FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” - UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PRISCILA TIEMI MAEDA SAITO

OTIMIZAÇÃO DO PROCESSAMENTO DE IMAGENS MÉDICAS UTILIZANDO A COMPUTAÇÃO PARALELA

MARÍLIA
2007

PRISCILA TIEMI MAEDA SAITO

OTIMIZAÇÃO DO PROCESSAMENTO DE IMAGENS MÉDICAS UTILIZANDO A COMPUTAÇÃO PARALELA

Monografia apresentada ao curso de Bacharelado em Ciência da Computação, do Centro Universitário Eurípides de Marília UNIVEM, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadora:
Profª. Drª Kalinka Regina Lucas Jaquie Castelo Branco

MARÍLIA
2007

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Priscila Tiemi Maeda Saito

OTIMIZAÇÃO DO PROCESSAMENTO DE IMAGENS MÉDICAS UTILIZANDO A COMPUTAÇÃO PARALELA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Resultado: ________ (_________________________)

Orientador: Kalinka R. L. J. Castelo Branco _________________________

1º. Examinador: Fátima L. S. Nunes Marques _________________________

2º. Examinador: Antônio Carlos Sementille _________________________

Marília, 22 de novembro de 2007.

Dedico a realização desse Trabalho de Conclusão de Curso...
...A Deus por ter me dado forças e condições para realizar este trabalho.
...Aos meus pais e à minha família, que me proporcionaram a educação que me permitiu chegar até aqui.

AGRADECIMENTOS

"O agradecimento é a memória do coração." Tao Tse

Primeiramente a Deus por ter me dado forças e competência para a realização deste trabalho.

Aos meus pais Kiokuni Saito e Yokie Maeda Saito, minha irmã Larissa Midori Maeda Saito e minha família, por todo amor, carinho e compreensão e por sempre estarem ao meu lado, me apoiando, me ajudando e contribuindo para a minha formação pessoal e profissional.

À minha orientadora Kalinka Regina Lucas Jaquie Castelo Branco, por ter me concedido a oportunidade de realizar o presente projeto, por sua orientação e paciência e pela confiança em meu trabalho. Obrigada por sempre me incentivar mesmo nas situações adversas.

À professora Fátima de Lourdes dos Santos Nunes Marques, pelo apoio e orientação neste e em outros trabalhos e pela disposição constante em ajudar sempre para melhoria destes.

Ao Pedro Henrique Bugatti pelo companheirismo, pelas sugestões, sempre me apoiando e me incentivando a enfrentar os desafios e a superar os obstáculos que surgiram. Obrigada pelos conselhos que me ajudaram a renovar as energias e continuar o trabalho.

Ao Ricardo José Sabatine, companheiro de laboratório e projetos, pelo auxílio e contribuições.

Aos meus amigos e colegas de graduação que de alguma maneira contribuíram para o desenvolvimento desse trabalho. Em especial Fábio, Kauê, Mariana, Pedro e Vanessa, pela ajuda, incentivo e maravilhosa convivência ao longo desses anos, compartilhando momentos de estudos e trabalhos.

Aos monitores dos laboratórios de Informática, Guilherme, Jorge e Luis Carlos, pela atenção, paciência, principalmente nos inúmeros testes realizados.

À FAPESP (processo nº. 2006/06671-0) – Fundação de Amparo à Pesquisa do Estado de São Paulo pela confiança e pelo apoio concedido à pesquisa.

A todos que de forma direta e indireta contribuíram para produção deste trabalho.

Muito Obrigada!

LIÇÃO DE VIDA

Que Deus não permita que eu perca o ROMANTISMO,
mesmo eu sabendo que as rosas não falam.
Que eu não perca o OTIMISMO,
mesmo sabendo que o futuro que nos espera não é assim tão alegre.
Que eu não perca a VONTADE DE VIVER,
mesmo sabendo que a vida é, em muitos momentos, dolorosa…
Que eu não perca a vontade de TER GRANDES AMIGOS,
mesmo sabendo que, com as voltas do mundo, eles acabam indo embora de nossas vidas…
Que eu não perca a vontade de AJUDAR AS PESSOAS,
mesmo sabendo que muitas delas são incapazes de ver, reconhecer e retribuir esta ajuda.
Que eu não perca o EQUILÍBRIO,
mesmo sabendo que inúmeras forças querem que eu caia.
Que eu não perca a VONTADE DE AMAR,
mesmo sabendo que a pessoa que eu mais amo, pode não sentir o mesmo sentimento por mim…
Que eu não perca a LUZ e o BRILHO NO OLHAR,
mesmo sabendo que muitas coisas que verei no mundo,
escurecerão meus olhos…
Que eu não perca a GARRA,
mesmo sabendo que a derrota e a perda são dois adversários extremamente perigosos…
Que eu não perca a RAZÃO,
mesmo sabendo que as tentações da vida são inúmeras e deliciosas.
Que eu não perca o SENTIMENTO DE JUSTIÇA,
mesmo sabendo que o prejudicado possa ser eu…
Que eu não perca o meu FORTE ABRAÇO,
mesmo sabendo que um dia meus braços estarão fracos…
Que eu não perca a BELEZA E A ALEGRIA DE VER,
mesmo sabendo que muitas lágrimas brotarão dos meus olhos
e escorrerão por minha alma…
Que eu não perca o AMOR POR MINHA FAMÍLIA,
mesmo sabendo que ela muitas vezes me exigiria esforços incríveis para manter a sua harmonia.
Que eu não perca a vontade de DOAR ESTE ENORME AMOR que existe em meu coração, mesmo sabendo que muitas vezes ele será submetido e até rejeitado.
Que eu não perca a vontade de SER GRANDE,
mesmo sabendo que o mundo é pequeno…
E acima de tudo…
Que eu jamais me esqueça que Deus me ama infinitamente,
que um pequeno grão de alegria e esperança dentro de cada um
é capaz de mudar e transformar qualquer coisa, pois….
A VIDA É CONSTRUÍDA NOS SONHOS
E CONCRETIZADA NO AMOR!

Francisco Cândido Xavier

SAITO, Priscila Tiemi Maeda. **Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela**. 2007. 160f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMO

O processamento de imagens exige um alto poder de processamento. Quando se trata de imagens médicas, esta questão é ainda mais realçada, visto que esta classe de imagens não pode permitir armazenamento com perdas de dados e, muitas vezes, exige precisão na sua aquisição. Essa precisão gera um conjunto ainda maior de dados, o que prejudica a avaliação da mesma, dada a demora em se efetuar a passagem de algum filtro, seja para suavização, atenuação ou outro. A computação paralela e distribuída, viabilizada por sistemas distribuídos e bibliotecas de passagem de mensagens, pode oferecer a potência computacional adequada a este tipo de aplicação. Nesse projeto são implementadas técnicas de processamento de imagens de forma seqüencial e paralela, utilizando a linguagem de programação Java juntamente com a API JAI (*Java Advanced Imaging*) e a biblioteca de troca de mensagens mpiJava. Após essa implementação, é construída uma base de comparação entre a aplicação seqüencial e a paralela, a fim de avaliar o ganho com o paralelismo. Pretende-se, assim, obter um estudo que defina requisitos e subsídios para a aplicação da computação paralela e distribuída no processamento de imagens médicas.

Palavras-Chave: Computação Paralela. Processamento de Imagens. Sistemas Distribuídos. Imagens Médicas. mpiJava.

SAITO, Priscila Tiemi Maeda. **Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela**. 2007. 160f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

Image processing demands a high capacity processing. In medical images, this question is still more enhanced, since this class of images can not allow storage with losses of data and, many times, still demands precision in its acquisition. This situation can generate a bigger set of data, which harms the evaluation of the same one, given the delay in effecting the passing of some filter, either for suavization, attenuation or another one. The parallel and distributed computing, made possible by distributed systems and message passing libraries, it can offer the adequate computational power to this application. In this project, images processing techniques are implemented in sequential and parallel form, using the Java programming language together with API JAI (Java Advanced Imaging) and the mpiJava message passing library. After this implementation, a comparison base is formed between the sequential and parallel aplication to evaluate the profit with the parallelism. It is intended to get a study that defines requirements and subsidies for the application of the parallel and distributed computing in the medical images processing.

Keywords: Parallel Computing. Image Processing. Distributed Systems. Medical Images. mpiJava.

SAITO, Priscila Tiemi Maeda. **Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela**. 2007. 160f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMEN

El procesamiento de imágenes requiere un gran poder de procesamiento. Cuando se trata de imágenes médicas, la cuestión es más realzada, aunque esta clase de imágenes no puede permitir almacenamiento con pierdas de datos y, muchas veces, requiere precisión en su adquisición. Esa precisión genera un conjunto aunque mayor de datos, lo que perjudica la avaluación de la misma, por demorarse en se efectuar el pasaje de filtros, sea para visualización, atenuación u otro. La computación paralela y distribuida, a través de sistemas distribuidos y bibliotecas de pasaje de mensaje, puede ofrecer la potenza computacional adecuada a esta aplicación. En este proyecto serán implementadas técnicas de procesamiento de imágenes de forma secuencial y paralela, utilizando el lenguaje de programación Java juntamente con la API JAI (*Java Advanced Imaging*) y la biblioteca de cambio de mensajes mpiJava. Pasado esa implementación, se pretende construir una base de comparación entre la aplicación secuencial con la paralela y avaluar el gano con el paralelismo. Se pretende, de ese modo, obtener un estudio que defina requisitos y subvenciones para la aplicación de computación paralela y distribuida en el procesamiento de imágenes medicas.

Palabras-Clave: Computación Paralela. Procesamiento de Imágenes. Sistemas Distribuidos. Imágenes Médicas. mpiJava.

LISTA DE ILUSTRAÇÕES

Figura 1.1 - Taxonomia de arquiteturas paralelas (TANENBAUM, 1999) 28

Figura 1.2 - Modelos de comunicação de computadores paralelos. (a) Multiprocessador; (b) Multicomputador (STALLINGS, 2003). 29

Figura 1.3 – Sistema Computacional Paralelo Distribuído 34

Figura 2.1 - Representação da vizinhança de um *pixel* considerando o ponto central como o *pixel* de interesse: (a) vizinhança-de-4; (b) vizinhança diagonal; (c) vizinhança-de-8. ... 38

Figura 2.2 - Exemplo genérico de uma máscara 3x3 40

Figura 2.3 - Algoritmo genérico para aplicação de templates (NUNES, 2006b). 40

Figura 2.4 – Algoritmo genérico para a implementação do Filtro de Mediana (NUNES, 2006a). 42

Figura 2.5 - Exemplos de suavização utilizando mediana da vizinhança. (a) imagem de Raios-X – original à esquerda e suavizada à direita; (b) imagem de RMN; (c) imagem de US. Os processamentos utilizaram máscara de tamanho 3x3 *pixels*. 43

Figura 2.6 – Região e máscaras para detecção de bordas: (a) Região de *pixels*; máscara para detecção de bordas: (a) horizontais; (b) verticais; (c) de 45º; (d) de 135º. 45

Figura 2.7 – Exemplos de detecção de bordas pelos operadores de Sobel modificado. (a) imagem de RMN – são apresentadas a imagem original e a resultante da detecção de bordas, respectivamente; (b) imagem de Raios-X; (c) imagem de Raios-X; (d) imagem de US. Os processamentos utilizaram máscara de tamanho 3x3 pixels. 46

Figura 3.1 - Principais classes do mpiJava (BAKER, 1999) 52

Figura 4.1. Estratégia de paralelização para o processamento das imagens médicas (SAITO et al., 2007j). 65

Figura 4.2. Implementação genérica do algoritmo paralelo utilizado para o processamento de imagens. 66

Figura 4.3 - Exemplos de suavização utilizando o filtro de mediana. (a) imagem mamográfica original; (b) imagem suavizada com máscara 3x3; (c) imagem suavizada com máscara 5x5; (d) imagem suavizada com máscara 7x7 (SAITO, et al., 2007f). 67

Figura 4.4 – Funcionamento do algoritmo filtro de mediana. 67

Figura 4.5. *Template* da estrutura do programa paralelo Mediana. 68

Figura 4.6. Template da estrutura referente ao processamento do mestre. 69

Figura 4.7. Representação do trecho de código referente à obtenção da imagem e do tamanho da máscara utilizada 70

Figura 4.8. Representação do trecho de código referente à divisão da imagem....................... 70

Figura 4.9. Representação da divisão da imagem: (a) distribuição das partes da imagem aos processos. (b) atribuição do resto da divisão ao penúltimo processo. 71

Figura 4.10. Representação do trecho de código referente à atribuição do resto da divisão da imagem. 71

Figura 4.11. Representação do trecho de código referente ao empacotamento dos dados a serem enviados. 72

Figura 4.12. Representação do trecho de código referente ao envio dos dados pelo mestre. .. 73

Figura 4.13. Representação do trecho de código referente ao recebimento dos dados processados pelos escravos... 74

Figura 4.14. Representação do trecho de código referente à união de cada parte da imagem processada pelos escravos. 75

Figura 4.15. Representação do trecho de código referente à união de cada parte da imagem processada pelos escravos. 76

Figura 4.16. Representação do trecho de código referente ao processamento escravo. 77

Figura 4.17. Trecho de código do método AplicaFiltro com o controle dos índices dos vetores. 78

Figura 4.18. Trecho de código do método AplicaFiltro com a ordenação (*shellsort)* do vetor de *pixels*. 78

Figura 4.19. Trecho de código do método AplicaFiltro com a obtenção do valor mediano. ... 79

Figura 4.20 – Representação das posições dos pixels de interesse iniciais para máscaras de tamanho: (a) 3x3; (b) 5x5; (c) 7x7. 79

Figura 4.21 – Exemplos de detecção de bordas pelos operadores de Sobel modificado com diferentes tamanhos de máscaras. (a) imagem original; (b) máscara 9x9; (c) 11x11 (SAITO, et al., 2007f).. 80

Figura 4.22 – Funcionamento do algoritmo filtro de detecção de bordas. 81

Figura 4.23 – Primeiro trecho de código do método AplicaFiltro da classe FiltroSobel (com a aplicação das máscaras).. 82

Figura 4.24 – Exemplo da utilização de máscaras (operadores de Sobel) de tamanho 9x9..... 83

Figura 4.25 – Segundo trecho de código do método AplicaFiltro da classe FiltroSobel. 84

Figura 5.1 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 500KB. 87

Figura 5.2 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 500KB. 87

Figura 5.3 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 500KB. 88

Figura 5.4 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 500KB. 88

Figura 5.5 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 500KB. 88

Figura 5.6 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 1 MB. 89

Figura 5.7 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 1 MB. 90

Figura 5.8 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 1 MB. 90

Figura 5.9 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 1 MB................ 90

Figura 5.10 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 1 MB. 91

Figura 5.11 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 11 MB...................... 92

Figura 5.12 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 11 MB...................... 92

Figura 5.13 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 11 MB...................... 93

Figura 5.14 – Média do tempo de execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 11 MB. 94

Figura 5.15 – Média do tempo de execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 11 MB. 94

Figura 5.16 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 21 MB...................... 95

Figura 5.17 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 21 MB...................... 95

Figura 5.18 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 21 MB...................... 95

Figura 5.19 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 21 MB. 96

Figura 5.20 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 21 MB. 96

Figura 5.21 - Gráfico da eficiência (em %) na paralelização do filtro de mediana utilizando imagens de diferentes tamanhos. 98

Figura 5.22 - Gráfico da eficiência (em %) na paralelização do filtro de detecção de bordas utilizando imagens de diferentes tamanhos. 98

Figura A.1 - Configuração do arquivo *hosts* 112

Figura A.2 - Configuração do arquivo *hosts.equiv* 113

Figura A.3 - Configuração do arquivo .rhosts 113

Figura A.4 - Configuração do arquivo *securetty* 114

Figura A.5 - Erros na execução de aplicações paralelas 114

Figura B.1 - Configuração das variáveis de ambiente CLASSPATH, PATH e LD_LIBRARY_PATH do mpiJava 117

Figura B.2 - Edição do arquivo *machines* 117

Figura B.3 - Programa de teste para a biblioteca mpiJava 118

Figura B.4 - Resultado da execução do programa de teste para a biblioteca mpiJava 118

Figura C.1 - Configuração da variável ambiente PATH na plataforma Linux 119

Figura C.2 - Programa de teste ao ambiente de programação Java 120

Figura D.1 - Configuração das variáveis de ambiente MPIR_HOME e PATH da instalação da biblioteca de passagem de mensagem MPI na plataforma Linux 121

Figura D.2 - Exemplo de criação do arquivo mpd.conf 122

Figura D.3 - Resultado da execução do programa cpi (MPICH) na plataforma Linux 123

Figura E.1 - Configuração das variáveis de ambiente CLASSPATH e MANPATH da API JAI na plataforma Linux 124

LISTA DE TABELAS

Tabela 1.1 - Taxonomia de Flynn para computadores paralelos (FLYNN; RUDD, 1996) 27

Tabela 3.1 - Tipos de dados básicos em mpiJava. 53

Tabela 3.2 - Rotinas de comunicação ponto-a-ponto bloqueantes e não bloqueantes do mpiJava 57

Tabela 3.3 - Operações de comunicação ponto-a-ponto mpiJava 58

Tabela 3.4 - Operações de Comunicação Coletivas mpiJava 59

Tabela 5.1 – Eficiência (em %) na paralelização do filtro de mediana. 97

Tabela 5.2 – Eficiência (em %) na paralelização do filtro de de detecção de bordas. 97

Tabela A.1 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB. 154

Tabela A.2 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB. 154

Tabela A.3 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB. 155

Tabela A.4 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB. 155

Tabela A.5 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB. .. 155

Tabela A.6 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB. 155

Tabela A.7 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB. 156

Tabela A.8 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB. 156

Tabela A.9 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB. 156

Tabela A.10 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB. 156

Tabela A.11 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB. 157

Tabela A.12 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB................157

Tabela A.13 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB................157

Tabela A.14 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB................157

Tabela A.15 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.....158

Tabela A.16 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB................158

Tabela A.17 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB................158

Tabela A.18 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB................158

Tabela A.19 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB................159

Tabela A.20 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.....159

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
CAD	*Computer-Aided Diagnosis*
CBIR	*Content-Based Image Retrieval*
CC-NUMA	*Cache Coherent NUMA*
COMA	*Cache Only Memory Access*
CORBA	*Common Object Request Broker Architecture*
COW	*Cluster of Workstations*
DICOM	*Digital Imaging and COmmunication in Medicine*
DNS	*Domain Name Service*
DSM	*Distributed Shared Memory*
EPs	Elementos de Processamento
JAI	*Java Advanced Imaging*
JDK	*Java Development Kit*
JNI	*Java Native Interface*
IPs	*Internet Protocols*
LAM	*Local Area Multicomputer*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MPI	*Message Passing Interface*
mpiJava	*Java Interface to MPI*
MPPs	*Massively Parallel Processing*
NC-NUMA	*Non-Coherent NUMA*
NFS	*Network File System*
NOW	*Network of Workstations*
NUMA	*Non-Uniform Memory Access*
OpenMP	*Open Multi-Processing*
ORNL	*Oak Ridge National Laboratory*
PVM	*Parallel Virtual Machine*
RGB	*Red Green Blue*
RMN	Ressonância Magnética Nuclear
RSH	*Remote Shell*

SD	Sistema Distribuído
SIAPDI	Serviço Integrado de Acesso e Processamento Distribuído de Imagens
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
SMP	*Symmetric Multiprocessor*
SPMD	*Single Process Multiple Data*
TC	Tomografia Computadorizada
UCPs	Unidade Central de Processamento
ULAs	Unidade Lógica e Aritmética
UMA	*Uniform Memory Access*
US	Ultra-Som
USa	Ultra-Sonografia

SUMÁRIO

INTRODUÇÃO **19**

Objetivos 20
Organização da monografia 20

1. COMPUTAÇÃO PARALELA E DISTRIBUÍDA **22**

1.1. Computação Paralela 22
1.1.1. Definições Básicas e Motivações 22
1.1.2. Conceitos Essenciais 23
1.1.2.1. Concorrência e Paralelismo 24
1.1.2.2. Granulosidade ou Níveis de Paralelismo 24
1.1.2.3. Abordagens de Programação Paralela 24
1.1.2.4. *Speedup* e Eficiência 25
1.1.3. Arquiteturas Paralelas 26
1.1.3.1. Taxonomias de Computadores Paralelos 26
1.1.3.2. Modelos de Comunicação 28
1.2. Sistemas Computacionais Distribuídos 30
1.3. Computação Paralela sobre Sistemas Distribuídos 33
1.4. Considerações Finais 35

2. PROCESSAMENTO DE IMAGENS MÉDICAS **36**

2.1. Considerações Iniciais 36
2.2. Definições Básicas 37
2.3. Conceitos Essenciais 38
2.3.1. Vizinhança de um *Pixel* 38
2.3.2. Conectividade entre *Pixels* 39
2.3.3. Domínios: Espacial e da Freqüência 39
2.3.4. *Templates* 40
2.4. Técnicas de Processamento de Imagens 41
2.4.1. Processamento de Nível Baixo – Suavização de Imagens 41
2.4.2. Processamento de Nível Médio - Segmentação 43
2.5. Considerações Finais 47

3. SUPORTE À COMPUTAÇÃO PARALELA DISTRIBUÍDA **49**

3.1. Ambientes de Passagem de Mensagens 49
3.1.1. PVM 50
3.1.2. MPI 50
3.2. mpiJava 51
3.2.3. Rotinas de Gerenciamento do Ambiente mpiJava 53
3.2.4. Rotinas de Comunicação no mpiJava 55
3.2.4.1. Rotinas de Comunicação Ponto-a-Ponto 55
Modo de Comunicação 56
Comunicação Bloqueante e Não-Bloqueante 57
3.2.4.2. Rotinas de Comunicação Coletiva 59
3.3. Trabalhos Correlatos 60
3.4. Considerações Finais 62

4. METODOLOGIA....63

4.1. Tecnologias Utilizadas 63
4.2. Modelo de Paralelismo para o Processamento de Imagens Médicas 64
4.3. Implementação dos Algoritmos de Processamento de Imagens....66
4.3.1. Algoritmo de Suavização – Filtro de Mediana....66
4.3.2. Algoritmo de Segmentação – Filtro de Detecção de Bordas....80
4.4. Considerações Finais 85

5. ANÁLISE DE DESEMPENHO DOS RESULTADOS OBTIDOS....86

5.1. Imagens de Tamanho 500 KB 87
5.2. Imagens de Tamanho 1 MB....89
5.3. Imagens de Tamanho 11 MB....91
5.4. Imagens de Tamanho 21 MB....94
5.5. Considerações Finais 98

6. CONCLUSÕES....100

6.1. Produção Bibliográfica....102
6.1.1. Artigos Completos Publicados em Periódicos 102
6.1.2. Trabalhos Completos Publicados em Anais de Congressos....102
6.1.3. Resumos Publicados em Anais de Congressos 103

REFERÊNCIAS 104

APÊNDICE A – CONFIGURAÇÃO DO SISTEMA....112

A.1. Configuração de Arquivos para a Comunicação entre as Máquinas (sem a Autenticação por Senha)....112
A.1.1. Configuração do Arquivo *hosts*....112
A.1.2. Configuração do Arquivo *hosts.equiv* 113
A.1.3. Configuração do Arquivo *.rhosts* 113
A.1.4. Configuração do Arquivo *securetty*....113
A.2. Configurações do Nível de Segurança....114

APÊNDICE B - INSTALAÇÃO DO MPIJAVA (*JAVA INTERFACE TO MPI*)....116

APÊNDICE C - INSTALAÇÃO DO AMBIENTE DE PROGRAMAÇÃO JAVA....119

APÊNDICE D - INSTALAÇÃO DO MPI (*MESSAGE PASSING INTERFACE*) 121

APÊNDICE E - INSTALAÇÃO DA API JAI (*JAVA ADVANCED IMAGING*)....124

APÊNDICE F – CÓDIGOS FONTE 125

F.1. Implementação Seqüencial do Filtro de Mediana 125
F.2. Implementação Paralela do Filtro de Mediana....128
F.3. Implementação Seqüencial do Filtro de Detecção de Bordas Sobel 137
F.4. Implementação Paralela do Filtro de Detecção de Bordas Sobel....142

ANEXO A – ANÁLISE ESTATÍSTICA (TESTES DE HIPÓTESES) 153

INTRODUÇÃO

O processamento de imagens é amplamente utilizado em muitas aplicações. Entretanto, essa área de conhecimento exige um intenso processamento, visto que as imagens apresentam em sua grande maioria tamanho elevado, constituindo matrizes enormes de pontos a serem processados.

Em se tratando de imagens médicas, esta questão é ainda mais crítica, uma vez que aplicações médicas requerem curto tempo de resposta e não se pode permitir armazenamento com perdas de dados caso a imagem seja utilizada para diagnóstico. Exigindo precisão na sua aquisição, essa classe de imagens gera um conjunto ainda maior de dados (BARBOSA, 2000). O tamanho elevado aliado à necessidade de passagem de algum filtro, seja para suavização, atenuação ou realce, aumenta o tempo de processamento dessas imagens, prejudicando a avaliação das mesmas (NUNES, 2001).

A necessidade de alto desempenho e alto poder computacional para o processamento dessas imagens pode ser resolvida por meio do uso de sistemas distribuídos e de bibliotecas de passagem de mensagens, que viabilizam a computação paralela sobre sistemas distribuídos (computação paralela e distribuída).

Os sistemas computacionais distribuídos aplicados à computação paralela permitem uma melhor relação custo/benefício para a computação paralela. Estes sistemas oferecem a potência computacional adequada às aplicações que não necessitam de uma máquina maciçamente paralela, porém necessitam de uma potência computacional maior que uma máquina seqüencial pode oferecer (BRANCO, 1999).

O MPI (*Message Passing Interface*) (SNIR *et al.*, 1996), proposto para ser um padrão internacional de biblioteca de passagem de mensagens, tem tido destaque na literatura, não só pela flexibilidade, mas também pelo fato de constituir um tipo de solução para o problema da portabilidade de programas paralelos entre sistemas diferentes. Além disso, possibilita eficiência e segurança em qualquer plataforma paralela (CÁCERES, 2001).

Existem muitas implementações de MPI em aplicações desenvolvidas em linguagens como Fortran, C e C++. Com o surgimento de Java (SUN, 2006b), inúmeras propostas foram apresentadas para a utilização de MPI nessa linguagem. A linguagem de programação Java apresenta um alto potencial, inclui funcionalidade para ajudar o desenvolvimento de aplicações distribuídas, e apresenta portabilidade, fator importante para o desenvolvimento

distribuído de aplicações paralelas. Sendo assim o ambiente de passagem de mensagens, mais especificamente neste trabalho é o mpiJava (MPIJAVA, 2007),

Objetivos

O presente trabalho de pesquisa tem como objetivo demonstrar a viabilidade na otimização do tempo de processamento de imagens, mais especificamente de imagens médicas. Para tanto, utiliza-se a programação paralela e distribuída, viabilizada por sistemas distribuídos e pela biblioteca de passagem de mensagens mpiJava.

Organização da monografia

Essa monografia está organizada da seguinte forma:

Capítulo 1 – Computação Paralela e Distribuída: apresenta a computação paralela e distribuída, destacando-se as arquiteturas utilizadas para a execução de aplicações paralelas. Conceitos essenciais referentes à programação paralela e sistemas distribuídos também são discutidos.

Capítulo 2 – Processamento de Imagens Médicas: descreve inicialmente conceitos básicos, referentes ao processamento de imagens, para a compreensão das técnicas de processamento de imagens propostas no trabalho. Logo em seguida, essas técnicas, são definidas.

Capítulo 3 – Suporte à Computação Paralela Distribuída: aborda os ambientes de passagem de mensagens como suporte à computação paralela distribuída. Esse capítulo apresenta também as características e a descrição de algumas funcionalidades do ambiente de passagem de mensagens utilizado nesse trabalho, bem como alguns trabalhos encontrados na literatura.

Capítulo 4 – Metodologia: apresenta as tecnologias utilizadas, o modelo de paralelismo para o processamento de imagens médicas, descrevendo a estratégia utilizada para a paralelização. Além disso, são apresentadas, de forma detalhada, as implementações de cada algoritmo estudado.

Capítulo 5 – Análise de Desempenho dos Resultados Obtidos: enfatiza as análises de desempenho das implementações dos filtros de mediana e de detecção de bordas Sobel.

Capítulo 6 – Conclusões: apresenta algumas conclusões finais e sugestões para atividades consideradas relevantes para a continuidade do projeto. Além disso, apresenta também a produção bibliográfica obtida com os resultados da paralelização das técnicas de processamento de imagens implementadas.

1. COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Neste capítulo são abordados conceitos inerentes à computação paralela distribuída. Para facilitar o estudo, computação paralela e sistemas distribuídos são tratados separadamente. Em seguida, a convergência das duas áreas é apresentada.

1.1. Computação Paralela

Embora as arquiteturas seqüenciais de von Neumann tenham sido um grande avanço tecnológico e largamente aprimoradas ao longo dos anos, elas demonstram deficiências quando utilizadas por certas aplicações que, em virtude do uso de algoritmos complexos e/ou conjuntos de dados extensos, demandam grande potência computacional; já outras aplicações são naturalmente paralelas. Com o objetivo principal de prover melhor desempenho e adequação para essas aplicações, surgiu a computação paralela (QUINN, 1994; DONGARRA et al., 2003; ALMASI; GOTTLIEB, 1994).

1.1.1. Definições Básicas e Motivações

Diversas definições de computação paralela podem ser encontradas na literatura. Segundo Almasi e Gottlieb (1994), computação paralela constitui-se de uma coleção de elementos de processamento (EPs) que se comunicam e cooperam entre si para resolver problemas de grande dimensão mais rapidamente. Hwang (1984) define processamento paralelo como: "forma eficiente do processamento de informações com ênfase na exploração de eventos concorrentes no processo computacional". Quinn (1987) por sua vez apresenta outra definição "computação paralela é o processamento de informações que enfatiza a manipulação concorrente dos dados, que pertencem a um ou mais processos que objetivam resolver um único problema".

Dessa forma, processamento paralelo consiste na divisão de uma determinada aplicação de maneira que esta possa ser executada por vários elementos de processamento, que por sua vez deverão cooperar entre si (comunicação e sincronismo), buscando uma maior eficiência (SANTANA, 1997).

Vários fatores explicam a necessidade do processamento paralelo. O principal deles, citado anteriormente, trata-se da busca por maior desempenho (ALMASI, 1994) (ZALUSKA, 1991). As diversas áreas nas quais a computação se aplica, sejam científicas, industriais, militares ou médicas, requerem cada vez mais poder computacional em virtude dos algoritmos complexos que são utilizados e do tamanho do conjunto de dados a ser processado (KIRNER, 1991).

Além da busca por maior desempenho, outros fatores motivam o desenvolvimento da computação paralela (ALMASI, 1994) (AMORIM, 1988). Entre eles:

- O desenvolvimento tecnológico permitindo a construção de microprocessadores de alto desempenho que agrupados possibilitam um ganho significativo de poder computacional e uma melhor relação custo/desempenho, quando comparadas aos caros supercomputadores (ZALUSKA, 1991);
- A modularidade permitindo o agrupamento desses processadores em módulos, de acordo com a natureza da aplicação. Além disso, tem-se um meio de extensão do sistema por meio da inclusão de novos módulos;
- A tolerância a falhas, obtida por meio da replicação de EPs, uma característica altamente desejada para aplicações que necessitam de confiabilidade.

Por outro lado a computação paralela apresenta também algumas desvantagens. Um computador paralelo tem pouca ou nenhuma utilidade sem um bom programa paralelo. Embora várias técnicas de programação paralela tenham sido desenvolvidas (FOSTER, 1995; DONGARRA et al., 2003), a construção de aplicações paralelas é ainda uma tarefa bastante custosa para o programador. A divisão de uma aplicação em processos que possam ser atribuídos aos EPs de forma eficiente e a sincronização são tarefas extremamente complexas.

1.1.2. Conceitos Essenciais

Esta seção apresenta alguns conceitos fundamentais, relacionados à programação paralela, e que serão adotados nesse trabalho.

1.1.2.1. Concorrência e Paralelismo

Concorrência existe quando dois ou mais processos inicializaram e não terminaram sua execução. Esses processos, geralmente, disputam pela utilização de um processador. Para que haja paralelismo é preciso que dois ou mais processos estejam em execução (alocando processador) num mesmo intervalo de tempo (ALMASI; GOTTLIEB, 1994). Dessa forma, processos paralelos são também concorrentes, porém o contrário nem sempre é verdadeiro.

O paralelismo consiste na utilização de um sistema paralelo composto por múltiplos elementos de processamento. Já a concorrência pode ocorrer tanto em sistemas paralelos quanto em sistemas com um único processador. Em sistemas uniprocessados tem-se um pseudo-paralelismo, em que na execução de cada processo em pequenos intervalos de tempo tem-se a impressão de que os processos são executados simultaneamente.

1.1.2.2. Granulosidade ou Níveis de Paralelismo

Granulosidade ou nível de paralelismo relaciona o tamanho da unidade de trabalho executada pelos processadores, sendo classificada em fina, média e grossa (QUINN, 1994).

A granulosidade fina consiste em um paralelismo ao nível de operações ou de instruções, sendo, geralmente, implementada em *hardware* e implica em um grande número de processadores pequenos e simples. Na granulosidade média, as unidades de trabalho são constituídas de blocos de instruções; geralmente, esse nível de paralelismo exige suporte de linguagens de programação paralela. No último nível da classificação, a granulosidade grossa, o paralelismo é dado ao nível dos processos ou tarefas, executando simultaneamente e geralmente se aplica a plataformas com poucos processadores grandes e complexos (KIRNER, 1991; NAVAUX, 1989).

1.1.2.3. Abordagens de Programação Paralela

Os processos e/ou *threads* que compõem uma aplicação paralela podem ser estruturados sob duas abordagens principais: o paralelismo de dados e o paralelismo funcional (HWANG; XU, 1998).

No paralelismo de dados, também denominado *Single Program Multiple Data* (SPMD), os dados a serem processados são divididos em porções, geralmente do mesmo tamanho. Cada processador executa o mesmo código, operando sobre uma porção diferente dos dados. A denominação paralelismo de dados deve-se ao fato de que o paralelismo obtido vem da decomposição dos dados entre diversos EPs. No paralelismo funcional ou *Multiple Program Multiple Data* (MPMD), cada tarefa possui um conjunto de instruções distinto que podem ou não operar sobre o mesmo conjunto de dados. O nome deve-se ao fato de que cada tarefa possui uma função diferente.

1.1.2.4. *Speedup* e Eficiência

Aumentar o desempenho é uma das metas da computação paralela. *Speedup* e eficiência são importantes métricas para verificar a qualidade de um programa paralelo. O s*peedup* permite determinar a relação existente entre o tempo dispensado para executar um algoritmo em um único processador e o tempo gasto para executá-lo em *p* processadores (equação 1.1). A eficiência determina a taxa de utilização dos processadores e é dada pela equação 1.2 (QUINN, 1994).

$$S(p) = \frac{T_1}{T_p} \tag{1.1}$$

Em que:

- $S(p)$ é o s*peedup* obtido pela razão entre T_1 e T_p com p processadores;
- T_1 é o tempo gasto para executar o programa seqüencial;
- T_p é o tempo gasto para executar o programa paralelo.

$$E(p) = \frac{S_p}{p} \tag{1.2}$$

Em que:

- $E(p)$ é a eficiência obtida pela razão entre $S(p)$ e p;
- p é o número de processadores utilizados.

Pelas equações observa-se que o caso ideal seria $S(p) = p$ e $E(p) = 1$, um ganho igual ao número de processadores utilizado e uma utilização de 100% dos processadores. Entretanto, o caso ideal é dificilmente alcançado, devido a diversos fatores, tais como: atraso na comunicação, balanceamento de carga inadequado, partes seqüenciais do programa paralelo, entre outros (BRANCO, 1999).

1.1.3. Arquiteturas Paralelas

Diferentes arquiteturas podem ser utilizadas para dar suporte à computação paralela. Elas se diferem em relação aos componentes utilizados, ao modo de interligá-los e até mesmo ao modelo de programação. Porém a maior divergência é a forma de interligá-los (TANENBAUM, 1999). Esse é o fator que apresenta grande influência nas características e no desempenho de um computador paralelo e que dá origem a diversas classificações de computadores paralelos.

1.1.3.1. Taxonomias de Computadores Paralelos

Devido à diversidade de computadores paralelos, faz-se necessária a introdução de uma classificação. Embora diversos esquemas foram propostos para classificar arquiteturas de computadores paralelos, nenhum teve ampla aceitação (TANENBAUM, 1999). Dentre os esquemas propostos, o mais citado é a taxonomia proposta por Flynn (1972).

A Taxonomia de Flynn (FLYNN; RUDD, 1996) consiste em fluxos de instruções e fluxos de dados. Um fluxo de instruções é uma seqüência de instruções executadas em que cada fluxo corresponde a um contador de programa (TANENBAUM, 1999). Um fluxo de dados consiste em um conjunto de operandos manipulado por um fluxo de instruções (TANENBAUM, 1999; QUINN, 1994). As quatro categorias que compõem a classificação de Flynn, podem ser observadas na Tabela 1.1.

Tabela 1.1 - Taxonomia de Flynn para computadores paralelos (FLYNN; RUDD, 1996)

Fluxo de Instruções	Fluxo de Dados	Categoria
Único	Único	SISD
Único	Múltiplo	SIMD
Múltiplo	Único	MISD
Múltiplo	Múltiplo	MIMD

Single Instruction, Single Data (SISD) em que um único fluxo de instrução opera em único fluxo de dados. Computadores seriais, baseados no modelo de von Neumann, são exemplos de arquiteturas SISD (QUINN, 1994).

Single Instruction, Multiple Data (SIMD) em que um único fluxo de instrução é executado por múltiplos elementos de processamento sobre conjuntos de dados distintos. Processadores vetoriais e matriciais são representantes dessa categoria.

Multiple Instruction, Single Data (MISD) em que múltiplos fluxos de instruções operam sobre um mesmo fluxo de dados. Existe uma divergência entre os autores sobre a existência de arquiteturas que possam ser classificadas como MISD. Segundo Flynn e Rudd (1996) e Zomaya (1996) essa categoria é composta por *arrays* sistólicos. Já Stallings (2003) afirma que essa estrutura nunca foi implementada.

Multiple Instruction, Multiple Data (MIMD) em que diferentes seqüências de instruções sobre diferentes conjuntos de dados são executadas simultaneamente por um conjunto de processadores. Grande parte dos computadores paralelos compõe essa categoria.

Apesar de ser bastante difundida, a taxonomia de Flynn não é suficiente para classificar os vários computadores modernos de forma adequada. Dessa forma, com o objetivo de incluir arquiteturas difíceis de serem acomodadas na taxonomia de Flynn, Duncan apresenta uma taxonomia dividida em arquiteturas síncronas e assíncronas (DUNCAN, 1990).

As arquiteturas síncronas coordenam operações concorrentes utilizando-se de relógio global, unidade de controle central ou controladoras de unidade vetorial. Esse grupo é composto pelas arquiteturas SIMD da taxonomia de Flynn, pelas arquiteturas vetoriais e sistólicas.

Nas arquiteturas assíncronas não há controle centralizado mantido por *hardware*; os processadores podem operar de maneira autônoma. Esse grupo é composto, basicamente, pelas arquiteturas MIMD da taxonomia de Flynn, sejam elas convencionais (MIMD com memória distribuída ou compartilhada) ou não-convencionais (MIMD/SIMD, fluxo de dados, redução ou dirigidas a demanda e frente de onda) (DUNCAN, 1990).

Embora a taxonomia de Duncan seja bastante abrangente, alguns autores (TANENBAUM, 1999; STALLINGS, 2003; GRAMA et al., 2003) preferem uma classificação mais simples composta por arquiteturas de computadores paralelos mais utilizados atualmente. Esta classificação é caracterizada por manter as quatro categorias da Taxonomia de Flynn e pela sub-divisão das categorias SIMD e MIMD, conforme pode ser observado na Figura 1.1.

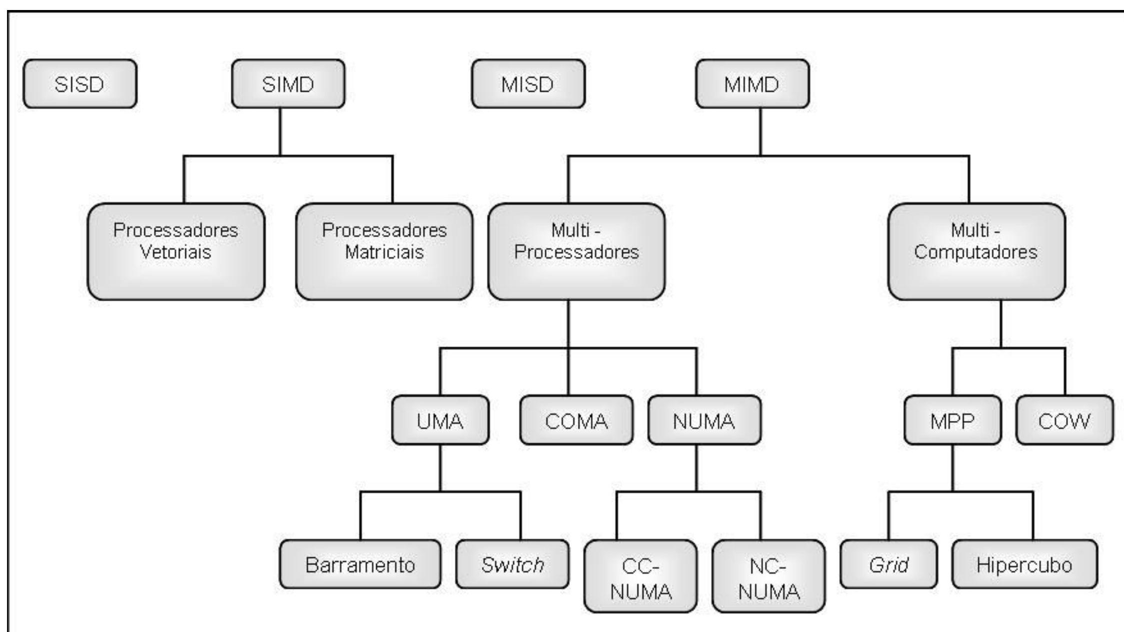


Figura 1.1 - Taxonomia de arquiteturas paralelas (TANENBAUM, 1999)

A categoria SIMD foi dividida em dois grupos: processadores vetoriais e matriciais. O primeiro grupo é composto por supercomputadores numéricos e máquinas que operam sobre vetores, realizando a mesma operação em cada elemento do vetor. O segundo grupo é composto por máquinas com uma unidade de controle principal que repassa a próxima instrução a ULAs independentes. A categoria MIMD é dividida, de acordo com o modelo de comunicação, em multiprocessadores e multicomputadores. Tais arquiteturas são discutidas na seção seguinte.

