Yu, 2006). Através dela, é possibilitada a comunicação do R com os demais nós (cada um dos processadores a quem se distribui as tarefas) através da interface MPI (ver Burns et al., 1994).

Como ilustração de sua utilização, observe-se o código 3.1; nele, apresentam-se alguns dos comandos básicos de acesso à interface de comunicação com

Código 3.1: Código utilizando a biblioteca Rmpi

```
#Carrega a biblioteca Rmpi
if ( !is.loaded ( "mpi_initialize" ) ) library ( Rmpi )
#Utiliza o maior número possível de nós
mpi.spawn.Rslaves ()
#Verificação da quantidade mínima de nós
if ( mpi.comm.size () < 5 ) {
    print ( "Inicialize o MPI com pelo menos 5 nós" )
    mpi.quit ()
}
#Envia um comando a todos os nós
mpi.remote.exec (
    cat ( "Olá! Eu sou o host de número", mpi.comm.rank (), "\n" )
)
mpi.close.Rslaves ()
```

os demais nós. Seu grafo de dependências está representado na figura 3.1, onde a marca "$*$". representa várias instruções iguais repetidas (apenas para aumentar a clareza da imagem).

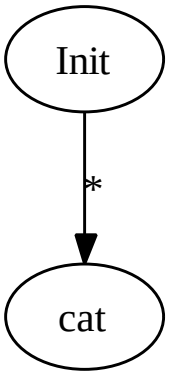

Figura 3.1: Grafo de dependências do código com Rmpi

De forma a se trabalhar com o código como um programa comum sem necessidade de intervenção humana (haja vista que o R é uma linguagem interpretada em forma de *shell*, ou seja, ele age como um interpretador de comandos), utiliza-se um arquivo de *script*. O arquivo deverá conter todas as instruções a serem utilizadas (ou referenciar as demais). Para sistemas derivados do UNIX© (Linux, etc.) basta referenciar como um *script* executável que referencia o R. Em plataforma Windows©, faz-se uso de arquivos de lote. Porém, como não é possível referenciar o R como interpretador de comandos, a chamada deverá ser explícita.

As ferramentas de suporte a orientação à objetos não foram analisadas, porém há histórico do trabalho de paralelização nesse tipo de estrutura em

Código 3.2: Pseudocódigo com `for` encadeado

```
1 Lx = imageWidth
2 Ly = imageHeight
3 for ( x in 1 : Lx )
4     for ( y in 1 : Ly )
5         aplicaFiltroNPixelAPixel ( x, y )
6         aplicaFiltroMPixelAPixel ( x, y )
7     end
8 end
```

Código 3.3: Laço `for` disfarçado em `repeat`

```
1  Lx = imageWidth
2  Ly = imageHeight
3  x = 0
4  Repeat
5      x = x + 1
6      y = 0
7      Repeat
8          y = y + 1
9          aplicaFiltroMPixelAPixel ( x, y )
10         aplicaFiltroNPixelAPixel ( x, y )
11     Until y = Ly
12 Until x = Lx
```

Mohapatra et al. (2005).

3.2 Detecção de paralelismo

Para verificar a possibilidade de gerar código paralelo a partir dos algoritmos seqüenciais de filtros de imagens, foram analisados os códigos destes filtros disponíveis em R. Nesta análise, foi observado que com grande freqüência ocorrem laços de repetição `for` encadeados com variáveis de controle alteradas no código (observe-se o código 3.2).

Para mapear tal caso de forma a explicitar a possibilidade de paralelização, optou-se nesse trabalho por criar a convenção da figura 3.2. Note-se a marca "$*$" no arco direcional dentre os laços `for` e os filtros. Elas indicam que as etapas podem ocorrer simultaneamente. Vale ressaltar que um laço deste tipo pode estar disfarçado em estruturas `repeat` ou `while` (vide código 3.3), porém com mesmo valor semântico.

