Análise Comparativa do Uso de Multi-Thread e OpenMP Aplicados a Operações de Convolução de Imagem

Dulcinéia Oliveira da Penha {dulcineia@pucmg.br}
João Batista Torres Corrêa {joaobtc@pucmg.br}
Carlos Augusto Paiva S. Martins {capsm@pucminas.br}

Laboratório de Sistemas Digitais e Computacionais
Instituto de Informática / Programa de Pós-Graduação em Engenharia Elétrica
Pontifícia Universidade Católica de Minas Gerais
Av. Dom José Gaspar 500, 30535-610
Belo Horizonte, MG, Brasil
Telefone / Fax: 55-31-33194305

Resumo

Este trabalho apresenta uma análise comparativa de diferentes métodos de implementações paralelas para sistemas de memória compartilhada, aplicados na operação de convolução de imagem. São analisadas implementações paralelas de convolução, usando os padrões WinThread e OpenMp, do ponto de vista de programabilidade e de desempenho. Este último é analisado em termos de tempo de resposta. Ao longo do desenvolvimento do trabalho, a implementação com o uso de OpenMp mostrou-se mais simples em relação à implementação com WinThread. Isto porque a segunda apresenta para o programador métodos de programação paralela mais explícitos do que a primeira. Em termos de desempenho, as implementações paralelas apresentaram um significativo ganho de desempenho em relação às implementações seqüenciais.

1. Introdução

Atualmente, muitas aplicações, tanto da área científica quanto da área comercial e industrial, exigem respostas em intervalos de tempo cada vez menores, ou até em tempo real. Desta forma, o processamento seqüencial em máquinas monoprocessadas, muitas vezes, não tem sido capaz de atender a crescente demanda por computação de alto desempenho [1] [2].

Estas aplicações aumentam a demanda por recursos computacionais necessários para armazenamento, transmissão e processamento de informações. Para resolver este problema, a solução é a utilização de sistemas computacionais de alto desempenho. Estes sistemas podem ser obtidos através de software e processadores de propósito geral (GPP - *general purpose processor*), software e circuitos de aplicação específica (ASIC, *application specific integrated circuit*), hardware dedicado e fixo (DSP - *digital signal processor*), e hardware reconfigurável [3] [4]. Processamento seqüencial e processamento paralelo [5] [6] ou processamento centralizado e processamento distribuído [7] [8], podem ser aplicados em todas estas soluções.

Sistemas paralelos são uma coleção de elementos de processamento que comunicam e cooperam entre si para resolver problemas com alta performance [5]. O paralelismo pode ser de duas formas: ILP (*instruction level parallelism*) ou PLP (*processor level parallelism*). O paralelismo ILP é aquele onde o paralelismo é obtido no nível das instruções, ou seja, mais de uma instrução é executada por vez. No paralelismo PLP, mais de um processador é utilizado e as tarefas são divididas entre eles. O uso de paralelismo é recomendado em aplicações cujas sub-tarefas são relativamente independentes.

Operações de processamento digital de imagem (PDI) são um exemplo significativo de aplicações que demandam uma grande quantidade de recursos computacionais para serem executadas. Uma das razões disto é que imagens digitais são compostas por grande quantidade de dados, normalmente armazenados na forma de matrizes. E, as operações realizadas sobre matrizes normalmente demandam um grande custo computacional [9]. Além disso, normalmente em operações de PDI, as mesmas operações matemáticas são realizadas sobre todos os pixels da imagem (elementos que compõem a matriz de representação da imagem), apresentando um paralelismo inerente à aplicação. Operações de PDI são usadas em muitas aplicações como visão computacional, áreas médica e meteorológica, e podem ser operações de realce, restauração, adição, multiplicação, filtragem, entre outras.

Atualmente, a utilização de hardwares específicos, como ASIC e DSP, vem apresentando um significativo ganho de desempenho para este tipo de aplicação. Por outro lado, operações digitais sobre imagens são naturalmente paralelas, podendo executar mais de uma operação ao mesmo tempo. Desta forma, na maioria das vezes, o uso de arquiteturas paralelas de propósito-geral utilizando memória compartilhada [6] vem apresentando resultados satisfatórios em termos de ganho de desempenho em PDI.