1.1.3.2. Modelos de Comunicação

A troca de informações entre as tarefas paralelas, ocorre de duas formas: acessando um espaço de memória compartilhado ou trocando mensagens, sendo suportados pelos

modelos multiprocessadores e multicomputadores, respectivamente. A diferença básica entre eles está no modo que a memória é acessível aos EPs, conforme pode ser observado na Figura 1.2.

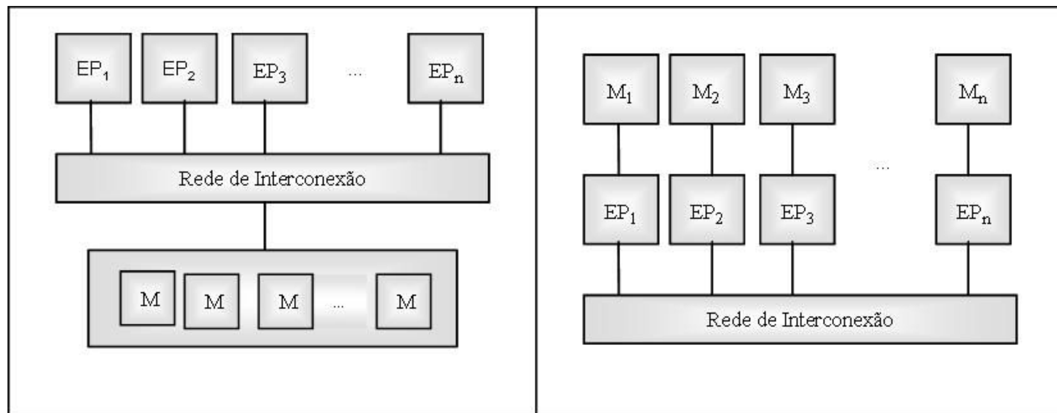


Figura 1.2 - Modelos de comunicação de computadores paralelos. (a) Multiprocessador; (b) Multicomputador (STALLINGS, 2003).

O modelo multiprocessador, conhecido também como sistema de memória compartilhada, é esquematizado na Figura 1.2(a). É possível observar que os EPs compartilham uma mesma memória física. Existe um único espaço de endereçamento virtual para todos os EPs, apesar dos módulos de memória estarem fisicamente separados.

O segundo modelo de comunicação multicomputadores, representado pela Figura 1.2(b), é também denominado como sistema de memória distribuída, devido à memória encontrar-se física e logicamente distribuída. Pela figura pode se observar que cada EP tem sua própria memória e seu próprio espaço de endereçamento, inacessível a outros EPs por meio de instruções *LOAD* e *STORE*.

Dessa forma, em sistemas com memória distribuída, a comunicação entre os EPs ocorre por meio de passagem de mensagens, utilizando a rede de interconexão. Isso gera grandes conseqüências sobre o *software*, tornando-o mais complexo. Já em sistemas de memória compartilhada, processos em diferentes EPs podem se comunicar apenas lendo e escrevendo na memória, facilitando a sincronização que pode ser feita utilizando-se semáforos e monitores, mecanismos já utilizados em computadores seriais.

Os multicomputadores ainda são mais fáceis de se construir e são escaláveis, apesar do desenvolvimento de aplicações ser mais trabalhoso.

A construção de sistemas híbridos tem sido realizada para resolver a dificuldade no desenvolvimento de aplicações, em que uma das abordagens utilizadas é conhecida como *Distributed Shared Memory* (DSM) (TANENBAUM, 1999). Nessa abordagem o sistema

operacional simula um sistema de memória compartilhada que permite à aplicação acessar a memória de todo o computador paralelo como um único espaço de endereçamento.

Os tipos de multiprocessadores existentes, observados na Figura 1.1, são: *Uniform Memory Access* (UMA), *Non-Uniform Memory Access* (NUMA) e *Cache Only Memory Access* (COMA).

Em arquiteturas UMA, o tempo de acesso de qualquer porção de memória para qualquer EP é o mesmo. Os multiprocessadores simétricos - *Symmetric MultiProcessor* (SMP) são exemplos de arquiteturas UMA.

Já em arquiteturas NUMA, os tempos de acesso a cada região de memória difere de EP para EP (STALLINGS, 2003), pois cada EP possui uma memória local e o espaço de endereçamento compartilhado é formado pela combinação destas (QUINN, 1994). Essas arquiteturas podem possuir coerência de *cache* em *hardware*, *Cache Coherent NUMA* (CC-NUMA), ou não possuir coerência de *cache*, *Non-Coherent NUMA* (NC-NUMA) (STALLINGS, 2003). Em arquiteturas COMA, o acesso à memória fisicamente remota é feito apenas por meio de *cache*.

Ainda na Figura 1.1, encontram-se os multicomputadores divididos em duas categorias. MPPs (*Massively Parallel Processing*), representam a primeira delas e consistem de sistemas com acoplamento forte formados por um grande número de UCPs interconectadas por uma rede proprietária de alta velocidade. Já a segunda categoria abrange NOW (*Network of Workstations*) e COW (*Cluster of Workstations*), sistemas com acoplamento fraco, constituindo-se de computadores autônomos interligados por uma rede de interconexão.

1.2. Sistemas Computacionais Distribuídos

Para dar suporte à computação paralela, utiliza-se um computador com múltiplos elementos de processamento, capazes de se comunicar e cooperar para resolver grandes problemas mais rapidamente (ALMASI; GOTTLIEB, 1994). Uma forma de se obter uma arquitetura com múltiplos EPs é a utilização de um sistema distribuído.

Um sistema computacional distribuído, segundo Tanenbaum (1992) e Colouris et al. (2006), é uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um sistema operacional distribuído que permitem compartilhamento transparente de recursos existentes no sistema. Dessa forma, um sistema distribuído aparenta ser um sistema centralizado, enquanto na realidade o sistema operacional

é executado em múltiplas unidades de processamento não dependentes que se comunicam (MÜLLENDER , 1993).

Surgidos da necessidade de compartilhar recursos de modo eficiente, os sistemas distribuídos tornaram-se uma alternativa, eficiente e de baixo custo, aos sistemas centralizados.

A utilização de sistemas distribuídos apresenta inúmeras vantagens em relação a um sistema centralizado, dentre as quais destacam-se:

- Econômicas, havendo um aproveitamento de máquinas potencialmente ociosas, além de ser mais barato interconectar vários processadores a adquirir um supercomputador;
- Distribuição inerente, algumas aplicações são distribuídas por natureza e, portanto mais facilmente implementadas nesse tipo de ambiente;
- Confiabilidade, aspecto de tolerância a falhas em que no caso de falha de máquinas, o funcionamento do sistema não é afetado de forma que ele venha a suspender sua execução, pode apresentar apenas uma redução no desempenho total;
- Crescimento incremental, em que o poder computacional pode ser aumentado pela inclusão de novos equipamentos;
- Flexibilidade, em que sistemas distribuídos são mais flexíveis que máquinas isoladas, por isso muitas vezes são utilizados até mesmo quando não se esteja buscando desempenho.

Por outro lado, os sistemas distribuídos apresentam também algumas desvantagens relacionadas com:

- **Rede de comunicação:** gargalo na rede de comunicação, a rede pode saturar ou causar outros problemas e proporcionar a perda de pacotes de dados;
- **Segurança:** pode ser comprometida devido ao compartilhamento de dados e grande número de usuários.

A ausência de memória compartilhada em um ambiente distribuído exige que a interação entre processadores ocorra de uma forma distinta do ambiente centralizado, ao invés de variáveis ou arquivos compartilhados utiliza-se troca de mensagens.

1.2.1. Características Essenciais em um Sistema Distribuído

A seguir são descritas as principais características de um sistema distribuído (SD) (TANENBAUM, 1995; MULLENDER, 1993; COLOURIS et al., 2001):

- **Transparência:** relacionada ao modo como o usuário e o programador de aplicações vêem o sistema. Um sistema distribuído é transparente se é visto como um sistema único ocultando a existência de componentes independentes e separados fisicamente. ANSA (1989) e ISO (1992) apresentam oito formas de transparência: acesso, localização, concorrência, replicação, falhas, mobilidade (ou migração), desempenho, propriedade escalar (*scaling*) (COULOURIS et al., 2001).
- **Tratamento de falhas:** é necessária em sistemas distribuídos, pois quando um componente do sistema falha, os demais continuam ativos, podendo gerar inconsistências. Não se sabe exatamente onde as tarefas estão sendo executadas, sendo assim a falha em uma máquina ou um erro que venha a ocorrer não deve interferir no funcionamento do sistema. Para lidar com as falhas algumas técnicas podem ser utilizadas, tais como: detecção de falhas, mascaramento de falhas (minimizar os efeitos de uma falha), recuperação de falhas (retornar o sistema para um estado correto antes da falha ocorrer) e redundância (replicar componentes, garantindo que, se um componente do sistema falhar, haverá outro, devidamente atualizado, para cumprir sua função).
- **Capacidade de expansão (*Scalability*):** característica de manter-se estável quando a quantidade de recursos e de usuários aumenta significativamente (COULOURIS et al., 2001). Várias técnicas têm sido utilizadas com o objetivo de possibilitar maior expansão do sistema, tais como: utilização de algoritmos descentralizados, evitando gargalos; replicação e estruturação hierárquica de servidores e dados, utilização de *caches*.

- **Concorrência:** característica natural dos sistemas distribuídos decorrente da existência de múltiplos usuários e do compartilhamento de recursos. A independência de recursos e a indefinição de localização de execução das tarefas, tornam o gerenciamento mais complicado devido à necessidade de sincronização de processos para que a consistência do sistema seja mantida.
- **Abertura (*openness*):** para ser considerado aberto, o sistema deve permitir a comunicação com outro sistema aberto por meio da utilização de regras padronizadas quanto a formato, conteúdo e significado das mensagens enviadas e recebidas (TANENBAUM, 1995). Em um sistema aberto novos componentes de *hardware* e *software* podem ser adicionados, independentemente de fabricantes individuais.

1.3. Computação Paralela sobre Sistemas Distribuídos

A computação paralela e distribuída, ou computação paralela sobre sistemas distribuídos, é a convergência de duas áreas surgidas com motivações e características distintas, a computação paralela e os sistemas distribuídos, trazendo benefícios para ambas (ZOMAIA, 1996).

Os sistemas distribuídos possuem várias características desejáveis na computação paralela, tais como: transparência de acesso aos recursos, confiabilidade e tolerância a falhas. Os avanços obtidos em capacidade de processamento, tecnologia de rede e em ferramentas de *software* tornaram os sistemas distribuídos uma infra-estrutura conveniente para o processamento paralelo (HARIRI; PARASHAR, 2004).

Dessa forma, com o intuito de aumentar ainda mais o desempenho, a flexibilidade e a versatilidade das máquinas paralelas, decidiu-se empregar a computação paralela sobre sistemas distribuídos, explorando as características de ambos e usufruindo suas características individuais. A computação paralela e distribuída surge como alternativa à utilização dos supercomputadores paralelos caros.

Apesar de possuir um meio de comunicação mais lento, fator que degrada o desempenho, tais sistemas, representados pela Figura 1.3, têm sido utilizados com sucesso para paralelizar aplicações que possuem pouca necessidade de comunicação entre os processos (ZALUSKA, 1991).

Na Figura 1.3 é possível observar que não existe endereçamento global, cada processador tem sua memória local (fracamente acoplado), desta forma a memória não é acessada diretamente por todos os processadores do sistema o que leva a um alto custo de comunicação (trocas de mensagens) e de sincronização.

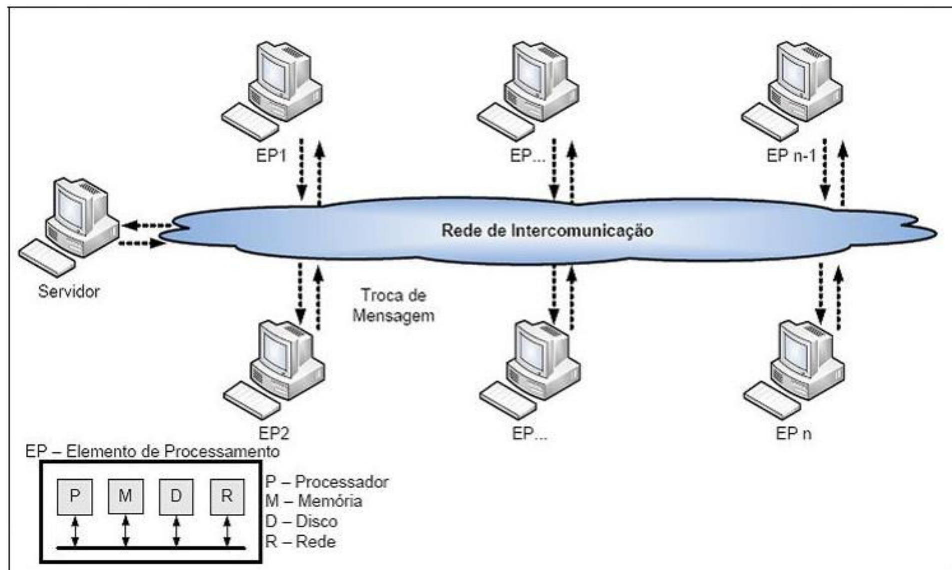


Figura 1.3 – Sistema Computacional Paralelo Distribuído

O aumento da utilização de sistemas computacionais distribuídos, associado aos conceitos de computação paralela, permite que tarefas possam ser realizadas explorando maior potência computacional com relevante redução dos custos.

Além da redução do custo com a utilização de *hardware* existente, a utilização da computação paralela sobre sistemas distribuídos apresenta outras vantagens, tais como: aumento de desempenho pela atribuição de tarefas de acordo com a arquitetura apropriada, exploração da heterogeneidade natural das aplicações, utilização de recursos conhecidos, recursos fornecidos pelas máquinas virtuais e o fato da computação paralela distribuída facilitar o trabalho corporativo (BRANCO, 1999).

Apesar das inúmeras vantagens apresentadas, existem alguns problemas. Técnicas e mecanismos antes utilizados na computação paralela nem sempre são convenientes na computação paralela distribuída. Fatores característicos dos sistemas distribuídos, tais como heterogeneidade, atraso na rede e carga de trabalho externa às aplicações paralelas, tornam a utilização e o gerenciamento desses sistemas uma tarefa complexa.

1.4. Considerações Finais

Os avanços tecnológicos obtidos nas últimas décadas, principalmente no que se refere ao desenvolvimento dos microprocessadores e das redes de comunicação, possibilitaram rápidas e profundas mudanças. A computação, que era baseada unicamente no uso de computadores seqüenciais e centralizados, é caracterizada, atualmente, pela grande conectividade dos recursos, tornando-se cada vez mais comum o uso de sistemas distribuídos (COLOURIS et al., 2006).

Houve um aperfeiçoamento muito grande dos componentes do modelo seqüencial de von Neumann. Porém, devido às deficiências demonstradas por esse modelo quando utilizadas por aplicações que necessitavam de um poder de processamento maior, surge a computação paralela, tendo como meta o aumento do poder de processamento de forma mais barata e aprimorada, principalmente a problemas essencialmente paralelos.

A convergência, apresentada entre a computação paralela e os sistemas distribuídos, trouxe inúmeras vantagens, principalmente para a primeira, como a redução de custos, o aumento de desempenho, a utilização mais adequada de recursos evitando ociosidade no sistema, entre outras. Tudo isso tem feito com que cada vez mais apareçam trabalhos em que a computação paralela distribuída pode ser aplicada para solucionar uma variedade de aplicações que necessitam de alto poder computacional.

O próximo capítulo descreve alguns conceitos relacionados ao processamento de imagens, aplicação esta que, na maioria das vezes, requer esse alto poder de processamento.

2. PROCESSAMENTO DE IMAGENS MÉDICAS

Neste capítulo, algumas considerações sobre processamento de imagens são tecidas inicialmente. Conceitos básicos, referentes ao processamento de imagens, também são descritos para a compreensão das técnicas de processamento de imagens propostas no trabalho. Logo em seguida, essas técnicas, são definidas.

2.1. Considerações Iniciais

Imagens médicas têm por finalidade o auxílio na composição do diagnóstico de anomalias e o fornecimento de material para acompanhamento de terapias.

Os sistemas de diagnóstico auxiliado por computador, *computer-aided diagnosis* (CAD), são sistemas computacionais com a finalidade de auxiliar o radiologista na tomada de decisão a respeito de um diagnóstico, obtido por resultados de uma análise computadorizada de imagens médicas (GIGER, 2000).

Sendo assim, esses sistemas são objetos de pesquisa de várias instituições. Chan et al. (1990), Dói et al. (1991), Ellis et al. (1993), Giger d MacMahon (1996) e Petrick et al. (1996) destacam a importância desses sistemas apresentando taxas de diagnósticos errados em programas de rastreamento e mostrando que a utilização de esquemas CAD pode melhorar o desempenho de radiologistas no diagnóstico médico.

No entanto, sistemas deste tipo, aplicados no dia-a-dia da prática médica, são poucos devido à necessidade de alto desempenho exigido por esta classe de sistema, tanto em nível de velocidade de execução quanto em nível de acerto nos resultados (NUNES, 2006b). Isso acontece, pois alguns tipos de erros não são admitidos, visto que esses resultados são utilizados na tomada de decisão, seja para diagnósticos ou escolha de tratamento.

Apesar disso, o avanço na aquisição, processamento e armazenamento de imagens médicas vêm permitindo o aperfeiçoamento de diagnósticos e tratamentos de doenças de naturezas diversas. Essas imagens podem ser de diversos tipos de modalidades, como Radiografia, Ultra-Sonografia (USa), Ressonância Magnética Nuclear (RMN), Tomografia Computadorizada (TC), entre outras (NUNES, 2006b).

Técnicas de processamento de imagens podem ser aplicadas com o objetivo de melhorar tais imagens e extrair delas informações úteis ao diagnóstico. Porém, no tratamento de imagens médicas, é necessário, antes de qualquer decisão, definir o objetivo a ser

alcançado. Somente após esta definição é possível traçar estratégias a partir da utilização de uma técnica de processamento, da combinação de várias delas ou, ainda, da criação de novas técnicas.

2.2. Definições Básicas

A formação da imagem, segundo Ballard e Brown (1982), ocorre quando um sensor registra a radiação que interagiu com objetos físicos. Portanto, a imagem é uma representação do objeto físico podendo, de acordo com as necessidades do interessado, ser armazenada, manipulada e interpretada.

No espaço bidimensional a imagem é matematicamente representada por $f(x,y)$, sendo x e y as coordenadas espaciais e f a função da intensidade luminosa, pois indica o brilho da imagem no ponto especificado. Nas imagens digitais, a representação é formada por um vetor de valores discretos e os valores de cada *pixel* são quantificados em um número de uma escala de variados níveis de cinza, onde é atribuído zero à cor mais escura (preto) e o maior valor se refere à cor mais clara da escala (branco).

Dessa forma, pode-se representar uma imagem como uma matriz onde cada ponto é um valor discreto, conforme mostra a equação 2.1, onde n e m correspondem à quantidade de colunas e linhas, respectivamente.

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \ldots & f(0,n-1) \\ f(1,0) & f(1,1) & \ldots & f(1,n-1) \\ . & & & \\ . & & & \\ . & & & \\ f(m-1,0) & f(m-1,1) & \ldots & f(m-1,n-1) \end{bmatrix} \quad (2.1)$$

O objetivo de se definir matematicamente a imagem é a possibilidade de manipulação de seu conteúdo a fim de transformá-la ou retirar dela informações importantes. Em Gonzalez (2002), o vasto conjunto de operações que se pode aplicar em uma matriz que representa uma imagem é denominado “Processamento de Imagens”.

Cada ponto ou elemento constituinte da matriz-imagem é chamado de *pixel*, abreviação de *picture element*. A medida de um *pixel* depende da resolução espacial com a

qual a imagem foi adquirida. O *pixel* é, então, a menor unidade sobre a qual podem-se realizar operações. Para essas operações são necessárias as definições de alguns conceitos básicos, apresentadas na seção seguinte.

2.3. Conceitos Essenciais

Esta seção apresenta alguns conceitos fundamentais, uma vez que as técnicas de processamento de imagens realizadas nesse projeto utilizam-se desses conceitos.

2.3.1. Vizinhança de um *Pixel*

Segundo Gonzalez e Woods (2002), a vizinhança de um *pixel p* nas coordenadas *(x,y)* é definida da seguinte forma:

- Vizinhança-de-4 de *p* (horizontal e vertical), representado por $N_4(p)$, é composta por seus vizinhos de coordenadas:

$$(x+1, y), (x-1, y), (x, y+1), (x, y-1)$$

- Vizinhança diagonal de *p*, denotada por $N_D(p)$, é composta por seus vizinhos de coordenadas:

$$(x+1, y+1), (x+1, y-1), (x-1, y+1), (x-1, y-1)$$

- Vizinhança-de-8 de *p* ou $N_8(p)$ é composta pelo conjunto de todos os *pixels* vizinhos, ou seja, pela vizinhança-de-4 mais a vizinhança diagonal $(N_4(p) \chi N_D(p))$.

Na Figura 2.1 são apresentados exemplos dos conceitos de vizinhança citados, em que os *pixels* da vizinhança são representados pelos espaços em cinza.

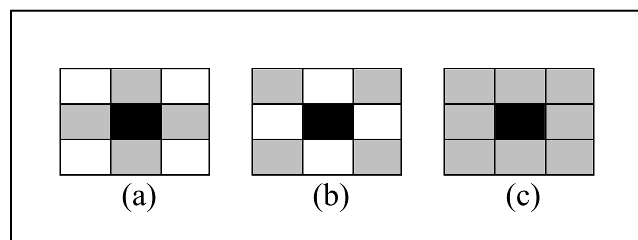


Figura 2.1 - Representação da vizinhança de um *pixel* considerando o ponto central como o *pixel* de interesse: (a) vizinhança-de-4; (b) vizinhança diagonal; (c) vizinhança-de-8.

2.3.2. Conectividade entre *Pixels*

A conectividade entre *pixels* é um conceito importante utilizado no estabelecimento das bordas de objetos e componentes de regiões em uma imagem (GONZALEZ, 2002). Dois *pixels* estão conectados se:

- São adjacentes (vizinhos-de-4);
- Seus níveis de cinza satisfazem um certo critério de similaridade, isto é, seus valores estão dentro de um conjunto pré-estabelecido de valores de cinza.

Seja $V = \{G1, G2, ..., GK\}$ o conjunto de k valores de níveis de cinza utilizado para definir a conectividade. São definidos três tipos de conectividade:

- Conectividade-4: Dois *pixels* p e q com valores em V e q ε $N4(p)$;
- Conectividade-8: Dois *pixels* p e q com valores em V e q ε $N8(p)$;
- Conectividade-m: Dois *pixels* p e q com valores em V e:
  - q ε $N4(p)$ ou
  - q ε $ND(p)$ e $N4(p)$ 1 $N4(p)$ = ι.

2.3.3. Domínios: Espacial e da Freqüência

A manipulação de imagens digitais pode ser realizada no domínio da freqüência ou no domínio espacial. O domínio espacial refere-se ao conjunto de *pixels* que compõem a imagem. As técnicas de processamento de imagens que trabalham neste domínio são aplicadas diretamente nos *pixels* da imagem. As técnicas que trabalham no domínio da freqüência têm como princípio básico o teorema da convolução, efetuando-se alterações na transformada de Fourier da imagem (NUNES, 2006b).

O domínio da freqüência, em geral, exige uma maior complexidade matemática, porém o processamento é mais rápido. Já o domínio espacial não exige grande complexidade matemática, mas o processamento é mais lento, uma vez que muitas vezes várias operações são requeridas para obter-se o novo valor de um *pixel*.

A abordagem neste trabalho é relacionada ao domínio espacial, no qual os *pixels* são considerados como uma matriz de atributos numéricos, conforme definido no início da Seção 2.2.

2.3.4. *Templates*

Templates podem ser utilizados para a manipulação no domínio espacial. São definidos, segundo Gonzalez e Woods (2002), como máscaras utilizadas para a realização de operações na vizinhança de um *pixel*. A Figura 2.2 ilustra uma máscara de tamanho 3x3.

C_1	C_2	C_3
C_4	C_5	C_6
C_7	C_8	C_9

Figura 2.2 - Exemplo genérico de uma máscara 3x3

Pode-se observar pela Figura 2.2 que o *template* é uma matriz cujo elemento central é posicionado no *pixel* de interesse (representado por C_5). Sendo que as operações realizadas consistem basicamente na multiplicação dos elementos da vizinhança, incluindo o *pixel* em questão pelos valores indicados nas posições correspondentes da matriz. A soma dos resultados obtidos substitui o valor do *pixel* de interesse na imagem resultante.

Exemplificando, considera-se c_i, i=1,...,9 os coeficientes da máscara em questão e x_i, i=1,...,9 os valores dos *pixels* sob a máscara, o resultado que será atribuído ao *pixel* central será: $c_1x_1 + c_2x_2 + ... + c_9x_9$.

Um algoritmo genérico para aplicação de um *template* é apresentado na Figura 2.3.

```
defina tamanho_template
para linha = 1 ate quantidade_linhas
   para coluna = 1 ate quantidade_colunas
      soma ← 0
      para indLinha = Linha - tamanho_template/2 ate Linha + tamanho_template/2
         para indColuna=Coluna - tamanho_template/2 ate Coluna + tamanho_Template/2
            soma ← soma + pixelindLinha,indColuna
         fim para
      fim para
      pixellinha,coluna ← soma
   fim para
fim para
```

Figura 2.3 - Algoritmo genérico para aplicação de templates (NUNES, 2006b).

Introduzidos os conceitos anteriores, pode-se aplicá-los ao processamento de imagens.

2.4. Técnicas de Processamento de Imagens

As técnicas de processamento de imagens dividem-se em três níveis de processamento, cada qual com suas funções específicas: o baixo nível, o nível médio e o alto nível.

O processamento de baixo nível é formado por procedimentos que são aplicados na imagem com o intuito de melhorar a sua qualidade, remover dados indesejáveis e realçar os dados importantes (GONZALEZ, 2002). Como exemplo pode-se citar as técnicas: *threshold*, quantização, *splitting*, equalização, suavização, entre outros.

O processamento de nível médio reúne procedimentos para a extração e caracterização de componentes como a identificação de formas significantes em uma imagem, utilizando um conjunto de técnicas denominado Segmentação (NUNES, 2006b).

Por fim, o alto nível, que requer um conjunto de procedimentos para o reconhecimento de padrões e representação, sendo responsável pela ligação da imagem com algum banco de conhecimento.

As seções subseqüentes apresentam técnicas aplicadas aos dois primeiros níveis, uma vez que as implementações realizadas nesse projeto se enquadram nos mesmos.

2.4.1. Processamento de Nível Baixo – Suavização de Imagens

Operações de suavização são usadas para diminuir efeitos resultantes do processo de aquisição da imagem (ruídos, por exemplo). Várias técnicas podem ser definidas para esta finalidade, como o filtro de média e mediana. Nessa seção será apresentada a filtragem mediana, uma das técnicas mais clássicas e utilizada nesse trabalho, como também, seus efeitos sobre algumas imagens médicas.

2.4.1.1. Filtragem Mediana

A filtragem mediana é uma técnica de suavização na qual cada *pixel* da imagem final é substituído pelo nível de cinza mediano em uma vizinhança do *pixel*. O nível mediano m de

um conjunto de valores é tal que metade dos valores no conjunto são menores que *m* e a outra metade é constituída de valores maiores que *m* (GONZALEZ, 2002).

Sendo assim, para que se obtenha o valor desse nível, é necessária a ordenação de todos os *pixels* da vizinhança, o que é relativamente lento, dependente principalmente do método de ordenação utilizado.

Segundo Gonzalez (2002), o efeito de borramento desta técnica é menor quando comparado ao resultado fornecido pela técnica do filtro de média. Dessa forma, um dos algoritmos implementados nesse trabalho será o de filtro de mediana. Na Figura 2.4 é apresentado um algoritmo genérico para implementação da técnica.

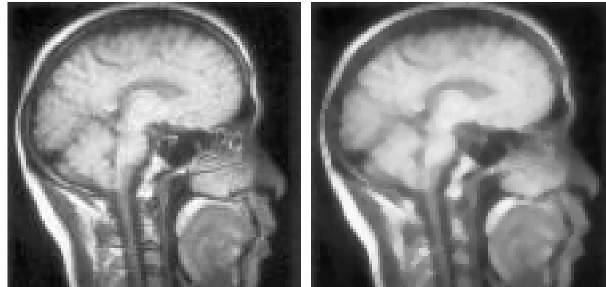
```
defina tamanho_template
para lin = 1 ate quant_linhas
   para col = 1 ate quant_colunas
      defina vet_elem_med com tam_template² elem
      para ind_lin = lin -tam_template/2 ate
         lin + tam_template/2
         para ind_col = col - tam_template/2 ate
         col + tam_template/2
         vet_elem_med ← pixels da vizinhanca
         fim para
      fim para
      ordena vet_elem_med
   fim para
fim para
```

Figura 2.4 – Algoritmo genérico para a implementação do Filtro de Mediana (NUNES, 2006a).

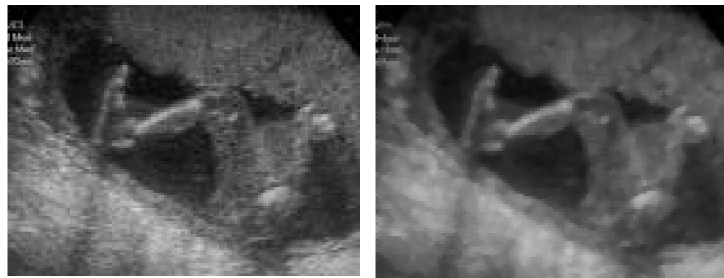
Uma das vantagens desta técnica é a eliminação de ruídos com o borramento mínimo das bordas que compõem os objetos representados na imagem. Assim as imagens mais beneficiadas são aquelas provenientes de ultra-som (US), que devem ser submetidas a um filtro de suavização devido ao ruído inerente da modalidade. Exemplos da aplicação da técnica, em algumas modalidades de imagens médicas citadas são apresentados na Figura 2.5.

Por apresentar um maior nível de ruído, devido à sua natureza, é possível observar, pela Figura 2.5, o efeito de suavização dos ruídos na imagem de US. Porém, nota-se também que outras estruturas foram suavizadas juntamente com os ruídos.

(a)



(b)



(c)

Figura 2.5 - Exemplos de suavização utilizando mediana da vizinhança. (a) imagem de Raios-X – original à esquerda e suavizada à direita; (b) imagem de RMN; (c) imagem de US. Os processamentos utilizaram máscara de tamanho 3x3 *pixels*.

2.4.2. Processamento de Nível Médio - Segmentação

Segundo Gonzalez e Woods (2002), segmentação é o processo que subdivide uma imagem em suas partes ou objetos constituintes. Ballard e Brown (1982) definem uma imagem segmentada como sendo o resultado do agrupamento de partes de uma imagem generalizada em unidades homogêneas considerando um ou mais aspectos. Portanto, segmentação é um conjunto de técnicas de processamento de imagens responsável por identificar as formas de interesse à pesquisa de uma imagem a fim de fornecer informações para a sua análise.

Para que a imagem possa ser analisada ela tem que estar segmentada em regiões, que são agrupamentos de *pixels* resultantes do processo de segmentação. Uma das tarefas mais

difíceis da visão computacional é realizar a segmentação de uma forma eficiente para todos os tipos de imagens. Isso porque em geral, as técnicas de segmentação são bastante limitadas, trazendo bons resultados para certo conjunto de imagens e falhando consideravelmente para outros tipos de imagens, de modo que uma segmentação não eficiente compromete todo o processamento posterior.

A seção seguinte aborda a técnica de detecção de bordas utilizada nesse nível e que será implementada nesse projeto.

2.4.2.1. Detecção de Bordas

Partindo da definição de borda como uma fronteira entre duas regiões com níveis de cinza relativamente distintos, os algoritmos utilizados para a detecção de bordas são estruturados de forma a detectar as descontinuidades existentes nas transições (NUNES, 2006b).

A detecção de bordas é outro exemplo de algoritmo que usa operações baseadas em vizinhança. Representar uma imagem por meio de suas bordas pode ser vantajoso, pois se reduz sensivelmente a quantidade de dados que estavam sendo armazenados na imagem sem perder muita informação. É útil principalmente quando se deseja conhecer informações a respeito de tamanho e forma dos objetos representados na imagem.

Existem vários algoritmos para detecção de bordas. Assim como os demais filtros no domínio espacial, esses algoritmos exigem uma série de operações aritméticas sobre o *pixel* e sua vizinhança. Para avaliação neste projeto foi utilizado o algoritmo de detecção de bordas fazendo uso dos operadores de Sobel horizontal e vertical (GONZALEZ, 2002). Porém houve algumas modificações nesse algoritmo e que são descritas nessa seção e de forma mais detalhada na seção 4.3.2.

Os operadores de Sobel executam operações na vizinhança do *pixel* fazendo com que o valor do *pixel* resultante seja diretamente proporcional à diferença existente entre o *pixel* e sua vizinhança. Assim, se o valor de um *pixel* for exatamente igual aos valores de seus vizinhos, o valor resultante no *pixel* processado será zero.

Para realizar estas operações são utilizadas máscaras de coeficientes. A Figura 2.6(a) ilustra a matriz de pontos da imagem. Além dos operadores de Sobel (máscaras horizontal e vertical), foram acrescentados, nesse algoritmo, as máscaras de 45º e de 135º. Tais máscaras são apresentadas, respectivamente, pela Figura 2.6(b), (c), (d) e (e). Para obter o valor do *pixel*

central, representado por X_5 na Figura 2.6(a), efetua-se a soma das multiplicações de cada *pixel* da vizinhança pelo coeficiente correspondente na máscara em questão, sendo atribuído ao *pixel* central X_5, a soma que apresentar maior valor.

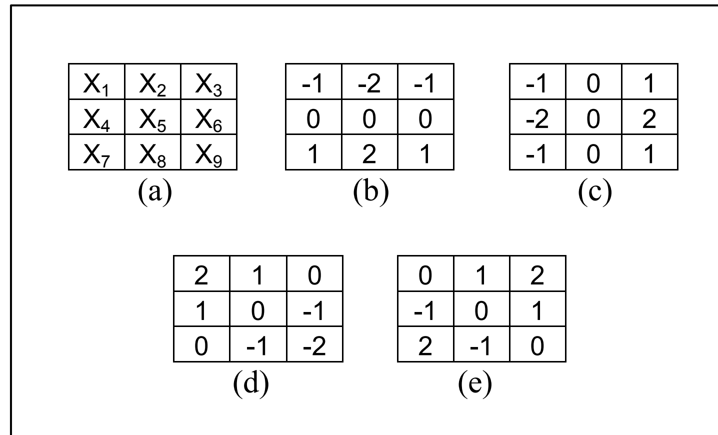


Figura 2.6 – Região e máscaras para detecção de bordas: (a) Região de *pixels*; máscara para detecção de bordas: (a) horizontais; (b) verticais; (c) de 45º; (d) de 135º.

No exemplo da Figura 2.6, considera-se uma vizinhança de 3x3 *pixels* em torno de um ponto central, representado por X_5 na Figura 2.6(a). As equações 2.2 a 2.5 definem os operadores horizontal e vertical (Sobel) e de 45º e 135º, respectivamente.

$$S_h = (x_7 + 2x_8 + x_9) - (x_1 + 2x_2 + x_3) \tag{2.2}$$

$$S_v = (x_3 + 2x_6 + x_9) - (x_1 + 2x_4 + x_7) \tag{2.3}$$

$$S_{45^o} = (x_2 + 2x_1 + x_4) - (x_6 + 2x_9 + x_8) \tag{2.4}$$

$$S_{135^o} = (x_2 + 2x_3 + x_6) - (x_4 + 2x_7 + x_8) \tag{2.5}$$

Onde:

- S_h, S_v, S_{45^o} *e* S_{135^o} são os valores resultantes dos operadores de detecção de bordas horizontal, vertical, de 45º e de 135º, respectivamente. Sendo que o maior valor, dentre estes quatros valores resultantes *(*S_h, S_v, S_{45^o} *e* S_{135^o}*)*, será atribuído ao *pixel* X_5;
- Os demais X_i*s* são os *pixels* pertencentes à vizinhança-de-8 de X_5.

Exemplos dos resultados obtidos com a aplicação desses operadores são apresentados na Figura 2.7.

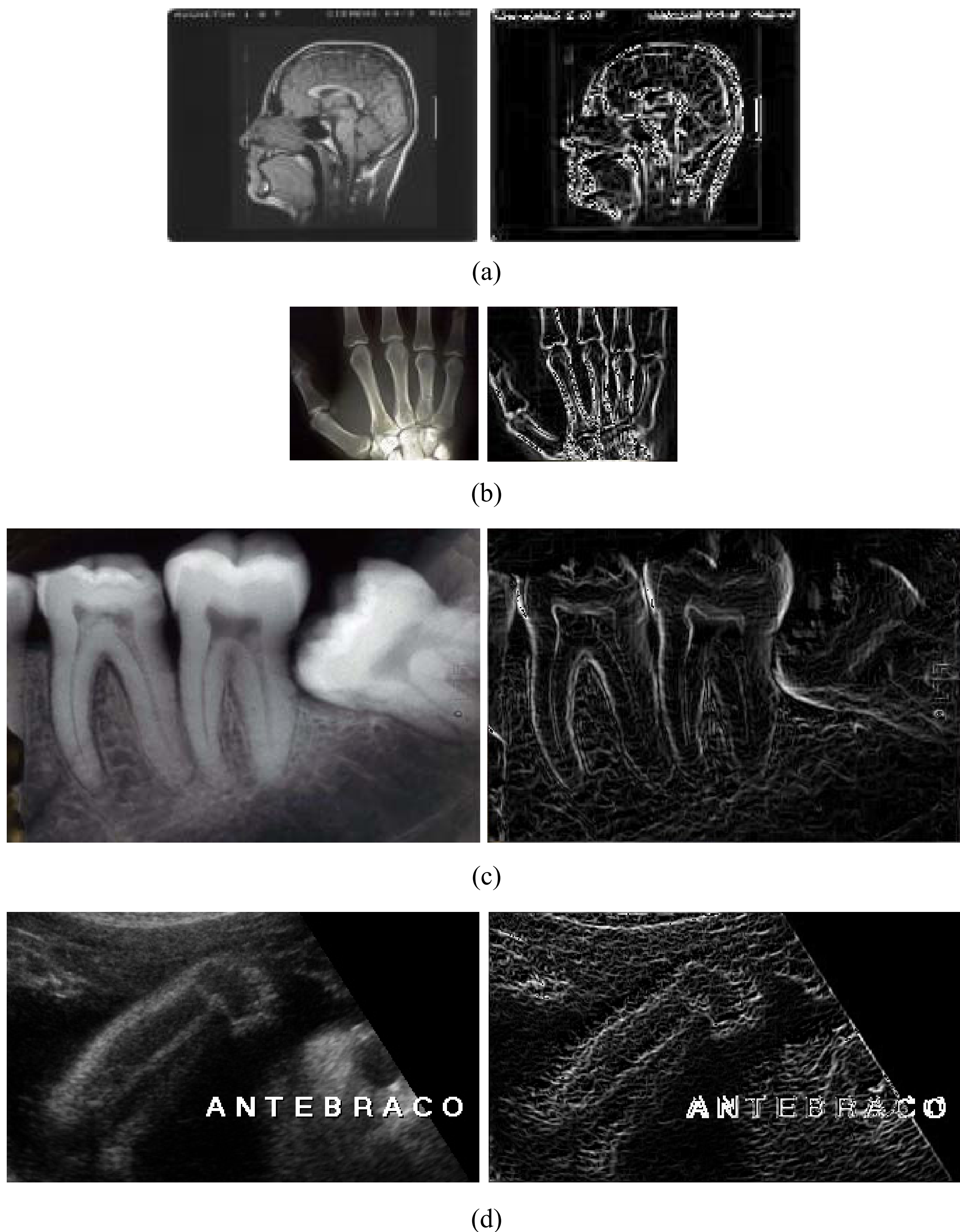


Figura 2.7 – Exemplos de detecção de bordas pelos operadores de Sobel modificado. (a) imagem de RMN – são apresentadas a imagem original e a resultante da detecção de bordas, respectivamente; (b) imagem de Raios-X; (c) imagem de Raios-X; (d) imagem de US. Os processamentos utilizaram máscara de tamanho 3x3 pixels.