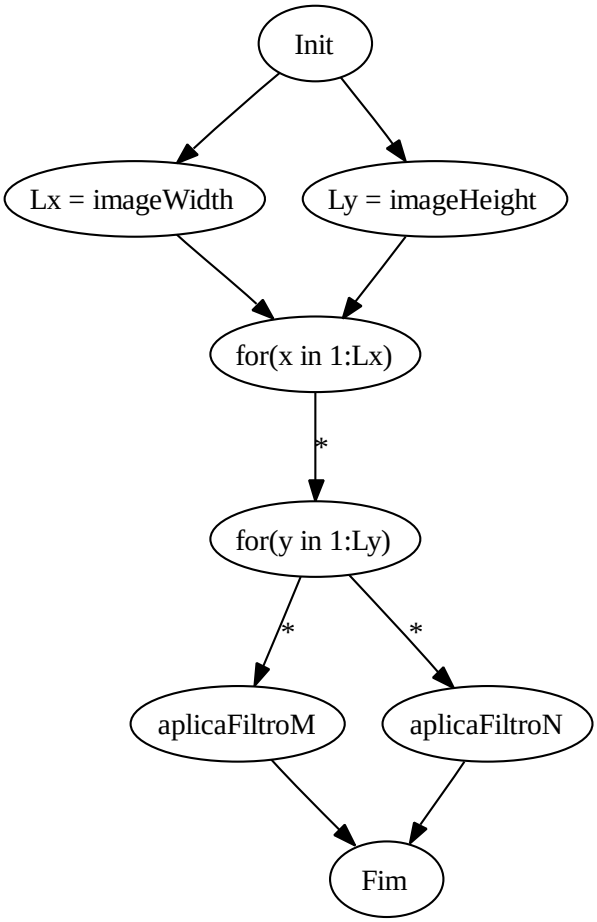

Figura 3.2: Grafo com laço `for` encadeado n vezes

3.3 Algoritmo de mapeamento

O algoritmo desenvolvido nesse trabalho para mapear um filtro de imagens implementado em `R` em um grafo de dependências consiste nos seguintes passos:

1. Iniciar com o vértice `Init`;
2. interpretar cada comando como um vértice;
3. interpretar cada dependência de controle como uma seta contínua;
4. ao se deparar com um laço `repeat`, caso a condição de saída seja um contador, convertê-lo em um laço `for`;
5. ao se deparar com um laço de repetição `while`, caso a condição de saída seja um contador, convertê-lo em um laço `for`;
6. ao se deparar com um laço de repetição `for`:

- caso seja possível determinar o número de repetições, gerar uma seqüência paralela para cada repetição;
- caso não seja possível determinar o número de repetições, gerar apenas uma seqüência com a marca "$*$";
- caso os comandos internos em paralelo possuam dependências de dados, ligá-los através de uma seta pontilhada;

7. chamadas a procedimentos seguem em ramos paralelos;
8. caso haja dependência de dados dentre os procedimentos, indicá-la por seta pontilhada;
9. demais casos seguir seqüencialmente.

De acordo com pressupostos de Barrett & Myers (2004), a convolução pode ser expressa através da transformada de Fourier (ver equação 2.2). Sabendo disso, validou-se o algoritmo de mapeamento para a forma normal necessária à transformada de Fourier, garantindo assim seu funcionamento para qualquer filtro convolucional.

3.4 Estudo de caso: filtro gaussiano

O filtro gaussiano consta como um dos mais importantes filtros convolucionais. Ele é definido por uma máscara quadrada cujos coeficientes são proporcionais a uma densidade gaussiana bivariada de média nula. Os parâmetros do filtro são o tamanho da máscara e o espalhamento dessa densidade; para filtros definidos sobre máscaras quadradas de lado K ímpar, quanto maior o espalhamento, medido por $\sigma > 0$, maior será o efeito de borramento na imagem filtrada.