O objetivo deste trabalho é analisar alguns mecanismos de suporte a paralelismo multiprocessador, usando programação com variáveis compartilhadas. Na análise, serão considerados: o ganho de desempenho em relação ao tempo de resposta, os métodos de programação, as facilidades e a transparência que estes mecanismos oferecem ao programador. Os mecanismos analisados são: programação multi-thread usando o padrão WinThread (padrão multi-thread para plataforma Windows), e programação com o uso da API (*Application Program Interface*) OpenMp [10].

A operação de PDI utilizada foi a convolução de imagem. Esta operação foi escolhida porque é uma das mais importantes operações na área de processamento digital de imagens. Algumas de suas aplicações típicas são: detecção de borda, reconhecimento de padrões, borramento de imagem, dentre outras [11] [12]. A máscara de convolução aplicada na imagem de entrada gerando uma imagem de saída filtrada caracteriza cada uma destas operações. As máscaras podem variar em relação ao valor de seus coeficientes e em relação ao seu tamanho. Dependendo dos coeficientes da máscara de entrada aplicada, resultará em uma imagem borrada, sem ruído ou outra imagem de saída.

2. Operação de convolução

A operação de filtragem no domínio do espaço é chamada convolução. O termo domínio do espaço refere-se à agregação de pixels que compõem uma imagem, e operações no domínio do espaço são procedimentos aplicados diretamente sobre esses pixels [11].

O filtro usado neste trabalho foi um passa-alta. A máscara de convolução que caracteriza um filtro passa-alta é constituída por coeficientes positivos no centro ou próximos a ele, e coeficientes negativos na sua periferia [11]. A operação de filtragem passa-alta produz um efeito de distinção nas bordas da imagem original. Isto acontece porque a aplicação da máscara passa-alta em regiões constantes ou com pequenas variações de tons de cinza, faz com que a saída seja 0 (zero) ou próximo disto [11]. Este resultado reduz significativamente o contraste global da imagem.

A operação de convolução é descrita pela Equação 1. Em uma imagem I de dimensão NxN e com uma máscara de dimensão KxK, a convolução é feita para cada pixel P[linha][coluna]. A máscara de convolução é aplicada a cada pixel da imagem de entrada, resultando em uma imagem de saída convoluída (filtrada).

$$P[x][y] = \sum_{u=-K/2}^{K/2} \sum_{v=-K/2}^{K/2} I[x+u][y+v] \times M[u][v]$$

Equação 1. Equação de convolução

Neste trabalho foi utilizado um filtro espacial de alto-reforço para realizar os testes e comparações. O filtro foi calculado utilizando-se a Equação 2 [11]. O valor w no centro da máscara usada é calculado por:

$$w = n \times A - 1$$

Equação 2. Valor central da máscara de convolução

onde, 'n' é o número de elementos da máscara. O valor de 'A' aplicado a todas as máscaras foi 1, constituindo a operação de filtragem passa-alta básica. Os outros coeficientes da máscara são –1. Para A=1 na Equação 2, a máscara 3x3 é representada na Figura 1.

$$\frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 1. Máscara 3x3 de convolução passa-alta

A Figura 3 apresenta o resultado da convolução da imagem original da Figura 2. Além disto, apresenta também a imagem negativa para questões de validação. A borda da imagem negativa não faz parte da imagem e foi acrescentada apenas para mostrar a sua dimensão.

Figura 2. Imagem Original

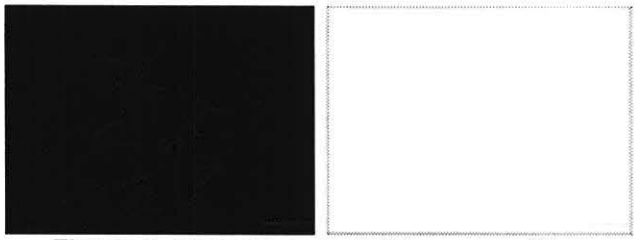

Figura 3. Imagem convoluída por um filtro passa-alta baseado na Equação 1, com A=1, e sua imagem negativa (com borda)

3. Programação paralela com variáveis compartilhadas

Em arquiteturas paralelas de memória compartilhada, variáveis compartilhadas podem ser utilizadas para comunicação entre os processos da aplicação. A utilização efetiva de sistemas paralelos é uma tarefa difícil, pois envolve o projeto de aplicações paralelas corretas e eficientes. Isto engloba vários problemas complexos como sincronização de processos, coerência de dados e ordenamento de eventos. O ideal para a utilização de sistemas paralelos é retirar do usuário (programador) a tarefa de controlar a execução paralela dos processos. Existem algumas formas de utilização do paralelismo que fornecem um certo nível de transparência ao programador.