Esta técnica apresenta simplicidade na aplicação pois não envolve complexidades computacionais. Entretanto, não apresenta boa imunidade de ruídos, pois a aplicação de máscaras os inclui nos cálculos aumentando efeitos como a suavização da imagem e, dessa

forma, a possível perda de definição de bordas, conforme observado em algumas imagens da Figura 2.7.

2.5. Considerações Finais

Este capítulo apresentou as técnicas a serem implementadas nesse projeto referentes aos níveis de processamento baixo e médio. O filtro de mediana é apresentado de forma conceitual, mostrando, dessa forma, como a imagem pode ser atenuada ou realçada visando à obtenção de melhores resultados no processamento intermediário. Neste processamento, de nível médio, é apresentada a técnica de detecção de bordas, cuja importância deve-se ao fato de que, ao se detectar uma borda é possível destacar os objetos de interesse de uma imagem, extraindo-se propriedades como formas e medidas.

As técnicas de processamento de imagens, entre outras especificidades computacionais, conforme mencionadas, são aplicadas com o objetivo de melhorar tais imagens e extrair delas informações úteis ao diagnóstico. O tempo de processamento computacional de uma imagem deve ser observado, principalmente quando são desenvolvidos sistemas com o objetivo de execução em tempo real.

O processamento no domínio espacial geralmente percorre a imagem e altera, de alguma forma, o valor de cada *pixel*. Quando uma imagem é grande, devido, principalmente à sua resolução espacial, deve-se otimizar o processamento a fim de torná-lo viável em tempo real.

É necessário o estabelecimento de algoritmos eficazes que possam fornecer, no menor tempo possível, um desempenho satisfatório em termos de acerto, a fim de contribuir, de fato, para o auxílio ao diagnóstico.

O desenvolvimento de sistemas de auxílio ao diagnóstico precisa vencer muitos desafios. Uma vez que as imagens médicas apresentam grande volume de dados para processamento e este processamento envolve, geralmente, milhares de operações em tempo real, é necessária a utilização de recursos computacionais que proporcionem alto desempenho. Por esses e outros motivos, ainda não foi divulgada a aplicação de sistemas de auxílio ao diagnóstico brasileiros em nível clínico (NUNES, 2006b).

A aplicação de técnicas de computação paralela é uma linha de pesquisa que pode ser promissora no sentido de auxiliar na minimização desta questão. O capítulo seguinte

apresenta o ambiente de passagem de mensagens como suporte para a computação paralela distribuída.

3. SUPORTE À COMPUTAÇÃO PARALELA DISTRIBUÍDA

A computação paralela sobre sistemas distribuídos exige uma camada de *software* que possa gerenciar o uso paralelo, pois existe a necessidade da passagem de informações entre as várias máquinas que compõem o ambiente.

Esse capítulo apresenta ambientes de passagem de mensagens existentes (que permitem a comunicação entre essas máquinas), as características e a descrição de algumas funcionalidades do ambiente de passagem de mensagens utilizado nesse trabalho, bem como a apresentação de trabalhos existentes na literatura.

3.1. Ambientes de Passagem de Mensagens

Ambientes de passagem de mensagens (ou interfaces de passagem de mensagens) são ambientes de programação paralela para memória distribuída portáteis, pois permitem o transporte de programas paralelos entre diferentes arquiteturas e sistemas distribuídos de maneira transparente (SANTANA, 1997). Tais ambientes provêem recursos necessários à programação paralela, como criação, comunicação e sincronização de processos.

Pode-se dizer também que são bibliotecas de comunicação que atuam como extensões de linguagens seqüenciais, como C e Fortran, possibilitando, dessa forma, a construção de aplicações paralelas (SANTANA, 1997).

As bibliotecas de passagem de mensagens mais utilizadas são o MPI (GROPP, 1995) (SNIR, 1996) e o PVM *(Parallel Virtual Machine)* (GEIST et al., 1994) (BEGUELIN, 1994). Estas bibliotecas provêem rotinas para iniciar e configurar o ambiente bem como enviar e receber mensagens de dados entre os elementos de processamento do sistema.

Ambas têm sido constantemente aperfeiçoadas, obtendo uma maior popularidade e aceitação, visto que proporcionam o desenvolvimento de aplicações paralelas a um custo relativamente baixo em relação às máquinas paralelas (BEGUELIN,1994) (SANTANA, 1997).

Além disso, os ambientes MPI e PVM, possibilitam flexibilidade, apresentando-se como uma solução para o problema da portabilidade de programas paralelos entre sistemas diferentes. Tais ambientes são discutidos nas seções seguintes.

3.1.1. PVM

Parallel Virtual Machine é um ambiente paralelo virtual caracterizado como uma das plataformas mais utilizadas (MCBRYAN, 1994), sendo considerado um padrão “de fato”, dado a sua utilização e popularidade em diversos setores (acadêmico, industrial, comercial, entre outros) (SUNDERAM, 1994) (GEIST et al., 1994).

Seu surgimento ocorreu em 1989 no ORNL (*Oak Ridge National Laboratory*), permitindo que um grupo de computadores de diferentes arquiteturas fosse conectado, formando assim, uma máquina paralela virtual (GEIST et al., 1994).

O PVM é composto por um conjunto de bibliotecas e ferramentas, que tem por objetivo a emulação de um sistema computacional concorrente heterogêneo, flexível e de propósito geral (BEGUELIN, 1994). Permite o desenvolvimento eficiente de programas paralelos e a manipulação transparente de mensagens por meio de uma rede de computadores com arquiteturas incompatíveis (GEIST et al., 1994).

Pode-se dividir o PVM em duas partes principais: a primeira é constituída por um conjunto de processos *daemon*, denominado *pvmd3* ou *pvmd*, que são executados em todos os elementos de processamento, a fim de: formar uma máquina paralela virtual, realizar a comunicação entre os processos criados e coordenar as tarefas em execução. A segunda parte do PVM é uma biblioteca de programação que trata funções básicas para: geração do paralelismo como troca de mensagens, criação e eliminação de processos, sincronização de tarefas, modificação da máquina virtual e envio e recebimento de mensagens (SUNDERAM; GEIST, 1999).

3.1.2. MPI

A existência de uma variedade de bibliotecas de passagem de mensagens, tornava difícil a construção de programas paralelos portáveis. Assim, fez-se necessário o estabelecimento de uma interface padrão para a implementação de ambientes de passagem de mensagens (SNIR et al., 1996).

Message Passing Interface é um padrão internacional de interface para troca de mensagens em máquinas paralelas com memória distribuída (GROPP, 1994) (FOSTER,

1995). Nesse padrão, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Sendo assim, a comunicação e a sincronização dos processos são garantidas.

Diversas versões do MPI foram publicadas. A versão 1.0 foi apresentada em 1993, tendo como exemplos de implementações o MPICH e o LAM. Em 1997, correções e novas funcionalidades, como a criação dinâmica de processos, E/S paralela e comunicação unilateral, foram acrescentadas à versão inicial, originando o MPI 2.0 (MPI, 1997).

O MPI baseia-se nas melhores características de todos os ambientes de passagem de mensagens e tenta explorar as vantagens de cada um deles. Programas escritos em MPI tendem a ser mais eficientes que os escritos em PVM, pelo fato de não haver sobrecarga na carga de processos em tempo de execução. Porém, há a necessidade de se explicitar a criação das tarefas, suas comunicações e destruição. Isso pode ser feito por meio de algumas funções.

As funções básicas da biblioteca de troca de mensagens têm diversas variações dentre as diferentes implementações existentes, sejam elas comerciais ou de domínio público. As implementações de domínio público mais comuns são LAM/MPI (BURNS et al., 1994; LAM, 2006) e MPICH (NEVIN, 1996).

Existem muitas implementações de MPI em aplicações desenvolvidas em linguagens como Fortran, C e C++. Com o surgimento de Java, inúmeras propostas foram apresentadas para a utilização dessas bibliotecas nessa linguagem na qual pode-se citar o mpiJava (BAKER, 1998), JMPI (MORIN, 2001), PJMPI (WEIQIN, 2000), JavaMPI (MINTCHEV, 1997), entre outros. Sendo o mpiJava o ambiente de passagem de mensagens utilizado neste trabalho, o mesmo será abordado em detalhes na próxima seção.

3.2. mpiJava

A escolha desse ambiente para o processamento paralelo dos algoritmos vem em decorrência do uso da linguagem de programação Java que contém fatores como portabilidade, permitindo a independência de plataforma, simplicidade, clareza nos códigos e a existência de APIs especializadas que possibilitam o uso, cada vez maior, desta linguagem em processamento de imagens (exemplo, utilização da API JAI nesse trabalho).

Além disso, dentro das formas de comunicação possíveis em Java, a biblioteca apresenta bons resultados (BAKER, 1999). O uso de mpiJava já foi validado em diversas implementações e avaliações de desempenho (TABOADA, 2003).

O mpiJava é uma interface, amplamente utilizada na computação paralela e distribuída, que permite fazer uso da orientação a objetos em Java juntamente com a biblioteca MPI (BAKER, 1998). Para tanto, as chamadas dos métodos obedecem à estrutura das funções definidas em MPI, o que torna a programação menos flexível. Também a portabilidade é atingida, uma vez que a chamada das funções MPI é específica para uma determinada arquitetura.

Em um nível de abstração mais baixo, mpiJava executa as funções nativas de uma implementação MPI, conforme a definição feita no momento da instalação.

Nesse trabalho as versões mpiJava e MPI utilizadas foram *mpiJava-1.2.5* (MPIJAVA, 2007) e *MPICH2-1.0.5* (MPICH2, 2006). As instalações e configurações de ambos encontram-se em (APÊNDICE B) e (APÊNDICE D), respectivamente.

O padrão mpiJava é baseado em objetos, seguindo o modelo de especificação em C++. Provê acesso a uma implementação nativa do MPI por meio do JNI (*Java Native Interface*). A hierarquia de classes, organizada seguindo a hierarquia de classes do C++, está definida na especificação do MPI-2 e consiste de seis classes principais: *MPI, Group, Comm, Datatype, Status e Request*, apresentadas na Figura 3.1.

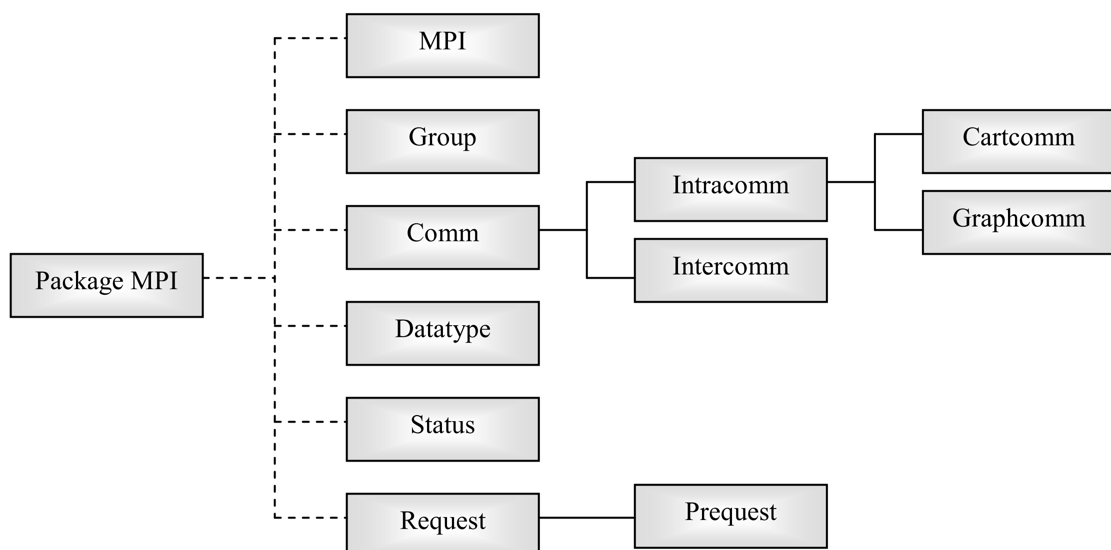


Figura 3.1 - Principais classes do mpiJava (BAKER, 1999)

A classe MPI possui apenas membros estáticos e é responsável pela realização de serviços globais, como a inicialização do MPI. A classe mais importante existente nesse pacote é a classe comunicadora *Comm*, pois todas as funções de comunicação em mpiJava são realizadas por objetos dessa classe ou de suas subclasses.

Como em MPI, os processos "existem" dentro dos chamados comunicadores. Dois processos podem trocar mensagens se, e somente se, eles pertencem ao mesmo comunicador. Todos os processos são automaticamente associados a um comunicador padrão, MPI.COMM_WORLD. Cada processo em um comunicador recebe uma identificação única denominada *rank* ou *id*, utilizada para especificar a origem e o destino das mensagens, no caso do processo pertencer a mais de um comunicador ele recebe uma identificação *(rank)* diferente em cada um deles.

Uma outra classe de extrema importância é a classe *Datatype*, responsável por descrever os tipos dos objetos dos *buffers* de mensagens passados para envio, recebimento ou qualquer outra função que envolva comunicação. Vários tipos de dados básicos estão pré-definidos no pacote. A correspondência entre estes e os tipos primitivos de Java podem ser observados na Tabela 3.1.

Tabela 3.1 - Tipos de dados básicos em mpiJava.

Tipo MPI	**Correspondente em Java**
MPI.BYTE	Byte
MPI.CHAR	Char
MPI.SHORT	Short
MPI.BOOLEAN	Boolean
MPI.INT	Int
MPI.LONG	Long
MPI.FLOAT	Float
MPI.DOUBLE	Double
MPI.PACKED	

3.2.3. Rotinas de Gerenciamento do Ambiente mpiJava

A principal finalidade de gerenciamento é iniciar e finalizar o ambiente mpiJava, entretanto, deve-se destacar também a importância das demais rotinas. Dessa forma, apresentam-se nesta seção algumas das principais rotinas de gerenciamento encontradas nesse ambiente.

As rotinas *MPI.Init(args)* e *MPI.Finalize()* são utilizadas, respectivamente, para iniciar e finalizar uma execução mpiJava. A rotina *MPI.Init(args)* deve ser chamada antes de qualquer rotina mpiJava ser acionada por cada processador. Depois que a rotina *MPI.Finalize()* é acionada, nenhuma outra poderá ser chamada ou acessada. Para verificar se o MPI foi realmente iniciado pode-se utilizar a rotina *MPI.Initialized()*.

A rotina *MPI.COMM_WORLD.Size()* determina o número de processos utilizados na aplicação e a *MPI.COMM_WORLD.Rank()* os identifica, devolvendo o identificador *(id)* do processo que executou a rotina utilizando um número inteiro, em que 0 # *id* # *numprocs* – 1. Já os processos pertencentes a um grupo, são identificados com um número inteiro precedido de um zero.

MPI.Get_processor_name() pode ser utilizado para retornar o nome do processador em que ele é chamado.

O empacotamento e desempacotamento dos dados podem ser realizados pelas seguintes rotinas, respectivamente:

```
int MPI.COMM_WORLD.Pack (Object inbuf, int offset, int incount, Datatype datatype, byte [] outbuf, int position)
```

Em que:

- *inbuf* deve ser um *array* de entrada com tipos de elementos primitivos;
- *offset* indica o elemento no *array* de entrada onde a mensagem começa;
- *incount* especifica o número de elementos, no *array* de entrada, a ser enviado;
- *datatype* corresponde ao tipo do dado do *array* de entrada;
- *outbuf* especifica o *array* de saída;
- *position* indica a posição inicial no *array* de saída.

O retorno da função de empacotamento é a posição final (o valor inicial incrementado pelo número de *bytes* escritos) no *array* de saída.

```
int MPI.COMM_WORLD.Unpack (byte [] inbuf, int position, Object outbuf, int offset, int outcount, Datatype datatype)
```

Em que:

- *inbuf* deve ser um *array* de entrada com tipos de elementos primitivos;
- *position* indica a posição inicial no *array* de entrada;
- *outbuf* especifica o *array* de saída;
- *offset* indica o elemento no *array* de saída onde a mensagem começa;

- *outcount* especifica o número de elementos no *array* de saída;
- *datatype* corresponde ao tipo do dado do *array* de saída;

O retorno da função de desempacotamento é a posição final (o valor inicial incrementado pelo número de *bytes* lidos) no *array* de entrada.

Outras rotinas de gerenciamento do ambiente podem ser encontradas em (BAKER, 1998).

3.2.4. Rotinas de Comunicação no mpiJava

Uma mensagem no mpiJava, como no MPI, é definida como um vetor de elementos de um determinado tipo. Ao ser enviada, deve-se indicar o endereço do primeiro elemento desse vetor e o número de elementos que o formam. O tipo desses elementos deve ser o mesmo e também deve ser indicado no envio.

O mpiJava também permite que se criem tipos definidos pelo usuário, tornando possível enviar mensagens compostas por elementos de tipos distintos, como por exemplo, estruturas *(structs)* (MACDONALD, 1996).

As funções de comunicação (ponto-a-ponto e coletiva) formam o núcleo básico do MPI (DONGARRA, 1995) (MACDONALD, 1996) (SNIR, 1996) (WALKER, 1994). O mpiJava realiza as funções de MPI como método da classe *Comm*.

3.2.4.1. Rotinas de Comunicação Ponto-a-Ponto

Na comunicação ponto-a-ponto, a transmissão da mensagem requer a participação de dois processos, um enviando e outro recebendo uma mensagem, realizando, respectivamente, um *send* (transmissor) e um *receive* (receptor).

Uma grande quantidade de rotinas ponto-a-ponto é oferecida pelo MPI. Elas se diferem em dois aspectos: modo de comunicação e bloqueio ou não bloqueio, sendo que para cada um desses existe uma rotina.

Modo de Comunicação

O modo de comunicação define o processamento de transmissão de uma mensagem e o critério que determina quando esta transmissão é considerada completada (NCSA, 2006). Para envio *(send)* existem quatro modos ou rotinas: padrão, síncrono, bufferizado e pronto. Para recepção *(receive)* existe apenas um. Com qualquer um dos modos de envio, quando o *send* é completado significa que o *buffer* de transmissão (posições de memória do processo transmissor onde estão os dados que estão sendo transmitidos) pode ser reutilizado com segurança. No modo de recepção, quando o *receive* é completado significa que o *buffer* de recepção (posições de memória do processo receptor onde são armazenados os dados recebidos) já contém os dados e eles estão disponíveis para uso.

Em uma comunicação, qualquer tipo de *send* e *receive* pode ser combinado. Os quatro modos de envio são descritos a seguir (NCSA, 2006; SOUZA, 1996; SOUZA, 1997):

- **Padrão *(standard)*:** o processo que está enviando a mensagem só pode continuar a execução de suas instruções depois que o *buffer* da aplicação existente no processo emissor estiver livre para ser reutilizado. É o modo mais eficiente de comunicação, podendo ser síncrono ou com *buffer*.
- **Bufferizado *(buffered)*:** os dados são copiados e armazenados em um *buffer*, definido pelo programador, até serem transmitidos.
- **Síncrono *(synchronous)*:** o processo *send* espera uma confirmação de recepção da mensagem, ficando bloqueado até que o *buffer* da aplicação do processo emissor esteja livre para ser reutilizado e que o processo receptor já tenha começado a receber a mensagem.
- **Pronto *(ready)*:** é necessário que o *receive()* correspondente tenha sido iniciado. Dessa forma, deve existir um protocolo de confirmação para o processo emissor do estado atual do processo receptor. Sendo assim, se este estiver pronto para receber a mensagem o emissor inicia a transmissão, caso contrário, fica esperando até que o outro processo esteja apto a receber os dados.

As rotinas utilizadas para os *sends* padrão, bufferizado, síncrono e pronto são, respectivamente, *MPI.COMM_WORLD.Send,* *MPI.COMM_WORLD.Bsend*, *MPI.COMM_WORLD.Ssend* e *MPI.COMM_WORLD.Rsend*. Já para o *receive* é

MPI.COMM_WORLD.Recv. Todas as rotinas de comunicação ponto-a-ponto, tanto de envio como de recepção seguem basicamente um mesmo padrão para os parâmetros utilizados durante a comunicação, conforme podem ser observadas na Tabela 3.2.

Além dessas rotinas de comunicação ponto-a-ponto apresentadas outras variantes também existem tais como rotinas de comunicação em dois sentidos e requisições persistentes. Detalhes dessas variantes podem ser encontrados em (CARPENTER et al., 1999).

Comunicação Bloqueante e Não-Bloqueante

Na comunicação bloqueante, o transmissor ou receptor fica bloqueado até que o envio ou recepção da mensagem seja completado. Assim, o código que sucede a chamada de uma rotina bloqueante só é executado quando ela tiver sido completada. Já na comunicação não-bloqueante, ele é executado imediatamente após o início da comunicação, possibilitando a sobreposição de computação e comunicação e conseqüentemente resultando em maior desempenho.

Para todos os modos de envio e para o modo de recepção, descritos na seção anterior, o MPI oferece versões bloqueantes correspondentes às rotinas apresentadas, e não bloqueantes, conforme pode ser observado na Tabela 3.3.

Tabela 3.2 - Rotinas de comunicação ponto-a-ponto bloqueantes e não bloqueantes do mpiJava

Tipo	Função Bloqueante	Função Não Bloqueante
SEND padrão	MPI.COMM_WORLD.Send	MPI.COMM_WORLD.Isend
SEND síncrono	MPI.COMM_WORLD.Ssend	MPI.COMM_WORLD.Issend
SEND bufferizado	MPI.COMM_WORLD.Bsend	MPI.COMM_WORLD.Ibsend
SEND pronto	MPI.COMM_WORLD.Rsend	MPI.COMM_WORLD.Irsend
RECEIVE	MPI.COMM_WORLD.Recv	MPI.COMM_WORLD.Irecv

Todas as rotinas de envio e recebimento não-bloqueantes têm os mesmos parâmetros das correspondentes bloqueantes. A Tabela 3.3 apresenta as operações de comunicação ponto-a-ponto mpiJava. Em seguida, cada um de seus parâmetros é definido:

Tabela 3.3 - Operações de comunicação ponto-a-ponto mpiJava

Operação	Sintaxe das Primitivas mpiJava
Send Padrão Bloqueante	MPI.COMM_WORLD.Send (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Síncrono Bloqueante	MPI.COMM_WORLD.Ssend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Bufferizado Bloqueante	MPI.COMM_WORLD.Bsend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Pronto Bloqueante	MPI.COMM_WORLD.Rsend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Padrão Não Bloqueante	MPI.COMM_WORLD.Isend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Síncrono Não Bloqueante	MPI.COMM_WORLD.Issend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Bufferizado Não Bloqueante	MPI.COMM_WORLD.Ibsend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Send Pronto Não Bloqueante	MPI.COMM_WORLD.Irsend (Object buf, int offset, int count, Datatype datatype, int dest, int tag);
Receive Bloqueante	MPI.COMM_WORLD.Recv (Object buf, int offset, int count, Datatype datatype, int source, int tag);
Receive Não Bloqueante	MPI.COMM_WORLD.Irecv (Object buf, int offset, int count, Datatype datatype, int source, int tag);

- ***buf***: nas primitivas de envio de mensagem, é o espaço de endereçamento alocado pelo programa para armazenar os dados que serão enviados. Nas primitivas de recebimento, é o espaço alocado para armazenar os dados recebidos. *buf* deve ser um *array* com tipos de elementos primitivos;
- ***offset***: indica o elemento no *array* onde a mensagem começa;
- ***count***: especifica o número de elementos, do tipo *datatype*, a ser enviado ou recebido, sendo útil quando se quer transmitir ou receber vários elementos de uma só vez, um vetor, por exemplo;
- ***datatype***: corresponde ao tipo do dado a ser enviado ou recebido, podendo ser um dos tipos definidos pelo MPI, conforme apresentado pela Tabela 3.1 ou um tipo de dado definido pelo usuário;
- ***dest***: indica qual processo receberá a mensagem enviada. Isso é feito especificando o *rank* ou *id* do processo destino (receptor);
- ***source***: indica qual processo enviou a mensagem. De maneira semelhante ao parâmetro *dest*, isso é feito especificando o *rank* ou *id* do processo origem (emissor);
- ***tag***: é um número inteiro não negativo escolhido pelo programador para identificar uma mensagem, distinguindo-a, dessa forma, de um mesmo

transmissor. O parâmetro *source* indica o processo emissor da mensagem e o *tag* indica qual mensagem será recebida.

Em um processo X, por exemplo, podem chegar mensagens provenientes de quaisquer outros processos, em qualquer ordem. Ao especificar, no *receive*, o *rank* ou *id* do processo de origem e um *tag*, somente a mensagem que combinar com os valores especificados por estes dois parâmetros será recebida. Quaisquer outras mensagens serão ignoradas e armazenadas para um recebimento posterior por meio de um *receive* que combine com os seus parâmetros.

3.2.4.2. Rotinas de Comunicação Coletiva

A comunicação coletiva envolve um grupo de processos e é especificado por meio de um comunicador. Em seus parâmetros são especificados quais processos participarão das trocas de mensagens. Por *default*, o MPI considera que todos os processos existentes estão envolvidos nas comunicações. Porém o programador pode definir um conjunto de processos de acordo com suas necessidades.

Existe uma variedade de rotinas coletivas, tais como: *broadcast*, aritméticas globais e sincronização. Além disso, existem também variantes dessas funções, como: *gather*, *scatter* e a combinação dessas duas. Essa grande variedade se explica pelo fato de se garantirem operações coletivas eficientes em todas as plataformas paralelas. Todas essas operações são bloqueantes e executam no modo padrão.

A seguir, cada uma das operações, mostradas na Tabela 3.4, são definidas (DONGARRA, 1995) (MACDONALD, 1996) (SNIR, 1996):

Tabela 3.4 - Operações de Comunicação Coletivas mpiJava

Operações	Sintaxe das Primitivas mpiJava
Broadcast	MPI.COMM_WORLD.Bcast (Object buf, int offset, int count, Datatype datatype, int root);
Scatter	MPI.COMM_WORLD.Scatter (Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root);
Gather	MPI.COMM_WORLD.Gather (Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root);
Allgather	MPI.COMM_WORLD.Algather (Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype);

- ***broadcast*:** é uma comunicação do tipo “1-para-todos”, em que o processo, chamado raiz do *broadcast*, transmite os dados armazenados em seu *buffer*, para todos os processos do mesmo grupo. Sendo que a quantidade de dados enviados pelo processo origem (emissor) deve ser exatamente a mesma recebida por cada um dos processos receptores;
- **s*catter*:** cada processo integrante do grupo, incluindo o raiz, recebe desse processo (*root)*, uma porção diferente do dado armazenado no buffer. A distribuição dos dados ocorre de acordo com a ordem de *rank*;
- **g*ather*:** ou coleta é a comunicação do tipo “todos-para-1”, em que os dados no *buffer* de transmissão de cada um dos processos de um comunicador são enviados para um único processo chamado raiz de coleta, que armazena estes dados no seu *buffer* de recepção em ordem de *rank*;
- **a*llgather*:** é semelhante à rotina g*ather*, no entanto, todos os processos do grupo recebem os dados.

Outras operações de comunicação coletiva, incluindo variantes de g*ather* e s*catter*, são oferecidas pelo MPI, permitindo, dessa forma, a distribuição e coleta de partes de diversos tamanhos para processos diferentes. Elas podem ser encontradas de forma detalhada em (CARPENTER et al., 1999).

3.3. Trabalhos Correlatos

A seguir apresenta-se uma rápida descrição de alguns trabalhos de importância relevante para as áreas de programação paralela e processamento de imagens e para o desenvolvimento dessa proposta de trabalho.

Bosque *et al.* (2000) apresentam a implementação de um sistema paralelo CBIR. Esse sistema manipula um banco de dados de imagens com cerca de 29 milhões de imagens bidimensionais RGB. Devido ao enorme volume de dados armazenados e a necessidade em se ganhar tempo no acesso e armazenamento desses dados, a aplicação foi destinada a um ambiente de memória distribuída e foi implementada em um cluster de 25 computadores. A estratégia de otimização consiste em um processo mestre distribuir os dados a serem processados entre os processos escravos. Cada um deles processa os dados e retornam os resultados parciais para o mestre à medida que terminam o processamento. Para a

comunicação das primitivas entre processos mestre e escravos foi utilizada a biblioteca MPI. As vantagens apresentadas pelo sistema CBIR foram a boa relação custo benefício e o alto nível de escalabilidade. O tempo de resposta diminuiu de cerca de 24 horas para 1 hora.

Skeletons algorítmicos, abstrações algorítmicas, foram desenvolvidos por Nicolescu *et al*. (2000) para a criação de um ambiente de processamento de imagens paralelas prontas para uso e para uma fácil implementação e desenvolvimento de aplicações de processamento de imagens paralelas. Estes *skeletons* foram implementados na linguagem de programação C, utilizando a biblioteca MPI. Obteve-se uma maior eficiência na utilização de paralelismo de ambos dados e tarefas quando comparadas ao uso de paralelismo somente de dados.

Giess *et al*. (1998) apresentam a implementação de um sistema paralelo para segmentação e visualização interativa de imagens médicas tridimensionais que são distribuídas em uma estação de clusters heterogêneos ou computadores pessoais. O programa foi implementado em ANSI-C, oferecendo uma alta performance. O uso de MPI e POSIX *Threads* permitiu a interoperabilidade em ambientes heterogêneos. Com o sistema proposto conseguiu-se a integração de vários algoritmos de processamento e visualização de imagens em um esquema de paralelização, evitando a redistribuição repetitiva de um grande volume de dados. O tempo de processamento reduzido obtido foi importante para a aceitação em uso clínico.

O SIAPDI, serviço integrado de acesso e processamento distribuído de imagens, foi desenvolvido por Barros Junior et al. (2001), permitindo a equipes médicas radiológicas acessarem um banco de imagens em conformidade ao padrão DICOM 3.0 *(Digital Communications in Medicine)*, provendo processamento de imagens de alto desempenho por um baixo custo. Equipamentos localizados na rede, independentemente do sistema operacional, podem ser utilizados para executar o processamento distribuído de imagens, repercutindo em uma redução no tempo final do processamento. Para o desenvolvimento e implementação do SIAPDI foi utilizada a tecnologia de objetos distribuídos baseado no padrão CORBA (*Common Object Request Broker Architecture*).

Pezzi (2005) aborda a paralelização do filtro de detecção de bordas. Nesse trabalho foi utilizada uma aplicação já existente, para verificar as dificuldades de paralelização dessa aplicação. A implementação no ambiente MPI-2 da biblioteca LAM, permitiu uma implementação simples da versão paralela do filtro a partir de uma aplicação seqüencial. Atingiu-se um *speedup* máximo utilizando seis máquinas. Porém, no aumento do número de nós, observou-se uma diminuição da eficiência, devido à sobrecarga do servidor NFS, que no caso era acessado concorrentemente pelos processos.

Schnorr (2003) propõe a paralelização de um filtro de mediana para imagens com ruídos do tipo “sal e pimenta” em um aglomerado de computadores. Com o objetivo de explorar todo o paralelismo disponível em um ambiente de execução paralelo, a implementação do filtro de mediana foi feito em etapas, explorando-se o paralelismo com passagem de mensagens em memória distribuída (utilizando a biblioteca MPI) e em memória compartilhada (utilizando a biblioteca OpenMP). As medidas de desempenho neste trabalho mostraram melhora no tempo de execução paralelo do algoritmo. No entanto, após o acréscimo de um determinado número de elementos de processamento a melhora no desempenho tornou-se pequena ou em alguns casos até negativa.

3.4. Considerações Finais

Para a realização da computação paralela sobre sistemas distribuídos é necessária a comunicação entre as várias máquinas que compõem o sistema. Existem diversos ambientes de passagens de mensagens especializados em fornecer recursos necessários à programação paralela tais como gerenciamento de recursos, comunicação, sincronização e gerenciamento das aplicações paralelas.

Os ambientes de passagem de mensagens mais populares e mais discutidos na literatura são o MPI e o PVM. A absoluta maioria dos programas que utilizam passagem de mensagens é escrita utilizando um desses ambientes (LASTOVETSKY, 2003) devido às vantagens apresentadas por estes, uma delas a portabilidade das aplicações neles geradas, permitindo sua execução em diferentes sistemas computacionais paralelos.

Esses ambientes permitem a escrita de aplicações em diversas linguagens. A escolha da linguagem de programação Java, nesse trabalho, deve-se às vantagens que ela oferece como portabilidade, simplicidade, clareza de código e a existência de APIs especializadas, como a API JAI, possibilitando a utilização cada vez maior desta linguagem em processamento de imagens.

Além disso, o mpiJava define um conjunto de rotinas que oferece diversos serviços: comunicação ponto-a-ponto, comunicação em grupo, suporte a grupos de processos, suporte a contextos de comunicação, entre outros.

Com base nas informações apresentadas nos capítulos anteriores, é proposto o modelo de paralelismo para a otimização no processamento de imagens médicas.

4. METODOLOGIA

Neste capítulo são apresentadas as características do desenvolvimento do projeto, tais como: tecnologias utilizadas, o modelo de paralelismo para o processamento de imagens médicas e a implementação paralela de cada algoritmo estudado.

4.1. Tecnologias Utilizadas

Para o desenvolvimento do projeto, foram utilizados, além do Laboratório de Arquitetura de Sistemas (LAS), os laboratórios disponíveis no UNIVEM com os devidos *softwares* necessários instalados para efetuar os testes de desempenho.

Para a avaliação dos estudos de casos foram utilizadas a biblioteca de passagem de mensagens mpiJava e a linguagem de programação Java como hospedeira para a biblioteca.

Foi utilizado o sistema operacional Linux, onde foi instalado e configurado o ambiente paralelo distribuído e também efetuada a implementação e as avaliações de desempenho dos algoritmos de processamento de imagens implementados. Os códigos-fonte dos algoritmos filtro de mediana e detecção de bordas Sobel, apresentam-se no Apêndice F.

Para a implementação dos algoritmos de processamento de imagens, de forma seqüencial e paralela, foram utilizados: o compilador da linguagem Java JDK (*Java Development Kit*) versão JDK-1_5_0_09; a API *(Application Program Interface)* JAI versão jai-1_1_3-beta; os *softwares* MPI versão mpich2-1.0.5 e mpiJava versão mpiJava-1.2.5. Suas instalações e configurações apresentam-se de forma detalhada nos Apêndices C, E, D e B, respectivamente. As configurações do sistema estão descritas no Apêndice A.

O desempenho dos algoritmos estudados e implementados foi validado por meio de análises estatísticas (Teste de Hipóteses), a fim de comprovar seu grau de significância. Os testes de hipóteses encontram-se no Anexo A.

A bibliografia conta com artigos publicados em periódicos, pesquisas na *internet* e livros, que foram de extrema importância para a aquisição de conhecimento sobre o assunto do presente projeto de pesquisa.

4.2. Modelo de Paralelismo para o Processamento de Imagens Médicas

O requisito básico de um sistema de processamento paralelo de imagens consiste em uma infra-estrutura que permita a execução eficiente de algoritmos neste domínio, sejam de nível baixo - que executam alterações globais na imagem; de nível médio – que identificam estruturas importantes na imagem ou de nível alto – relacionados com o reconhecimento de padrões. Essa infra-estrutura é composta essencialmente de funções de comunicação e distribuição de dados adequados ao processamento de imagens (BARBOSA, 2000).

Os filtros de processamento de imagens podem ser executados tanto no domínio da freqüência (considerando a definição matemática da imagem) quanto no domínio espacial, considerando o conjunto de *pixels* que formam a imagem. No entanto, o domínio espacial é o mais utilizado devido principalmente à facilidade de implementação, mas em contrapartida, exige alto poder de processamento, visto que as imagens constituem, na maioria das vezes, matrizes enormes de pontos a serem processados.

Neste trabalho, a escolha dos algoritmos é devido ao objetivo de processamento, buscando executar um filtro que exigisse uma quantidade elevada de operações a fim de que a diferença entre a implementação seqüencial e paralela pudesse ser evidenciada. É neste contexto que as técnicas de processamento de imagens e, em especial, aquelas desenvolvidas para aplicação em imagens médicas, podem se beneficiar dos conceitos de paralelização de imagens.

Quando se trata de imagens médicas a importância do paralelismo é ainda maior, visto que esta classe de imagens não pode permitir armazenamento com perda de dados e, muitas vezes, exige precisão na sua aquisição, gerando um conjunto ainda maior de dados.

Uma proposta de paralelização eficiente seria a divisão da imagem em blocos distribuídos pelos processadores, de forma a processar ao mesmo tempo vários blocos de uma imagem.

Definir o tipo de paralelismo que melhor se adapte ao processamento proposto é tarefa que permite obtenção de melhor desempenho. Desse modo, o paralelismo de dados é o que melhor se enquadra, tornando-se o mais interessante neste caso, pois se pode definir que o processador execute as mesmas tarefas sobre diferentes dados aproximadamente do mesmo tamanho, tendo um único fluxo de controle (SPMD).

O desafio a ser vencido, neste caso e, mais especificamente em imagens médicas, é a divisão da imagem em blocos e a posterior junção desses blocos sem perdas de processamento.

A Figura 4.1 ilustra uma estratégia de paralelização em que uma imagem médica é dividida em blocos pelo mestre. Os blocos são subseqüentemente transmitidos por meio da biblioteca mpiJava aos escravos. Estes têm a incumbência de processá-los e devolvê-los já processados ao mestre, o qual os une, reconstituindo a imagem.

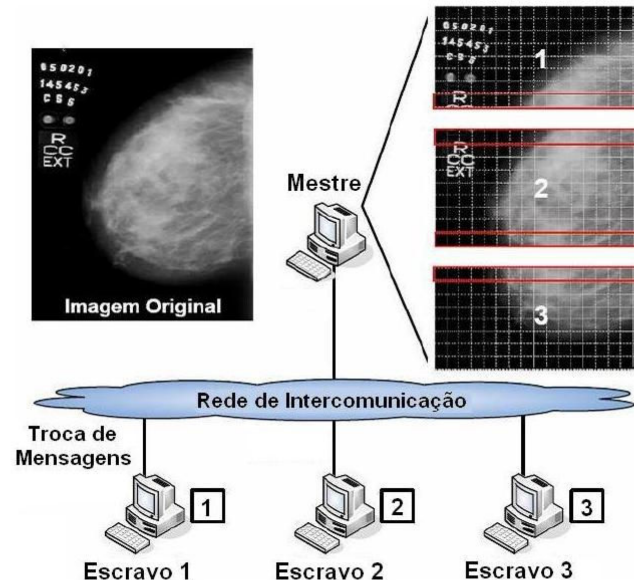


Figura 4.1. Estratégia de paralelização para o processamento das imagens médicas (SAITO et al., 2007j).

Pode-se perceber também que os blocos apresentam tamanhos variados (como observado pelo número de linhas em cada um dos blocos 1, 2 e 3), dependendo da característica de cada algoritmo. Os algoritmos dos filtros de mediana e de detecção de bordas, abordados neste trabalho, fazem uso de máscaras coeficientes que operam sobre uma "vizinhança" de pontos da imagem. Sendo assim, há necessidade de alguma redundância nos blocos para o processamento (destacada em vermelho na Figura 4.1).

Um algoritmo paralelo genérico, em que o mestre efetua a distribuição aos escravos, é apresentado na Figura 4.2. Os detalhes da implementação dos algoritmos paralelos são discutidos na seção 4.3.

```
abre imagem
define num_processos
caso seja mestre
   para i = 0 até num_processos
       tam_bloco = (altura_imagem div num_processos) + máscara
       define bloco com tam_bloco
       para j = 0 até tam_bloco
          copia pixels da imagem no bloco
   envia(bloco)
   para k = 0 até num_processos
      recebe(bloco)
      para j = 0 até tam_bloco
         copia pixels do bloco na imagem
caso seja escravo
   recebe(bloco)
   processa(bloco)
   envia(bloco)
```

Figura 4.2. Implementação genérica do algoritmo paralelo utilizado para o processamento de imagens.

4.3. Implementação dos Algoritmos de Processamento de Imagens

Os algoritmos de filtro de mediana e de detecção de bordas foram implementados na linguagem de programação Java, de forma seqüencial e paralela. Nesta seção são apresentados os principais aspectos e alguns trechos de código das implementações desses algoritmos, bem como uma análise do seu funcionamento e de suas respectivas imagens resultantes.

4.3.1. Algoritmo de Suavização – Filtro de Mediana

Conforme descrito na seção 2.4, uma das vantagens da utilização do filtro de mediana é a eliminação de ruídos. A Figura 4.3 apresenta as imagens resultantes da aplicação do filtro, utilizando-se diferentes tamanhos de máscaras (3x3, 5x5 e 7x7). Tais imagens foram utilizadas para a análise dos testes de desempenho.