Os coeficientes do filtro gaussiano são dados por

$$a_{i,j} = Z_{K,\sigma} \exp\left\{-\frac{i^2+j^2}{2\sigma^2}\right\},$$

onde $-(K-1)/2 \leq i,j \leq (K-1)/2$ e $Z_{K,\sigma}$ é a constante de padronização que garante $\sum_{i,j} a_{i,j} = 1$.

Este filtro é apropriado para combater ruído aditivo, mas introduz um certo borramento na imagem de saída. Os coeficientes são todos positivos e simétricos em relação ao centro da máscara.

A seguir mostramos as máscaras para os filtros de lados $K = 3$ (primeira linha) e $K = 5$ (segunda linha) com $\sigma = 1$ e $\sigma = 10$ (da esquerda para a direita,

Código 3.4: Função R que gera máscaras de convolução gaussianas

```
mascaraGaussiana ← function ( k, sigma ) {
    cont ← 0
    a ← matrix ( 0, k, k )
    for ( i in 1 : k )
        for (j in 1 : k ) {
            a[i,j] ← exp ( -( ( i*i+j*j )/( 2*sigma*sigma ) ) )
            cont ← cont + a[i,j]
        }
    z ← 1 / cont;
    for ( i in 1 : k )
        for ( j in 1 : k )
            a[i,j] ← exp ( -( ( i*i+j*j )/( 2*sigma*sigma ) ) ) * z
    return ( a )
}
```

Código 3.5: Função R que efetua convoluções

```
convolucionar ← function ( k, l, imagem, mascara ) {
    max ← dim ( mascara ) [1]
    g ← matrix ( 0, max, max )
    kM ← k + trunc ( max / 2 ) + 1
    lM ← l + trunc ( max / 2 ) + 1
    for ( i in 1 : max )
        for ( j in 1 : max )
            g[i, j] ← g[i,j] + mascara[i,j] * imagem[kM - i,lM - j]
    return (g[k,l])
}
```

respectivamente):

$$\begin{pmatrix} 0.075 & 0.12 & 0.075 \\ 0.124 & 0.20 & 0.124 \\ 0.075 & 0.12 & 0.075 \end{pmatrix}, \begin{pmatrix} 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 \end{pmatrix},$$

$$\begin{pmatrix} 0.0030 & 0.013 & 0.022 & 0.013 & 0.0030 \\ 0.0133 & 0.060 & 0.098 & 0.060 & 0.0133 \\ 0.0219 & 0.098 & 0.162 & 0.098 & 0.0219 \\ 0.0133 & 0.060 & 0.098 & 0.060 & 0.0133 \\ 0.0030 & 0.013 & 0.022 & 0.013 & 0.0030 \end{pmatrix}, \begin{pmatrix} 0.039 & 0.040 & 0.040 & 0.040 & 0.039 \\ 0.040 & 0.040 & 0.041 & 0.040 & 0.040 \\ 0.040 & 0.041 & 0.041 & 0.041 & 0.040 \\ 0.040 & 0.040 & 0.041 & 0.040 & 0.040 \\ 0.039 & 0.040 & 0.040 & 0.040 & 0.039 \end{pmatrix}.$$

Para introduzir o processo de detecção de paralelismo, implementa-se uma função geradora de máscaras (código 3.4) e uma convolucionadora (código 3.5).

Utilizando essas funções, define-se uma função que filtra uma imagem con-

Código 3.6: Filtro gaussiano codificado em R

```
1  filtroGaussiano ← function ( imagem, k, sigma ) {
2      mascara ← mascaraGaussiana ( k, sigma )
3      min ← ( k + 1 ) / 2
4      Xmax ← dim ( imagem ) [1] - min
5      Ymax ← dim ( imagem ) [2] - min
6      g ← imagem
7      for ( x in min : Xmax )
8          for ( y in  min : Ymax )
9              g [x,y] ← convolucionar ( min, min, imagem ( x-( k-1 )
10 /2 : x+( k-1 )/2, j-( k-1 )/2 : j+( k-1 )/2 ) , mascara )
11     return ( g )
12 }
```

volucionalmente com uma máscara gaussiana no código 3.6.