Uma das formas de se fornecer essa transparência ao usuário é através do uso de sistemas operacionais multiprocessados aplicando paralelismo SMP (*Symmetric Multiprocessors*) [6]. Do ponto de vista de softwares, muitas técnicas são conhecidas e utilizadas para programação de multiprocessadores [13]. Para comunicação, um processador escreve dados na memória, para serem lidos por todos os outros processadores. Para sincronização, podem ser usadas sessões críticas, implementando semáforos ou monitores para prover a exclusão mútua [14] necessária.

Os sistemas operacionais Unix [15] [16], Windows NT, Windows 2000 [17], entre outros, suportam sistemas multiprocessadores. Nestes sistemas, a execução paralela é ativada pela geração de múltiplas threads que são processadas paralelamente. O número de threads, em princípio, é independente do número de processadores [18].

3.1. Programação multi-thread

Diz-se que threads são processos "leves". Na verdade, da mesma forma que processos são partes de um programa, threads são partes de um processo, ou seja, um conjunto de instruções dentro de um processo. Por serem "leves", elas são (relativamente) baratas, em termos de custo de CPU, para serem criadas e destruídas.

Todas as threads criadas por um processo compartilham o mesmo espaço de memória. A comunicação entre as threads acontece através de variáveis compartilhadas. Programas podem ter variáveis compartilhadas entre as threads (ou entre os processos) e variáveis privadas para cada thread (por exemplo, os contadores do laço de repetição "*for*").

As várias threads de um programa podem ser executadas paralelamente (se não forem dependentes). O uso de threads traz, para os programadores, a facilidade de escrever aplicações concorrentes, que rodam em máquinas monoprocessadas e multiprocessadas transparentemente, tendo a vantagem do processador adicional quando este existe.

A grande dificuldade encontrada para o desenvolvimento de aplicações usando threads é o gerenciamento dos recursos do sistema através destas. O programador precisa se preocupar com vários detalhes, desde a criação das threads e seu disparo, até o gerenciamento da sincronização e ordenamento.

Existem dois padrões de programação multi-thread, um para plataformas Unix, o padrão pthread, e outro para plataformas Windows (WinThread). O padrão pthread apresenta um grau desejável de padronização para diferentes compiladores. Já a programação com o uso de WinThread varia significativamente de acordo com o compilador. Desta forma, além da desvantagem da não portabilidade entre diferentes sistemas operacionais, ainda existe a falta de padronização entre diferentes compiladores.

3.2. API OpenMp

A API OpenMP é usada para programação multi-thread em ambientes de memória-compartilhada para plataformas UNIX e Microsoft Windows NT. Aplicações que usam OpenMp são portáveis pois são especificadas para C/C++ e Fortran. Esta API provê diretivas de compilador e rotinas de bibliotecas e variáveis, embutidas no código-fonte C/C++ e Fortran, para escopo de dados, especificação e compartilhamento da carga de trabalho e criação e sincronização de threads. Também oferece chamadas de funções para setar e obter informações sobre as threads. Algumas variáveis ajudam a controlar o comportamento do programa paralelo em tempo de execução.

Entre as vantagens desta API, pode-se destacar a padronização e a portabilidade. A primeira é possível porque OpenMp é definida por um grupo formado pelos principais vendedores de hardware e software [10]. Além disso, provê ao programador um ambiente de programação mais amigável do que lidar diretamente com threads. Isto se deve principalmente à geração automática do código de gerenciamento e disparo das threads pelo compilador, a partir da especificação de regiões paralelas.

Esta especificação é feita pelo programador através do uso de diretivas OpenMp.