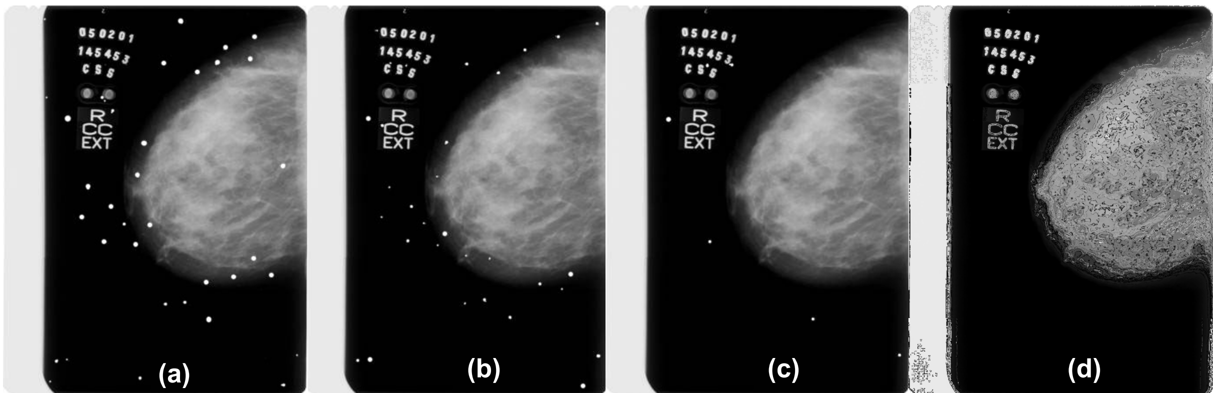


Figura 4.3 - Exemplos de suavização utilizando o filtro de mediana. (a) imagem mamográfica original; (b) imagem suavizada com máscara 3x3; (c) imagem suavizada com máscara 5x5; (d) imagem suavizada com máscara 7x7 (SAITO, et al., 2007f).

Na Figura 4.3(a), observa-se que a quantidade de ruído presente na imagem original é grande. Com a aplicação do filtro, pode-se observar pelas Figuras 4.3(b), 4.3(c) e 4.3(d), uma diminuição desse ruído. Sendo que, quanto maior o tamanho da máscara utilizada, mais suavizada será a imagem resultante, e conseqüentemente, menos ruído essa imagem apresentará.

O funcionamento do filtro de mediana, utilizando-se máscara de tamanho 3x3, é representado pela Figura 4.4.

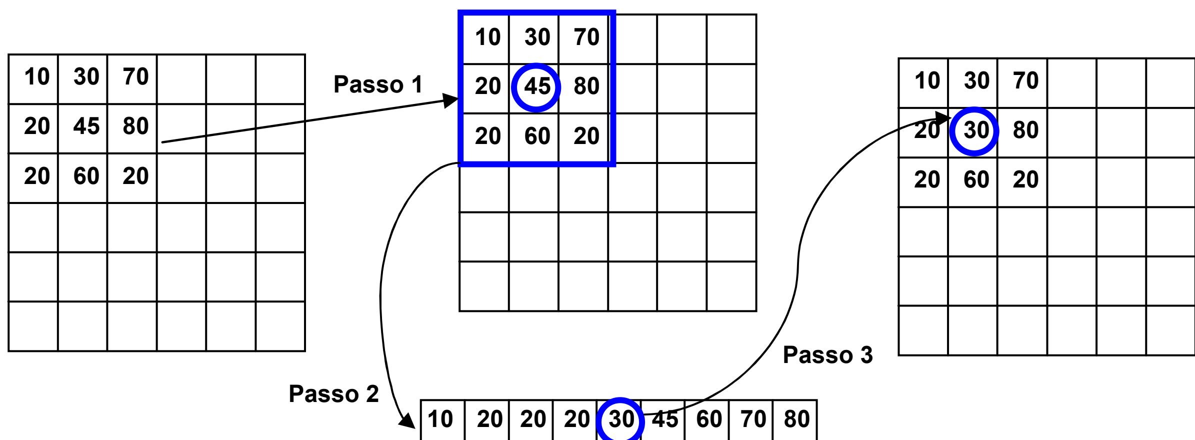


Figura 4.4 – Funcionamento do algoritmo filtro de mediana.

O passo 1 na Figura 4.4, diz respeito à aplicação de uma máscara de tamanho 3x3, em que o *pixel* de interesse é representado pelo círculo azul. O passo 2 refere-se à ordenação dos *pixels* contidos na área da imagem envolvida pela máscara, obtendo-se, dessa forma, a mediana dos níveis de cinza da vizinhança do *pixel* de interesse. E o passo 3 consiste na substituição do valor da intensidade do *pixel* de interesse pela mediana obtida no passo 2.

A Figura 4.5 ilustra a estrutura do programa *Mediana* utilizado para a aplicação paralela do filtro de suavização.

```
class Mediana {

   // Método que aplica o filtro de suavização em cada parte da imagem.
   public static int[] AplicaFiltro (...) {
      ....
      ....
      ....

      // Retorna o vetor de pixels 'vetor' com seus novos valores.
      return(vetor);
   }

   static public void main(String[] args) throws MPIException {

      // Inicialização do MPI.
      MPI.Init(args);

      // Definição das variáveis globais.

      ....
      ....
      ....

      // Retorna o rank do processo que a chamou no grupo do comunicador.
      rank = MPI.COMM_WORLD.Rank();

      // Retorna o número de processos no grupo do comunicador.
      p = MPI.COMM_WORLD.Size();

      // Parte de processamento do mestre.
      if (rank == 0) {

         ....
         ....
         ....

      }

      // Parte de processamento dos escravos.
      else {

         ....
         ....
         ....

      }

      // Finalização do MPI.
      MPI.Finalize();
   }
}
```

Figura 4.5. *Template* da estrutura do programa paralelo Mediana.

Conforme pode ser observado pela Figura 4.5, algumas rotinas de gerenciamento do mpiJava devem ser executadas. Primeiramente, a rotina *MPI.Init(args)*, responsável pela inicialização do MPI, deve ser chamada. Nenhuma outra rotina deve ser executada antes dela.

Iniciado o MPI, as variáveis globais necessárias para a execução da aplicação são declaradas. Em seguida, outras rotinas de gerenciamento são executadas. Uma delas, *MPI.COMM_WORLD.Rank()*, retornando o *rank* do processo que a chamou. Outra rotina é a *MPI.COMM_WORLD.Size()*, responsável pelo retorno do número de processos existentes na aplicação.

Para a paralelização do algoritmo, é necessária a divisão da imagem, pelo mestre, em partes a serem processadas pelos escravos. Conforme observado na Figura 4.5, o programa encontra-se dividido em duas partes, uma referente ao processamento do mestre e a outra ao processamento dos escravos. Para um melhor detalhamento e entendimento, a parte relativa ao processamento do mestre foi dividida em três partes, referentes ao envio dos dados, ao recebimento dos dados já processados pelos escravos e a união de cada parte da imagem. A Figura 4.6 ilustra a parte do processamento relativa ao mestre.

```
if (rank == 0) {      /* Se o rank for mestre */

   // Obtenção da imagem e do tamanho da máscara a ser utilizada.

   ....
   ....
   ....

   // Divisão da imagem em partes.

   ....
   ....
   ....

   for (int i = 1; i < p; i++) {

      // Empacotamento dos valores necessários para a aplicação do filtro

      ....
      ....
      ....

      // Envio da mensagem aos escravos.

      ....
      ....
      ....

      // Mestre recebe as mensagens de cada um dos escravos.
      for (int i = 1; i<p; i++) {

         // Junção de cada parte da imagem processsada.
         // Gravação da imagem resultante suavizada.
      }
   }
}
```

Figura 4.6. Template da estrutura referente ao processamento do mestre.

Na Figura 4.7, as linhas 3 a 9, referem-se à obtenção da imagem a ser processada e do tamanho da máscara a ser utilizada. A declaração e atribuição de valores necessários para se trabalhar com as imagens são definidas nas linhas 11 a 22.

```
1  if (rank == 0) {   // Se rank for mestre.
2
3     String nome=JOptionPane.showInputDialog("Nome da Figura Original") + ".tif";
4     PlanarImage image = JAI.create("fileload",nome);
5     String mas = JOptionPane.showInputDialog(" Valor da mascara: \n"
6                                       + "1 - mascara 3X3 \n"
7                                       + "2 - mascara 5X5 \n"
8                                       + "3 - mascara 7X7 ");
9     mascara = Integer.parseInt(mas);
10
11    width = image.getWidth();
12    height = image.getHeight();
13    SampleModel sm = image.getSampleModel();
14    nbands = sm.getNumBands();
15    Raster inputRaster = image.getData();
16    WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
17    int[] pixels = new int[nbands*width*height];
18    int[] pixels2 = new int[nbands*width*height];
19    ColorModel cm = image.getColorModel();
20    TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,sm,cm);
21    inputRaster.getPixels(0,0,width,height,pixels);
22    inputRaster.getPixels(0,0,width,height,pixels2);
```

Figura 4.7. Representação do trecho de código referente à obtenção da imagem e do tamanho da máscara utilizada.

Obtidos a imagem a ser processada e o tamanho da máscara a ser utilizada, divide-se a imagem em partes. Na Figura 4.8 é representado o trecho de código relativo à forma como foi realizada essa divisão.

```
1  nlinhas = height/(p-1);
2
3  resto = (height%(p-1));
4
5  start = 0;
6  start = System.currentTimeMillis();
```

Figura 4.8. Representação do trecho de código referente à divisão da imagem.

A variável *nlinhas* é responsável pelo controle da altura de cada parte da imagem dividida, recebendo, dessa forma, a altura da imagem original dividida pelo número de processos existentes na aplicação menos 1. Retira-se um do número de processos, devido ao mestre não realizar o processamento de nenhuma parte da imagem. Portanto, deixa-se um processo apenas para a divisão da imagem em partes, distribuição e união dessas partes, pelo mestre.

A variável *resto* recebe o resto da divisão anterior (altura da imagem pelo número de processos menos 1), indicando, dessa forma, o número de linhas de *pixels* que restaram para serem processadas. Na Figura 4.9 é representada a divisão da imagem em uma aplicação com 4 processos em execução.

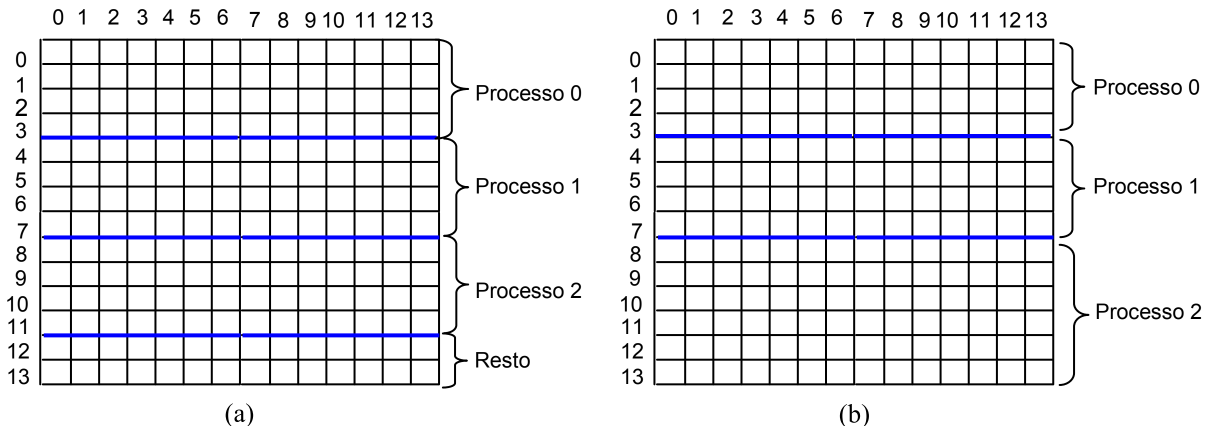


Figura 4.9. Representação da divisão da imagem: (a) distribuição das partes da imagem aos processos. (b) atribuição do resto da divisão ao penúltimo processo.

A imagem, de tamanho 14x14 *pixels*, é dividida em 3 processos, sendo assim cada um deles (cada parte da imagem) teria uma altura igual a 4, porém devem-se acrescentar as linhas redundantes necessárias ao processamento das técnicas (filtragens mediana e detecção de bordas) que trabalham com operações de vizinhança, conforme já mencionado. Restando, dessa forma, duas linhas de *pixels* a serem processadas (Figura 4.9a). Conforme já mencionado, o mpiJava é estático, dessa forma, a aplicação já começa com o número de processos definido, não havendo como criar novos processos em tempo de execução. Portanto, para o processamento dessas duas linhas restantes, ou de qualquer quantidade de linhas que restam da divisão da imagem, o penúltimo processo (p-1) recebe o resto dessa divisão, conforme ilustrado pelas Figuras 4.9b e 4.10.

```
1     for (int i = 1; i < p; i++) {
2
3        if (resto >= 1) {
4           if (i == (p - 1)) {
5              nlinhas = nlinhas + (resto);
6           }
7        }
```

Figura 4.10. Representação do trecho de código referente à atribuição do resto da divisão da imagem.

Dividido a imagem em partes, cada uma delas com altura definida pela variável *nlinhas*, parte da imagem e alguns valores necessários para o processamento podem ser

enviados para cada um dos escravos. Para tanto, deve-se empacotá-los. A Figura 4.11 ilustra o empacotamento desses dados.

```
1      if (p == 2) {
2         tam_vetor[0] = (width*nlinhas*nbands);
3      }
4      else
5         tam_vetor[0] = ((width*nlinhas*nbands)+(chave*mascara*width*nbands));
6
7      int [] vetor = new int[tam_vetor[0]];
8
9      pos = 0;
10
11     vetor_aux[0] = mascara;
12     vetor_aux[1] = width;
13     vetor_aux[2] = height;
14     vetor_aux[3] = nbands;
15     vetor_aux[4] = ((vetor.length) / (width*nbands));
16     vetor_aux[5] = vetor.length;
17
18     System.arraycopy (pixels2, (ele - (ch1*mascara*width*nbands)), vetor, 0,
19                       vetor.length);
20     tag = i;
21     pos = MPI.COMM_WORLD.Pack(vetor_aux,0,vetor_aux.length,MPI.INT,buffer,pos);
22     pos = MPI.COMM_WORLD.Pack(vetor,0,vetor.length,MPI.INT,buffer,pos);
23
24     apos[0] = pos;
```

Figura 4.11. Representação do trecho de código referente ao empacotamento dos dados a serem enviados.

Nas linhas 1 a 5 da Figura 4.11, é definido o tamanho do vetor que conterá os *pixels* da parte da imagem. Caso o número de processos seja igual a dois, considera-se que um processo é reservado ao mestre (para partição, distribuição e união das partes da imagem) e o outro é reservado ao processamento escravo. Sendo assim, o tamanho do vetor deve corresponder à imagem inteira. Caso contrário, conforme apresentado pela linha 5, o tamanho do vetor recebe a multiplicação do número de linhas (altura da imagem) pela largura e pelo número de bandas de cor que a imagem possui, mais algumas linhas (a chave multiplicada pela máscara, largura e número de bandas de cor) de acordo com a mascara utilizada e o número do processo a ser executado.

A variável *chave* (linha 5 da Figura 4.11) recebe os valores 1 ou 2 de acordo com a parte da imagem trabalhada. Se a parte a ser processada é relativa às partes internas da imagem, a variável *chave* recebe o valor 2, pois há a necessidade de redundância de duas linhas de *pixels*, uma delas é referente à última linha do processo anterior e a outra é referente à primeira linha do processo posterior, conforme citado anteriormente e descrito na Figura 4.1. Caso seja a primeira ou a última parte da imagem, atribui-se o valor 1 à variável *chave*. Devido à necessidade de redundância de apenas uma linha de *pixel*. Em se tratando da

primeira parte da imagem, a linha de *pixel* é relativa à primeira linha do processo posterior. Na última parte da imagem é relativa à última linha do processo anterior.

Nas linhas 11 a 16 da Figura 4.11, a variável *vetor_aux* recebe os valores necessários para o processamento e que devem ser empacotados, tais como: máscara, largura, altura, *nbands*, altura da parte da imagem e o tamanho do vetor contendo parte dos *pixels* da imagem. As linhas 18 e 19 ilustram o recebimento desses *pixels* nesse vetor. No primeiro processo, os *pixels* da variável *pixels2* são copiados em *vetor*, começando-se na posição zero de *pixels2* e percorrendo-se até a posição final de acordo com o tamanho do vetor. Nos processos seguintes, começa-se na posição de *pixels2* em parou no processo anterior, colocando tal *pixel* na posição zero de *vetor*.

O empacotamento do *vetor_aux* e do vetor contendo os *pixels* da imagem são representados pelas linhas 21 e 22. Para o empacotamento é utilizada a rotina *MPI.COMM_WORLD.Pack(parâmetros)*, em que os parâmetros são: o dado a ser empacotado, este deve ser sempre um vetor; a posição inicial do vetor a ser enviado; a posição final do vetor (tamanho) a ser enviado; o tipo dos dados contidos no vetor; onde os vetores irão ser empacotados e por fim a posição inicial em que os dados podem ser colocados.

Realizado o empacotamento, os dados podem ser enviados com o comando *MPI.COMM_WORLD.Isend(parâmetros)*, conforme as linhas 1 e 3 da Figura 4.12. Primeiramente, é necessário o envio do tamanho da mensagem a ser enviada (linha 1), logo em seguida a mensagem propriamente dita (linha 3), pois o recebimento da mensagem exige que se saiba antecipadamente o tamanho da mensagem a ser recebida.

```
1       MPI.COMM_WORLD.Isend(apos, 0, 1, MPI.INT, i, tag);
2       tag += 900;
3       MPI.COMM_WORLD.Isend(buffer, 0, pos, MPI.PACKED, i, tag);
4
5       ele += (nlinhas * width * nbands);
6       ch1 = 1;
7       chave = 2;
8
9       if ((i + 1) == (p - 1)) {
10         chave = 1;
11      }
12   }
13   nlinhas = nlinhas - (resto);
```

Figura 4.12. Representação do trecho de código referente ao envio dos dados pelo mestre.

Os parâmetros da rotina de envio são: a mensagem a ser enviada, a posição inicial e a posição final (tamanho) da mensagem; o tipo da mensagem; a identificação do processo *(i)* e por fim a identificação da mensagem *(tag)*.

Na linha 2, observa-se que após o envio da primeira mensagem, o valor do tag é incrementado em 900 apenas para que haja um relacionamento (correspondência) entre as duas mensagens a serem enviadas. Dessa forma, garante-se sempre que o tamanho enviado pela primeira mensagem seja realmente referente aos dados empacotados da segunda mensagem (no mesmo processo).

Após o envio das mensagens, as variáveis *ele* e *ch1* têm seus valores alterados. Tais variáveis são utilizadas para o controle das posições dos *pixels* da imagem, fazendo com que cada processo receba parte da imagem corretamente, conforme pode ser observado pela linha 18 da Figura 4.11. Em que no primeiro processo, a variável *vetor* recebe os *pixels* da imagem, começando na posição zero e indo até o tamanho do vetor. Nos processos seguintes, começa-se na posição de *pixels2*, que contém os *pixels* da imagem, em que parou no processo anterior, dessa forma, as variáveis *ele* e *ch1* servem para que a cada processo, a posição, do *pixel* a ser copiado em *vetor*, seja alterada de acordo com o tamanho da parte da imagem em questão.

Na linha 7 da Figura 4.12, a variável *chave*, já mencionada, recebe o valor 2, devido à necessidade do acréscimo de duas linhas de *pixels* para o processamento das partes internas da imagem. Para o processamento da última parte da imagem, é necessário o acréscimo de apenas uma linha de *pixels*, referente à última linha do penúltimo processo. Para tanto, utiliza-se a condição das linhas 9 e 10 da Figura 4.12.

Ao término da divisão e distribuição dos processos para cada escravo, a variável *nlinhas* recebe o seu valor inicial, sem o acréscimo da variável *resto* (linha 13 da Figura 4.12).

A segunda parte referente ao processamento do mestre, conforme citado anteriormente, é o recebimento dos dados já processados por cada um dos escravos, podendo ser observado pela Figura 4.13.

```
1  for (int i = 1; i<p; i++) {
2
3     status = MPI.COMM_WORLD.Recv (size, 0, 1, MPI.INT, MPI.ANY_SOURCE, MPI.ANY_TAG);
4
5     aux_tag = status.tag;
6     tag = status.tag;
7     tag += 900;
8     src = status.source;
9     status = MPI.COMM_WORLD.Recv (buffer, 0, size[0], MPI.PACKED, src, tag);
10    pos = 0;
11    pos = MPI.COMM_WORLD.Unpack (buffer, pos, tam_vetor, 0, 1, MPI.INT);
12    int [] vetor = new int[tam_vetor[0]];
13    pos = MPI.COMM_WORLD.Unpack (buffer, pos, vetor, 0, vetor.length, MPI.INT);
```

Figura 4.13. Representação do trecho de código referente ao recebimento dos dados processados pelos escravos.

Para o recebimento das mensagens, utiliza-se o comando *MPI.COMM_WORLD.Recv(parâmetros)*. Os parâmetros são: os dados a serem recebidos; as posições inicial e final (tamanho) da mensagem; o tipo da mensagem; o indicativo de qual escravo (processador) a enviou (por meio do comando *MPI.ANY_SOURCE*) e por fim a identificação da mensagem (por meio do comando *MPI.ANY_TAG*).

A primeira mensagem recebida pelo mestre (linha 3) refere-se ao tamanho dos dados que serão recebidos na próxima mensagem. Portanto, para o recebimento da segunda mensagem, é necessário que haja um controle de forma que o tamanho enviado pela primeira mensagem seja realmente referente à segunda mensagem recebida. Para tanto, incrementa-se em 900 o valor do identificador das mensagens (*tag*), sendo que as mensagens recebidas devem se referir à mesma fonte (*src*).

As linhas 11 e 13 da Figura 4.13 referem-se ao desempacotamento dos dados recebidos. Primeiramente, o tamanho do vetor, logo em seguida o vetor contendo os *pixels* já processados pelos escravos.

A terceira e última parte referente ao processamento do mestre, conforme citado anteriormente, é a união das partes das imagens processadas por cada um dos escravos. A Figura 4.14 ilustra a união de cada uma dessas partes.

```
1          if (p == 2) {
2             pixels2 = vetor;
3          }
4          else {
5             if (aux_tag == 1) {
6                ind = 0;
7                comeco = 0;
8                fim = tam_vetor[0] - (mascara * width * nbands);
9             }
10            else {
11               ind = mascara * width * nbands;
12               comeco = (nlinhas * (aux_tag - 1) ) * width * nbands;
13               if ( aux_tag == (p - 1) )
14                  fim = tam_vetor[0] - (mascara * width * nbands);
15               else
16                  fim = tam_vetor[0] - (2 * mascara * width * nbands);
17            }
18            System.arraycopy(vetor, ind, pixels2, comeco, fim);
19         }
20    }
```

Figura 4.14. Representação do trecho de código referente à união de cada parte da imagem processada pelos escravos.

Se o número de processos é igual a 2, conforme já mencionado, um processo é referente ao mestre e o outro ao escravo, portanto envia-se e recebe-se a imagem inteira, dessa forma, *pixels2* recebe o vetor contendo todos os *pixels* da imagem. Caso contrário cada vetor

recebido deve ser copiado em *pixels2* na sua posição correspondente. Para garantir que a imagem seja reconstituída de maneira correta, são utilizadas as variáveis *ind*, *começo* e *fim*. Em que *ind* é responsável pelo controle das posições de *vetor* e *come*co e *fim* são responsáveis pelo controle das posições de *pixels2*.

Reconstituída a imagem, a variável *end* recebe o valor do tempo final de processamento, conforme ilustrado na linha 2 da Figura 4.15. Para obtenção do tempo de execução total do processamento, subtrai-se o valor do tempo final pelo valor do tempo inicial (contido na variável *start*). A Figura 4.15 também apresenta a gravação da imagem resultante com o nome *suavizada* (linha 14).

```
1        long end;
2        end = System.currentTimeMillis();
3        try {
4           FileOutputStream writer = new FileOutputStream ("resultado.txt", true);
5           PrintWriter saida = new PrintWriter(writer);
6           saida.println((end-start));
7           saida.close();
8           writer.close();
9        } catch( IOException e) { }
10
11       outputRaster.setPixels(0, 0, width, height, pixels2);
12       tiledImage.setData(outputRaster);
13
14       JAI.create("filestore", tiledImage, "suavizada", "TIFF");
15       System.out.println("Tempo Total: " + (end - start) );
16    }
```

Figura 4.15. Representação do trecho de código referente à união de cada parte da imagem processada pelos escravos.

Conforme mencionado anteriormente, o programa encontra-se dividido em duas partes, uma referente ao processamento do mestre e a outra ao processamento dos escravos.

A parte de processamento dos escravos constitui-se basicamente do recebimento e desempacotamento das mensagens enviadas pelo mestre; da chamada ao método *AplicaFiltro*(), responsável pela aplicação do filtro em cada parte da imagem e do empacotamento e envio de cada parte da imagem já processada pelo escravo. A Figura 4.16 ilustra a parte do processamento relativa aos escravos.

Após o recebimento e desempacotamento das mensagens o método *AplicaFiltro* é executado por cada um dos escravos nas suas respectivas partes da imagem a serem processadas. As Figuras 4.17, 4.18 e 4.19 ilustram os trechos que compõem o método *AplicaFiltro*.

O método *AplicaFiltro* recebe como parâmetros as variáveis *máscara*, *distância*, *tam_vet_ord*, *vetor*, *alt_parte*, *width* e *nbands*, referentes ao tamanho da máscara, limite para

se percorrer a vizinhança do *pixel*, tamanho do vetor de *pixels* a ser ordenado, altura da parte da imagem, largura da imagem e o número de bandas de cor que a imagem possui, respectivamente.

```
1     else {
2        int distancia;
3        int tam_vet_ord;
4
5        status = MPI.COMM_WORLD.Recv(size, 0, 1, MPI.INT, source, MPI.ANY_TAG);
6        int aux_tag;
7        aux_tag = status.tag;
8        tag = status.tag;
9        tag += 900;
10       status = MPI.COMM_WORLD.Recv (buffer, 0, size[0], MPI.PACKED, source, tag);
11       pos = 0;
12
13       pos = MPI.COMM_WORLD.Unpack (buffer, pos, vetor_aux, 0, vetor_aux.length,
14                                   MPI.INT);
15       mascara = vetor_aux[0];
16       width = vetor_aux[1];
17       height = vetor_aux[2];
18       nbands = vetor_aux[3];
19       alt_parte = vetor_aux[4];
20       tam_vetor[0] = vetor_aux[5];
21       int [] vetor = new int[tam_vetor[0]];
22       pos = MPI.COMM_WORLD.Unpack(buffer, pos, vetor, 0, vetor.length, MPI.INT);
23
24       distancia = mascara + 1;
25       tam_vet_ord = ((mascara + distancia) * (mascara + distancia));
26
27       vetor = AplicaFiltro (mascara, distancia, tam_vet_ord, vetor, alt_parte,
28                            width, nbands);
29
30       dest = 0;
31       pos = 0;
32       pos = MPI.COMM_WORLD.Pack (tam_vetor, 0, 1, MPI.INT, buffer, pos);
33       pos = MPI.COMM_WORLD.Pack (vetor, 0, vetor.length, MPI.INT, buffer, pos);
34       apos[0] = pos;
35       tag = aux_tag;
36       MPI.COMM_WORLD.Send (apos, 0, 1, MPI.INT, dest, tag);
37       tag += 900;
38       MPI.COMM_WORLD.Send (buffer, 0, pos, MPI.PACKED, dest, tag);
39    }
40    MPI.Finalize();
41    }
42 }
```

Figura 4.16. Representação do trecho de código referente ao processamento escravo.

A Figura 4.17 apresenta o primeiro trecho do método *AplicaFiltro*, responsável pelo controle dos índices dos vetores que devem indicar somente o *pixel* de interesse e/ou seus vizinhos.

A variável *offset* (linha 5 da Figura 4.17) é utilizada para o controle do índice de *vetor* (Figura 4.19), fazendo com que somente o *pixel* de interesse sofra alteração, recebendo o valor mediano obtido com a ordenação do vetor *(vet_ord)* composto pelo *pixel* de interesse e sua vizinhança.

```
1  public static int[] AplicaFiltro (int mascara, int distancia, int
2           tam_vet_ord, int[] vetor, int alt_parte, int width, int nbands) {
3     for(int h = mascara; h < alt_parte - mascara; h++)
4        for (int w = mascara; w < width - mascara; w++) {
5           offset = h * width * nbands + w * nbands;
6           int [] vet_ord = new int [tam_vet_ord];
7           ele = 0;
8           for (int a = h - mascara; a < h + distancia; a++)
9              for (int l = w - mascara; l < w + distancia; l++) {
10                offset2 = a * width * nbands + l * nbands;
11                vet_ord[ele] = vetor[offset2 + 0];
12                ele++;
13             }
14    ....
15    ....
16    ....
```

Figura 4.17. Trecho de código do método AplicaFiltro com o controle dos índices dos vetores.

A variável *offset2* (linha 10 da Figura 4.17) é responsável pelo controle do índice de *vetor*, de forma que o vetor de ordenação (*vet_ord*) contenha apenas os elementos referentes ao *pixel* de interesse e sua vizinhança.

Obtido os valores do *pixel* de interesse e sua vizinhança, o vetor denominado *vet_ord* é ordenado, conforme pode ser observado pelo segundo trecho de código apresentado pela Figura 4.18. Em que o método de ordenação utilizado é o *Shellsort*.

```
1     public static int[] AplicaFiltro (int mascara, int distancia, int
2        tam_vet_ord, int[] vetor, int alt_parte, int width, int nbands) {
3
4        ....
5        ....
6        ....
7
8        int i, j, hi = 1, value;
9        do {
10          hi = 3 * hi + 1;
11       } while (hi < vet_ord.length);
12
13       do {
14          hi /= 3;
15          for ( i = hi; i < vet_ord.length; i++) {
16             value = vet_ord[i];
17             j = i - hi;
18             while (j >= 0 && value < vet_ord[ j ]) {
19                vet_ord[j + hi] = vet_ord[j];
20                j -= hi;
21             }
22             vet_ord[j + hi] = value;
23          }
24       } while (hi > 1);
25
26       ....
27       ....
28       ....
```

Figura 4.18. Trecho de código do método AplicaFiltro com a ordenação (*shellsort)* do vetor de *pixels*.

O terceiro trecho de código (Figura 4.19) refere-se à obtenção do valor mediano do vetor ordenado *vet_ord* e a atribuição do valor obtido ao *pixel* de interesse.

```
1     public static int[] AplicaFiltro (int mascara, int distancia, int
2        tam_vet_ord, int[] vetor, int alt_parte, int width, int nbands) {
3
4        ....
5        ....
6        ....
7
8        for (int band = 0; band < nbands; band++) {
9           vetor[offset + band] = vet_ord[tam_vet_ord/2];
10       }
11       return(vetor);
12    }
```

Figura 4.19. Trecho de código do método AplicaFiltro com a obtenção do valor mediano.

O método *AplicaFiltro* é executado por cada um dos escravos nas suas respectivas partes da imagem a serem processadas. Em cada uma dessas partes, percorrem-se todos os *pixels* pertencentes a elas. É importante enfatizar que o *pixel* de interesse inicial deve localizar-se no ponto central da máscara. Dessa forma, para uma máscara de tamanho 3x3, o *pixel* de interesse inicial será o elemento localizado na linha 1, coluna 1, como ilustra a Figura 4.20(a). As Figuras 4.20(b) e 4.20(c) ilustram as respectivas posições dos *pixels* de interesse iniciais para as máscaras de tamanho 5x5 e 7x7, respectivamente.

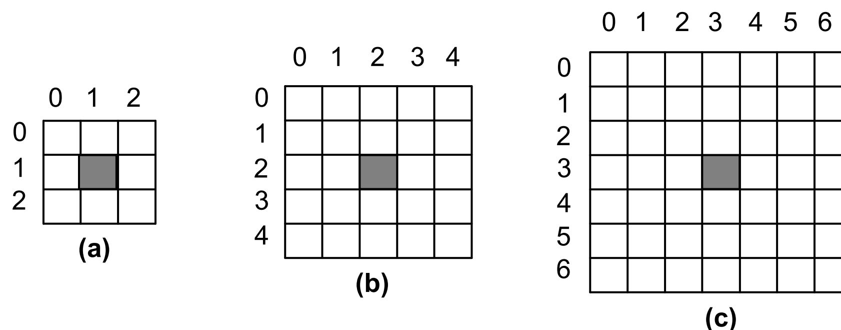


Figura 4.20 – Representação das posições dos pixels de interesse iniciais para máscaras de tamanho: (a) 3x3; (b) 5x5; (c) 7x7.

Após a aplicação do filtro em cada uma das partes da imagem e o retorno dos valores dos *pixels* modificados pelo método *AplicaFiltro()*, cada um dos escravos empacotam e enviam as mensagens, contendo os *pixels* já processados por eles, ao mestre. Cabe ao mestre à união das partes processadas a fim de reconstituir a imagem. Após a reconstituição, o MPI pode ser finalizado com o comando *MPI.Finalize()*.

4.3.2. Algoritmo de Segmentação – Filtro de Detecção de Bordas

Conforme descrito na seção 2.4, o filtro de detecção de bordas é útil quando se deseja conhecer informações referentes ao tamanho e forma dos objetos representados na imagem. Representar uma imagem por meio de suas bordas pode ser vantajoso, pois reduz-se sensivelmente a quantidade de dados que estavam sendo armazenados na imagem, sem perder muita informação. A Figura 4.21 apresenta as imagens resultantes da aplicação do filtro, utilizando-se diferentes tamanhos de máscaras (9x9 e 11x11). Tais imagens foram utilizadas para a análise dos testes de desempenho.

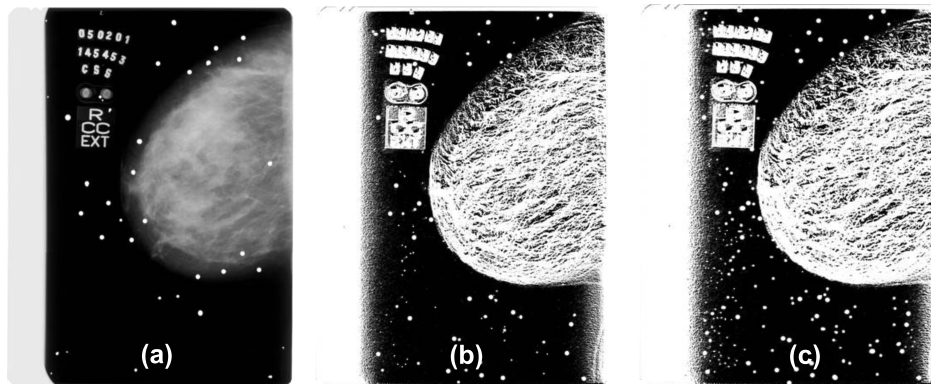


Figura 4.21 – Exemplos de detecção de bordas pelos operadores de Sobel modificado com diferentes tamanhos de máscaras. (a) imagem original; (b) máscara 9x9; (c) 11x11 (SAITO, et al., 2007f).

Na Figura 4.21(a), observa-se que a quantidade de ruído presente na imagem original é grande. Com a aplicação do filtro, pode-se observar pelas Figuras 4.21(b) e 4.21(c), um realce em suas bordas. Sendo que, quanto maior o tamanho da máscara utilizada, mais realçada será a imagem resultante, porém realça-se também o ruído presente na imagem original.

O funcionamento do algoritmo de detecção de bordas, utilizando-se máscara de tamanho 3x3, é representado pela Figura 4.22, lembrando-se que houve modificações nesse algoritmo. Foram utilizados os operadores de Sobel horizontal e vertical e acrescentados mais dois operadores de 45º e de 135º, as demais alterações realizadas são descritas, de forma detalhada, nessa seção.

O passo 1 na Figura 4.22, diz respeito à aplicação de máscaras horizontal, vertical, de 45º e de 135º de tamanho 3x3 em uma região de *pixels* da imagem, em que o *pixel* de

interesse é representado pelo círculo azul. Nesse passo, para cada uma das máscaras, obtém-se a soma das multiplicações de cada elemento da região de *pixels* pelo elemento da máscara correspondente. Realizado o cálculo de cada soma (s_h, s_v, s_{35}, s_{135}), o passo 2 consiste em obter o maior valor dentre as quatro somas calculadas. No passo 3, o valor do *pixel* de interesse é substituído pelo valor obtido no passo 2.

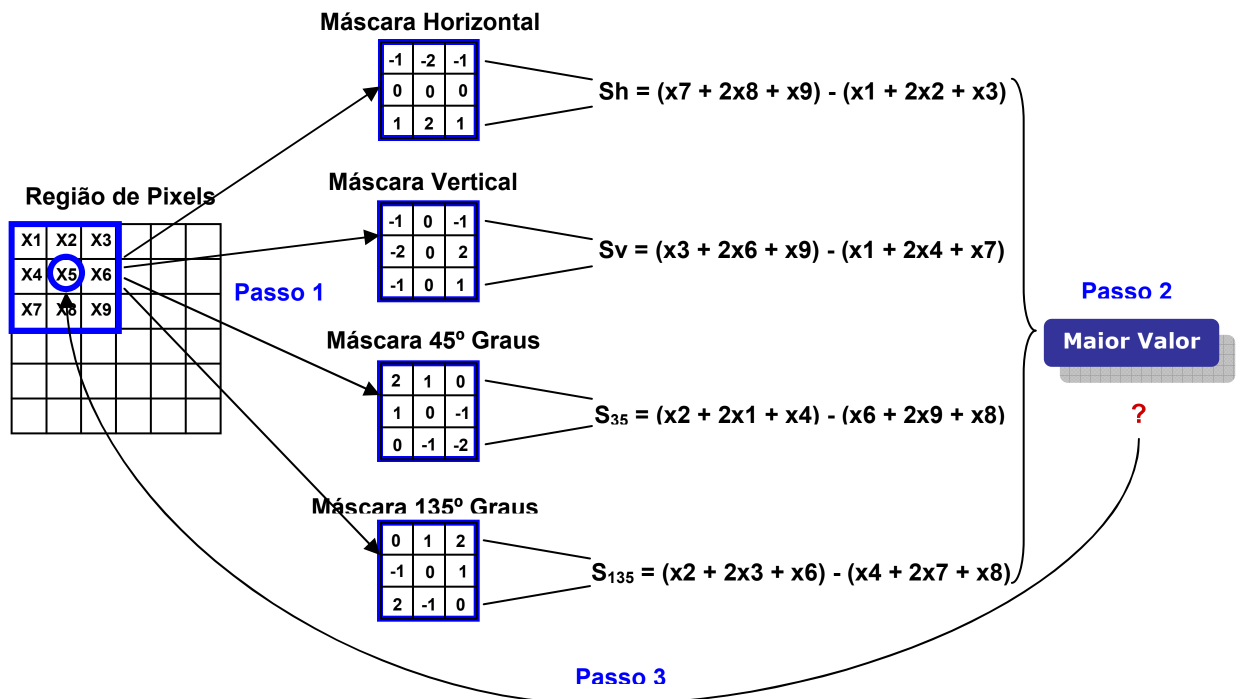


Figura 4.22 – Funcionamento do algoritmo filtro de detecção de bordas.

O programa *Sobel*, utilizado para a aplicação paralela do filtro de detecção de bordas, apresenta a mesma estrutura do programa *Mediana*, ao invés do método *AplicaFiltro*, responsável pela aplicação do filtro mediana na imagem, o programa *Sobel* possui a classe *FiltroSobel*, nessa classe existe o método *AplicaFiltro*, responsável pela aplicação do filtro em questão. Dessa forma, nessa seção será abordada de forma detalhada a implementação da classe *FiltroSobel*.

Para um melhor entendimento, o método *AplicaFiltro* foi dividido em 2 partes. A primeira relativa à aplicação das máscaras (horizontal, vertical, de 45º e de 135º) e a segunda referente à substituição do valor do *pixel* de interesse pela maior soma obtida com a aplicação das máscaras. As Figuras 4.23 e 4.25, ilustram os trechos de código que compõem o método *AplicaFiltro* da classe *FiltroSobel*.

O método *AplicaFiltro* recebe como parâmetros as variáveis *máscara*, *alt_parte*, *width, nbands e vetor*, referentes ao tamanho da máscara, altura da parte da imagem, largura da imagem, número de bandas de cor que a imagem possui e o vetor de *pixels* a ser processado, respectivamente.

```
1     class FiltroSobel {
2
3     // declaração das variáveis
4     ....
5     ....
6     ....
7
8     public FiltroSobel(){}
9
10    public int[]  AplicaFiltro(int mascara, int alt_parte, int width,
11                              int nbands, int vetor[] ){
12
13       vetor2 = new int[vetor.length];
14       tam_vet_ord = ((mascara * 2) + 1) * ((mascara * 2) + 1);
15       int offset, offset2;
16       int veth[] = {...};
17       int vetv[] = {...};
18       int vetds[] = {...};
19       int vetdi[] = {...};
20
21       for(int h=mascara; h<alt_parte-mascara; h++)
22          for (int w=mascara; w<width-mascara; w++) {
23             offset = h * width * nbands + w * nbands;
24             int [] vet_ord = new int [tam_vet_ord];
25             int ele = 0;
26             for (int a=h-mascara; a<=h+mascara; a++)
27                for (int l=w-mascara; l<=w+mascara; l++) {
28                   offset2 = a * width * nbands + l * nbands;
29                   vet_ord[ele] = vetor[offset2+0];
30                   ele++;
31                }
32
33             // inicialização das variáveis somah, somav, somads, somadi.
34
35            for (int i=0; i<vet_ord.length; i++) {
36                somav +=   vet_ord[i] * vetv[i];
37                somah +=   vet_ord[i] * veth[i];
38                somads += vet_ord[i] * vetds[i];
39                somadi += vet_ord[i] * vetdi[i];
40          }
41          ....
42          ....
43          ....
```

Figura 4.23 – Primeiro trecho de código do método AplicaFiltro da classe FiltroSobel (com a aplicação das máscaras).

Conforme pode ser observado pela Figura 4.23, as variáveis *offset* e *offset2*, presentes no método *AplicaFiltro* do programa *Mediana*, descrito na seção anterior, também são utilizadas no método *AplicaFiltro* da classe *FiltroSobel* para o controle dos índices dos vetores que devem indicar somente o *pixel* de interesse e/ou seus vizinhos, sobre os quais serão aplicadas as máscaras, representadas pelos vetores *veth, vetv, vetds e vetdi*.