Seguindo o algoritmo proposto na seção 3.3, pode-se construir o grafo de dependências do código 3.6, resultando na figura 3.3. Neste grafo pode-se observar que as atividades que seguem em vértices não-seqüenciais podem ser executadas em paralelo.

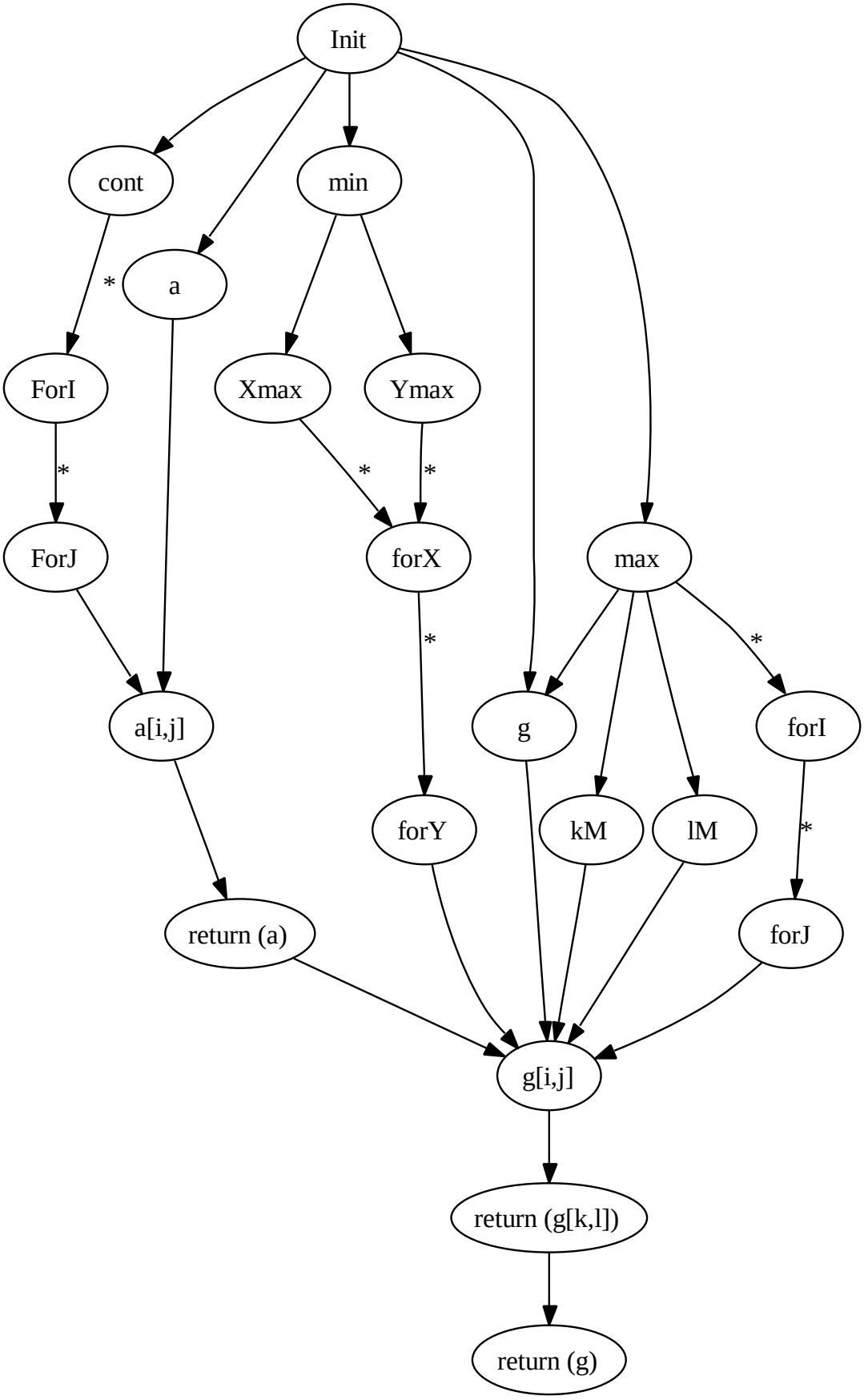

Figura 3.3: Grafo de dependências da aplicação de um filtro gaussiano

Capítulo 4

Análise de paralelização em um grafo de dependências

A partir de um grafo de dependências gerado de acordo com o algoritmo proposto no capítulo anterior (seção 3.3). Esta análise está baseada em conceitos como os propostos por Quinn (2003) e Grama et al. (2003). De forma a prover uma solução eficiente, o grafo deve ser o mais otimizado o possível. É proposta uma ferramenta de paralelização de software baseada no código R.

4.1 Grafos cíclicos

No tocante a interpretação dos grafos, há uma grande dificuldade na interpretação de um grafo cíclico (na figura 4.1, observe-se o ciclo $a \rightarrow b \rightarrow d \rightarrow a$), comum em situações de recursividade. De forma a solucioná-la, Hwang & Saltz (2003) propõem métodos automatizados para identificar os ciclos e eliminá-los sem que a semântica do programa se altere. Estes ciclos constituem um problema clássico no tratamento de grafos de dependências. Ao solucioná-lo, não se espera encontrar demasiados empecilhos à análise de paralelização.