Porém, OpenMp não garante fazer o uso mais eficiente de sistemas de memória compartilhada. Um dos motivos disto é que, atualmente, não há nenhuma construção que trate da localidade dos dados.

Todos os programas OpenMP começam com um único processo: a thread mestre. Quando se define uma região paralela, a thread mestre vai criando outras threads, através de chamadas de sistema. Estas threads então, são executadas em paralelo (ou de forma concorrente, no caso de máquinas monoprocessadas). Ao final da região paralela especificada, todas as threads são sincronizadas e finalizadas, deixando somente a thread mestre. No caso de OpenMp para Fortran, esta finalização é explícita através da diretiva *Join*, enquanto que, para C/C++ a finalização é implícita.

Assim como na programação com threads, a utilização de OpenMP também permite que o mesmo programa possa ser executado em paralelo ou seqüencialmente. Além disso, paralelismo aninhado pode ser usado.

A API OpenMp foi criada no início dos anos 90, a partir da idéia de embutir em Fortran e C/C++ diretivas que estendessem essas linguagens para execuções paralelas em máquinas de memória-compartilhada. Isto porque os usuários destas máquinas gostariam de que fosse possível especificar através de diretivas, em programas seriais, quais laços de repetição poderiam ser paralelizados. Desta forma, o compilador seria responsável por paralelizar automaticamente os loops para os processadores do sistema SMP.

Apesar da padronização, todas as implementações de OpenMp são similares funcionalmente, mas algumas podem divergir.

4. Implementação da convolução

A convolução de imagem é uma operação de processamento de imagem no domínio do espaço. Para cada pixel da imagem de entrada, teremos uma operação realizada entre este pixel, seus vizinhos e os coeficientes da máscara de convolução, a fim de produzir um resultado que será o pixel convoluído da imagem de saída. Esta operação é descrita pela Equação 1. Para descartar efeitos de borda, os pixels das bordas da imagem de entrada foram desconsiderados.

O algoritmo seqüencial que realiza a operação de convolução [19] foi implementado como mostra a Figura 4. Ele é composto por um laço de repetição quádruplo. Esta implementação foi feita para o Sistema Operacional Windows 2000 utilizando o compilador Borland C++ Builder 5.0. O programa que implementa o algoritmo seqüencial foi chamado de KMT-IPS, descrito em [19]. Uma outra implementação seqüencial foi feita para o Sistema Operacional Conectiva Linux 7 utilizando o compilador GNU gcc.

Para ambas as implementações seqüenciais, os dois laços de repetição mais externos percorrem os pixels da imagem de entrada e os dois laços internos percorrem os pixels da máscara de convolução. Cada pixel da imagem de saída é calculado da seguinte forma: os pixels da imagem de entrada são multiplicados pelos seus respectivos coeficientes da máscara de convolução e então são somados. Ao final deste somatório, um fator de divisão é aplicado, resultando no pixel convoluído, que constituirá a imagem final convoluída.

```
Four-loop
for (input image rows)
  for (input image collums)
    for (convolution mask rows)
      for (convolution mask collums)
        ... ...
```

Figura 4. Algoritmo de convolução básico com laço quádruplo

Esta implementação é bastante simples. Existem duas matrizes, uma que armazena a imagem de entrada e outra que armazena o filtro (máscara de convolução). Existe ainda uma terceira matriz, temporária, onde os pixels da imagem de saída vão sendo armazenados à medida que são calculados pela convolução. Ao término da operação de convolução sobre todos os pixels da imagem de entrada, a matriz temporária, que contém os pixels convoluídos, é copiada para esta matriz original substituindo a imagem de entrada.

5. Implementações paralelas

Atualmente, máquinas paralelas estão se tornando cada vez mais acessíveis às pessoas e o uso de paralelismo em muitas aplicações vem se tornando comum. A convolução é uma operação naturalmente paralela, de forma que, a operação para convoluir um pixel não afeta a convolução dos outros pixels. Assim, cada operação para calcular um pixel da imagem de saída (imagem convoluída) pode ser executada em paralelo. Usando o modelo de variável compartilhada, pode-se compartilhar a imagem de entrada e a máscara de convolução entre os elementos de processamento (EPs). A tarefa de convoluir a imagem é dividida igualmente entre os EPs, através da distribuição das linhas da imagem de entrada a ser convoluída. Então, cada EP é responsável por calcular as linhas da imagem de saída que lhe foram atribuídas, usando a imagem de entrada.