Um exemplo dessas máscaras é apresentado pela Figura 4.24, utilizando-se máscaras de tamanho 9x9.

```
int veth[] =
        {
          4, 5, 6, 7, 8, 7, 6, 5, 4,
          3, 4, 5, 6, 7, 6, 5, 4, 3,
          2, 3, 4, 5, 6, 5, 4, 3, 2,
          1, 2, 3, 4, 5, 4, 3, 2, 1,
          0, 0, 0, 0, 0, 0, 0, 0, 0,
         -1,-2,-3,-4,-5,-4,-3,-2,-1,
         -2,-3,-4,-5,-6,-5,-4,-3,-2,
         -3,-4,-5,-6,-7,-6,-5,-4,-3,
         -4,-5,-6,-7,-8,-7,-6,-5,-4,
        };

     int vetv[] = {
         4, 3, 2, 1, 0, -1, -2, -3, -4,
         5, 4, 3, 2, 0, -2, -3, -4, -5,
         6, 5, 4, 3, 0, -3, -4, -5, -6,
         7, 6, 5, 4, 0, -4, -5, -6, -7,
         8, 7, 6, 5, 0, -5, -6, -7, -8,
         7, 6, 5, 4, 0, -4, -5, -6, -7,
         6, 5, 4, 3, 0, -3, -4, -5, -6,
         5, 4, 3, 2, 0, -2, -3, -4, -5,
         4, 3, 2, 1, 0, -1, -2, -3, -4,
     };

     int vetds[] =
     {
        8, 7, 6, 5, 4, 3, 2, 1, 0,
        7, 6, 5, 4, 3, 2, 1, 0,-1,
        6, 5, 4, 3, 2, 1, 0, 1,-2,
        5, 4, 3, 2, 1, 0,-1,-2,-3,
        4, 3, 2, 1, 0,-1,-2,-3,-4,
        3, 2, 1, 0,-1,-2,-3,-4,-5,
        2, 1, 0,-1,-2,-3,-4,-5,-6,
        1, 0,-1,-2,-3,-4,-5,-6,-7,
        0,-1,-2,-3,-4,-5,-6,-7,-8,
     };

     int vetdi[] =
     {
         0, 1, 2, 3, 4, 5, 6, 7, 8,
        -1, 0, 1, 2, 3, 4, 5, 6, 7,
        -2,-1, 0, 1, 2, 3, 4, 5, 6,
        -3,-2,-1, 0, 1, 2, 3, 4, 5,
        -4,-3,-2,-1, 0, 1, 2, 3, 4,
        -5,-4,-3,-2,-1, 0, 1, 2, 3,
        -6,-5,-4,-3,-2,-1, 0, 1, 2,
        -7,-6,-5,-4,-3,-2,-1, 0, 1,
        -8,-7,-6,-5,-4,-3,-2,-1, 0,
     };
```

Figura 4.24 – Exemplo da utilização de máscaras (operadores de Sobel) de tamanho 9x9.

Para os demais tamanhos de máscaras, segue-se o mesmo padrão. Em *vetdi*, os elementos destacados em negrito, correspondem aos elementos de uma máscara de tamanho 3x3. Englobando os elementos mais externos, destacados em azul, têm-se os elementos de uma máscara de tamanho 5x5 e assim sucessivamente.

As linhas 36 a 39 da Figura 4.23, são as linhas em que ocorre a aplicação do filtro propriamente dita. Nessas linhas, para cada um dos operadores (máscaras) horizontal, vertical, de 45º e de 135º, obtêm-se as somas das multiplicações de cada elemento de uma região de *pixels* (contidos em *vet_ord*) pelos elementos correspondentes das máscaras (contidos em *vetv, veth, vetds, vetdi*).

Obtido os valores de cada uma das somas *(somav, somah, somads e somadi)*, é necessária a verificação da soma que possui o maior valor, atribuindo-o, dessa forma, ao *pixel* de interesse, conforme pode ser observado pelo segundo trecho de código apresentado pela Figura 4.25.

```
1       public int[]  AplicaFiltro(int mascara, int alt_parte, int width,
2                                       int nbands, int vetor[] ){
3          ....
4          ....
5          ....
6
7          for (int band = 0; band < nbands; band++) {
8             int rn = 0;
9             int rd = 0;
10            if (somah < somav) {
11               rn = somav;
12            }
13            else {
14               rn = somah;
15            }
16            if (somads < somadi) {
17               rd = somadi;
18            }
19            else {
20               rd = somads;
21            }
22            if (rn < rd) {
23               if (rd < 0)
24                  vetor2[offset + band] = 0;
25               else
26                  if (rd > 65535)
27                  vetor2[offset + band] = 65535;
28                  else
29                     vetor2[offset + band] = rd;
30            }
31            else {
32               if (rn < 0)
33                  vetor2[offset + band] = 0;
34               else
35                  if (rn > 65535)
36                     vetor2[offset + band] = 65535;
37                  else
38                     vetor2[offset + band] = rn;
39            }
40         }
41      }
42      return(vetor2);
43      }
44    }
```

Figura 4.25 – Segundo trecho de código do método AplicaFiltro da classe FiltroSobel.

São utilizadas duas variáveis *(rn e rd)*, para auxiliarem nas diferentes comparações necessárias para a obtenção do maior valor dentre as quatro somas. A variável *rn* é responsável por verificar a maior soma vertical ou horizontal. Já a variável *rd* é responsável por verificar a maior soma de 45º ou de 135º.

Obtidos esses dois valores *(rn e rd)*, compara-se qual deles possui maior valor (linha 22). Sendo que aquele que possuir maior valor substituirá o *pixel* de interesse. Antes disso, porém, é necessária a verificação dos limites mínimos e máximos que um *pixel* pode ter. Caso o valor obtido seja menor que zero, deve-se atribuir ao *pixel* de interesse o valor zero, pois não é permitido valor negativo ao *pixel*. Da mesma forma, caso o valor ultrapasse o valor máximo (65536) permitido, o *pixel* de interesse deve receber tal valor (linhas 27 e 36 da Figura 4.25).

Todo esse procedimento é realizado em todas as regiões de *pixels* que constituem cada parte da imagem, retornando, ao mestre, a variável *vetor2* que corresponde à parte da imagem com os valores dos *pixels* já alterados pela aplicação do filtro de detecção de bordas Sobel.

4.4. Considerações Finais

Esse capítulo apresentou o modelo de paralelismo para o processamento de imagens médicas, descrevendo a estratégia de paralelização utilizada para a otimização no processamento dessas imagens. Foi apresentado o passo a passo da implementação paralela de cada algoritmo estudado, bem como uma análise do funcionamento desses algoritmos e suas respectivas imagens resultantes.

Os *templates* gerados podem auxiliar na confecção de outros algoritmos. Com intuito de avaliar o desempenho das implementações em paralelo, o próximo capítulo faz uma análise de desempenho dos resultados obtidos por meio da comparação das implementações em paralelo e em seqüencial.

5. ANÁLISE DE DESEMPENHO DOS RESULTADOS OBTIDOS

A análise de desempenho dos algoritmos de processamento de imagens foi realizada por meio de diferentes testes reais em um ambiente paralelo distribuído composto inicialmente por três máquinas homogêneas (Pentium IV de 2.7 GHz com 512Mbytes, interligadas por uma rede *ethernet* de 100Mb/s) e posteriormente foram acrescidas máquinas idênticas para compor um ambiente de 10 máquinas.

Foram utilizadas imagens mamográficas (radiografia das mamas) de diferentes tamanhos (500 KB, 1 MB, 11 MB e 21MB), no formato TIFF, com resolução de contraste de 16 bits, consistindo em matrizes com tamanho médio 432x580 *pixels*, 625x840 *pixels*, 2500x3000 *pixels* e 2855x3835 *pixels*, respectivamente. Tais imagens fazem parte de um banco de imagens desenvolvido pelo LAPIMO (Laboratório de Processamento de Imagens Médicas e Odontológicas, da EESC/USP).

O filtro de mediana foi avaliado com máscaras de tamanhos diferentes 3x3, 5x5 e 7x7 utilizando o algoritmo de ordenação *shellsort*. O filtro de detecção de bordas foi avaliado com máscaras de tamanhos 9x9 e 11x11. Observa-se que o tamanho da máscara é o que define o tamanho da vizinhança a ser considerada e dependendo do tipo da imagem, uma maior vizinhança pode levar a um melhor resultado de processamento.

Após a realização dos testes obteve-se uma média dos 30 tempos de processamento tanto para aplicação seqüencial quanto para a paralela para os diferentes tipos de máscaras. Quando da avaliação da execução em paralelo foram efetuados testes com até dez máquinas e para cada uma delas, testes com até 11 processos sendo iniciados em paralelo, sendo possível, assim, realizar a comparação de desempenho de cada uma das possíveis combinações, por meio das medidas relacionadas ao tempo (em milissegundos), *speedup* e eficiência. Além disso, todos os resultados obtidos foram avaliados estatisticamente, por meio de testes de hipóteses (JAIN, 1991) (ANEXO A), para comprovar seu grau de significância.

As próximas seções apresentam os resultados da execução seqüencial e paralela das técnicas implementadas utilizando-se imagens de diferentes tamanhos.

5.1. Imagens de Tamanho 500 KB

As Figuras 5.1 a 5.5 apresentam os resultados da execução seqüencial e paralela dos filtros mediana (com máscaras de tamanho 3x3, 5x5 e 7x7) e detecção de bordas (com máscaras de tamanho 9x9 e 11x11), respectivamente, utilizando imagens de tamanho 500 KB.

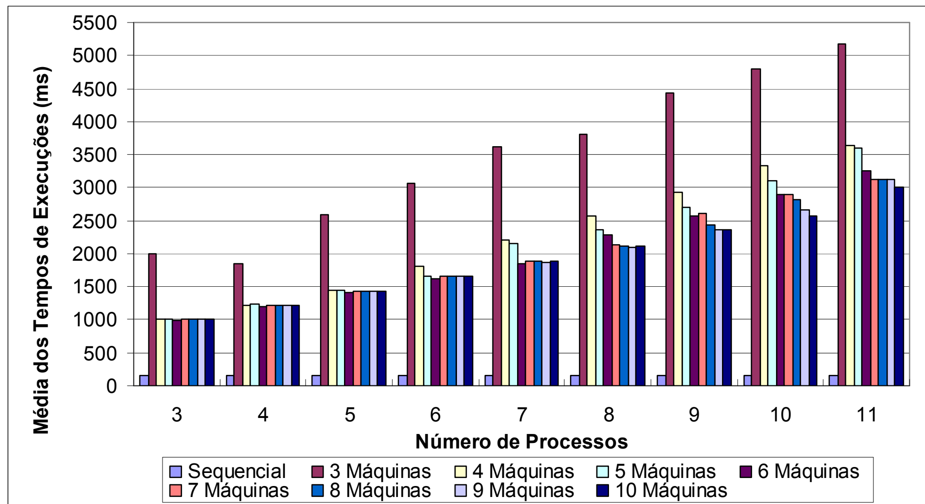


Figura 5.1 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 500KB.

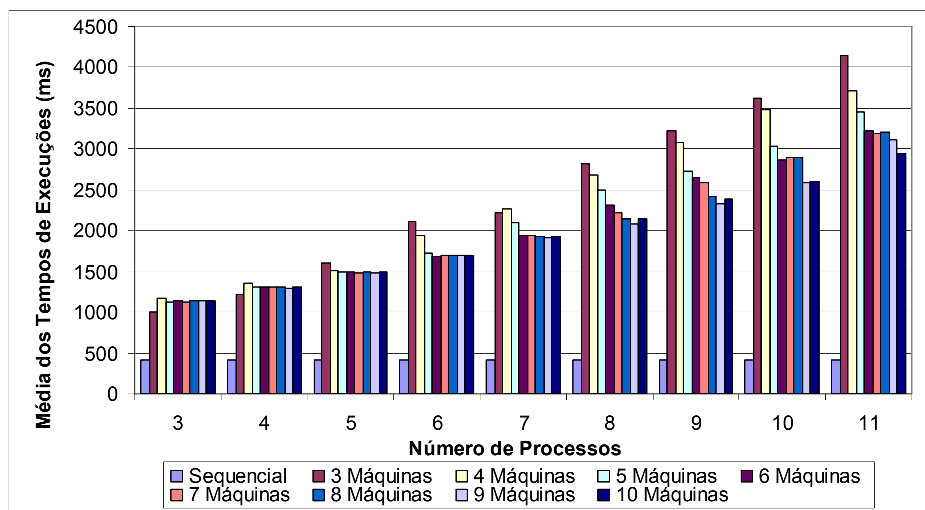


Figura 5.2 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 500KB.

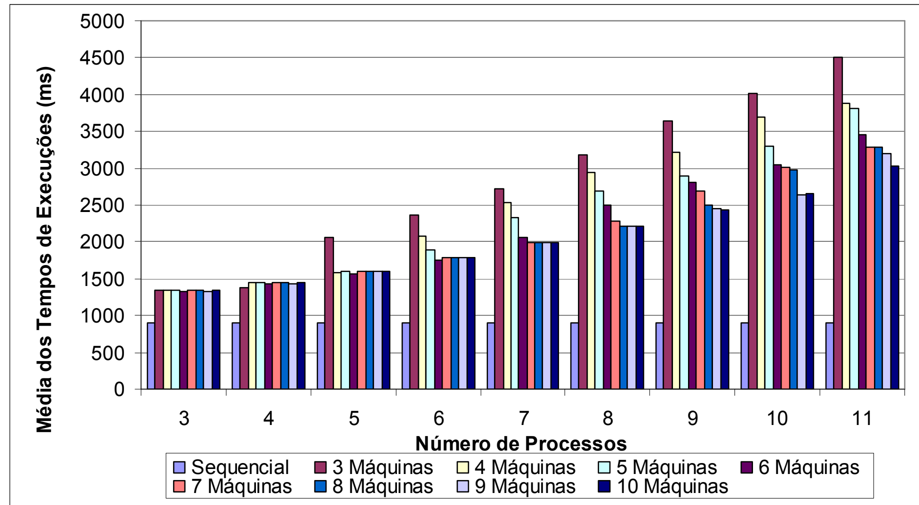


Figura 5.3 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 500KB.

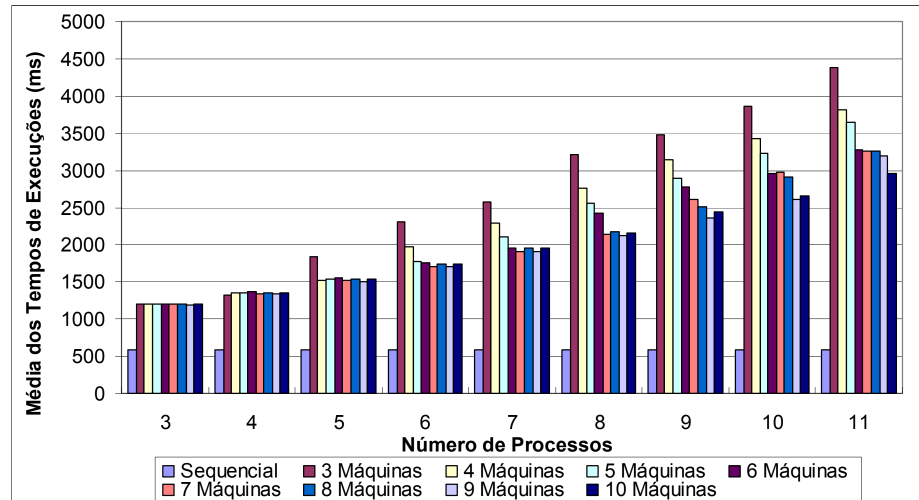


Figura 5.4 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 500KB.

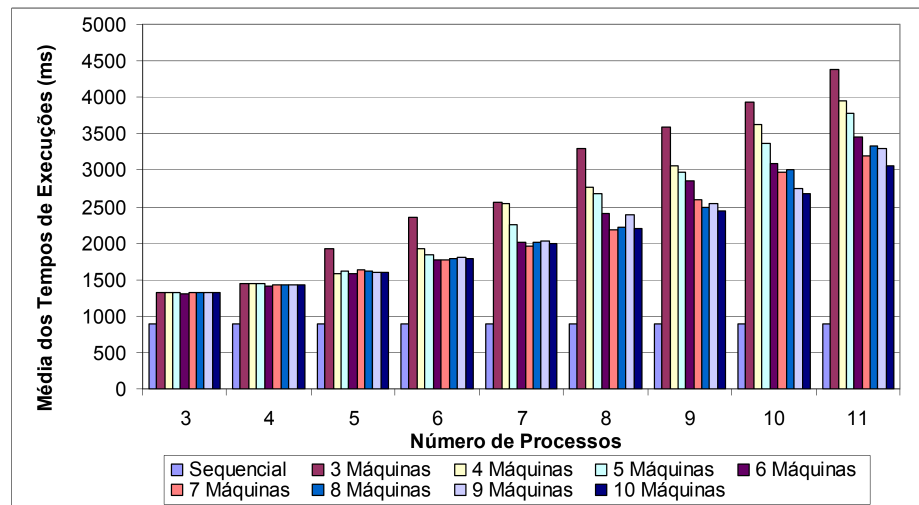


Figura 5.5 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 500KB.

Pelas figuras é possível observar que o tempo de execução do algoritmo seqüencial é significativamente melhor (comprovação dada a partir das Tabelas A.1 a A.5 do Anexo A) que o tempo do mesmo em paralelo independentemente do tamanho da máscara, do número de máquinas e da quantidade de processos iniciados em paralelo.

Uma vez que tanto a quantidade de cálculos efetuados pelos filtros de mediana e de detecção de bordas quanto o tamanho da imagem utilizada (tamanho 500 KB) são relativamente pequenos, a paralelização do mesmo não impõe melhoria, pois se consome mais tempo com comunicação para envio de dados por meio da rede do que no cálculo da máscara propriamente dita, ou seja, o tempo utilizado para o envio de cada parte da imagem (subvetor) é maior que o tempo gasto pelos escravos para realizar o processamento (formação e ordenação desses subvetores), o que a torna uma aplicação mais voltada para comunicação do que para processamento.

5.2. Imagens de Tamanho 1 MB

As Figuras 5.6 a 5.10 apresentam os resultados da execução seqüencial e paralela dos filtros de mediana e de detecção de bordas, utilizando imagens de tamanho 1 MB com os mesmos tamanhos de máscaras citados na seção anterior.

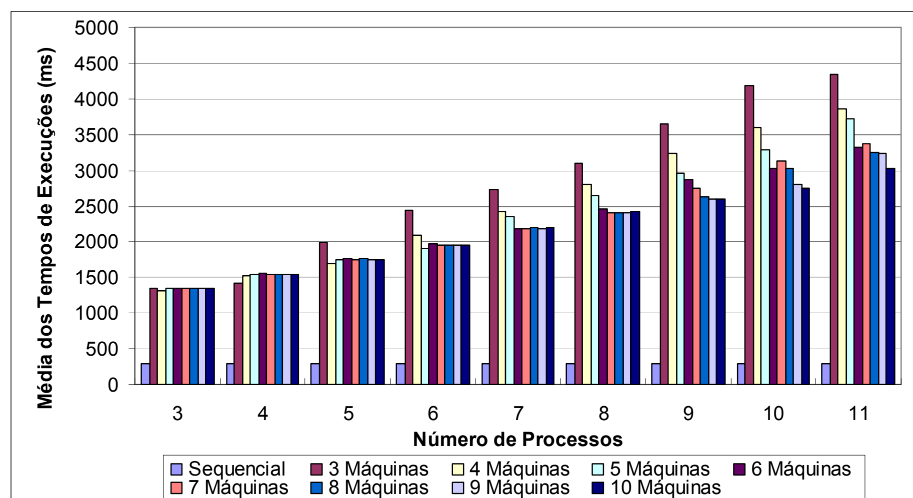


Figura 5.6 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 1 MB.

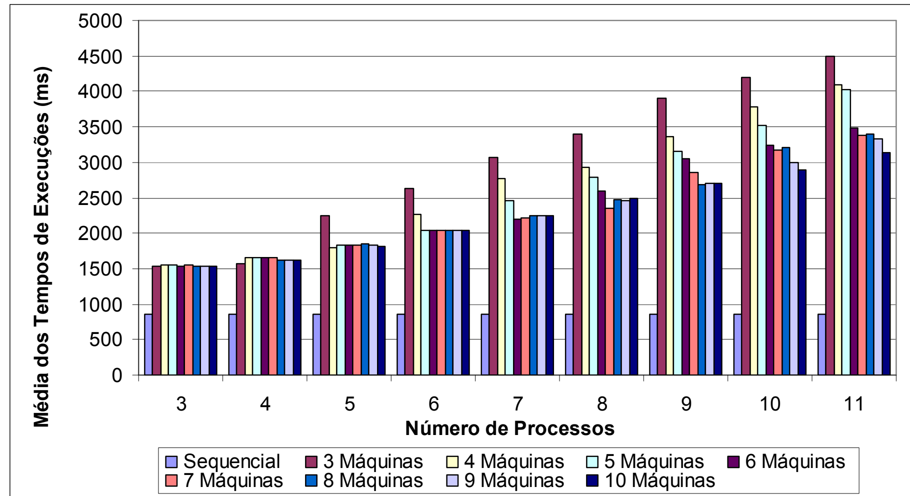


Figura 5.7 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 1 MB.

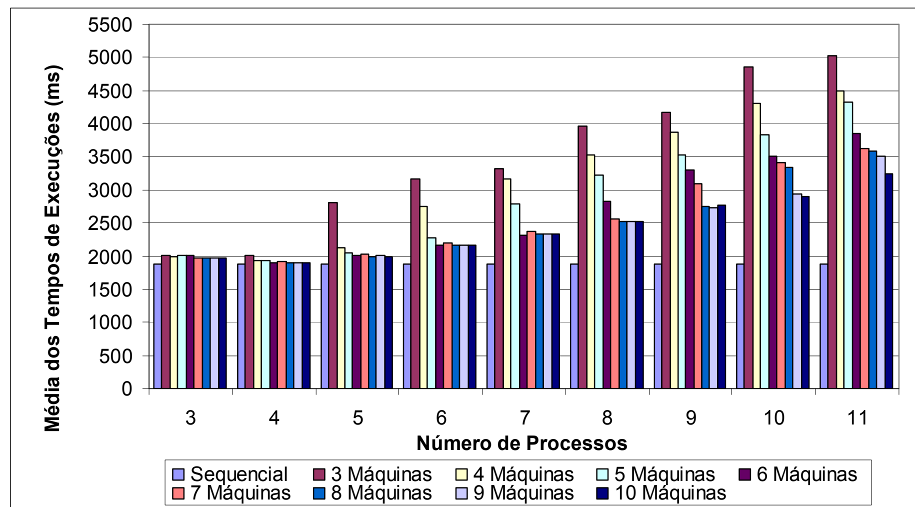


Figura 5.8 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 1 MB.

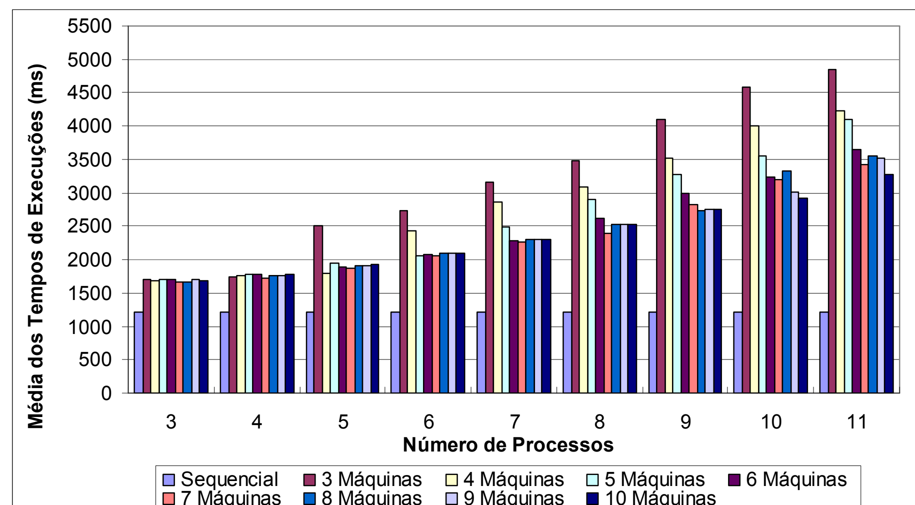


Figura 5.9 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 1 MB.

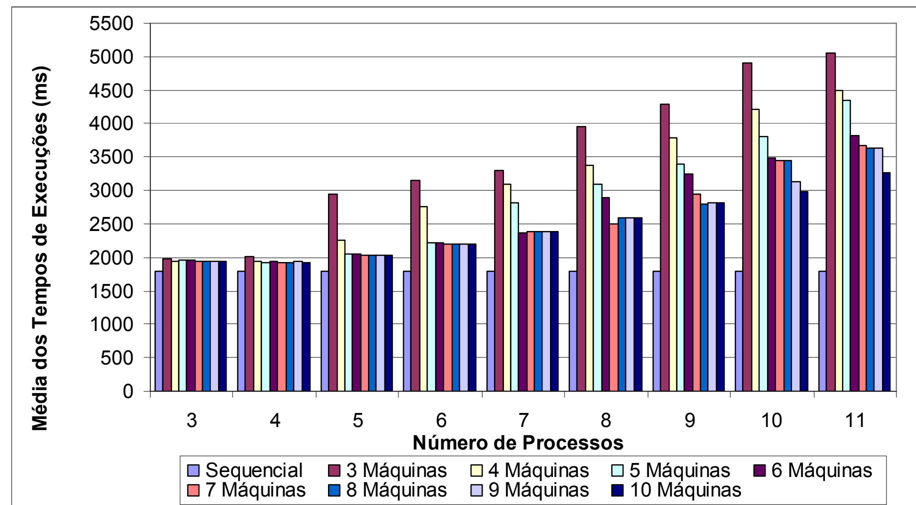


Figura 5.10 – Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 1 MB.

Conforme pode ser observado pelas figuras, a paralelização de imagens de tamanho 1 MB (independentemente do tamanho da máscara, do número de máquinas e número de processos utilizados), ainda não impõe melhoria (comprovação dada a partir das Tabelas A.6 a A.10 do Anexo A), da mesma maneira que as imagens de tamanho 500 KB, devido ao tamanho da imagem e aos cálculos realizados serem pequenos.

5.3. Imagens de Tamanho 11 MB

As Figuras 5.11 a 5.15 apresentam os resultados da execução seqüencial e paralela dos filtros utilizando imagens de tamanho 11 MB.

Na Figura 5.11 é possível observar que com a utilização de imagens de 11 MB, da mesma forma com a utilização de imagens de tamanhos 500 KB e 1 MB, o tempo de execução do algoritmo seqüencial é ainda significativamente melhor (comprovação dada a partir da Tabela A.11 do Anexo A) que o tempo do mesmo em paralelo independentemente do número de máquinas e da quantidade de processos iniciados em paralelo, utilizando máscara de tamanho 3x3.

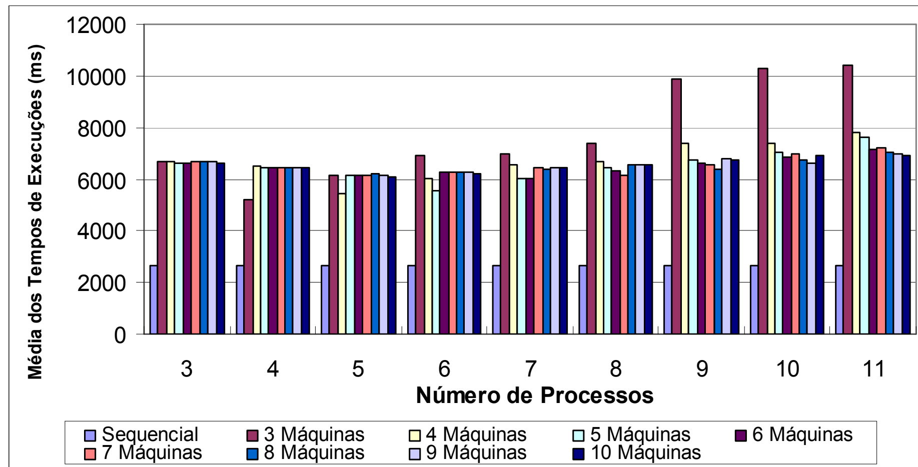


Figura 5.11 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 11 MB.

Os resultados das mesmas execuções anteriores, levando-se em consideração as máscaras 5x5 e 7x7, são apresentados pelas Figuras 5.12 e 5.13, respectivamente.

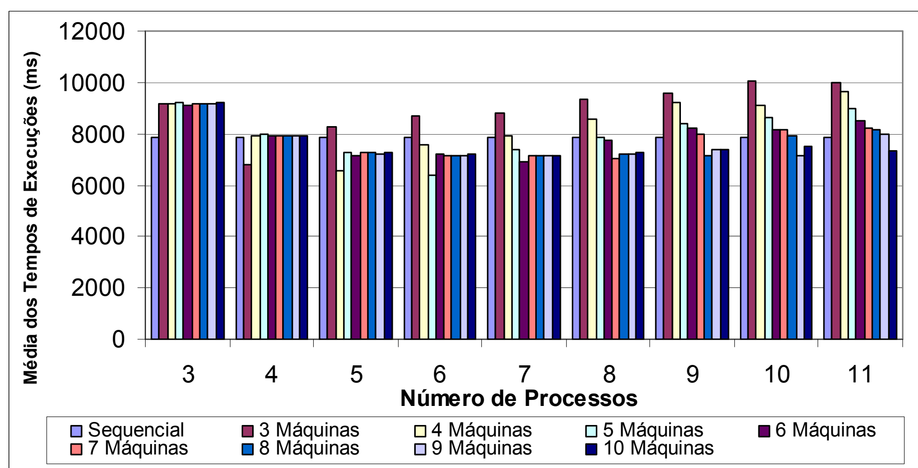


Figura 5.12 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 11 MB.

Pela Figura 5.12 é possível observar que em alguns casos o tempo de execução do algoritmo paralelo passa a ser significativamente melhor (comprovação dada a partir da Tabela A.12 do Anexo A) que o tempo seqüencial. Isso pode ser observado principalmente para situações em que o número de processos equivale ou é maior que o número de máquinas.

À medida que se aumenta o número de processos para certa quantidade de máquinas, o desempenho diminui, tornando-se, em alguns casos, maior que o tempo médio seqüencial. Isso acontece porque quando se aumenta o número de processos pode haver uma sobrecarga na comunicação, passando a ser um fator evidente de queda de desempenho.

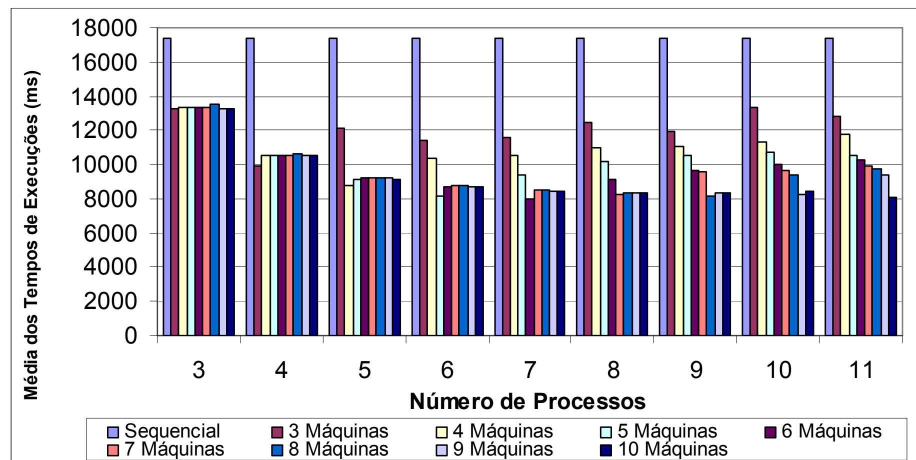


Figura 5.13 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 11 MB.

Diferentemente das máscaras 3x3 e 5x5, quando utilizada a máscara 7x7, independentemente do número de processos iniciados e do número de máquinas, o tempo médio paralelo é sempre melhor que o tempo seqüencial (comprovação dada a partir da Tabela A.13 do Anexo A), uma vez que o processamento é alto e o aumento no número de processos ainda implica em ganho de desempenho sobre a sobrecarga de comunicação.

Essa diferença e significativa melhora de desempenho, quando se faz uso de uma arquitetura paralela distribuída, dá-se pelo fato dos cálculos efetuados pelas máscaras serem volumosos, o que garante uma melhoria do seu uso em paralelo uma vez que a comunicação imposta se torna menos relevante quando comparada ao ganho em relação aos cálculos efetuados. Isso significa que a quantidade de comunicação exigida exerce menor influência na avaliação, visto que o volume de dados a serem transmitidos é menor que o volume de dados a ser processado.

O número de processos mostrou ser um fator muito importante. O melhor resultado é obtido quando se tem um número de processos maior (em uma unidade) do que o número de máquinas participantes. Isso é devido ao mpiJava utilizar um processo mestre que é contabilizado, mas não efetua a tarefa escrava propriamente dita.

Pelas Figuras 5.14 e 5.15 (com a utilização de máscaras de tamanho 9x9 e 11x11, respectivamente), é possível observar que independentemente do número de processos iniciados, o tempo médio paralelo é sempre melhor que o seqüencial (comprovação dada a partir das Tabelas A.14 e A.15 do Anexo A).

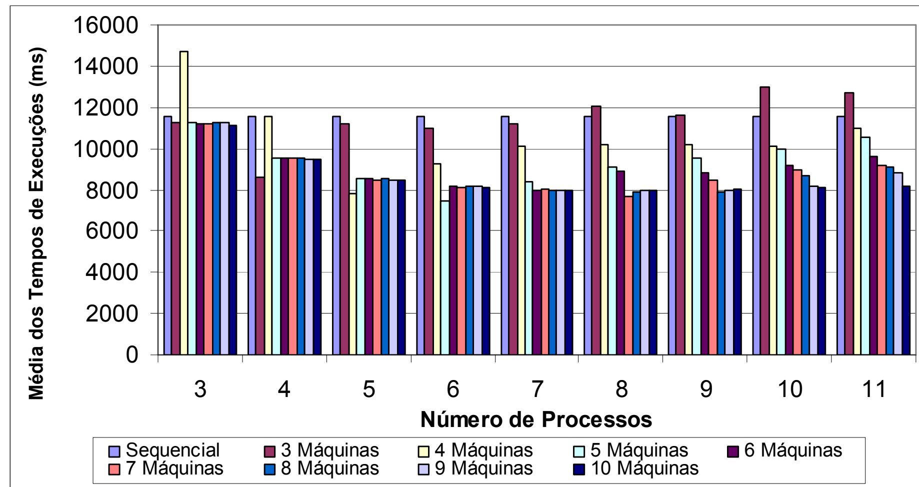


Figura 5.14 – Média do tempo de execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 11 MB.

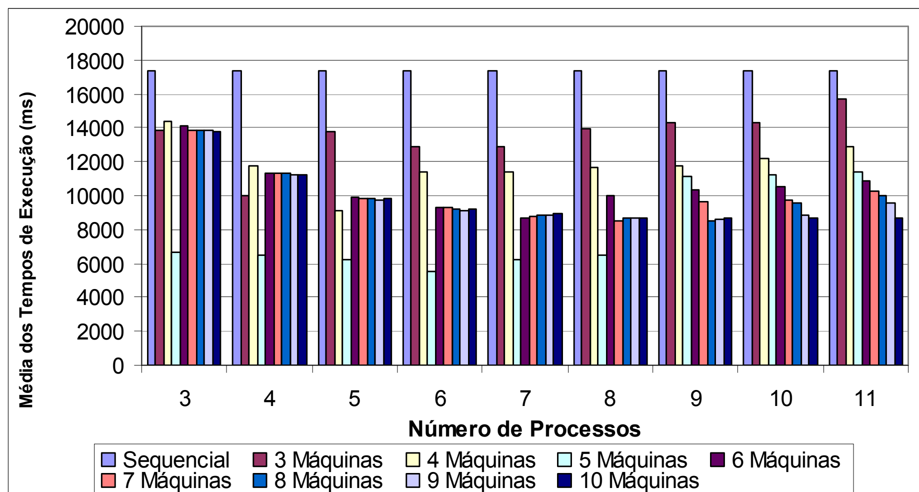


Figura 5.15 – Média do tempo de execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 11 MB.

Como as técnicas apresentadas (filtro de mediana e de detecção de bordas) trabalham no domínio espacial é interessante observar, ainda, que o tamanho da máscara exerce influência no desempenho do processamento, pois o esforço computacional aumenta proporcionalmente a esta dimensão devido ao incremento na quantidade de operações matemáticas que devem ser realizadas.

5.4. Imagens de Tamanho 21 MB

As Figuras 5.16 a 5.20 apresentam os resultados da execução seqüencial e paralela dos filtros utilizando imagens de tamanho 21 MB.

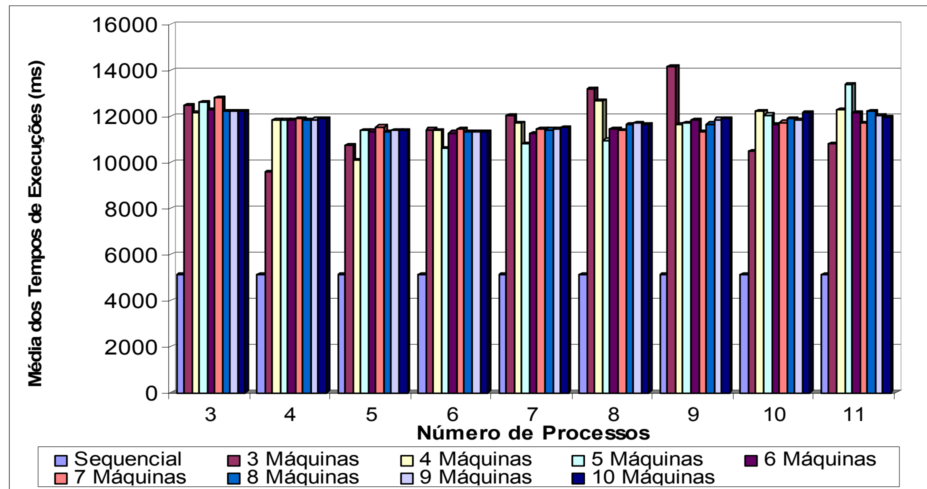


Figura 5.16 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 3x3 utilizando imagens de 21 MB.

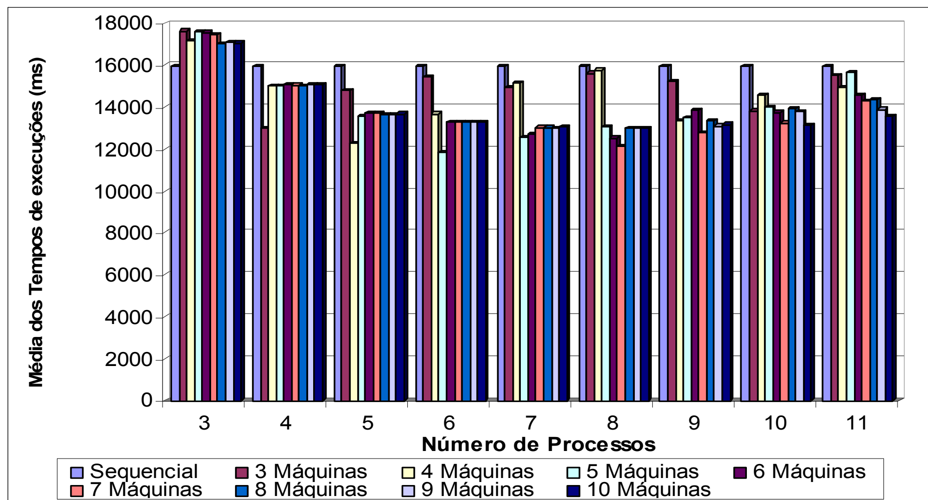


Figura 5.17 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 5x5 utilizando imagens de 21 MB.

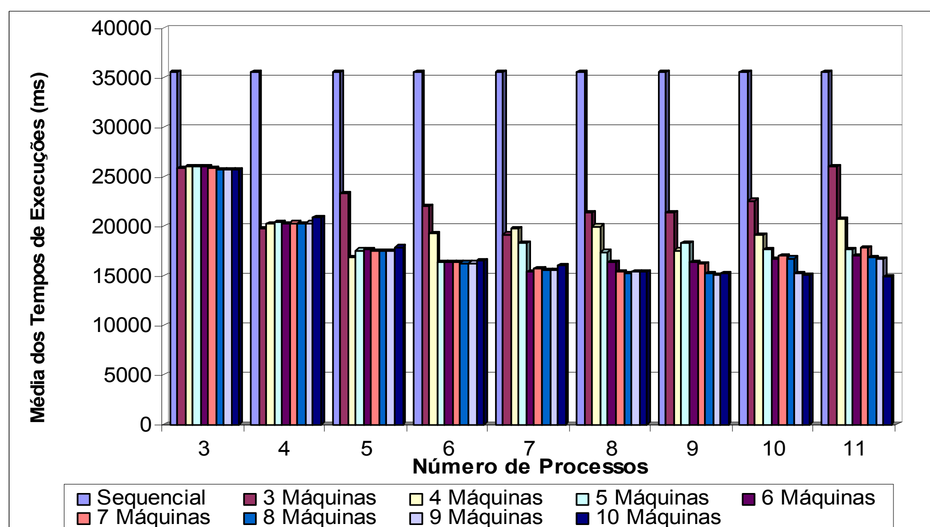


Figura 5.18 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro mediana de máscara 7x7 utilizando imagens de 21 MB.