Numa perspectiva de pré-processamento, Burstall & Darlington (1977) propõem métodos algorítmicos de conversão recursivo $\rightarrow$ seqüencial. Ele pretende remover os rótulos de recursão. Demais princípios interessantes podem ser observados em Knuth (1974).

Outra idéia consiste em utilizar os conceitos de Johnson & Pingali (1993), onde se definem grafos de dependências, explanando sobre como contruí-los. A partir daí, segue numa análise de fluxo, explicando como percorrer um grafo a entender sua lógica. Dessa forma pode-se verificar se as alterações propostas em Hwang & Saltz (2003) estão corretas, quando se desfaz o ciclo de um grafo.

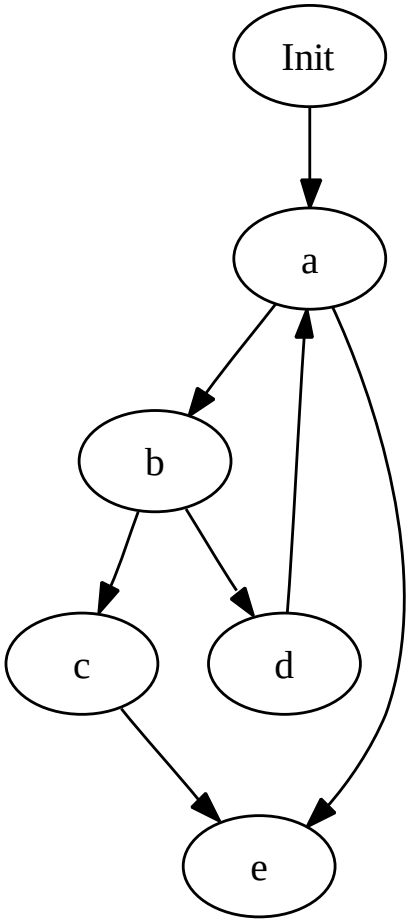

Figura 4.1: Exemplo de grafo de dependências cíclico

Código 4.1: Exemplo de instrução *forall*

```
1 for ( ind in 1 : 30 )
2     matriz[ind] ← sqrt ( ind )
```

4.2 Técnicas de análise de paralelismo

Partindo do grafo obtido com o algoritmo de conversão código seqüencial → grafo de dependências, dá-se início a análise do grafo. Apresentam-se tipos de paralelismo passíveis de identificação.