Foram realizadas duas implementações da operação de convolução em paralelo. A primeira utilizando explicitamente programação multi-thread, através do padrão WinThread do compilador Borland C++ Builder 5.0 [19], para o Windows. A segunda implementação foi

realizada utilizando diretivas OpenMp para o Linux, que significa utilizar programação multi-thread implicitamente.

5.1. Implementação com WinThread

Como citado anteriormente, a implementação usando multi-thread foi feita para o Windows utilizando o compilador Borland C++ Builder 5.0, assim como na implementação da versão seqüencial para plataformas Windows.

Para implementar a convolução em paralelo utilizando WinThread, foi necessário analisar a operação e a parte do código a ser paralelizada, considerando o modelo de programação com variáveis compartilhadas. As variáveis compartilhadas foram a matriz de armazenamento da imagem de entrada, a matriz temporária e o filtro de convolução. As threads são criadas de forma que cada uma recebe como parâmetro a linha inicial da parte da imagem de saída que lhe cabe calcular. Com este parâmetro, cada thread calcula a linha final da sua sub-imagem de saída. A parte da imagem compreendida entre a linha inicial e a linha final atribuída a uma thread, será convoluída por esta.

Uma das principais dificuldades encontradas para paralelizar a operação de convolução utilizando WinThread foi o fato de ter que conhecer o modelo de variáveis compartilhadas e analisar as partes da operação a serem paralelizadas, para desenvolver corretamente a aplicação. Além disso, existe a desvantagem de o programador ter que se preocupar explicitamente com: criação e gerenciamento de variáveis, como por exemplo o vetor de threads, criação das threads, inicialização das mesmas, e controle de acessos às partes críticas do programa (seções críticas). Para o controle de seções críticas, existe a necessidade da utilização de semáforos ou funções providas por bibliotecas próprias.

5.2. Implementação com OpenMp

A implementação da operação de convolução com OpenMp foi feita para o Sistema Operacional Linux, utilizando o pré-compilador OdinMp, versão 1.02 [23].

```
#pragma omp parallel\
    private (tid, numethreads)
```

Figura 5. Especificação da região paralela na implementação da operação de convolução utilizando OpenMp

Para esta implementação, foram utilizadas as diretivas básicas da API. O laço for mais externo da operação, descrito na Seção 4, foi inserido em uma região paralela, como apresentado na Figura 5. Este laço, que percorre a imagem de entrada linha a linha, foi paralelizado através do uso da diretiva *"omp for"* como mostrado na Figura 6. Desta forma, durante a execução do programa, as linhas da imagem de entrada serão divididas entre as threads criadas.

```
#pragma omp for\
    shared  (Temp, Imagem)\
    private (linha_img, coluna_img,\
        linha_filtro, coluna_filtro)
```

Figura 6. Paralelização do laço for da operação de convolução

O código de criação e gerenciamento das threads e divisão das linhas da imagem de entrada entre estas foi gerado automaticamente quando o programa contendo as diretivas foi pré-compilado com o OdinMp.

O número de threads disparadas durante a execução é setado através da função apresentada na Figura 7. O número de threads utilizadas foi quatro. Com este número, espera-se melhor aproveitamento do tempo de processamento, evitando excessivas trocas de contexto.

```
omp_set_num_threads (int numthreads)
```

Figura 7. Função OpenMp para setar número de threads disparadas durante a execução da região paralela

6. Testes e resultados

O computador utilizado para executar todos os testes foi um Intel Dual Pentium III 933MHz com 1024MB de memória primária. Para cada teste, primeiramente a imagem de entrada foi carregada para posterior execução do mesmo.

Para a obtenção dos tempos médios de resposta, cada implementação proposta de convolução foi executada quatro vezes, sendo que o primeiro tempo de cada uma das execuções foi descartado. Fazendo isto, considera-se que foi removido o tempo gasto com as tarefas inerentes ao Sistema Operacional, que poderiam influenciar na análise do tempo de resposta da aplicação.