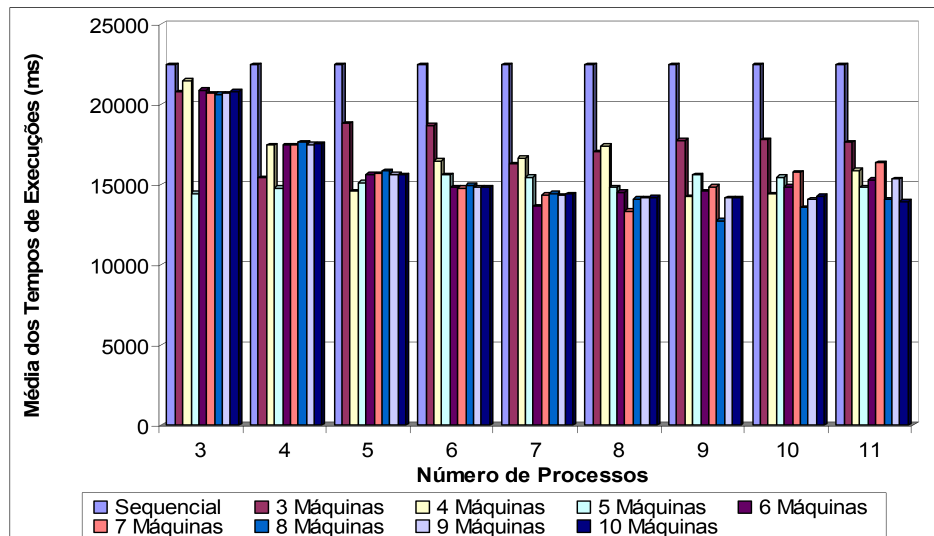


Figura 5.19 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 9x9 utilizando imagens de 21 MB.

Pelas Figuras 5.16 a 5.20 é possível observar que o tempo de execução paralelo dos filtros, independentemente do número de máquinas utilizadas, do número de processos iniciados e do tamanho das máscaras (incluindo a 3x3) apresenta um melhor desempenho se comparado ao tempo seqüencial (comprovação dada a partir das Tabelas A.16 a A.20 do Anexo A). A imagem utilizada é grande, dessa forma, os cálculos efetuados pelas máscaras são volumosos, garantindo, dessa forma, uma melhoria bastante significativa do uso em paralelo uma vez que a comunicação imposta se torna desprezível quando comparada aos cálculos realizados.

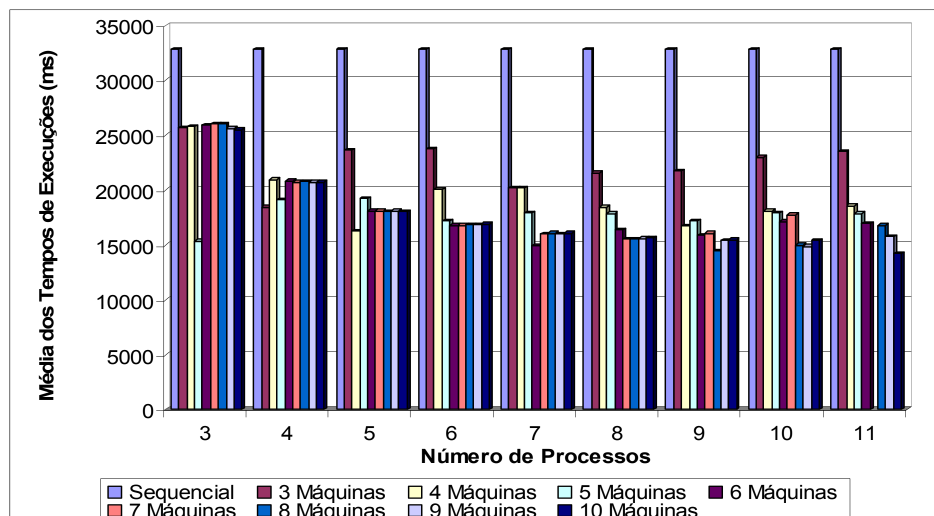


Figura 5.20 - Gráfico da execução da aplicação seqüencial comparada com a aplicação paralela do filtro de detecção de bordas de máscara 11x11 utilizando imagens de 21 MB.

A seguir estão compiladas algumas considerações sobre avaliações de desempenho entre tamanhos de imagens e máscaras. Lembrando que foram utilizadas imagens de tamanhos 500 KB, 1 MB, 11 MB e 21 MB e máscaras de tamanhos 3x3, 5x5, 7x7, 9x9 e 11x11. Foram calculadas as medidas *speedup* e eficiência, e essas medidas foram representadas em percentuais, conforme pode ser observado pela Tabela 5.1.

Para a obtenção dos percentuais apresentados pelas Tabelas 5.1 e 5.2, foram calculados, para cada quantidade de máquinas, os *speedups* e eficiência dos seus melhores tempos de execução (em milissegundos). Obtida a eficiência de cada uma das quantidades de máquinas utilizadas foi realizada uma média desses valores, representando os percentuais apresentados.

Tabela 5.1 – Eficiência (em %) na paralelização do filtro de mediana.

Tamanho	Imagem			
Máscara	500 KB	1 MB	11 MB	21 MB
3x3	-686,66	-452,76	-193,51	-187,01
5x5	-244,48	-179,57	80,74	74,31
7x7	-146,86	-100,58	45,89	42,03

Tabela 5.2 – Eficiência (em %) na paralelização do filtro de de detecção de bordas.

Tamanho	Imagem			
Máscara	500 KB	1 MB	11 MB	21 MB
9x9	-205,40	-138,10	64,44	56,90
11x11	-146,31	-107,87	31,92	43,32

Exemplificando, para imagens de tamanho 21 MB, utilizando-se máscara de tamanho 7x7, obtiveram-se os melhores tempos de execução para cada quantidade de máquinas utilizadas. Para 3 máquinas, o melhor tempo médio (utilizando-se 4 processos) foi 19749,5 milissegundos. Os demais valores foram 16819,70; 16331,1; 15391; 15399,2; 15203,3; 15294,1 e 14931,6, para 4, 5, 6, 7, 8, 9 e 10 máquinas respectivamente.

Foi calculado o *speedup* de cada um desses valores, obtendo-se, assim, uma média dos valores da eficiência encontrados por meio dos *speedups*. A média obtida foi de 0,75699025, determinando que a paralelização do filtro de detecção de bordas Sobel, utilizando-se imagem de tamanho 21MB e máscara de tamanho 7x7, é 75,6990% mais eficiente que a aplicação do mesmo de forma seqüencial.

Pelas Tabelas, é possível perceber que o ganho de desempenho do tempo de processamento paralelo cresce em proporções muito grandes em relação ao tamanho da

imagem utilizada. Esse crescimento pode ser melhor observado pelas curvas, referentes às máscaras 3x3, 5x5 e 7x7, na Figura 5.21 e às máscaras 9x9 e 11x11, na Figura 5.22.

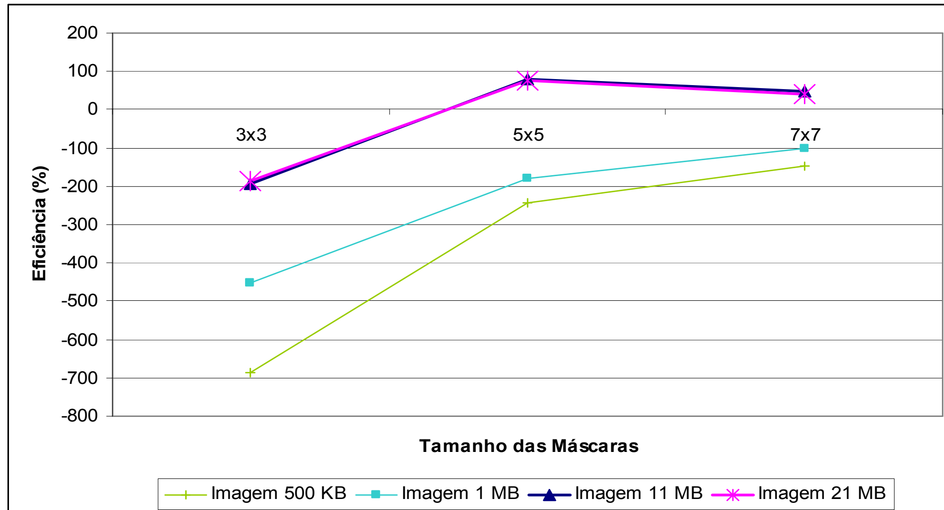


Figura 5.21 - Gráfico da eficiência (em %) na paralelização do filtro de mediana utilizando imagens de diferentes tamanhos.

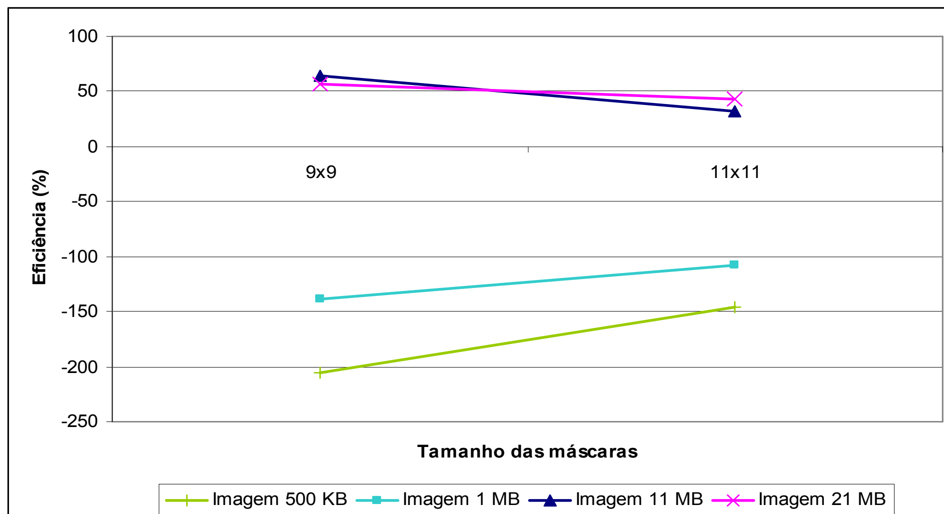


Figura 5.22 - Gráfico da eficiência (em %) na paralelização do filtro de detecção de bordas utilizando imagens de diferentes tamanhos.

5.5. Considerações Finais

Esse capítulo deu ênfase às análises de desempenho das implementações dos filtros de mediana e de detecção de bordas Sobel.

O tamanho da máscara utilizada na implementação dos filtros exerce influência no desempenho do processamento, pois o esforço computacional aumenta proporcionalmente a esta dimensão devido ao incremento na quantidade de operações matemáticas realizadas.

Observa-se também que a proporção de aumento de tamanho da imagem se reflete no tempo de processamento paralelo de forma exponencial. Quanto maior o tamanho da imagem, melhor o resultado obtido com a paralelização.

O capítulo seguinte apresenta as conclusões que se chegam a partir dos resultados obtidos com a análise de desempenho realizada nesse capítulo.

6. CONCLUSÕES

O presente trabalho de pesquisa teve como objetivo principal demonstrar a viabilidade na otimização do tempo de processamento de imagens, mais especificamente de imagens médicas. Para tanto, utilizou-se da programação paralela e distribuída, viabilizada por sistemas distribuídos e pela biblioteca de passagem de mensagens mpiJava.

Os objetivos, para algumas situações, foram alcançados e comprovados por meio de testes das implementações dos algoritmos estudados. Partindo-se dos resultados obtidos pode-se verificar que existe um ganho em se fazer uso do processamento paralelo distribuído quando se pensa em processamento de imagens médicas.

Os algoritmos de suavização (filtro mediana) e de segmentação (detecção de bordas) foram estudados de forma detalhada para a obtenção de maior eficiência nas implementações realizadas. Após o domínio detalhado dos algoritmos, os mesmos foram implementados numa primeira instância na linguagem Java de forma seqüencial, em seguida, suas versões paralelas foram implementadas para realizar as avaliações de desempenho.

Nessa avaliação foram utilizados critérios que avaliaram tempo de execução, número de máquinas, número de processos, tamanho de máscaras e tamanho das imagens utilizadas.

Na análise de desempenho realizada por meio de testes das implementações dos algoritmos na forma seqüencial e paralela, foram obtidos os seguintes resultados e informações:

- Independentemente do tipo de algoritmo de ordenação utilizado (*bubblesort* ou *shellsort*), na implementação do algoritmo filtro mediana, os tempos de execução em paralelo apresentaram-se satisfatórios, dependendo dos tamanhos das imagens e das máscaras utilizadas (SAITO et al., 2007f).
- A média dos tempos de execução da implementação paralela do algoritmo utilizando o método *shellsort* é menor que a média dos tempos utilizando o método *bubblesort*, visto que a versão seqüencial deste último é muito mais lenta que a do *shellsort* (SAITO et al., 2007f).
- Apesar do resultado não ser favorável à execução paralela dos filtros utilizando-se imagens pequenas, e em alguns casos na aplicação de máscaras de tamanho 3x3 e 5x5 (filtro mediana), foi possível observar que para processamentos intensos, como é o caso da utilização de máscaras de tamanhos 7x7 (filtro mediana), 9x9 e 11x11 (filtro de detecção de bordas), o uso do processamento paralelo é bastante

vantajoso (SAITO et al., 2007c, SAITO et al., 2007d, SAITO et al., 2007e, SAITO et al., 2007g).

- A melhor média dos tempos de execuções é obtida quando se tem um processo a mais (em uma unidade) que o número de máquinas, conforme mencionado no capítulo anterior.
- À medida que se aumenta o número de processos, diminui-se o desempenho, sendo um resultado já esperado, demonstrando o impacto que o tempo de comunicação entre as máquinas exerce sobre o tempo de processamento.

Acredita-se que a utilização de outros filtros (no domínio espacial) também possa prover melhora significativa de desempenho para a versão paralela quando comparada à execução seqüencial. Com base nisso, o desenvolvimento de outros algoritmos de processamento de imagens, mais complexos, pode ser realizado em uma próxima etapa de projeto.

Acredita-se também que a paralelização dos algoritmos de ordenação, utilizados na implementação do filtro mediana, possa prover melhora significativa no desempenho dos mesmos, uma vez que já se tem comprovado na literatura (CORTÉS, 1999) que a paralelização de algoritmos de ordenação permite um aumento de desempenho quando comparado a sua versão seqüencial.

Demais testes com outras modalidades de imagens, diferentes das mamográficas utilizadas até então, podem ser proferidos, permitindo, dessa forma, a obtenção de um conjunto maior de dados a partir do qual mais informações poderão ser extraídas.

Embora a computação paralela distribuída viabilize a transferência de dados e o uso de múltiplas máquinas, possibilitando os resultados obtidos e já apresentados, ela não provê por si própria (o ambiente de passagem de mensagem utilizado provê somente o escalonamento *round-robin*) mecanismos eficientes para a distribuição de processos a processadores (escalonamento de processos objetivando o balanceamento de cargas) (Branco, 2004).

O uso de escalonadores de processos ou até mesmo de mecanismos para prover uma distribuição voltada ao balanceamento de cargas deve prover uma melhora ainda mais significativa em relação aos resultados obtidos (Ferrari & Zhou, 1987; Branco, 2004; Branco *et al.*, 2006), e é nesse sentido que estudos devem ser efetuados para averiguar a qual classe de aplicação (*CPU-Bound*, *Memory-Bound*, *Network-Bound* ou *Disk-Bound*) as aplicações de processamento de imagens se enquadram e posteriormente averiguar quais índices de carga e/ou desempenho podem e devem ser utilizados para que ocorra uma melhora ainda maior no tempo de processamento desses algoritmos em paralelo.

6.1. Produção Bibliográfica

Trabalhos com os resultados da paralelização das filtragens mediana e detecção de bordas e a comparação de dois ambientes de passagem de mensagens, mpiJava e JPVM, resultaram em artigos e foram publicados em alguns eventos científicos.

6.1.1. Artigos Completos Publicados em Periódicos

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Otimização de Algoritmos de Processamento de Imagens Médicas Utilizando a Computação Paralela. Revista Diciplinarum Scientia. , v.1, p.1 - 8, 2007.

6.1.2. Trabalhos Completos Publicados em Anais de Congressos

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela In: CISCI 2007: 6a. Conferencia Iberoamericana en Sistemas, Cibernética e Informática, 2007, Orlando, Flórida. **6a. Conferencia Iberoamericana en Sistemas, Cibernética e Informática: CISCI 2007**. , 2007. v.1. p.1 – 1

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Otimização do Processamento de Imagens Médicas: Uma Abordagem Utilizando Java In: III Workshop de Visão Computacional – WVC’ 2007, 2007, São José do Rio Preto. **Anais do III Workshop de Visão Computacional – WVC’ 2007**. , 2007.

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Processamento de Imagens Médicas: Otimização no Tempo de Execução Usando Computação Paralela Distribuída In: III Simpósio de Instrumentação e Imagens Médicas, 2007, São Carlos. **Anais do III Simpósio de Instrumentação e Imagens Médicas**. , 2007.

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Uma Abordagem Utilizando mpiJava para a Paralelização de Algoritmos de Processamento de Imagens In: VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007, Gramado - Rio Grande do Sul. **Anais do VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho**. , 2007.

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Otimização de Algoritmos de Processamento de Imagens Médicas Utilizando a

Computação Paralela In: VI Simpósio de Informática da Região Centro – SIRC 2007, 2007, Santa Maria. **Anais do VI SIRC/RS 2007 - VI Simpósio de Informática da Região Centro - Centro Universitário Franciscano (UNIFRA)**. , 2007.

- SAITO, Priscila Tiemi Maeda, SABATINE, R. J., NUNES, F. L. S., BRANCO, K. R. L. J. C. Uso da Computação Paralela Distribuída para Melhoria no Tempo de Processamento de Imagens Médicas In: XIV ERI/PR - XIV Escola Regional de Informática da SBC, 2007, Guarapuava. **Anais do XIV ERI/PR - XIV Escola Regional de Informática da SBC**. , 2007. v.1. p.36 - 47

- SABATINE, R. J., SAITO, Priscila Tiemi Maeda, NUNES, F. L. S., BRANCO, K. R. L. J. C. Utilização da Computação Paralela Distribuída na Otimização do Processamento de Imagens Médicas Fazendo Uso do JPVM In: VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007, Gramado - Rio Grande do Sul. **Anais do VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho**. , 2007.

6.1.3. Resumos Publicados em Anais de Congressos

- SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Paralelização de Algoritmos de Processamento de Imagens Médicas Fazendo uso de mpiJava.* In: 15º SIICUSP – 15º Simpósio Internacional de Iniciação Científica da Universidade de São Paulo., 2007, São Carlos. Anais do Simpósio Internacional de Iniciação Científica da USP, 2007b.

- SAITO, Priscila Tiemi Maeda, BRANCO, K. R. L. J. C., NUNES, F. L. S., SABATINE, R. J. Paralelização de Algoritmos de Processamento de Imagens Médicas no Domínio Espacial - Uma Abordagem Usando Java In: XV CIC UFSCar - XV Congresso de Iniciação Científica da UFSCar - 7ª Jornada Científica da UFSCar, 2007, São Carlos. **Anais de Eventos da UFSCar - XV Congresso de Iniciação Científica da UFSCar - 7ª Jornada Científica da UFSCar**. , 2007. v.3. p.52 - 52

- SAITO, Priscila Tiemi Maeda, BRANCO, K. R. L. J. C. Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela In: VIII SIC UNIVEM – VIII Seminário de Iniciação Científica, 2007, Marília. **Anais do Seminário de Iniciação Científica do UNIVEM**, 2007.

- SABATINE, R. J., BRANCO, K. R. L. J. C., NUNES, F. L. S., SAITO, Priscila Tiemi Maeda. Utilização da Computação Paralela Distribuída na Otimização do Processamento de Imagens Médicas Utilizando JPVM In: XIV CIC UFSCar - XIV Congresso de Inciação Cientifica da UFSCar, 2007, São Carlos. **Anais de Eventos da UFSCar - XIV Congresso de Inciação Cientifica da UFSCar**. , 2007. v.3. p.49 - 49

REFERÊNCIAS

ALMASI, G. S., GOTTLIEB, A. *Highly Parallel Computing.* 2a. ed. The Benjamin Cummings Publishing Company, Inc., 1994.

ACHCAR, J. A.; RODRIGUES, J. *Introdução à Estatística para Ciências e Tecnologia.* ICMSC-USP, São Carlos – Apostila de Consulta, 1995.

AMORIM, C. L. *Uma Introdução a Computação Paralela e Distribuída*. VI Escola de Computação, 1988.

ANSA. *The Advanced Network Systems Architecture (ANSA) Reference Manual*. Castle Hill, Cambridge, England, 1989.

BAKER, M.; CARPENTER, B.; FOX, G.; KO, S. H.; LIM, S. (1999). *mpiJava: An Object-Oriented Java Interface to MPI*. Presented at International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999.

BAKER, M.; CARPENTER, B.; KO, S. H.; LI, X. (1998). *mpiJava: A Java Interface to MPI.* Submitted to First UKWorkshop on Java for High Performance Network Computing, Europar 1998.

BALLARD, D. H.; BROWN, C. M. (1982). *Computer Vision*. Englewood Cliffs, New Jersey, Prentice-Hall Inc, 1982.

BARBOSA, J. M. G. *Paralelismo em Processamento e Análise de Imagem Médica*. 2000, 240f. Tese (Doutorado) - Departamento de Engenharia Electrotécnica e de Computadores, Faculdade de Engenharia da Universidade do Porto, 2000.

BARROS JUNIOR, E. M. ; RIBEIRO, L. A.; GURGACZ, C. V.; WANGENHEIM A. V. *SIAPDI: Um Sistema de Processamento Distribuído de Imagens Médicas com CORBA*. III WRNP2 Workshop RNP2, p.1-4, Florianópolis, 21 e 22 de maio de 2001.

BEGUELIN, A.; GEIST, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for NetWork Parallel Computing*. The MIT Press, s. 1, 1994.

BRANCO, K. R. L. J. C.; SANTANA, M. J.; SANTANA, R. H. C.; BRUSCHI, S. M. *PIV and WPIV: Performance Index For Heterogeneous Systems Evaluation.* In: 2006 IEEE International Symposium on Industrial Electronics. Volume: 1. p. 323-328, 2006.

BRANCO, K. R. L. J. C. *Índices de carga e desempenho em ambientes paralelos/distribuídos – modelagem e métricas.* São Paulo: Instituto de Ciências Matemáticas e Computação, 2004, 260 f. Grau: Tese (Doutorado em Ciência da Computação e Matemática Computacional) Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, São Carlos, 2004.

BRANCO, K. R. L. J. C. *Extensão da Ferramenta de Apoio à Programação Paralela (F.A.P.P.) para Ambientes Paralelos Virtuais*. 1999, 152f. Dissertação (Mestrado) - Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 1999.

BOSQUE, J. L.; ROBLES, O. D.; RODRÍGUEZ, A.; PASTOR, L. *Study of a Parallel CBIR Implementation using MPI*. Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception, p.1-10, 2000.

BURNS, G.; DAOUD, R.; VAIGL, J. *LAM: An Open Cluster Environment for MPI.* In: Proceedings of Supercomputing Symposium. [s.n.], 1994. p. 379–386. Disponível em: <http://www.lam-mpi.org/download/files/lam-papers.tar.gz>. Acesso em: 16 jan. 2006.

CÁCERES, E. N.; MONGELLI, H.; SONG, S. W. *Algoritmos Paralelos Usando CGM/PVM/MPI: Uma Introdução*. In: As Tecnologias da Informação e a Questão Social. Ed. Porto Alegre: Sociedade Brasileira de Computação, 2001.

CARPENTER, B.; FOX, G.; KO, S. H.; LIM, S. *mpiJava 1.2: API Specification*. 1999. Disponível em: http://grid.u-strasbg.fr/p2pmpi/documentation/www.hpjava.org/mpiJava-spec/mpiJava-spec.html. Acesso em: 16 janeiro 2007.

CHAN, H-P.; DOI, K.; VYBORNY, C. J.; SCHMIDT, R. A.; METZ, C.E.; LAN, K. L.; OGURA, T.; WU, Y.; MACMAHON, H. (1990). *Improvement in Radiologists' Detection of Clustered Microcalcification on Mammograms: The Potential of Computer-Aided Diagnosis*. Investigative Radiology, v. 25, p. 1102-1110, 1990.

CORTÉS, O. A. C. (1999). *Desenvolvimento e Avaliação de Algoritmos Numéricos Paralelos.* Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - USP, São Carlos, São Paulo, 1999.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems: Concepts and Design*. 3. ed. Harlow: Addison Wesley, 771 p, 2001.

______. *Distributed Systems: Concepts and Design.* 4. ed. Harlow: Addison Wesley, 944 p, 2006.

DOI, K. (1991). *Computer-Aided Diagnosis: Present and Future. A New Horizon on Medical Physics and Biomedical Engineering*. H. Abe; K. Atsumi; T. Iiuma; M. Saito; M. Inoue, eds. Elsevier Science Publishers B.V., p. 59-66, 1991.

DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. (Ed.). *Sourcebook of Parallel Computing*. San Francisco: Morgan Kaufmann Publishers, 842 p, 2003.

DONGARRA, J. J.; OTTO, S. W.; SNIR, M.; WALKER, D. *An Introduction to the MPI Standard.* University of Tennessee Technical Report CS-95-274, http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html,1995.

DUNCAN, R. A S*urvey of Parallel Computer Architectures.* In: *Computer*. Los Alamitos, CA, USA: IEEE Computer Society Press, (Survey & Tutorial Series, 2). P. 5–16, 1990.

ELLIS, I. O.; GALEA, M. H.; LOCKER, A. (1993). *Early Experience in Breast Cancer Screening: Emphasis on Development of Protocols for Ttriple Assessment*. The Breast, v. 2, p. 148-153, 1993.

FERRARI, D.; ZHOU, S. *An Empirical Investigation of Load Indices for Load Balancing Applications.* In Proceedings of Performance'87, the 12th Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation. 1987.

FLYNN, M. J. *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, v. C, n. 21, pp. 948-960, 1972.

FLYNN, M. J.; RUDD, K. W. *Parallel Architectures. ACM Computing Surveys*, v. 28, n. 1, p. 67–70, mar. 1996.

FOSTER, I. *Designing and Bulding Parallel Programs: Concepts and Tools for Parallel Software Engineering*. New York: Addison-Wesley Publishing Company, 1995. Disponível em: <http://www-unix.mcs.anl.gov/dbpp/text/book.html>. Acesso em: 16 out. 2006.

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. *PVM 3 User's Guide and Reference Manual*. Oak National Laboratory, Setembro, 1994.

GIESS, C.; MAYER, A.; EVERS, H.; MEINZER, H. P. *"Medical Image Processing and Visualization on Heterogeneous Clusters of Symmetric Multiprocessors using MPI and POSIX Threads"*. Parallel Processing Symposium, 1998. Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing 1998.

GIGER, M.L. (2000). *Computer-Aided Diagnosis of Breast Lesions in Medical Images*. Computing in Science & Engineering, v. 2, n. 5, p. 39-45, 2000.

GIGER, M. L.; MACMAHON, H. (1996). *Image Processing and Computer-Aided Diagnosis*. Radiologic Clinics of North America, v.34, n.3, p. 565-595, 1996.

GONZALEZ, R. C.; WOODS, R. E. (2002). *Digital Image Processing*. Addison-Wesley Publishing Company, Massachusetts, 2a. Edição, 2002.

GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. *Introduction to Parallel Computing*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 689 p, 2003.

GROPP, W., SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

GROPP, B.; LUSK, R.; SKJELLUM, T.; DOSS, N. *Portable MPI Model Implementation.* Argonne National Laboratory, July 1994.

HARIRI, S.; PARASHAR, M. *Tools and Environments for Parallel and Distributed Computing*. Hoboken, New Jersey: Wiley-Interscience, 232 p, 2004.

HWANG, K.; BRIGGS, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill International Editions, 1984.

HWANG, K.; XU, Z. *Scalable Parallel Computing: Technology, Architecture, Programming*. 1. ed. New York: McGraw-Hill, 802 p, 1998.

ISO, INTERNATIONAL STANDARDS ORGANIZATION. *Basic Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use*. ISO/IEC JTC1/SC212/WG7 CD 10746-1, 1992.

JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

KIRNER, C. *Arquitetura de Sistemas Avançados de Computação*. Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, pp. 307-353, 1991.

LAM. *LAM/MPI Parallel Computing.* Disponível em: <http://www.lam-mpi.org>. Acesso em: 09 Novembro 2006.

LASTOVETSKY, A. L. *Parallel Computing on Heterogeneous Networks.* New York, NY, USA: John Wiley & Sons, Inc., 440 p, 2003.

MACDONALD, N.; MINTY, E.; HARDING, T.; BROWN, S. *Writing Message-Passing Parallel Programs with MPI.* Technical report, Edinburg Parallel Computing Centre, The University of Edinburgh. Course notes. 1996.

MCBRYAN, O. A. *An Overview of Message Passing Environments.* Parallel Computing, v. 20, pp. 417-444, 1994.

MINTCHEV, S. *Writing Programs in JavaMPI.* TR MAN-CSPE-02, Univ. of Westminster, London, UK, 1997.

MORIN, S.; KOREN, I.; KRISHNA, C. M. *JMPI: Implementing the Message Passing Standard in Java.* In IPDPS 02: Proceedings of the 16th International Parallel and Distributed Processing Symposium, p.191, IEEE Computer Society, 2002.

MPI. *MPI-2: Extensions to the Message-Passing Interface.* 1997. Disponível em: <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>. Acesso em: 14 Outubro 2003.

MPICH2. *Download Document Home Page.* Disponível em: http://www-unix.mcs.anl.gov/mpi/mpich2/. Acesso em dezembro, 2006.

MPIJAVA. *The HPJava Project document home page.* Disponível em: http://www.hpjava.org/mpiJava/doc/api/. Acesso em janeiro, 2007.

MÜLLENDER, S. J. *Distributed Systems*. 2ª ed., ACM PRESS Frontier Series, Addison-Wesley Publishing Company, 1993.

NAVAUX, P. O. A. *Introdução ao Processamento Paralelo*. RBC- Revista Brasileira de Computação, v. 5, no 2, pp.31-43, Outubro, 1989.

NCSA – NATIONAL CENTER FOR SUPERCOMPUTING APPLICATION AT THE UNIVERSITY OF ILLINOIS. *Introduction to MPI.* Disponível em http://webct.ncsa.edu:8900/public/MPI/index.html. Acesso em: 17 dezembro 2006.

NEVIN, N. J. *The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster.* Columbus, Ohio, 1996.

NICOLESCU, C.; JONKER, P. *"A Data and Task Parallel Image Processing Environment for Distributed Memory Systems"*, International Conference on Parallel Processing Workshops (ICPPW'01), p.39-44, IEEE Computer, 2001.

NUNES, F. L. S. (2006a). *Introdução ao Processamento de Imagens Médicas para Auxílio ao Diagnóstico*. Breitman, K.; Anido, R. (Org). Atualizações em Informática. 1 ed. Rio de Janeiro: PUC-Rio, v. 1, p. 73-126, 2006.

NUNES, F. L. S. (2006b). *Processamento de Imagens Médicas para Sistemas de Auxílio ao Diagnóstico*. Rodello, I., A.; Brega, J. R. F.; Branco, K. R. L. J. C. (Org). ERI – Escola Regional de Informática São Paulo/Oeste. 1 ed. São Paulo: Marília/Bauru, cap. 4, p. 83-135, 2006.

NUNES, F. L. S. M. *"Investigações em Processamento de Imagens Mamográficas para Auxílio ao Diagnóstico de Mamas Densas"*. 2001. 230f. Tese (Doutorado) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2001.

PETRICK, N.; CHAN, H-P.; SAHINER, B. and WEI, D. (1996) *An adaptive density-weighted contrast enhancement filter for mammographic breast mass detection*, IEEE Transactions on Medical Imaging, v.15, n.1, p.59-67, 1996.

PEZZI, G. P. *Aplicação Paralela de um Filtro Gráfico*. 2005. Trabalho do Curso de Pós - Graduação em Ciência da Computação apresentado à Universidade Federal do Rio Grande do Sul – UFRGS, 2005.

QUINN, M.J. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, 1987.

______. *Parallel Computing: Theory and Practice*. 2. ed. New York: McGraw Hill, 1994. 446 p.

SABATINE, R. J.; BRANCO, K. R. L. J. C.; NUNES, F. L. S.; SAITO, P. T. M. *Utilização da Computação Paralela Distribuída na Otimização do Processamento de Imagens Médicas Utilizando JPVM*. In: XIV CIC UFSCar - XIV Congresso de Iniciação Cientifica da UFSCar, 2007, São Carlos. Anais de Eventos da UFSCar - XIV Congresso de Iniciação Cientifica da UFSCar, v.3. p.49 – 49, 2007a.

SABATINE, R. J.; SAITO, P. T. M.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Utilização da Computação Paralela Distribuída na Otimização do Processamento de Imagens Médicas Fazendo Uso do JPVM.* In: VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007, Gramado - Rio Grande do Sul. Anais do VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007b.

SAITO, P. T. M.; BRANCO, K. R. L. J. C. *Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela*. In: VIII SIC UNIVEM – VIII Seminário de Iniciação Científica, 2007, Marília. Anais do Seminário de Iniciação Científica do UNIVEM, 2007a.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Paralelização de Algoritmos de Processamento de Imagens Médicas Fazendo uso de mpiJava.* In: 15º SIICUSP – 15º Simpósio Internacional de Iniciação Científica da Universidade de São Paulo., 2007, Marília. Anais do Simpósio Internacional de Iniciação Científica da USP, 2007b.

SAITO, P. T. M.; BRANCO, K. R. L. J. C.; NUNES, F. L. S.; SABATINE, R. J. *Paralelização de Algoritmos de Processamento de Imagens Médicas no Domínio Espacial - Uma Abordagem Usando Java*. In: XV CIC UFSCar - XV Congresso de Iniciação Científica da UFSCar - 7ª Jornada Científica da UFSCar, 2007, São Carlos. Anais de Eventos da UFSCar - XV Congresso de Iniciação Científica da UFSCar - 7ª Jornada Científica da UFSCar, v.3. p.52 – 52, 2007c.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Otimização de Algoritmos de Processamento de Imagens Médicas Utilizando a Computação Paralela.* Revista Diciplinarum Scientia, v.1, p.1 - 8, 2007d.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Otimização de Algoritmos de Processamento de Imagens Médicas Utilizando a Computação Paralela*. In: VI Simpósio de Informática da Região Centro – SIRC 2007, 2007, Santa Maria. Anais do VI SIRC/RS 2007 - VI Simpósio de Informática da Região Centro - Centro Universitário Franciscano (UNIFRA), 2007e.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Otimização do Processamento de Imagens Médicas: Uma Abordagem Utilizando Java*. In: III Workshop de Visão Computacional – WVC' 2007, 2007, São José do Rio Preto. Anais do III Workshop de Visão Computacional – WVC' 2007, 2007f.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela*. In: CISCI 2007: 6a. Conferencia Iberoamericana en Sistemas, Cibernética e Informática, 2007, Orlando,

Flórida. 6a. Conferencia Iberoamericana en Sistemas, Cibernética e Informática: CISCI 2007, v.1. p.1 – 1, 2007g.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Processamento de Imagens Médicas: Otimização no Tempo de Execução Usando Computação Paralela Distribuída*. In: III Simpósio de Instrumentação e Imagens Médicas, 2007, São Carlos. Anais do III Simpósio de Instrumentação e Imagens Médicas, 2007h.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Uma Abordagem Utilizando mpiJava para a Paralelização de Algoritmos de Processamento de Imagens*. In: VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007, Gramado - Rio Grande do Sul. Anais do VIII WSCAD - VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007i.

SAITO, P. T. M.; SABATINE, R. J.; NUNES, F. L. S.; BRANCO, K. R. L. J. C. *Uso da Computação Paralela Distribuída para Melhoria no Tempo de Processamento de Imagens Médicas*. In: XIV ERI/PR - XIV Escola Regional de Informática da SBC, 2007, Guarapuava. Anais do XIV ERI/PR - XIV Escola Regional de Informática da SBC, v.1. p.36 – 47, 2007j.

SANTANA, R. H. C.; SANTANA, M. J.; SOUZA, M. A.; SOUZA, P. S. L; PIEKARSKI, A. E. T. *Computação Paralela.* Universidade de São Paulo. Departamento de Ciências de Computação e Estatística. São Carlos. 1997.

SCHNORR, L. M. *Paralelização do Filtro de Mediana*. 2003. Trabalho do Curso de Pós – Graduação em Ciência da Computação apresentado à universidade Federal do Rio Grande do Sul – UFRGS, 2003.

SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. *MPI: The Complete Reference*. The MIT Press. 1996.

SOUZA, M. A. *Avaliação das Rotinas de Comunicação Ponto-a-Ponto do MPI.* 1996. 152f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 1996.

SOUZA, P. S. L. *MPI: Um Padrão para Ambientes de Passagem de Mensagens.* 1997. 12f. Monografia de qualificação (Doutorado em Física) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 1997.

STALLINGS, W. *Arquitetura e Organização de Computadores: Projeto para o Desempenho.* 5. ed. São Paulo: Prentice Hall, 786 p. Tradução: Carlos Camarão de Figueiredo e Lucília Camarão de Figueiredo, 2003.

SUNDERAM, V. S.; GEIST, G. A. *Heterogeneous Parallel and Distributed Computing.* Parallel Computing, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, n. 25, p. 1699–1721, 1999.

SUNDERAM, V. S., GEIST, A., DONGARRA, J., MANCHEK, R. *The PVM Concurrent Computing System: Evolution, Experiences and Trends*. Parallel Computing, v. 20, pp. 531-545, 1994.

SUN Microsystems, *JAVA document home page*. Disponível em: http://java.sun.com/j2se/1.5.0/docs/api/. Acesso em novembro, 2006b.

TABOADA, G. L.; TOURINO, J.; DOALLO, R. (2003) *Performance Modeling and Evaluation of Java Message-Passing Primitives on a Cluster.* Lecture Notes in Computer Science, v.2840, p.29-36, 2003.

TANENBAUM, A. M. *Modern Operating Systems*. 2ª ed., Prentice Hall International Inc, 1992.

TANENBAUM, A. S. *Distributed Operating Systems*. New Jersey: Prentice Hall, 614p, 1995.

TANENBAUM, A. S. *Structured Computer Organization*. 4. ed. Upper Saddle River, NJ: Prentice Hall, 1999.

ZALUSKA, E. J. *Research Lines in Distributed Computing Systems and Concurrent Computation*. Anais dos Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, pp. 132-155, 1991.

ZOMAYA, A. Y. (Ed.). *Parallel and Distributed Computing Handbook*. New York: McGraw-Hill, 1198 p. (Computer Engineering Series), 1996.

WALKER, D. W. *The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers*. Parallel Computing, v. 20, pp 657-673, 1994.

WEIQIN, T.; HUA, Y.; WENSHENG, Y. *PJMPI: Pure Java Implementation of MPI.* Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region, 2000.

APÊNDICE A – CONFIGURAÇÃO DO SISTEMA

A.1. Configuração de Arquivos para a Comunicação entre as Máquinas (sem a Autenticação por Senha)

Para que a comunicação entre as máquinas ocorra e sem a utilização de senha, alguns arquivos devem ser alterados ou criados, caso eles não existam. As seções seguintes apresentam a configuração de cada um desses arquivos, para retirada de senha no acesso por meio do protocolo *rsh (remote shell)*. É de extrema importância que todos esses arquivos sejam replicados em todas as máquinas que participarão do processo.