Um comando `for` que não possui iterações dependentes é denominado uma instrução *forall*. Da mesma forma, um laço `do - while` também pode representar um laço `for`, mantendo o conceito. Este representa uma situação ideal à paralelização, como pode-se ver no código 4.1.

Observe-se, no entanto, que uma situação de iterações dependentes representa um caso deveras complexo de se paralelizar (ver código 4.2). O trabalho de Lisper (1991) explicita como identificar esse caso.

Conforme pode-se observar na figura 4.2, a expressão é claramente passí-

Código 4.2: Exemplo de `for` encadeado

```
1 for ( i in 1 : 30 )
2     for ( j in 1 : 30 )
3         matriz[i,j] ← sqrt ( i * j )
```

vel de paralelização, porém gerar um código capaz de representá-la torna-se difícil.

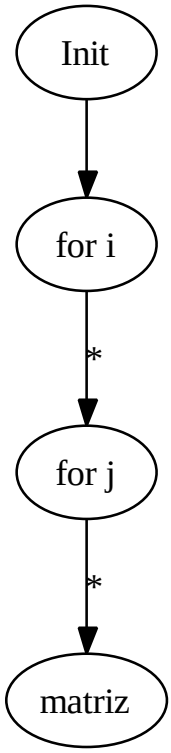

Figura 4.2: Grafo de dependências de laços `for` encadeados

Métodos descritos por Foster (1995) propõem a conversão das estruturas encadeadas em estruturas menores, de forma a transcrever repetições encadeadas em laços distintos. Adverte-se que o nível de complexidade da tarefa de paralelizar automaticamente cresce consideravelmente.

Gatu & Kontoghiorghes (2003) apresentam um conjunto de técnicas capaz de paralelizar qualquer conjunto de iterações. Dessa forma, uma análise desse trabalho pode auxiliar na construção de novas regras para melhorar o desempenho da análise de paralelização, não só gerando código mais claro como também mais eficiente.

No trabalho de Pancake (1996) são apresentadas 16 situações em forma de estudo de caso que tendem a expor casos clássicos de quando é compensatório ou não paralelizar um programa. Este apresenta as vantagens impulsionadas pelo advento da computação paralela em uníssomo às condições ideais de

execução de um programa.

4.3 Definição da granularidade de instruções

Uma tarefa que merece um pouco de atenção é a definição de um bloco de comandos a ser enviado a cada nó de processamento, ou seja, definir o tamanho do grão de instruções. A necessidade dessa avaliação é exposta no trabalho de Kruatrachue & Lewis (1988), baseado em um trabalho anterior (ver Kruatrachue, 1987) apresenta um método independente de linguagem de programação capaz de obter resultados extremamente satisfatórios. Ele aplica um agendador de tarefas otimizado capaz de escalonar as atividades. Para isso, o algoritmo proposto neste trabalho assume que há um tamanho fixo ideal à todos os blocos de instruções do programa.

McGregor & Riehl (1989) apresentam um outro método para determinação do tamanho do bloco, já baseado no de Kruatrachue & Lewis (1988). Nele, descrevem-se algoritmos capazes de determinar o tamanho do bloco de forma *ad-hoc*.

Capítulo 5

Resultados e conclusão

Após o desenvolvimento do algoritmo de conversão código seqüencial $\rightarrow$ grafo de dependências e sua respectiva análise, discorre-se agora sobre as vantagens dessa abordagem. A seguir, apresentam-se propostas de trabalhos futuros.

Este trabalho iniciou com uma análise dos filtros de imagens convolucionais, de forma a compreender que tipos de estruturas são geralmente encontrados em seus algoritmos. Após esse levantamento, analisou-se quais dessas estruturas são passíveis de paralelização e de que forma. Definiu-se o algoritmo de conversão de um filtro de imagens convolucional seqüencial a um grafo de dependências como cerne deste trabalho, a partir do qual discorreu-se sobre a análise de viabilidade da paralelização.