Os testes de tempo de resposta foram realizados utilizando três diferentes tamanhos de imagem (640x480, 1152x864 e 2048x2048 pixels), com aplicação de três tamanhos de máscara de convolução 2D (3x3, 5x5 e 7x7).

Todas as implementações apresentadas neste trabalho, tanto para o Sistema Operacional Linux, quanto para o Windows, foram realizadas utilizando a linguagem C/C++ e os compiladores apropriados, já citados nos tópicos que descrevem cada uma das implementações.

Para a realização dos testes para plataforma Windows foi utilizado o software KMT-IPS. Deste, foram utilizadas

a implementação seqüencial da operação de convolução e a implementação paralela multi-thread da mesma, com padrão WinThread. Além disto, a operação de convolução foi implementada seqüencialmente para Linux e posteriormente usando diretivas OpenMp.

As implementações propostas neste trabalho realizaram corretamente a operação de convolução. Os resultados alcançados em todas elas estão dentro dos padrões apresentados na Seção 2 (A operação de convolução), através da Figura 2 e da Figura 3.

Conforme foi discutido nas seções anteriores, as implementações seqüenciais da operação de convolução, tanto para Windows quanto para Linux, são bastante simples. Já as implementações paralelas exigem mais do programador, que precisa se preocupar com cada detalhe da programação e da execução da aplicação. A programação com o uso de threads é bastante complicada. Por outro lado, implementações paralelas com o uso de OpenMp apresentam-se significativamente mais fáceis do que com o uso explícito de threads. O uso das diretivas de compilação e das funções de biblioteca disponibilizadas por esta API é relativamente simples. Utilizando diretivas OpenMp, o programador não precisa se preocupar com os detalhes relativos à criação, sincronização, gerenciamento, finalização das threads, entre outros. Isto porque o pré-compilador do OpenMp gera automaticamente o código para o tratamento e gerenciamento destes detalhes.

Tabela 1. Tempos de resposta das implementações para o Linux

Tipo	Dimensão Imagem	Máscara	Tempo Resposta
seqüencial	640x480	3x3	333,33 ms
paralelo	640x480	3x3	126,67 ms
seqüencial	1152x864	3x3	1090,00 ms
paralelo	1152x864	3x3	416,67 ms
seqüencial	2048 x 2048	3x3	5113,33 ms
paralelo	2048 x 2048	3x3	1733,33 ms
seqüencial	640x480	5x5	753,33 ms
paralelo	640x480	5x5	236,67 ms
seqüencial	1152x864	5x5	2440,00 ms
paralelo	1152x864	5x5	750,00 ms
seqüencial	2048 x 2048	5x5	10880,00 ms
paralelo	2048 x 2048	5x5	3286,67 ms
seqüencial	640x480	7x7	1320,00 ms
paralelo	640x480	7x7	370,00 ms
seqüencial	1152x864	7x7	4350,00 ms
paralelo	1152x864	7x7	1210,00 ms
seqüencial	2048 x 2048	7x7	18840,00 ms
paralelo	2048 x 2048	7x7	5333,33 ms

A Tabela 1 apresenta os tempos de resposta das implementações da convolução para o Linux, seqüencial e paralela (OpenMp). A Tabela 2 apresenta os mesmos tempos para as implementações para o Windows, seqüencial e paralela (WinThread). Analisando estas tabelas, percebe-se um significativo ganho de desempenho para as duas implementações paralelas em relação às seqüências, tanto para o Linux quanto para o Windows. Para uma imagem de 2048x2048, aplicando uma máscara de convolução de tamanho 7x7, o speedup alcançado com a implementação paralela usando OpenMp (em relação à implementação seqüencial para Linux) chegou a 3,59. Conclui-se, portanto, que o uso de paralelismo melhora significativamente a performance deste tipo de aplicação, independente do padrão de programação paralela utilizado.