A.1.1. Configuração do Arquivo *hosts*

O arquivo *hosts* encontra-se no diretório */etc*. Esse arquivo deve conter os *IPs (Internet Protocols)*, os nomes e os apelidos atribuídos às máquinas participantes do processo. Os *IPs* de cada máquina podem ser obtidos pelo comando *ifconfig*. Cada um dos nomes e apelidos são colocados conforme foram definidos no *DNS (Domain Name Service)*, eles podem ser obtidos pelo comando *hostname* como também serem alterados em *desktop*, *administração*, *rede*, *DNS*. Dessa forma, consegue-se prover uma maior transparência para o usuário, pois o reconhecimento entre as máquinas participantes ocorre pelos nomes atribuídos a elas, deixando o *IP* mascarado. Um exemplo de configuração do arquivo *hosts* é representado pela Figura A.1.

```
# <IP>            <hostname>       <hostname>
  192.168.17.20   lab07_01        lab07_01
  192.168.17.62   lab07_02        lab07_02
  192.168.17.61   lab07_03        lab07_03
  192.168.17.59   lab07_04        lab07_04
  192.168.17.26   lab07_05        lab07_05
  192.168.17.19   lab07_06        lab07_06
  192.168.17.64   lab07_07        lab07_07
  192.168.17.56   lab07_08        lab07_08
  192.168.17.58   lab07_09        lab07_09
  192.168.17.32   lab07_10        lab07_10
```

Figura A.1 - Configuração do arquivo *hosts*

A.1.2. Configuração do Arquivo *hosts.equiv*

O arquivo *hosts.equiv* encontra-se também no diretório */etc*. Os nomes *(hostname)* atribuídos às máquinas devem ser acrescentados conforme apresentado na Figura A.2. Estabelece-se dessa forma a relação de confiança entre as máquinas, para que haja relacionamento de equivalência entre elas sem a necessidade de autenticação por senha.

```
# <hostname>
lab07_01
lab07_02
lab07_03
lab07_04
lab07_05
lab07_06
lab07_07
lab07_08
lab07_09
lab07_10
```

Figura A.2 - Configuração do arquivo *hosts.equiv*

A.1.3. Configuração do Arquivo *.rhosts*

Esse arquivo deverá constar em cada diretório de trabalho do usuário como /home e /root. A existência do ponto "." na frente do arquivo, deixa-o escondido ao comando *ls*. Esse arquivo será utilizado pelo protocolo *rsh* para a execução de comandos remotos e por algumas aplicações de monitoramento. O formato do arquivo é apresentado na Figura A.3.

```
lab07_01
lab07_02
lab07_03
lab07_04
lab07_05
lab07_06
lab07_07
lab07_08
lab07_09
lab07_10
```

Figura A.3 - Configuração do arquivo .rhosts

A.1.4. Configuração do Arquivo *securetty*

É necessária também a edição do arquivo *securetty*, acrescentando *rsh* e *rlogin* ao final do arquivo, um em cada linha, conforme pode ser observado na Figura A.4. Incluem-se

dessa forma as chamadas aos protocolos *rsh* e *rlogin*, habilitando o acesso sem senhas entre as máquinas. Esse arquivo encontra-se no diretório */etc*.

```
tty1
tty2
tty3
tty4
tty5
tty6
...
rsh
rlogin
```

Figura A.4 - Configuração do arquivo *securetty*

Deve-se verificar se o acesso entre as máquinas, por meio do protocolo rsh, está funcionando sem a necessidade de autenticação por senha. Isso pode ser feito pelo seguinte comando:

rsh <hostname>

Em que:

- *hostname* é o *hostname* da máquina que se deseja acessar.

A.2. Configurações do Nível de Segurança

Dependendo da configuração em que se encontra o sistema, pode ser que ocorram problemas na execução das aplicações. Erros como o apresentado na Figura A.5, por exemplo.

```
[root@lab07_08 Hello]# mpirun -n 2 java Hello
lab07_08_32990 (mpd_sockpair 233): connect error with -3 Temporary failure in name
resolution
lab07_08_32990 (mpd_sockpair 233): connect error with -3 Temporary failure in name
resolution
lab07_08_mpdman_1 (mpd_sockpair 226): connect -3 Temporary failure in name
resolution
lab07_08_mpdman_0 (mpd_sockpair 226): connect -3 Temporary failure in name
resolution
Exception in thread "main" java.lang.UnsatisfiedLinkError:
/usr/local/mpiJava/lib/libmpijava.so: /usr/local/mpiJava/lib/libmpijava.so: cannot
restore segment prot after reloc: Permission denied
        at java.lang.ClassLoader$NativeLibrary.load(Native Method)
        at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:1751)
        at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1676)
        at java.lang.Runtime.loadLibrary0(Runtime.java:822)
        at java.lang.System.loadLibrary(System.java:993)
        at mpi.MPI.<clinit>(MPI.java:52)
        at Hello.main(Hello.java:15)
```

Figura A.5 - Erros na execução de aplicações paralelas

Nesse caso, as configurações do sistema em relação ao nível de segurança devem ser verificadas. Isso pode ser feito em *Desktop, Configurações do Sistema, Nível de Segurança*. Uma janela de configuração do nível de segurança é aberta. Na aba *Opções do Firewall*, em nível de segurança, deve-se desabilitar o *firewall* (caso esteja habilitado). Na aba *SELINUX*, há duas opções: *ativado (modificação requer reinicialização)* e a opção *enforcing corrente: disabled*. Ambas devem estar desabilitadas. Realizada as devidas alterações, é necessária a reinicialização do sistema. Pode-se verificar se as alterações foram bem sucedidas pelo comando *setenforce 0*, estando no diretório */etc/sysconfig*. Nesse mesmo diretório, o arquivo *SELINUX* deve conter a linha *SELINUX=disabled*.

APÊNDICE B - INSTALAÇÃO DO MPIJAVA (*JAVA INTERFACE TO MPI*)

A instalação do ambiente de troca de mensagens mpiJava requer a instalação de um ambiente de programação Java e uma versão qualquer do MPI. Tendo ambos instalados, configurados e funcionando de forma correta, pode-se dar início à instalação do mpiJava.

As instalações detalhadas do Java e do MPI estão descritas nos Apêndices C e D respectivamente. Nesse trabalho foi utilizada a versão *mpiJava-1.2.5* (MPIJAVA, 2007).

Feito o *download* do arquivo *mpiJava-1.2.5.tar.gz*, é necessária a sua descompactação. O seguinte comando pode ser utilizado:

tar –zxvf mpiJava-1.2.5.tar.gz

Um subdiretório *mpiJava* é criado. Nesse diretório deve-se configurar o *software* para a plataforma utilizada, de acordo com a versão do MPI instalada. O seguinte comando pode ser utilizado:

- Para a versão MPICH (MPICH é o default): *./configure*;
- Para a versão LAM: *./configure--with-mpi=lam*;
- Para a versão SunHPC: *./configure--with-mpi=sunhpc*;
- Para a versão AIX + POE: *./configure--with-mpi=sp2*.

Em seguida, deve-se compilar o *software* utilizando o comando *make*. Após a compilação, o *makefile* colocará as classes geradas no diretório *~/lib/classes/mpi* e uma biblioteca no diretório *~/lib*.

Dessa forma, é necessário editar as variáveis de ambiente do arquivo *.bash_profile*, acrescentando:

~/mpiJava/lib/classes à variável ambiente CLASSPATH.

~/mpiJava/src/scripts à variável ambiente PATH.

~/mpiJava/lib à variável ambiente LD_LIBRARY_PATH.

Um exemplo de edição das variáveis de ambiente, é definido na Figura B.1. Considerando-se que o *software* mpiJava foi instalado no diretório */usr/local*.

```
CLASSPATH=.:/usr/local/mpiJava/lib/classes
PATH=/usr/local/mpiJava/src/scripts:$PATH
LD_LIBRARY_PATH=/usr/local/mpiJava/lib
export PATH CLASSPATH LD_LIBRARY_PATH
```

Figura B.1 - Configuração das variáveis de ambiente CLASSPATH, PATH e LD_LIBRARY_PATH do mpiJava

Para que as modificações nas variáveis de ambiente sejam efetivamente realizadas, é necessária a reinicialização do Linux.

No diretório onde se encontra o programa a ser executado, deve-se existir o arquivo *machines* com os nomes *(hostname)* atribuídos às máquinas participantes do processo, conforme pode ser observado na figura B.2. Nesse caso, foram colocadas as máquinas definidas anteriormente no apêndice A.

```
lab07_01
lab07_02
lab07_03
lab07_04
lab07_05
lab07_06
lab07_07
lab07_08
lab07_09
lab07_10
```

Figura B.2 - Edição do arquivo *machines*

O programa apresentado na Figura B.3 pode ser executado para verificar se a instalação foi bem sucedida. Outros programas de teste, que acompanham o pacote e estão localizados em *~/mpiJava/examples*, podem ser executados.

A compilação e a execução do programa podem ser feitas com os seguintes comandos, respectivamente:

javac Hello.java

mpirun –np 4 java Hello

Em que:

Hello.java é o no me do programa a ser executado;

Hello é a classe que foi gerada com a compilação;

O valor “4” é o número de processos a ser utilizado.

```
import mpi.* ;

class Hello {
  static public void main(String[] args) throws MPIException {

  MPI.Init(args);

   int my_rank;         // Rank of process
   int source;          // Rank of sender
   int dest;            // Rank of receiver
   int tag=50;          // Tag for messages
   int myrank = MPI.COMM_WORLD.Rank();
   int      p = MPI.COMM_WORLD.Size();

  if(myrank != 0) {
          dest = 0;
          String myhost = MPI.Get_processor_name();
     char [] message =
             ("Greetings from process "+myrank+" on "+myhost).toCharArray();
     MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR,dest, tag) ;
  }

  else {  // my_rank == 0
          for (source =1;source < p;source++) {
     char [] message = new char [60] ;
     Status s =
        MPI.COMM_WORLD.Recv(message, 0, 60, MPI.CHAR, MPI.ANY_SOURCE, tag) ;
     int nrecv = s.Get_count(MPI.CHAR);
     String s1 = new String(message);
     System.out.println("received: " + s1.substring(0,nrecv) + " : ") ;
          }
  }
  MPI.Finalize();
 }
}
```

Figura B.3 - Programa de teste para a biblioteca mpiJava

O resultado para a execução do programa apresentado na Figura B.3 pode ser observado na Figura B.4.

```
received: Greetings from process 2 on lab07_01 :
received: Greetings from process 3 on lab07_02 :
received: Greetings from process 1 on lab07_03 :
```

Figura B.4 - Resultado da execução do programa de teste para a biblioteca mpiJava

APÊNDICE C - INSTALAÇÃO DO AMBIENTE DE PROGRAMAÇÃO JAVA

Recomenda-se a utilização de versões superiores à versão 1.4 do JDK. Nesse caso foi utilizada a versão *jdk-1.0.5_09* para Linux (SUN, 2006b).

É necessária a descompactação do arquivo *jdk-1.0.5_09-linux-i586.rpm.bin*. O seguinte comando pode ser utilizado:

./jdk-1.0.5_09-linux-i586.rpm.bin

O arquivo *jdk-1.0.5_09-linux-i586.rpm* é criado. Porém antes de descompactá-lo, é necessário transformá-lo em executável. Isso é possível com os seguintes comandos:

chmod +x jdk-1.0.5_09-linux-i586.rpm
rpm –ivh jdk-1.0.5_09-linux-i586.rpm

Após a instalação do JDK, deve-se adicionar *~/jdk-1.0.5_09/bin* à variável de ambiente PATH. Dessa forma, o *script mpiJava/configure* pode encontrar os comandos *java*, *javac* e *javah*.

Um exemplo de configuração da variável de ambiente PATH é demonstrado pela Figura C.1, considerando-se que *jdk-1.0.5*_09 é um subdiretório de */usr/local/java*.

```
JAVA_HOME=/usr/local/java/jdk1.0.5_09
PATH=$JAVA_HOME/bin:$PATH
export JAVA_HOME PATH
```

Figura C.1 - Configuração da variável ambiente PATH na plataforma Linux

Para que as modificações nas variáveis de ambiente sejam efetivamente realizadas, é necessária a reinicialização do *Linux*.

Realizada as modificações, é possível verificar se as variáveis de ambiente foram configuradas de forma correta pelo comando *java –version*. A versão exibida deve corresponder à versão instalada.

Na Figura C.2 é apresentada um simples programa *Welcome.java* para teste.

```
// Programa Teste Welcome.java
public class Welcome {
       public static void main( String args[] ) {
           System.out.println( "Welcome to Java Programming!" );
       }
}
```

Figura C.2 - Programa de teste ao ambiente de programação Java

O programa pode ser compilado e executado pelos comandos *javac Welcome.java* e *java Welcome*, respectivamente. O resultado deve ser *Welcome to Java Programming!*

APÊNDICE D - INSTALAÇÃO DO MPI (*MESSAGE PASSING INTERFACE*)

Para a instalação do mpiJava é necessária a instalação de qualquer versão do MPI. Nesse caso, foi utilizada a versão *MPICH2-1.0.5* (MPICH2, 2006).

Feito o *download* do arquivo *mpich2-1.0.5.tar.gz*, é necessária sua descompactação. O seguinte comando pode ser utilizado:

tar –zxvf mpich2-1.0.5.tar.gz

Um subdiretório *mpich2-1.0.5* é criado. Nesse diretório deve-se configurar a biblioteca de passagem de mensagens MPI. A configuração pode ser feita com o seguinte comando:

./configure—prefix=/usr/local

Em que:

- O *configure* analisa se todos os requisitos para a instalação estão disponíveis para a compilação e configura os parâmetros de compilação de acordo com o sistema;
- O *prefix* serve para especificar o diretório onde o pacote será instalado;
- */usr/local* é o caminho, nesse caso, do diretório onde deverá ser instalada a aplicação.

Em seguida, deve-se compilar o *software* utilizando o comando *make*. Para copiar os arquivos gerados na compilação no local especificado deve-se utilizar o comando *make install*.

Após a instalação, deve-se acrescentar *~/mpich2-1.0.5/bin* à variável de ambiente PATH, conforme pode ser observado na Figura D.1. Tal variável pode ser editada no arquivo *.bash_profile*.

```
MPIR_HOME=/usr/local/mpich2-1.0.5
PATH=$MPIR_HOME/bin:$PATH
export PATH
```

Figura D.1 - Configuração das variáveis de ambiente MPIR_HOME e PATH da instalação da biblioteca de passagem de mensagem MPI na plataforma Linux

Para que as modificações nas variáveis de ambiente sejam efetivamente realizadas deve-se reinicializar o *Linux*.

É necessária a criação de um arquivo contendo *MPD_SECRETWORD=<secretword>*. Como usário *root*, o nome do arquivo deve ser *mpd.conf* e estar localizado em */root*. Como usuário qualquer, o nome do arquivo deve ser *.mpd.conf* (com ponto antes do nome) e estar localizado no diretório *home*. Na Figura D.2 pode se observar um exemplo da criação do arquivo *mpd.conf*, em que *mpipass* é a senha escolhida.

```
MPD_SECRETWORD=mpipass
```

Figura D.2 - Exemplo de criação do arquivo mpd.conf

Escolhida a senha, deve-se configurar as permissões do arquivo *mpd.conf* para que somente o proprietário possa visualizá-lo ou modificá-lo. Para isso pode ser utilizado o seguinte comando:

chmod 600 mpd.conf

Para testar se o sistema está configurado corretamente até o momento é aconselhável verificar se o servidor foi inicializado corretamente por meio dos comandos:

mpd &

mpdtrace

mpdallexit

Em que:

- *mpd &* inicia a execução do servidor MPI;
- *mpdtrace* mostra as conexões ativas que no caso é apenas um servidor;
- *mpdallexit* fecha o servidor MPI.

Antes de executar qualquer aplicação é necessário iniciar a execução dos servidores em todas as máquinas que participarão do processo.

No mestre: *mpd &*;

No escravo: *mpd –h hostname –p port &*.

Em que:

- *hostname* é o *hostname* do servidor mestre;
- port é a porta em que ele está sendo executado.

Estas informações podem ser encontradas por meio dos comandos *hostname* e *mpdtrace –l*, respectivamente.

Verificado o funcionamento do *rsh*, sem senha, é recomendável que se teste a instalação do MPI antes de se instalar o mpiJava. Alguns aplicativos de testes acompanham o pacote MPICH, podendo ser executados para verificar se tudo está funcionando de forma correta.

O primeiro teste básico de passagem de mensagens pode ser realizado utilizando-se o programa para cálculo de PI paralelizado, denominado CPI. Em que cada máquina calcula o valor de PI e compara com o valor 3.14159265358979. O programa CPI é um dos aplicativos de testes que acompanham o pacote MPICH e pode ser encontrado em *~/mpich2-1.0.5/examples*.

A compilação e a execução podem ser realizadas com os seguintes comandos, respectivamente:

mpicc cpi.c –o cpi

mpirun –np 3 cpi

Pode-se especificar também o caminho em que se encontra o programa a ser executado. Dessa forma, o comando para execução do programa CPI seria *mpirun –np 3 ~/mpich2-1.0.5/examples/cpi*.

Na execução dos programas, erros podem vir a ocorrer, caso isso aconteça, as configurações do sistema relacionadas às regras de segurança devem ser verificadas, conforme apresentado no apêndice A.2

O resultado da execução do programa é apresentado na Figura D.3:

```
Process 0 of 3 on lab06_12
pi is approximately 3.1415926544231323, Error is 0.0000000008333392
wall clock time = 0.000768
Process 1 of 3 on lab06_13
Process 2 of 3 on lab06_14
```

Figura D.3 - Resultado da execução do programa cpi (MPICH) na plataforma Linux

Nesse momento, tem-se a certeza de que tudo está funcionando perfeitamente. Podendo-se assim, dar prosseguimento à instalação do mpiJava descrita no apêndice B.

APÊNDICE E - INSTALAÇÃO DA API JAI (*JAVA ADVANCED IMAGING*)

Faz-se necessária a instalação da API JAI, devido a possibilidade de representação, processamento e visualização de imagens que esta API permite. Nesse caso, foi utilizada a versão *jai-1_1_3-beta* (SUN, 2006a).

Feito o *download* do arquivo *jai-1_1_3-beta_lib-linux-i586-jdk.bin*, é necessária sua descompactação, que deve ser feita no mesmo diretório em que se encontra o compilador Java. O seguinte comando pode ser utilizado:

./jai-1_1_3-beta_lib-linux-i586-jdk.bin

Em seguida, deve-se acrescentar ao arquivo *.bash_profile*: *~/JAVA_HOME/lib/tools.jar* à variável de ambiente CLASSPATH e *~/JAVA_HOME/man* à variável de ambiente MANPATH, conforme pode ser observado na Figura E.1.

```
JAVA_HOME=/usr/local/java/jdk1.0.5_09
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=.:$JAVA_HOME/lib/tools.jar
MANPATH=$MANPATH:$JAVA_HOME/man
export JAVA_HOME PATH CLASSPATH MANPATH
```

Figura E.1 - Configuração das variáveis de ambiente CLASSPATH e MANPATH da API JAI na plataforma Linux

APÊNDICE F – CÓDIGOS FONTE

As seções seguintes apresentam os códigos-fonte das implementações seqüenciais seguidas das paralelas dos filtros de mediana e de detecção de bordas, respectivamente.

F.1. Implementação Seqüencial do Filtro de Mediana

```
import mpi.* ;
import java.io.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.media.jai.*;
import java.awt.image.*;
import com.sun.media.jai.widget.DisplayJAI;

/*
* Esta é uma classe que implementa o algoritmo sequencial de suavização (filtro de mediana).
*/
class MedianaSequencial {

    static public void main(String[] args) throws MPIException {

        File arquivo = new File ("resultMedSeq.txt");
        int n = 0;
        int size_vetor = 0;
        long start;

        // Obtenção da imagem a ser processada e do tamanho da máscara a ser utilizada
        String nome = JOptionPane.showInputDialog("Nome da Figura Original: ") + ".tif";
        PlanarImage image = JAI.create("fileload", nome);
        String masc = JOptionPane.showInputDialog("Entre com o valor da mascara: \n"
                                                  //+ "1 - mascara 3X3 \n"
                                                  //+ "2 - mascara 5X5 \n"
                                                  //+ "3 - mascara 7X7 ");
        int mascara =  Integer.parseInt(masc);

        // Declaração e atribuição de valores necessários para se trabalhar com as imagens.
        int width = image.getWidth();
        int height = image.getHeight();
        SampleModel sm = image.getSampleModel();
        int nbands = sm.getNumBands();
        Raster inputRaster = image.getData();
        WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
        int[] pixels = new int[nbands * width * height];
        int[] pixels2 = new int[nbands * width * height];
        ColorModel cm = image.getColorModel();
        TiledImage tiledImage = new TiledImage(0, 0, width, height, 0, 0, sm, cm);
        inputRaster.getPixels(0, 0, width, height, pixels);
        inputRaster.getPixels(0, 0, width, height, pixels2);
```

```
n = mascara + 1;
size_vetor = ((mascara * 2) + 1) * ((mascara * 2) + 1);

// Variável start recebe o valor do tempo inicial de processamento.
start = 0;
start = System.currentTimeMillis();

/**
 * Controle dos índices dos vetores que devem indicar somente o pixel de interesse e/ou seus
 * vizinhos.
 */
int offset, offset2;
for(int h = mascara; h < height - mascara; h++)
    for (int w = mascara; w < width - mascara; w++) {
    /**
     * A variável offset é utilizada para o controle do índice de pixels2, fazendo com que somente
     * o pixel de interesse sofra alteração recebendo o valor mediano obtido com a ordenação do
     * vetor (vetor) composto pelo pixel de interesse e sua vizinhança.
     */
        offset = h * width * nbands + w * nbands;
        int [] vetor = new int [size_vetor];
        int ele = 0;
        for (int a = h - mascara; a < h + n; a++)
            for (int l = w - mascara; l < w + n; l++) {
                /**
                 * A variável offset2 é responsável pelo controle do índice de pixels2, de forma que
                 * o vetor de ordenação (vetor) contenha apenas os elementos referentes ao pixel
                 * de interesse e sua vizinhança.
                 */
                offset2 = a * width * nbands + l * nbands;
                vetor[ele] = pixels[offset2 + 0];
                ele++;
            }

        /**
         * Trecho que realiza a ordenação do pixel de interesse e sua vizinhança.
         * Ordenação do vetor 'vetor' utilizando o método de ordenação Shellsort.
         */
        int i , j , hi = 1, value ;
        do {
            hi = 3 * hi + 1;
        } while (hi < vetor.length);
        do {
            hi /= 3;
            for (i = hi; i < vetor.length; i++) {
                value = vetor[i];
                j = i - hi;
                while (j >= 0 && value < vetor[j]) {
                    vetor[j + hi] = vetor[j];
                    j -= hi;
                }
                vetor[j + hi] = value;
            }
        } while (hi > 1);
```

```
            /**
             * Trecho que realiza a ordenação do pixel de interesse e sua vizinhança.
             * Ordenação do vetor 'vetor' utilizando o método de ordenação Bubblesort.
             */

            /*
            for (int t = vetor.length - 1; t > vetor.length/2; t--) {
                int troca = 0;
                for (int r = 0; r < t; r++) {
                    if (vetor[r] > vetor[r + 1]) {
                        troca = 1;
                        int aux = vetor[r];
                        vetor[r] = vetor[r + 1];
                        vetor[r + 1] = aux;
                    }
                }
                if (troca == 0) {
                    break;
                }
            }
            */

            /**
             * Obtenção do valor mediano do vetor ordenado 'vetor' e a atribuição do valor obtido
             * ao pixel de interesse.
             */
            for (int band = 0; band < nbands; band++) {
                pixels2[offset + band] = vetor[size_vetor/2];
            }
        }

    // Variável end recebe o valor do tempo final de processamento.
    long end;
    end = System.currentTimeMillis();
    try {
        FileOutputStream writer = new FileOutputStream ("resultado.txt", true);
        PrintWriter saida = new PrintWriter(writer);
        // Cálculo do tempo total de aplicação do filtro.
        saida.println((end - start));
        saida.close();
        writer.close();
    } catch( IOException e) { }

    outputRaster.setPixels(0, 0, width, height, pixels2);
    tiledImage.setData(outputRaster);
    // Armazenamento da imagem resultante.
    nome = JOptionPane.showInputDialog("Nome da Figura Modificada: ") + ".TIF";
    JAI.create("filestore",tiledImage,nome,"TIFF");
    PlanarImage image2 = JAI.create("fileload", nome);
    DisplayJAI dj2 = new DisplayJAI(image2);
    // Exibição do tempo total de processamento.
    System.out.println("Total Time: " + (end - start) );
  }
}
```

F.2. Implementação Paralela do Filtro de Mediana

```
import mpi.* ;
import java.io.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.media.jai.*;
import java.awt.image.*;
//import com.sun.media.jai.widget.DisplayJAI;

/**
 * Esta é uma classe que implementa o algoritmo paralelo de suavização (filtro de mediana).
 */
class MedShell2 {

/**
 * AplicaFiltro - Método que aplica o filtro de suavização em cada parte da imagem.
 */
    public static int[] AplicaFiltro (int mascara, int distancia, int tam_vet_ord, int[] vetor, int alt_parte,
                                      int width, int nbands) {
        int ele;

        /**
         * Controle dos índices dos vetores que devem indicar somente o pixel de interesse e/ou seus
         * vizinhos.
         */
        int offset, offset2;
        for(int h = mascara; h < alt_parte - mascara; h++)
            for (int w = mascara; w < width - mascara; w++) {

                /**
                 * A variável offset é utilizada para o controle do índice de vetor, fazendo com que
                 * somente o pixel de interesse sofra alteração recebendo o valor mediano obtido com a
                 * ordenação do vetor (vet_ord) composto pelo pixel de interesse e sua vizinhança.
                 */
                offset = h * width * nbands + w * nbands;
                int [] vet_ord = new int [tam_vet_ord];
                ele = 0;
                for (int a = h - mascara; a < h + distancia; a++)
                    for (int l = w - mascara; l < w + distancia; l++) {

                        /**
                         * A variável offset2 é responsável pelo controle do índice de vetor, de forma que
                         * o vetor de ordenação (vet_ord) contenha apenas os elementos referentes ao pixel
                         * de interesse e sua vizinhança.
                         */
                        offset2 = a * width * nbands + l * nbands;
                        vet_ord[ele] = vetor[offset2 + 0];
                        ele++;
                    }
```

```
        /**
         * Trecho que realiza a ordenação do pixel de interesse e sua vizinhança.
         * Ordenação do vetor 'vet_ord' utilizando o método de ordenação Shellsort.
         */
        int i , j , hi = 1, value;
        do {
            hi = 3 * hi + 1;
        } while ( hi < vet_ord.length );
        do {
            hi /= 3;
            for ( i = hi; i < vet_ord.length; i++) {
                value = vet_ord[ i ];
                j = i - hi;
                while (j >= 0 && value < vet_ord[ j ]) {
                    vet_ord[ j + hi ] = vet_ord[ j ];
                    j -= hi;
                }
                vet_ord[ j + hi ] = value;
            }
        } while ( hi > 1 );


        /**
         * Trecho que realiza a ordenação do pixel de interesse e sua vizinhança.
         * Ordenação do vetor 'vet_ord' utilizando o método de ordenação Bubblesort.
         */

        /*
        for (int t = vet_ord.length-1; t>vet_ord.length/2; t--) {
            int troca = 0;
            for (int r=0; r<t; r++) {
                if (vet_ord[r]>vet_ord[r+1]) {
                    troca = 1;
                    int aux = vet_ord[r];
                    vet_ord[r] = vet_ord[r+1];
                    vet_ord[r+1] = aux;
                }
            }
            if (troca == 0) {
                break;
            }
        }
        */


        // Após a ordenação dos pixels, obtém-se o valor mediano do vetor ordenado 'vet_ord'.
        // Atribuindo tal valor ao pixel de interesse.
        for (int band = 0; band < nbands; band++) {
            vetor[offset + band] = vet_ord[tam_vet_ord/2];
        }
    }

    // Retorna o vetor de pixels 'vetor' com seus novos valores.
    return(vetor);
}
```

```
static public void main(String[] args) throws MPIException {

    // Inicializa o MPI.
    MPI.Init(args) ;

    File arquivo = new File ("resultado.txt");
    int rank;                              // identificador dos processos.
    int source;                            // rank do fonte.
    int p;                                 // numero de processos.
    int dest;                              // rank de destino.
    int tag;                               // identificador das mensagens.
    long start;                            // variavel para contagem do tempo inicial.
    int nlinhas;                           // número de linhas de pixels.

    /**
     * Chave p/ controle.
     * chave = 1 indica uma linha de pixels.
     * chave = 2 indica duas linhas de pixels.
     */
    int chave = 1;
    final int BUF_SIZE = 16000000;         // tamanho do buffer de saida.
    byte buffer[] = new byte[BUF_SIZE];    // buffer de saida.
    int size[] = new int[1];               // recebe o tamanho da mensagem a ser enviada.
    int pos;                               // posicao inicial no buffer de saida.
    int apos[] = new int[1];               // recebe a posicao final do buffer de saida.
    Status status;                         // status da mensagem.
    int nitens = 6;                        // numero de itens do vetor.
    int mascara;                           // tamanho da mascara utilizada.
    int height;                            // altura da imagem.
    int width;                             // largura da imagem.
    int nbands;                            // número de bandas de cor que a imagem possui.
    int alt_parte;                         // altura da parte da imagem
    int tam_vetor[] = new int [1];         // contem o tamanho de 'vetor'
    int vetor_aux[] = new int[nitens];     // contém nitens necessarios para o processamento.

    // Retorna o rank do processo chamador no grupo do comunicador.
    rank = MPI.COMM_WORLD.Rank() ;

    // Retorna o número de processos no grupo do comunicador.
    p = MPI.COMM_WORLD.Size() ;

    // Inicialização das variáveis ‘source = 0’ (mestre) e ‘dest = 1’ (escravos).
    source = 0;
    dest = 1;

    /**
     * Se rank for zero, parte de processamento do mestre.
     */
    if (rank == 0) {

        /**
         * Obtenção da imagem a ser processada e do tamanho da máscara a ser utilizada
         * para a aplicação do filtro.
         */
```

```
String nome = JOptionPane.showInputDialog("Nome da Figura Original")+".tif";
PlanarImage image = JAI.create("fileload",nome);
String mas = JOptionPane.showInputDialog("Entre com o valor da mascara: \n"
                                        + "1 - mascara 3X3 \n"
                                        + "2 - mascara 5X5 \n"
                                        + "3 - mascara 7X7 ");
mascara =  Integer.parseInt( mas );

// Declaração e atribuição de valores necessários para se trabalhar com as imagens.
width = image.getWidth();
height = image.getHeight();
SampleModel sm = image.getSampleModel();
nbands = sm.getNumBands();
Raster inputRaster = image.getData();
WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
int[] pixels = new int[nbands*width*height];
int[] pixels2 = new int[nbands*width*height];
ColorModel cm = image.getColorModel();
TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,sm,cm);
inputRaster.getPixels(0,0,width,height,pixels);
inputRaster.getPixels(0,0,width,height,pixels2);

pixels2 = pixels;
/**
 * 'nlinhas' é a altura da parte da imagem, ou seja, é a altura da imagem dividida pelo número
 * de processos menos 1, retira-se 1 do número de processos, pois o mestre não processa
 * nenhuma parte da imagem, apenas divide, distribui e une as partes da imagem.
 */
nlinhas = height/(p-1);
int resto;
/**
 * 'resto' recebe o valor do resto da divisão (altura da imagem pelo número de processos
 * menos 1). Indicando, dessa forma, o número de linhas de pixels que restaram para serem
 * processadas.
 */
resto = (height%(p-1));
int ele = 0;
int ch1 = 0;

// Variável start recebe o valor do tempo inicial de processamento.
start = 0;
start = System.currentTimeMillis();

for (int i = 1; i < p; i++) {
    /**
     * Caso o resto da divisão seja maior ou igual a 1, e o processo sendo executado seja o
     * penúltimo processo (p-1), deve-se incrementar ao número de linhas (nlinhas) o valor
     * da variável 'resto' que é o número de linhas de pixels que restaram da divisão.
     */
    if (resto >= 1) {
        if (i == (p - 1)) {
            nlinhas = nlinhas + (resto);
        }
    }
```

```
/**
 * Se o número de processos for igual a 2, considera-se que um processo é referente ao
 * mestre e o outro ao escravo portanto deve-se enviar a imagem inteira para ser
 * processada pelo escravo. Dessa forma, 'tam_vetor' recebe a multiplicação dos valores
 * da largura da imagem pelo número de linhas (altura da imagem) e pelo número de
 * bands. Caso contrário 'tam_vetor' recebe parte da imagem mais algumas linhas
 * de acordo com a máscara utilizada e o número do processo a ser executado.
 */
if (p == 2) {
    tam_vetor[0] = (width * nlinhas * nbands);
}
else
    tam_vetor[0] = ((width * nlinhas * nbands) + (chave * mascara * width * nbands));


// Declaração da variável 'vetor', contendo os pixels de parte da imagem.
int [] vetor = new int[tam_vetor[0]];

pos  = 0;

/**
 * 'vetor_aux' recebe valores necessários para o processamento tanto no mestre como nos
 * escravos, tais como: máscara, largura, altura, nbands, altura da parte da imagem e o
 * tamanho do vetor contendo parte dos pixels da imagem.
 */
vetor_aux[0] = mascara;
vetor_aux[1] = width;
vetor_aux[2] = height;
vetor_aux[3] = nbands;
vetor_aux[4] = ((vetor.length) / (width * nbands));
vetor_aux[5] = vetor.length;

/**
 * 'vetor' recebe parte dos valores dos pixels contidos em pixel2 de acordo com o tamanho
 * do vetor, começando do elemento zero no primeiro processo e percorrendo até o
 * tamanho do vetor no processo seguinte, começa do elemento em que parou o processo
 * anterior.
 */
System.arraycopy(pixels2, (ele - (ch1 * mascara * width * nbands)), vetor, 0,
                                    vetor.length);
tag = i;

/**
 * Empacotamento dos dados a serem enviados.
 */
pos = MPI.COMM_WORLD.Pack(vetor_aux, 0, vetor_aux.length, MPI.INT, buffer,
                                                pos);
pos = MPI.COMM_WORLD.Pack(vetor, 0, vetor.length, MPI.INT, buffer, pos);
apos[0] = pos;

/**
 * Primeira mensagem - Envio do tamanho dos dados empacotados a serem enviados.
 */
MPI.COMM_WORLD.Isend(apos, 0, 1, MPI.INT, i, tag);
```

```
    /**
     * O valor do 'tag' é incrementado em 900, apenas para que haja uma correspondência
     * entre as duas mensagens a serem enviadas (a primeira com o tamanho e em seguida a
     * mensagem com os dados empacotados), dessa forma, garante-se que o tamanho
     * enviado pela primeira mensagem seja realmente referente aos dados empacotados da
     * segunda mensagem (no mesmo processo).
     */
    tag += 900;

    /**
     * Segunda mensagem - Envio do buffer contendo todos os dados empacotados (
     * elementos necessarios ao processamento).
     */
    MPI.COMM_WORLD.Isend(buffer, 0, pos, MPI.PACKED, i, tag);

    ele += (nlinhas * width * nbands);
    ch1 = 1;

    /**
     * 'chave' recebe o valor 2, devido aos proximos processos necessitarem o acréscimo de
     * duas linhas de pixels para o processamento, uma pertencente à última linha do processo
     * anterior e a outra pertencente à primeira linha do processo posterior.
     */
    chave = 2;

    /**
     * Para o último processo, como no primeito processo, 'chave' recebe o valor 1, devido ao
     * primeiro processo necessitar apenas de mais uma linha de pixel para o processamento,
     * referente à última linha do penúltimo processo.
     */
    if ((i + 1) == (p - 1)) {
        chave = 1;
    }
}

nlinhas = nlinhas - (resto);


// Recebimento das partes da imagem processadas pelos escravos.
for (int i = 1; i<p; i++) {
    int src;
    int aux_tag;
    int ind;
    int comeco;
    int fim;

    /**
     * Recebimento da mensagem contendo o tamanho da próxima mensagem.
     */
    status = MPI.COMM_WORLD.Recv (size, 0, 1, MPI.INT, MPI.ANY_SOURCE,
                            MPI.ANY_TAG);

    aux_tag = status.tag;
    tag = status.tag;
```

```
/**
 * 'tag' é incrementado em 900 para correspondência entre as duas mensagens a serem
 * recebidas, uma com o tamanho dos dados empacotados e a outra com os dados
 * propriamente ditos.
 */
tag += 900;
src = status.source;

/**
 * Recebimento da mensagem contendo os dados empacotados.
 */
status = MPI.COMM_WORLD.Recv (buffer, 0, size[0], MPI.PACKED, src, tag);

pos = 0;

/**
 * Desempacotamento dos dados recebidos. Desempacota-se primeiramente o tamanho do
 * vetor, logo em seguida o vetor.
 */
pos = MPI.COMM_WORLD.Unpack (buffer, pos, tam_vetor, 0, 1, MPI.INT);
int [] vetor = new int[tam_vetor[0]];
pos = MPI.COMM_WORLD.Unpack (buffer, pos, vetor, 0, vetor.length, MPI.INT);

/**
 * União de cada parte da imagem processada.
 */
if (p == 2) {
/**
 * Se o número de processos for igual a 2, um processo é referente ao mestre e o outro ao
 * escravo. Envia-se e recebe-se a imagem inteira.
 */
    pixels2 = vetor;
}
else {
/**
 * Caso contrário, cada vetor recebido deve ser copiado em pixels2 na sua posição
 * correspondente. Se o 'aux_tag' for 1, trata-se da primeira parte da imagem. A
 * variável ind é responsável pelo controle das posições de vetor, as variáveis comeco e
 * fim são responsáveis pelo controle das posições de pixels2.
 */
    if (aux_tag == 1) {
        ind = 0;
        comeco = 0;
        fim = tam_vetor[0] - (mascara * width * nbands);
    }
    else {
        ind = mascara * width * nbands;
        comeco = (nlinhas * (aux_tag - 1) ) * width * nbands;
        if ( aux_tag == (p - 1) )
            fim = tam_vetor[0] - (mascara * width * nbands);
        else
            fim = tam_vetor[0] - (2 * mascara * width * nbands);
    }
    System.arraycopy(vetor, ind, pixels2, comeco, fim);
}
```

```
        }

        // Variável end recebe o valor do tempo final de processamento.
        long end;
        end = System.currentTimeMillis();
        try {
            FileOutputStream writer = new FileOutputStream ("resultado.txt", true);
            PrintWriter saida = new PrintWriter(writer);

            // Cálculo do tempo total de aplicação do filtro.
            saida.println((end-start));
            saida.close();
            writer.close();
        } catch( IOException e) { }

        outputRaster.setPixels(0, 0, width, height, pixels2);
        tiledImage.setData(outputRaster);

        // Armazenamento da imagem resultante com o nome suavizada.tiff
        JAI.create("filestore", tiledImage, "suavizada", "TIFF");

        // Exibição do tempo total de processamento.
        System.out.println("Tempo Total: " + (end - start) );
}

/**
* Se rank diferente de zero, parte de processamento dos escravos.
*/
else {
    int distancia;
    int tam_vet_ord;

    // Recebimento da mensagem contendo o tamanho da próxima mensagem enviada pelo mestre.
    status = MPI.COMM_WORLD.Recv(size, 0, 1, MPI.INT, source, MPI.ANY_TAG);

    int aux_tag;
    aux_tag = status.tag;
    tag = status.tag;
    tag += 900;

    //Recebimento da mensagem contendo os dados empacotados necessários para o processamento.
    status = MPI.COMM_WORLD.Recv (buffer, 0, size[0], MPI.PACKED, source, tag);
    pos = 0;

    // Desempacotamento da mensagem contendo os dados empacotados.
    pos = MPI.COMM_WORLD.Unpack (buffer, pos, vetor_aux, 0, vetor_aux.length, MPI.INT);
    mascara = vetor_aux[0];
    width = vetor_aux[1];
    height = vetor_aux[2];
    nbands = vetor_aux[3];
    alt_parte = vetor_aux[4];
    tam_vetor[0] = vetor_aux[5];
    int [] vetor = new int[tam_vetor[0]];
    pos = MPI.COMM_WORLD.Unpack(buffer, pos, vetor, 0, vetor.length, MPI.INT);
```

```
        distancia = mascara + 1;
        tam_vet_ord = ((mascara + distancia) * (mascara + distancia));

        // Chamada ao método que aplica o filtro mediana
        vetor = AplicaFiltro (mascara, distancia, tam_vet_ord, vetor, alt_parte, width, nbands);

        dest = 0;
        pos = 0;

        /**
         * Empacotamento do vetor de pixels após a aplicação do filtro mediana. Primeiramente o
         * tamanho do vetor de pixels, em seguida o vetor com os pixels processados.
         */
        pos = MPI.COMM_WORLD.Pack (tam_vetor, 0, 1, MPI.INT, buffer, pos);
        pos = MPI.COMM_WORLD.Pack (vetor, 0, vetor.length, MPI.INT, buffer, pos);
        apos[0] = pos;
        tag = aux_tag;

        // Envio do tamanho e sua respectiva mensagem.
        MPI.COMM_WORLD.Send (apos, 0, 1, MPI.INT, dest, tag);
        tag += 900;
        MPI.COMM_WORLD.Send (buffer, 0, pos, MPI.PACKED, dest, tag);
    }

    // Finalização do MPI.
    MPI.Finalize();
    }
}
```