Através do algoritmo definido na seção 3.3, qualquer código seqüencial de um filtro de imagens convolucional pode ser facilmente transposto a um grafo de dependências. O algoritmo foi discutido como resultado parcial desse trabalho em Vieira et al. (2007). A partir do grafo, ficam explícitas as estruturas passíveis de paralelização.

O uso de um grafo de dependências para representar um filtro gaussiano apresenta redução considerável no número de operações seqüenciais, de 28 a 9 operações por pixel. Em uma imagem com, por exemplo, 1000×1000 pixels, utilizando-se uma máscara de lado $\ell = 3$, a economia em um cluster com cem processadores (1 nó *master* e 99 *slaves*), o número total de operações seqüenciais se reduz da ordem de 10^6 a 10^4. Assim sendo, o ganho no tempo de processamento é significativo e será tanto mais relevante quanto mais interativa for a aplicação, isto é, quando se trata de uma abordagem exploratória.

Com o uso da interface `Rmpi` oferecida por Yu (2006), pode-se implementar a comunicação do `R` com a interface `MPI` (ver Burns et al., 1994), capaz de gerenciar múltiplos processadores. Dessa forma o código pode ser facilmente

paralelizado na plataforma R, como pode ser observado no código 3.1.

Como forma de dar continuidade à pesquisa, propõem-se os seguintes trabalhos a serem desenvolvidos futuramente.

- Análise do ganho de desempenho obtido ao usar paralelização, levando em conta os custos de comunicação. Como bibliografia preliminar, têm-se os trabalhos de McGregor & Riehl (1989), Kruatrachue (1987), Kruatrachue & Lewis (1988), Pancake (1996) e Gatu & Kontoghiorghes (2003).

- Definição do algoritmo de conversão grafo de dependências → código R, de forma a transpassar o paradigma da computação paralela. A bibliografia inicial é constituída pelas referências utilizadas nesse trabalho.

- Automação de todo o processo de conversão código R monolítico → código R paralelo, tornando a computação paralela transparente ao usuário. Ou seja, é desejável definir um sistema que receba como entrada um filtro convolucional em código R e retorne como saída um código R paralelo. A bibliografia inicial é a mesma utilizada nesse trabalho.

- Validar o algoritmo de conversão código seqüencial → grafo de dependências (ver seção 3.3) para uma perspectiva maior que a dos filtros convolucionais. Ampliar também de forma a absorver o paradigma da orientação a objetos. Permanece como bibliografia inicial a utilizada nesse trabalho, com um destaque a trabalhos que seguem a linha de Mohapatra et al. (2005).

Referências Bibliográficas

Allen, R., Callahan, D. & Kennedy, K. (1987), Automatic decomposition of scientific programs for parallel execution, *in* 'POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, New York, NY, EUA, pp. 63–76.

Banerjee, U., Eigenmann, R., Nicolau, A. & Padua, D. A. (1993), 'Automatic program parallelization', *Proceedings of the IEEE* **81**(2), 211–243. URL `citeseer.ist.psu.edu/banerjee93automatic.html`.

Barrett, H. H. & Myers, K. J. (2004), *Foundations of Image Science*, Pure and Applied Optics, Wiley-Interscience, NJ.

Bergmark, D. & Pancake, C. M. (1990), 'Do parallel languages respond to the needs of scientific programmers?', *Computer* **23**(12), 13–23.

Blume, W., Eigenmann, R., Hoeflinger, J., Padua, D., Petersen, P., Rauchwerger, L. & Tu, P. (1994), 'Automatic detection of parallelism – a grand challenge for high-performance computing', *IEEE Parallel & Distributed Technology* **2**, 37–47.

Burns, G., Daoud, R. & Vaigl, J. (1994), Lam: An open cluster environment for `MPI`, *in* 'Proceedings of Supercomputing Symposium', pp. 379–386. URL `http://www.lam-mpi.org/download/files/lam-papers.tar.gz`.

Burstall, R. M. & Darlington, J. (1977), 'A transformation system for developing recursive programs', *Journal of the ACM* **24**(1), 44–67.