Tabela 2. Tempos de resposta das implementações para Windows

Tipo	Dimensão Imagem	Máscara	Tempo Resposta
seqüencial	640x480	3x3	281,33 ms
paralelo	640x480	3x3	187,00 ms
seqüencial	1152x864	3x3	901,00 ms
paralelo	1152x864	3x3	583,00 ms
seqüencial	2048 x 2048	3x3	3734,33 ms
paralelo	2048 x 2048	3x3	2427,00 ms
seqüencial	640x480	5x5	422,00 ms
paralelo	640x480	5x5	250,00 ms
seqüencial	1152x864	5x5	1359,67 ms
paralelo	1152x864	5x5	801,67 ms
seqüencial	2048 x 2048	5x5	5739,67 ms
paralelo	2048 x 2048	5x5	3391,00 ms
seqüencial	640x480	7x7	588,33 ms
paralelo	640x480	7x7	333,67 ms
seqüencial	1152x864	7x7	1927,33 ms
paralelo	1152x864	7x7	1083,67 ms
seqüencial	2048 x 2048	7x7	8219,00 ms
paralelo	2048 x 2048	7x7	4594,00 ms

Tabela 3. Speedup das implementações paralelas em relação às seqüenciais

		Speed-up	
Dimensão Imagem	Máscara	Windows	Linux
640x480	3x3	1,50	2,63
	5x5	1,55	3,18
	7x7	1,54	3,57
1152x864	3x3	1,69	2,62
	5x5	1,70	3,25
	7x7	1,69	3,60
2048 x 2048	3x3	1,76	2,95
	5x5	1,78	3,31
	7x7	1,79	3,53

A Tabela 3 apresenta os valores do speedup alcançado pelas implementações paralelas em relação às seqüenciais, para Linux e para Windows. Através da análise desta tabela, percebe-se que o speedup alcançado pela versão paralela usando OpenMp (em relação à versão seqüencial para Linux), é maior do que o speedup

da implementação paralela usando WinThread (em relação à seqüencial para Windows). Este resultado pode ter sido influenciado pelo fato de que foram utilizados, neste trabalho, compiladores e sistemas operacionais diferentes.

Um outro fator importante, que também pode ser notado na Tabela 3, é que quanto maior o tamanho da máscara de convolução aplicada à imagem, maior é o speedup alcançado pela implementação paralela em relação à seqüencial, tanto para o Linux (OpenMp) quanto para o Windows (WinThread). Isto pode ser explicado porque o número de operações realizadas aumenta de acordo com o tamanho da máscara. Desta forma, o uso de paralelismo pode melhorar ainda mais o speedup para máscaras de convolução maiores.

7. Conclusões

O objetivo proposto neste trabalho foi alcançado, uma vez que os mecanismos escolhidos foram comparados através da sua utilização na implementação de uma importante aplicação da área de processamento digital de imagem. Para a análise comparativa dos mecanismos, foram considerados: o ganho de desempenho em relação ao tempo de resposta, os métodos de programação, a programabilidade e a transparência que estes mecanismos oferecem ao programador.

Conforme foi discutido na apresentação dos resultados, a implementação paralela da operação de convolução traz um resultado satisfatório em termos de ganho de desempenho. No que diz respeito à transparência desejada do paralelismo para o programador, a implementação com o uso de OpenMp mostrou-se mais adequada porque é mais fácil de ser entendida e utilizada. Porém, nem sempre o resultado alcançado em relação ao tempo de resposta foi melhor para a aplicação paralela com o uso desta API.

Entre as contribuições deste trabalho estão, a implementação paralela da convolução de imagem usando a API OpenMp e a análise comparativa do uso de multi-thread e OpenMp aplicados a operações de convolução de imagem.

Algumas questões inerentes aos diferentes sistemas operacionais e compiladores escolhidos não foram consideradas. Desta forma, não são discutidas as diferenças de tempo de execução para as implementações seqüenciais no Linux e no Windows, pois estão fora do escopo deste trabalho.

8. Trabalhos futuros

Um trabalho futuro já iniciado é a implementação da operação de convolução de imagem usando explicitamente o padrão multi-thread para Unix, pthread. Da mesma forma, o uso da API OpenMp para a implementação da convolução para Windows. E posteriores estudos e comparações de desempenho para ambas as implementações e ambientes.