Eigenmann, R., Hoeflinger, J. & Padua, D. (1998), 'On the automatic parallelization of the perfect benchmarks', *IEEE Transactions on Parallel and Distributed Systems* **9**, 5–23.

Ferrante, J. & Ottenstein, K. J. (1987), 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems* **9**, 319–349.

Foster, I. (1995), *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, EUA.

Gansner, E., Koutsofios, E. & North, S. (2002), *Drawing graphs with dot*.

Gatu, C. & Kontoghiorghes, E. J. (2003), 'Parallel algorithms for computing all possible subset regression models using the QR decomposition', *Parallel Computing* **29**(4), 505–521.

Golub, G. H. & Loan, C. F. V. (1996), *Matrix computations*, 3 ed., Johns Hopkins University Press, Baltimore, MD, EUA.

Grama, A., Gupta, A., Karypis, G. & Kumar, V. (2003), *Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2 ed., Addison Wesley.

Hwang, Y.-S. & Saltz, J. H. (2003), 'Identifying parallelism in programs with cyclic graphs', *Journal of Parallel and Distributed Computing* **63**, 337–355.

IEEE (2006), 'The world's leading professional association for the advancement of technology'. URL `http://www.ieee.org`, última consulta em janeiro de 2007.

Johnson, R. & Pingali, K. (1993), Dependence-based program analysis, *in* 'PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation', ACM Press, New York, NY, EUA, pp. 78–89.

Knuth, D. E. (1974), 'Structured programming with go to statements', *ACM Computing Surveys* **6**(4), 261–301.

Kruatrachue, B. (1987), Static task scheduling and grain packing in parallel processing systems, Tese de doutorado.

Kruatrachue, B. & Lewis, T. (1988), 'Grain size determination for parallel processing', *IEEE Software* **5**(1), 23–32.

Lisper, B. (1991), Detecting static algorithms by partial evaluation, *in* 'PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation', ACM Press, New York, NY, EUA, pp. 31–42.

Lätt, J. & Chopard, B. (2004), 'VLADYMIR: a `C++` matrix library for data-parallel applications', *Future Generation Computer Systems* **20**(6), 1023–1039.

Martins, C. B. (2000), Detecção de paralelismo a partir de semântica denotacional e de grafos de dependência, Dissertação (mestrado em informática), Departamento de Informática Pontifícia Universidade Católica do Rio de Janeiro.

McGregor, J. D. & Riehl, A. M. (1989), 'Automatic determination of grain size for efficient parallel processing', *Communications of the ACM* **32**, 1073–1078.

Mohapatra, D. P., Mall, R. & Kumar, R. (2005), 'Computing dynamic slices of concurrent object-oriented programs', *Information and Software Technology* **47**, 805–817.

Pancake, C. M. (1996), 'Is parallelism for you?', *IEEE Computational Science & Engineering* **3**(2), 18–37.

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

Richards, J. A. & Jia, X. (1999), *Remote Sensing Digital Image Analysis*, Springer, Berlin.

Tanembaum, A. S. (2001), *Modern Operating Systems*, 2 ed., Prentice Hall.

Vieira, B. L., de Almeida, E. S. & Frery, A. C. (2007), Detecção de paralelismo para filtros convolucionais, *in* 'Anais do XIII Simpósio Brasileiro de Sensoriamento Remoto'. No prelo.

Yu, H. (2006), *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*. URL http://www.stats.uwo.ca/faculty/yu/Rmpi, R package version 0.5-2.

Este trabalho foi preparado em LaTeX utilizando uma modificação do estilo IC-UFAL. As referências bibliográficas foram digitadas no JabRef e administradas pelo BibTeX com uma modificação do estilo 'agsm'.
O texto utiliza fonte Bookman e as equações a família tipográfica Euler em corpo de 12 pontos. O documento foi impresso em papel branco de gramatura 90 utilizando uma impressora laser.