Outros trabalhos futuros que podem ser citados são: realização de outras implementações usando OpenMp e testes combinando o uso desta API com outros métodos de programação paralela, como passagem de mensagem, para ambientes paralelos heterogêneos compostos por multiprocessadores e multicomputadores.

8. Agradecimentos

Agradecemos à ProPPG (Pró-reitoria de Pesquisa e de Pós-Graduação da PUC-Minas) pelas bolsas Probic P-2001/113 e P-2002/84 e ao LSDC (Laboratório de Sistemas Digitais e Computacionais) [20] da PUC-Minas pelo suporte e infra-estrutura fornecidos para os testes e desenvolvimento dos projetos de pesquisa [21] [22] dos autores.

9. Referências

[1] SBAC-PAD 2001. Proceedings: 13º Symposium on Computer Architecture and High Performance Computing: editors Alba Cristina M. A. de Melo, Mário Antônio R. Dantas, Jairo Panetta. Brasília: Departamento de Ciência da Computação da UNB, 2001.

[2] WSCAD 2001. Proceedings: II Workshop de Sistemas Computacionais de Alto Desempenho: editores Alba Cristina M. A. de Melo, Alberto F. Souza, Mário Antônio R. Dantas. Brasília: Departamento de Ciência da Computação da UNB, 2001.

[3] CORE 2000. "Computação Reconfigurável" Editado por Edward D. Moreno Ordonez, Jorge Luiz e Silva, Fundação de Ensino Eurípedes Soares da Rocha (FEESR) 2000.

[4] SCR 2001. Anais do I Seminário de Computação Reconfigurável, Instituto de Informática, PUC Minas, 2001.

[5] G.S. Almasi and A. Gottlieb, "Highly Parallel Computing" 2.ed, Benjamin/Cummings, 1994.

[6] K. Hwang and Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming", McGraw-Hill, 1998.

[7] A. Tanenbaum, "Distributed Operating Systems", Prentice Hall, New Jersey, Upper Saddle River, 1995.

[8] S.J. Mullender, "Distributed-Operating Systems", ACM Computing Surveys, Vol. 28, No. 1, March 1996.

[9] C.A.P.S. Martins, "*Subsistema de exibição de imagens digitais com desacoplamento de resolução - SEID-DR*", Tese de Doutorado, Universidade de São Paulo, SP, 1998.

[10] URL: www.openmp.org

[11] R.C. Gonzalez and R.E. Woods, *"Processamento de Imagens Digitais"* 3.ed, Nova Iorque, Editora Edgard Blucher, 2000.

[12] N.K. Ratha, A.K. Janin, and D.T. Rover, "Convolution on Splash 2 ", Michigan State University, *IEEE,* 1995.

[13] K.K. Yue, D.J. Lilja, "An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory

Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 12, Dezembro 1997.
[14] A. Tanenbaum and A. Woodhull, "Operating Systems Design and Implementation", 2.ed., Editora Prentice Hall, Upper Saddle River, New Jersey, 1997.
[15] C. Schimmel, "UNIX Systems Architectures Symetric Multiprocessing and Caching for Kernel Programmers", Addison-Wesley Professional Computing Series, 1994.
[16] URL: http://www.linux.org
[17] URL: http://www.microsoft.com/windows/default.asp
[18] "Introduction to OpenMP", Advanced Computational Research Laboratory, Faculty of Computer Science, UNB Fredericton, New Brunswick.
[19] J. B. T. Corrêa, C. A. P. S. Martins, "*Performance Optimization on Digital Image Filtering*", International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA), 2002.
[20] Laboratório de Sistemas Digitais e Computacionais. URL http://www.lsdc.inf.pucminas.br
[21] D.O. Penha, C.A.P.S. Martins, *"Estudo e Implementação de Mecanismos de Suporte a Paralelismo em Sistemas Operacionais"*, Projeto PROBIC P-2002/84, LSDC - PUC Minas, Belo Horizonte, MG
[22] J.B.T. Corrêa, C.A.P.S. Martins, *"Estudo e análise de sistemas computacionais reconfiguráveis aplicados em processamento de imagens"*, Projeto PROBIC P-2001/113, LSDC - PUC Minas, Belo Horizonte, MG
[23] URL: http://vvv.it.kth.se/labs/cs/odinmp/