F.3. Implementação Seqüencial do Filtro de Detecção de Bordas Sobel

```
import java.io.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.media.jai.*;
import java.awt.image.*;
import com.sun.media.jai.widget.DisplayJAI;

/*
* Esta é uma classe que implementa o algoritmo sequencial de segmentação (filtro de detecção de
bordas utilizando o operador Sobel modificado).
*/
class SobelSequencial {

    static public void main(String[] args)  {

        File arquivo = new File ("resultSobSeq.txt");
        int size_vetor = 0;
        long start;
        long end;

        String nome = JOptionPane.showInputDialog("Nome da Figura Original: ")+".TIF";
        PlanarImage image = JAI.create("fileload", nome);
        String masc = JOptionPane.showInputDialog("Entre com o valor da mascara: \n"
                                                   //+ "1 - mascara 3X3 \n"
                                                   //+ "2 - mascara 5X5 \n"
                                                   //+ "3 - mascara 7X7 ");
                                                   //+ "4 - mascara 9X9 \n");
                                                   //+ "5 - mascara 11X11 \n");
        int mascara =  Integer.parseInt(masc);

        // Declaração e atribuição de valores necessários para se trabalhar com as imagens.
        int width = image.getWidth();
        int height = image.getHeight();
        SampleModel sm = image.getSampleModel();
        int nbands = sm.getNumBands();
        Raster inputRaster = image.getData();
        WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
        int[] pixels = new int[nbands * width * height];
        int[] pixels2 = new int[nbands * width * height];
        ColorModel cm = image.getColorModel();
        TiledImage tiledImage = new TiledImage(0, 0, width, height, 0, 0, sm, cm);
        inputRaster.getPixels(0, 0, width, height, pixels);
        inputRaster.getPixels(0, 0, width, height, pixels2);

        size_vetor = ((mascara * 2) + 1) * ((mascara * 2) + 1);

        // Variável start recebe o valor do tempo inicial de processamento.
        start = 0;
        start = System.currentTimeMillis();
```

```
/**
 * Controle dos índices dos vetores que devem indicar somente o pixel de interesse e/ou seus
 * vizinhos.
 */
int offset, offset2;

/**
 * Declaração das máscaras (operadores de Sobel) aplicadas às regiões de pixels da
 * imagem.
 */
int veth[] =
          {
            5, 6, 7, 8, 9,10, 9, 8, 7, 6, 5,
            4, 5, 6, 7, 8, 9, 8, 7, 5, 6, 4,
            3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3,
            2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2,
            1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           -1,-2,-3,-4,-5,-6,-5,-4,-3,-2,-1,
           -2,-3,-4,-5,-6,-7,-6,-5,-4,-3,-2,
           -3,-4,-5,-6,-7,-8,-7,-6,-5,-4,-3,
           -4,-5,-6,-7,-8,-8,-9,-7,-5,-6,-4,
           -5,-6,-7,-8,-9,-10,-9,-8,-7,-6,-5
          };

int vetv[] =
          {
           5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5,
           6, 5, 4, 3, 2, 0, -2, -3, -4, -5, -6,
           7, 6, 5, 4, 3, 0, -3, -4, -5, -6, -7,
           8, 7, 6, 5, 4, 0, -4, -5, -6, -7, -8,
           9, 8, 7, 6, 5, 0, -5, -6, -7, -8, -9,
          10, 9, 8, 7, 6, 0, -6, -7, -8, -9, -10,
           9, 8, 7, 6, 5, 0, -5, -6, -7, -8, -9,
           8, 7, 6, 5, 4, 0, -4, -5, -6, -7, -8,
           7, 6, 5, 4, 3, 0, -3, -4, -5, -6, -7,
           6, 5, 4, 3, 2, 0, -2, -3, -4, -5, -6,
           5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5
          };

int vetds[] =
          {
          10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
           9, 8, 7, 6, 5, 4, 3, 2, 1, 0,-1,
           8, 7, 6, 5, 4, 3, 2, 1, 0, 1,-2,
           7, 6, 5, 4, 3, 2, 1, 0,-1,-2,-3,
           6, 5, 4, 3, 2, 1, 0,-1,-2,-3,-4,
           5, 4, 3, 2, 1, 0,-1,-2,-3,-4,-5,
           4, 3, 2, 1, 0,-1,-2,-3,-4,-5,-6,
           3, 2, 1, 0,-1,-2,-3,-4,-5,-6,-7,
           2, 1, 0,-1,-2,-3,-4,-5,-6,-7,-8,
           1, 0,-1,-2,-3,-4,-5,-6,-7,-8,-9,
           0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10
          };
```

```
int vetdi[] =
          {
              0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
             -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
             -2,-1, 0, 1, 2, 3, 4, 5, 6, 7, 8,
             -3,-2,-1, 0, 1, 2, 3, 4, 5, 6, 7,
             -4,-3,-2,-1, 0, 1, 2, 3, 4, 5, 6,
             -5,-4,-3,-2,-1, 0, 1, 2, 3, 4, 5,
             -6,-5,-4,-3,-2,-1, 0, 1, 2, 3, 4,
             -7,-6,-5,-4,-3,-2,-1, 0, 1, 2, 3,
             -8,-7,-6,-5,-4,-3,-2,-1, 0, 1, 2,
             -9,-8,-7,-6,-5,-4,-3,-2,-1, 0, 1,
            -10,-9,-8,-7,-6,-5,-4,-3,-2,-1, 0
          };

for(int h = mascara; h < height - mascara; h++)
   for (int w = mascara; w < width - mascara; w++) {

      /**
       * A variável offset é utilizada para o controle do índice de pixels2, fazendo com que
       * somente o pixel de interesse sofra alteração.
       */
      offset = h * width * nbands + w * nbands;
      int [] vetor = new int [size_vetor];
      int ele = 0;
      for (int a = h - mascara; a <= h + mascara; a++)
         for (int l = w - mascara; l <= w + mascara; l++) {

            /**
             * A variável offset2 é responsável pelo controle do índice de pixels, de forma que
             * vetor contenha apenas os elementos referentes ao pixel de interesse e sua
             * vizinhança.
             */
            offset2 = a * width * nbands + l * nbands;
            vetor[ele] = pixels[offset2 + 0];
            ele++;
         }

         int somah = 0;
         int somav = 0;
         int somads = 0;
         int somadi = 0;

         /**
          * Para cada um dos operadores (máscaras) horizontal, vertical, de 45º e de 135º,
          * obtêm-se as somas das multiplicações de cada elemento de uma região de pixels
          * (contidos em vetor) pelos elementos correspondentes das máscaras (contidos em
          * vetv, veth, vetds, vetdi).
          */
         for (int i=0; i<vetor.length; i++) {
            somah += vetor[i] * veth[i];
            somav += vetor[i] * vetv[i];
            somads += vetor[i] * vetds[i];
            somadi += vetor[i] * vetdi[i];
         }
```

```
            /**
             * Verificação da soma que possui o maior valor, atribuindo-o, dessa forma, ao pixel
             * de interesse.
             */
            for (int band = 0; band < nbands; band++) {
                int rn = 0;
                int rd = 0;

                if (somah < somav) {
                  rn = somav;
                }
                else {
                  rn = somah;
                }

                if (somads < somadi) {
                  rd = somadi;
                }
                else {
                  rd = somads;
                }

                if (rn < rd) {

                /**
                 * Caso o valor obtido seja menor que zero, deve-se atribuir ao pixel de interesse o
                 * valor zero, pois não é permitido valor negativo ao pixel. Caso o valor ultrapasse
                 * o valor máximo (65536) permitido, o pixel de interesse deve receber tal valor.
                 */
                    if (rd < 0)
                        pixels2[offset + band] = 0;
                    else
                        if (rd > 65535)
                            pixels2[offset + band] = 65535;
                        else
                            pixels2[offset + band] = rd;
                }
                else {
                    if (rn < 0)
                        pixels2[offset + band] = 0;
                    else
                        if (rn > 65535)
                            pixels2[offset + band] = 65535;
                        else
                            pixels2[offset + band] = rn;
                }
            }
    }

// Variável end recebe o valor do tempo final de processamento.
end = System.currentTimeMillis();

try {
```

```
            FileOutputStream writer = new FileOutputStream ("resultado.txt", true);
            PrintWriter saida = new PrintWriter(writer);

            // Cálculo do tempo total de aplicação do filtro.
            saida.println((end - start));
            saida.close();
            writer.close();
        } catch( IOException e) { }

        outputRaster.setPixels(0, 0, width, height, pixels2);
        tiledImage.setData(outputRaster);

        // Armazenamento da imagem resultante.
        nome = JOptionPane.showInputDialog("Nome da Figura Modificada: ") + ".TIF";
        JAI.create("filestore",tiledImage,nome,"TIFF");
        PlanarImage image2 = JAI.create("fileload", nome);
        DisplayJAI dj2 = new DisplayJAI(image2);

        // Exibição do tempo total de processamento.
        System.out.println("Tempo Total: " + (end - start) );
    }
}
```

F.4. Implementação Paralela do Filtro de Detecção de Bordas Sobel

```
import mpi.* ;
import java.io.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.media.jai.*;
import java.awt.image.*;
import com.sun.media.jai.widget.DisplayJAI;

/**
 * Esta é uma classe que implementa o algoritmo paralelo de segmentação (filtro de detecção de
 * bordas utilizando o operador Sobel modificado).
 */
class Sobel {

    static public void main(String[] args) throws MPIException {
        // Rotina que inicializa o MPI.
        MPI.Init(args);
        File arquivo = new File ("resultado.txt");

        int rank;                              // identificador dos processos.
        int source;                            // rank do fonte.
        int p;                                 // numero de processos.
        int dest;                              // rank de destino.
        int tag;                               // identificador das mensagens
        long start;                            // variavel para contagem do tempo inicial.
        int nlinhas;                           // numero de linhas de pixels.

        /**
         * Chave p/ controle.
         * chave = 1 indica uma linha de pixels.
         * chave = 2 indica duas linhas de pixels.
         */
        int chave = 1;
        final int BUF_SIZE = 16000000;         // tamanho do buffer de saida.
        byte buffer[] = new byte[BUF_SIZE];    //buffer de saida.
        int size[] = new int[1];               // recebe o tamanho da mensagem a ser enviada.
        int pos;                               // posicao inicial no buffer de saida.
        int apos[] = new int[1];               // recebe a posicao final do buffer de saida.
        Status status;                         // status da mensagem.
        int nitens = 6;                        // numero de itens do vetor.
        int mascara;                           // tamanho da mascara utilizada.
        int height;                            // altura da imagem.
        int width;                             // largura da imagem.
        int nbands;                            // número de bandas de cor que a imagem possui.
        int alt_parte;                         // altura da parte da imagem.
        int tam_vetor[] = new int [1];         // contem o tamanho de 'vetor'.
        int vetor_aux[] = new int[nitens];     // contém nitens necessários para o processamento.

        // Retorna o rank do processo chamador no grupo do comunicador.
        rank = MPI.COMM_WORLD.Rank() ;
```

```
// Retorna o número de processos no grupo do comunicador.
p = MPI.COMM_WORLD.Size() ;

// Inicialização das variáveis 'source = 0' (mestre) e 'dest = 1' (escravos).
source = 0;
dest = 1;

/**
 * Se rank for igual a zero, parte de processamento do mestre.
 */
if (rank == 0) {

    /**
     * Obtenção da imagem a ser processada e do tamanho da máscara a ser utilizada
     * para a aplicação do filtro.
     */
    String nome = JOptionPane.showInputDialog("Nome da Figura Original")+".tif";
    PlanarImage image = JAI.create("fileload",nome);
    String mas = JOptionPane.showInputDialog("Entre com o valor da mascara: \n"
                                    + "1 - mascara 3X3 \n"
                                    + "2 - mascara 5X5 \n"
                                    + "3 - mascara 7X7 \n ");
                                    + "4 - mascara 9X9 \n");
                                    + "5 - mascara 11X11 \n");
    mascara =  Integer.parseInt( mas );

    // Declaração e atribuição de valores necessários para se trabalhar com as imagens.
    width = image.getWidth();
    height = image.getHeight();
    SampleModel sm = image.getSampleModel();
    nbands = sm.getNumBands();
    Raster inputRaster = image.getData();
    WritableRaster outputRaster = inputRaster.createCompatibleWritableRaster();
    int[] pixels = new int[nbands*width*height];
    int[] pixels2 = new int[nbands*width*height];
    ColorModel cm = image.getColorModel();
    TiledImage tiledImage = new TiledImage(0,0,width,height,0,0,sm,cm);
    inputRaster.getPixels(0,0,width,height,pixels);
    inputRaster.getPixels(0,0,width,height,pixels2);

    pixels2 = pixels;

    /**
     * 'nlinhas' é a altura da parte da imagem, ou seja, é a altura da imagem dividida pelo número
     * de processos menos 1, retira-se 1 do número de processos, pois o mestre não processa
     * nenhuma parte da imagem, apenas divide, distribui e une as partes da imagem.
     */
    nlinhas = height/(p-1);
    int resto;
    /**
     * 'resto' recebe o valor do resto da divisão (altura da imagem pelo numero de processos
     * menos 1). Indicando, dessa forma, o número de linhas de pixels que restaram para serem
     * processadas.
     */
```

```
resto = (height%(p-1));

int ele = 0;
int ch1 = 0;

// Variável start recebe o valor do tempo inicial de processamento.
start = 0;
start = System.currentTimeMillis();

for (int i=1; i<p; i++) {

    /**
     * Caso o resto da divisão seja maior ou igual a 1, e o processo sendo executado seja o
     * penúltimo processo (p-1), deve-se incrementar ao número de linhas (nlinhas) o valor
     * da variável 'resto' que é o número de linhas de pixels que restaram da divisão.
     */
    if (resto>=1) {
        if (i == (p-1)) {
            nlinhas = nlinhas + (resto);
        }
    }


    /**
     * Se o número de processos for igual a 2, considera-se que um processo é referente ao
     * mestre e o outro ao escravo portanto deve-se enviar a imagem inteira para ser
     * processada pelo escravo. Dessa forma, 'tam_vetor' recebe a multiplicação dos valores
     * da largura da imagem pelo número de linhas (altura da imagem) e pelo número de
     * bands. Caso contrário 'tam_vetor' recebe parte da imagem mais algumas linhas
     * de acordo com a máscara utilizada e o número do processo a ser executado.
     */
    if (p==2) {
        tam_vetor[0] = (width*nlinhas*nbands);
    }
    else
        tam_vetor[0] = ( (width*nlinhas*nbands) + (chave*mascara*width*nbands) );


    // Declaração da variável 'vetor', contendo os pixels de parte da imagem.
    int [] vetor = new int[tam_vetor[0]];

    pos  = 0;

    /**
     * 'vetor_aux' recebe valores necessários para o processamento tanto no mestre como nos
     * escravos, tais como: mascara, largura, altura, nbands, altura da parte da imagem e o
     * tamanho do vetor contendo parte dos pixels da imagem.
     */
    vetor_aux[0] = mascara;
    vetor_aux[1] = width;
    vetor_aux[2] = height;
    vetor_aux[3] = nbands;
    vetor_aux[4] = ((vetor.length)/(width*nbands));
    vetor_aux[5] = vetor.length;
```

```
/**
 * 'vetor' recebe parte dos valores dos pixels contidos em pixel2 de acordo com o tamanho
 * do vetor, começando do elemento zero no primeiro processo e percorrendo até o
 * tamanho do vetor no processo seguinte, comeca do elemento em que parou o processo
 * anterior.
 */
System.arraycopy(pixels2, (ele - (ch1*mascara*width*nbands)), vetor, 0, vetor.length);
tag = i;

/**
 * Empacotamento dos dados a serem enviados.
 */
pos = MPI.COMM_WORLD.Pack(vetor_aux,0,vetor_aux.length,MPI.INT,buffer,pos);
pos = MPI.COMM_WORLD.Pack(vetor,0,vetor.length,MPI.INT,buffer,pos);
apos[0] = pos;

/**
 * Primeira mensagem - Envio do tamanho dos dados empacotados a serem enviados.
 */
MPI.COMM_WORLD.Isend(apos, 0, 1, MPI.INT, i, tag);

/**
 * O valor do 'tag' é incrementado em 900, apenas para que haja uma correspondência
 * entre as duas mensagens a serem enviadas (a primeira com o tamanho e em seguida a
 * mensagem com os dados empacotados), dessa forma, garante-se que o tamanho
 * enviado pela primeira mensagem seja realmente referente aos dados empacotados da
 * segunda mensagem (no mesmo processo).
 */
tag += 900;

/**
 * Segunda mensagem - Envio do buffer contendo todos os dados empacotados (
 * elementos necessários ao processamento).
 */
MPI.COMM_WORLD.Isend(buffer, 0, pos, MPI.PACKED, i, tag);

/**
 * Variáveis ele e ch1 são utilizadas para o controle das posições dos pixels da imagem,
 * fazendo com que cada processo receba parte da imagem corretamente.
 */
ele += (nlinhas*width*nbands);
ch1 = 1;

/**
 * 'chave' recebe o valor 2, devido aos próximos processos necessitarem o acréscimo de
 * duas linhas de pixels para o processamento, uma pertencente à última linha do processo
 * anterior e a outra pertencente à primeira linha do processo posterior.
 */
chave = 2;

/**
 * Para o último processo, como no primeiro processo, 'chave' recebe o valor 1, devido ao
 * primeiro processo necessitar apenas de mais uma linha de pixel para o processamento,
 * referente à últma linha do penúltimo processo.
 */
```

```
        if ((i+1) == (p-1)) {
            chave = 1;
        }
    }

    nlinhas = nlinhas - (resto);

        // Recebimento das partes da imagem processadas pelos escravos.
        for (int i=1; i<p; i++) {
        int src;
        int aux_tag;
        int ind;
        int comeco;
        int fim;

        /**
         * Recebimento da mensagem contendo o tamanho da próxima mensagem.
         */
        status = MPI.COMM_WORLD.Recv(size, 0, 1, MPI.INT, MPI.ANY_SOURCE,
                            MPI.ANY_TAG);
        aux_tag = status.tag;
        tag = status.tag;

        /**
         * 'tag' é incrementado em 900 para correspondência entre as duas mensagens a serem
         * recebidas, uma com o tamanho dos dados empacotados e a outra com os dados
         * propriamente ditos.
         */
        tag += 900;
        src = status.source;

        /**
         * Recebimento da mensagem contendo os dados empacotados.
         */
        status = MPI.COMM_WORLD.Recv(buffer, 0, size[0], MPI.PACKED, src, tag);

        pos = 0;

        /**
         * Desempacotamento dos dados recebidos. Desempacota-se primeiramente o tamanho do
         * vetor, logo em seguida o vetor.
         */
        pos = MPI.COMM_WORLD.Unpack(buffer,pos,tam_vetor,0,1,MPI.INT);
        int [] vetor = new int[tam_vetor[0]];
        pos = MPI.COMM_WORLD.Unpack(buffer,pos,vetor,0,vetor.length,MPI.INT);

        /**
         * Inião de cada parte da imagem processada.
         */
        if (p==2) {

        /**
         * Se o número de processos for igual a 2, um processo é referente ao mestre e o outro ao
         * escravo. Envia-se e recebe-se a imagem inteira.
         */
```

```
            pixels2 = vetor;
        }
        else {

        /**
         * Caso contrário, cada vetor recebido deve ser copiado em pixels2 na sua posição
         * correspondente. Se o 'aux_tag' for 1, trata-se da primeira parte da imagem. A
         * variável ind é responsável pelo controle das posições de vetor, as variáveis comeco e
         * fim são responsáveis pelo controle das posições de pixels2.
         */
            if (aux_tag == 1) {
                ind = 0;
                comeco = 0;
                fim = tam_vetor[0] - (mascara*width*nbands);
            }
            else {
                ind = mascara * width * nbands;
                comeco = (nlinhas * (aux_tag-1)) * width * nbands;
                if (aux_tag==(p-1))
                    fim = tam_vetor[0] - (mascara * width * nbands);
                else
                    fim = tam_vetor[0] - (2 * mascara * width * nbands);
            }
            System.arraycopy(vetor, ind, pixels2, comeco, fim);
        }
    }

    // Variável end recebe o valor do tempo final de processamento.
    long end;
    end = System.currentTimeMillis();
    try {
        FileOutputStream writer = new FileOutputStream ("resultado.txt", true);
        PrintWriter saida = new PrintWriter(writer);

        // Cálculo do tempo total de aplicação do filtro.
        saida.println((end - start));
        saida.close();
        writer.close();
    } catch( IOException e) { }

    outputRaster.setPixels(0,0,width,height,pixels2);
    tiledImage.setData(outputRaster);

    // Armazenamento da imagem resultante com o nome realçada.tiff
    JAI.create("filestore", tiledImage, "realçada", "TIFF");
    PlanarImage image2 = JAI.create("fileload",nome);
    DisplayJAI dj2 = new DisplayJAI(image2);

    // Exibição do tempo total de processamento.
    System.out.println("Tempo Total: "+ (end-start) );
}

/**
* Se rank for diferente de zero, parte de processamento dos escravos.
*/
```

```
    else {
        int ele;
        int distancia;
        int tam_vet_ord;

        // Recebimento da mensagem contendo o tamanho da próxima mensagem enviada pelo mestre.
        status = MPI.COMM_WORLD.Recv(size, 0, 1, MPI.INT, source, MPI.ANY_TAG);
        int aux_tag;
        aux_tag = status.tag;
        tag = status.tag;
        tag += 900;

       // Recebimento da mensagem contendo os dados empacotados necessários para o processamento.
        status = MPI.COMM_WORLD.Recv(buffer,0,size[0],MPI.PACKED,source,tag);
        pos = 0;

        // Desempacotamento da mensagem contendo os dados empacotados.
        pos = MPI.COMM_WORLD.Unpack(buffer,pos,vetor_aux,0,vetor_aux.length,MPI.INT);

        mascara = vetor_aux[0];
        width = vetor_aux[1];
        height = vetor_aux[2];
        nbands = vetor_aux[3];
        alt_parte = vetor_aux[4];
        tam_vetor[0] = vetor_aux[5];
        int [] vetor = new int[tam_vetor[0]];
        int [] vetor2 = new int[tam_vetor[0]];
        pos = MPI.COMM_WORLD.Unpack(buffer,pos,vetor,0,vetor.length,MPI.INT);
        FiltroSobel fsobel = new FiltroSobel();

        // Chamada à classe que aplica o filtro de detecção de bordas.
        vetor2 = fsobel.AplicaFiltro(mascara,alt_parte,width,nbands,vetor);
        dest = 0;
        pos = 0;

        /**
         * Empacotamento do vetor de pixels após a aplicação do filtro de detecção de bordas.
         * Primeiramente o tamanho do vetor de pixels, em seguida o vetor com os pixels processados.
         */
        pos = MPI.COMM_WORLD.Pack(tam_vetor,0,1,MPI.INT,buffer,pos);
        pos = MPI.COMM_WORLD.Pack(vetor2,0,vetor2.length,MPI.INT,buffer,pos);
        apos[0] = pos;
        tag = aux_tag;

        // Envio do tamanho e sua respectiva mensagem.
        MPI.COMM_WORLD.Send(apos, 0, 1, MPI.INT, dest, tag);
        tag += 900;
        MPI.COMM_WORLD.Send(buffer, 0, pos, MPI.PACKED, dest, tag);
    }

    // Finaliza o MPI.
    MPI.Finalize();
    }
}
```

```
/*
 * Esta é uma classe responsável pela aplicação do filtro de detecção de bordas
 */
class FiltroSobel {
  private   int tam_vet_ord = 0;
  private  int somah = 0;
  private   int somav = 0;
  private   int somads = 0;
  private   int somadi = 0;
  private   int[] vetor2;

public FiltroSobel(){}

/*
 * AplicaFiltro - Método que aplica o filtro de segmentação (filtro de detecção de bordas) em cada
 * parte da imagem.
 */
public int[]  AplicaFiltro(int mascara, int alt_parte, int width, int nbands, int vetor[] ){
     vetor2 = new int[vetor.length];
     tam_vet_ord = ((mascara * 2) + 1) * ((mascara * 2) + 1);

     /**
      * Controle dos índices dos vetores que devem indicar somente o pixel de interesse e/ou seus
      * vizinhos.
      */
     int offset, offset2;

     /**
      * Declaração das máscaras (operadores de Sobel) aplicadas às regiões de pixels da
      * imagem.
      */
     int veth[] =
                {
                   5, 6, 7, 8, 9,10, 9, 8, 7, 6, 5,
                   4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4,
                   3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3,
                   2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2,
                   1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  -1,-2,-3,-4,-5,-6,-5,-4,-3,-2,-1,
                  -2,-3,-4,-5,-6,-7,-6,-5,-4,-3,-2,
                  -3,-4,-5,-6,-7,-8,-7,-6,-5,-4,-3,
                  -4,-5,-6,-7,-8,-9,-8,-7,-6,-5,-4,
                  -5,-6,-7,-8,-9,-10,-9,-8,-7,-6,-5
                };
     int vetv[] =
                {
                   5,4,3,2,1,0,-1,-2,-3,-4,-5,
                   6,5,4,3,2,0,-2,-3,-4,-5,-6,
                   7,6,5,4,3,0,-3,-4,-5,-6,-7,
                   8,7,6,5,4,0,-4,-5,-6,-7,-8,
                   9,8,7,6,5,0,-5,-6,-7,-8,-9,
                  10,9,8,7,6,0,-6,-7,-8,-9,-10,
                   9,8,7,6,5,0,-5,-6,-7,-8,-9,
                   8,7,6,5,4,0,-4,-5,-6,-7,-8,
                   7,6,5,4,3,0,-3,-4,-5,-6,-7,
                   6,5,4,3,2,0,-2,-3,-4,-5,-6,
                   5,4,3,2,1,0,-1,-2,-3,-4,-5
                };
```

```
int vetds[] =
             {
               10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
               9, 8, 7, 6, 5, 4, 3, 2, 1, 0,-1,
               8, 7, 6, 5, 4, 3, 2, 1, 0, 1,-2,
               7, 6, 5, 4, 3, 2, 1, 0,-1,-2,-3,
               6, 5, 4, 3, 2, 1, 0,-1,-2,-3,-4,
               5, 4, 3, 2, 1, 0,-1,-2,-3,-4,-5,
               4, 3, 2, 1, 0,-1,-2,-3,-4,-5,-6,
               3, 2, 1, 0,-1,-2,-3,-4,-5,-6,-7,
               2, 1, 0,-1,-2,-3,-4,-5,-6,-7,-8,
               1, 0,-1,-2,-3,-4,-5,-6,-7,-8,-9,
               0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10
             };

int vetdi[] =
             {
               0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
              -2,-1, 0, 1, 2, 3, 4, 5, 6, 7, 8,
              -3,-2,-1, 0, 1, 2, 3, 4, 5, 6, 7,
              -4,-3,-2,-1, 0, 1, 2, 3, 4, 5, 6,
              -5,-4,-3,-2,-1, 0, 1, 2, 3, 4, 5,
              -6,-5,-4,-3,-2,-1, 0, 1, 2, 3, 4,
              -7,-6,-5,-4,-3,-2,-1, 0, 1, 2, 3,
              -8,-7,-6,-5,-4,-3,-2,-1, 0, 1, 2,
              -9,-8,-7,-6,-5,-4,-3,-2,-1, 0, 1,
              -10,-9,-8,-7,-6,-5,-4,-3,-2,-1, 0
             };

   for(int h=mascara; h<alt_parte-mascara; h++)
      for (int w=mascara; w<width-mascara; w++) {

         /**
          * A variável offset é utilizada para o controle do índice de vetor2, fazendo com que
          * somente o pixel de interesse sofra alteração.
          */
         offset = h * width * nbands + w * nbands;
         int [] vet_ord = new int [tam_vet_ord];
         int ele = 0;
         for (int a=h-mascara; a<=h+mascara; a++)
            for (int l=w-mascara; l<=w+mascara; l++) {

            /**
             * A variável offset2 é responsável pelo controle do índice de vetor, de forma que
             * vet_ord contenha apenas os elementos referentes ao pixel de interesse e sua
             * vizinhança.
             */
               offset2 = a * width * nbands + l * nbands;
               vet_ord[ele] = vetor[offset2+0];
               ele++;
            }

         int somah = 0;
         int somav = 0;
         int somads = 0;
         int somadi = 0;
```

```
/**
 * Para cada um dos operadores (máscaras) horizontal, vertical, de 45º e de 135º,
 * obtêm-se as somas das multiplicações de cada elemento de uma região de pixels
 * (contidos em vet_ord) pelos elementos correspondentes das máscaras (contidos em
 * vetv, veth, vetds, vetdi).
 */
for (int i=0; i<vet_ord.length; i++) {
    somav += vet_ord[i] * vetv[i];
    somah += vet_ord[i] * veth[i];
    somads += vet_ord[i] * vetds[i];
    somadi += vet_ord[i] * vetdi[i];
}

/**
 * Verificação da soma que possui o maior valor, atribuindo-o, dessa forma, ao pixel
 * de interesse.
 */
for (int band = 0; band < nbands; band++) {
    int rn = 0;
    int rd = 0;
    if (somah < somav) {
        rn = somav;
    }
    else {
        rn = somah;
    }
    if (somads < somadi) {
        rd = somadi;
    }
    else {
        rd = somads;
    }
    if (rn < rd) {

    /**
     * Caso o valor obtido seja menor que zero, deve-se atribuir ao pixel de interesse o
     * valor zero, pois não é permitido valor negativo ao pixel. Caso o valor ultrapasse
     * o valor máximo (65536) permitido, o pixel de interesse deve receber tal valor.
     */
        if (rd < 0)
            vetor2[offset + band] = 0;
        else
            if (rd > 65535)
            vetor2[offset + band] = 65535;
            else
                vetor2[offset + band] = rd;
    }
    else {
        if (rn < 0)
            vetor2[offset + band] = 0;
        else
            if (rn > 65535)
                vetor2[offset + band] = 65535;
            else
```

```
                    vetor2[offset + band] = rn;
                }
            }
        }

        // Retorna o vetor de pixels 'vetor' com seus novos valores.
        return(vetor2);
    }
}
```

ANEXO A – ANÁLISE ESTATÍSTICA (TESTES DE HIPÓTESES)

A seguir são compiladas algumas considerações sobre o desempenho obtido por meio de uma análise estatística, cujo objetivo é verificar se as diferenças de desempenho das aplicações são estatisticamente significativas. Dessa forma, para os experimentos apresentados no capítulo 5, utiliza-se a técnica estatística de Teste de Hipóteses.

Essa técnica é desenvolvida a partir de duas hipóteses, H_0 ou Hipótese de nulidade e H_1 ou hipótese alternativa, formuladas sobre valores que se deseja comparar.

O primeiro passo para a utilização do Teste de Hipóteses consiste na definição dessas duas hipóteses que, após realizar o cálculo dos testes, uma delas será aceita e a outra rejeitada. A hipótese H_0 ou hipótese de nulidade é geralmente formulada procurando-se "discordar" dos valores obtidos com o experimento. A hipótese H_1 ou hipótese alternativa é aquela que geralmente "concorda" com os valores obtidos no experimento quando comparados "in-loco". Assim, a hipótese H_0 é a negação da hipótese H_1.

A hipótese H_0 utilizada nas comparações pode ser exemplificada da seguinte maneira: "o desempenho da aplicação paralela é melhor que o desempenho da aplicação sequencial". Essa hipótese H_0 é utilizada quando os valores obtidos com a aplicação paralela são menores que os obtidos com a aplicação seqüencial.

Para se realizar a análise estatística dos tempos coletados fazem-se, portanto, os seguintes testes de hipóteses.

- para amostras onde o tempo da aplicação paralela < tempo da aplicação seqüencial:

  H_0: $\mu paralela \geq \mu sequencial$

  H_1: $\mu paralela < \mu sequencial$

- para amostras onde o tempo da aplicação paralela > tempo da aplicação seqüencial:

  H_0: $\mu paralela \leq \mu sequencial$

  H_1: $\mu paralela > \mu sequencial$

Em que: μparalela e μsequencial são as médias dos tempos de execução da aplicação paralela com o ambiente de troca de mensagem mpiJava e da aplicação seqüencial respectivamente.

Considerando-se que os 30 tempos obtidos para cada média comparada possuem uma distribuição normal, a estatística dos testes de hipóteses acima é dada por:

$$Z = \frac{\overline{X}_{mpiJava} - \overline{X}_{seq}}{\sqrt{\frac{S^2_{mpiJava}}{n_{mpiJava}} + \frac{S^2_{seq}}{n_{seq}}}}$$

Equação A.1

Em que: $\overline{X}_{mpiJava}$ e $\overline{X}_{seq}$ são as médias amostrais dos tempos obtidos; $S^2_{mpiJava}$ e S^2_{seq} representam o desvio padrão amostral; e $n_{mpiJava}$ e n_{seq} representam o tamanho das amostras (neste caso 30).

Para um nível de significância (α) igual a 0.01 (probabilidade de estar correto 99% das vezes que a análise estatística for feita), rejeita-se a hipótese nulidade quando Z ultrapassar o limite fornecido por $Z_{0,01}$, o qual é 2.57. O valor de $Z_{0,01}$ = 2.57 é fornecido pela Tabela de Distribuição Normalizada (Achcar & Rodrigues 1995). Rejeita-se a hipótese nulidade H0 caso Z ≤-2.57, ou então, Z ≥2.57.

As Tabelas A.1 a A.20 referem-se aos testes de hipóteses e aos valores de *speedup* e eficiência obtidos com os experimentos apresentados no capítulo 5.

Tabela A.1 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB.

500 KB	Máquinas								
3x3	1	3	4	5	6	7	8	9	10
Média (ms)	144,47	1853,60	1438,93	1658,33	1837,67	2139,13	2444,47	2673,33	3011,67
Desvio Padrão	5,30	203,59	97,85	90,60	68,38	102,31	138,47	111,30	152,82
Variância	28,12	41449,44	9574,06	8208,36	4675,82	10467,45	19172,65	12386,76	23353,69
Hipótese α = 0,01		457,99	493,63	598,71	763,96	744,70	742,92	907,04	883,10
Speedup		0,08	0,10	0,09	0,08	0,07	0,06	0,05	0,05
Eficiência		0,03	0,03	0,02	0,01	0,01	0,01	0,01	0,00

Tabela A.2 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB.

500 KB	Máquinas								
5x5	1	3	4	5	6	7	8	9	10
Média (ms)	411,60	1223,93	1511,33	1723,47	1938,93	2225,13	2414,87	2582,47	2947,60
Desvio Padrão	5,30	103,34	37,97	119,80	141,03	138,34	83,70	104,28	134,13
Variância	28,12	10678,46	1441,56	14352,92	19888,33	19136,65	7005,32	10874,92	17989,71
Hipótese α = 0,01		307,03	676,56	461,00	495,20	593,62	839,75	816,83	842,87
Speedup		0,34	0,27	0,24	0,21	0,18	0,17	0,16	0,14
Eficiência		0,11	0,07	0,05	0,04	0,03	0,02	0,02	0,01

Tabela A.3 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB.

500 KB	Máquinas								
7x7	1	3	4	5	6	7	8	9	10
Média (ms)	905,80	1377,20	1575,27	1887,20	2064,80	2277,33	2504,53	2638,07	3027,00
Desvio Padrão	5,30	25,82	118,93	177,84	186,18	153,87	134,67	112,88	147,88
Variância	28,12	666,69	14143,53	31626,69	34662,83	23677,16	18137,05	12741,53	21869,87
Hipótese α = 0,01		295,25	226,24	275,56	318,53	411,93	510,54	599,37	648,87
Speedup		0,66	0,58	0,48	0,44	0,40	0,36	0,34	0,30
Eficiência		0,22	0,14	0,10	0,07	0,06	0,05	0,04	0,03

Tabela A.4 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB.

500 KB	Máquinas								
9x9	1	3	4	5	6	7	8	9	10
Média (ms)	581,20	1324,20	1524,47	1777,00	1960,40	2140,33	2509,13	2607,33	2959,47
Desvio Padrão	6,41	125,20	104,33	114,80	24,07	95,02	154,07	83,77	122,78
Variância	41,09	15674,43	10885,32	13178,40	579,57	9029,69	23737,72	7017,16	15075,72
Hipótese α = 0,01		250,84	347,15	420,67	967,46	599,56	589,42	826,34	810,37
Speedup		0,44	0,38	0,33	0,30	0,27	0,23	0,22	0,20
Eficiência		0,15	0,10	0,07	0,05	0,04	0,03	0,02	0,02

Tabela A.5 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 500 KB.

500 KB	Máquinas								
11x11	1	3	4	5	6	7	8	9	10
Média (ms)	890,33	1450,33	1578,93	1842,47	2009,47	2182,93	2483,67	2751,33	3062,13
Desvio Padrão	110,43	186,30	125,32	141,40	147,56	67,15	124,32	138,24	155,72
Variância	12195,16	34709,29	15704,60	19994,65	21773,98	4509,66	15456,09	19110,36	24249,18
Hipótese α = 0,01		125,91	173,69	232,37	269,85	375,67	402,76	457,07	515,58
Speedup		0,61	0,56	0,48	0,44	0,41	0,36	0,32	0,29
Eficiência		0,20	0,14	0,10	0,07	0,06	0,04	0,04	0,03

Tabela A.6 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB.

1 MB	Máquinas								
3x3	1	3	4	5	6	7	8	9	10
Média (ms)	290,27	1414,00	1700,47	1905,67	2174,93	2400,73	2634,80	2810,20	3034,00
Desvio Padrão	12,14	111,18	176,94	107,31	98,18	87,78	105,37	129,64	110,99
Variância	147,40	12360,00	31308,65	11514,62	9638,60	7705,53	11102,83	16806,29	12319,20
Hipótese α = 0,01		391,92	397,19	572,45	694,96	817,70	837,65	819,65	957,64
Speedup		0,21	0,17	0,15	0,13	0,12	0,11	0,10	0,10
Eficiência		0,07	0,04	0,03	0,02	0,02	0,01	0,01	0,01

Tabela A.7 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB.

1 MB	Máquinas								
5x5	1	3	4	5	6	7	8	9	10
Média (ms)	852,13	1561,73	1800,00	2033,07	2188,00	2346,20	2681,73	2988,47	3130,87
Desvio Padrão	12,63	194,69	215,90	194,37	26,92	44,11	142,10	183,11	157,95
Variância	159,45	37906,06	46614,80	37779,66	724,53	1946,03	20192,46	33527,45	24949,72
Hipótese α = 0,01		190,87	242,84	317,90	822,75	768,19	569,66	591,40	675,73
Speedup		0,55	0,47	0,42	0,39	0,36	0,32	0,29	0,27
Eficiência		0,18	0,12	0,08	0,06	0,05	0,04	0,03	0,03

Tabela A.8 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB.

1 MB	Máquinas								
7x7	1	3	4	5	6	7	8	9	10
Média (ms)	1886,13	2015,73	2119,13	2275,53	2313,60	2554,93	2758,20	2944,87	3235,60
Desvio Padrão	72,25	393,12	310,95	268,87	210,28	125,47	197,53	183,74	193,10
Variância	5220,25	154543,26	96692,12	72290,78	44219,17	15743,66	39019,49	33760,12	37289,44
Hipótese α = 0,01		23,27	46,10	81,66	98,49	184,21	205,63	256,28	320,84
Speedup		0,94	0,89	0,83	0,82	0,74	0,68	0,64	0,58
Eficiência		0,31	0,22	0,17	0,14	0,11	0,09	0,07	0,06

Tabela A.9 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB.

1 MB	Máquinas								
9x9	1	3	4	5	6	7	8	9	10
Média (ms)	1210,20	1736,73	1794,07	2050,40	2275,93	2402,33	2722,33	3012,87	3279,53
Desvio Padrão	16,43	261,10	153,91	180,86	188,69	108,36	159,27	157,68	175,79
Variância	270,03	68172,60	23689,26	32709,97	35602,20	11742,22	25367,82	24861,98	30903,85
Hipótese α = 0,01		122,41	173,26	231,67	288,20	413,31	441,82	529,11	578,05
Speedup		0,70	0,67	0,59	0,53	0,50	0,44	0,40	0,37
Eficiência		0,23	0,17	0,12	0,09	0,07	0,06	0,04	0,04

Tabela A.10 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 1 MB.

1 MB	Máquinas								
11x11	1	3	4	5	6	7	8	9	10
Média (ms)	1782,53	2005,47	2256,40	2225,73	2370,00	2494,00	2790,47	3127,33	3265,47
Desvio Padrão	38,50	337,84	352,57	263,85	205,95	32,47	199,03	224,05	136,62
Variância	1482,38	114135,98	124302,24	69618,06	42416,00	1054,40	39611,18	50199,42	18664,38
Hipótese α = 0,01		44,51	92,81	98,72	145,52	327,08	253,29	321,44	434,01
Speedup		0,89	0,79	0,80	0,75	0,71	0,64	0,57	0,55
Eficiência		0,30	0,20	0,16	0,13	0,10	0,08	0,06	0,05

Tabela A.11 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.

11 MB	Máquinas								
3x3	1	3	4	5	6	7	8	9	10
Média (ms)	2650,67	5208,87	5390,00	5568,87	6051,07	6156,33	6386,53	6602,67	6932,40
Desvio Padrão	18,36	276,25	131,27	67,31	320,38	278,82	166,88	110,02	295,31
Variância	337,02	76315,58	17230,53	4531,05	102640,86	77741,42	27848,25	12105,29	87210,64
Hipótese α = 0,01		577,24	867,34	1221,08	715,56	787,60	1063,10	1350,86	936,33
Speedup		0,51	0,49	0,48	0,44	0,43	0,42	0,40	0,38
Eficiência		0,17	0,12	0,10	0,07	0,06	0,05	0,04	0,04

Tabela A.12 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.

11 MB	Máquinas								
5x5	1	3	4	5	6	7	8	9	10
Média (ms)	7871,20	6798,40	6563,93	6364,93	6727,67	7046,53	7150,60	7202,00	7419,53
Desvio Padrão	14,64	143,53	156,81	114,93	471,48	465,18	379,12	405,72	488,64
Variância	214,29	20600,91	24590,33	13209,53	222298,09	216395,98	143733,84	164606,93	238773,58
Hipótese α = 0,01		-330,37	-386,67	-512,50	-200,87	-145,81	-140,64	-126,41	-77,98
Speedup		1,16	1,20	1,24	1,17	1,12	1,10	1,09	1,06
Eficiência		0,39	0,30	0,25	0,19	0,16	0,14	0,12	0,11

Tabela A.13 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.

11 MB	Máquinas								
7x7	1	3	4	5	6	7	8	9	10
Média (ms)	17357,93	9918,53	8861,87	8160,80	8070,87	8030,60	8051,13	8321,07	8059,73
Desvio Padrão	19,92	165,66	104,18	87,14	388,56	556,23	437,33	506,74	377,71
Variância	396,73	27443,18	10853,72	7592,96	150981,72	309392,51	191258,78	256783,53	142666,73
Hipótese α = 0,01		-2115,05	-2953,79	-3442,65	-1779,66	-1505,00	-1685,66	-1525,11	-1805,95
Speedup		1,75	1,96	2,13	2,15	2,16	2,16	2,09	2,15
Eficiência		0,58	0,49	0,43	0,36	0,31	0,27	0,23	0,22

Tabela A.14 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.

11 MB	Máquinas								
9x9	1	3	4	5	6	7	8	9	10
Média (ms)	11540,67	8583,60	7843,67	7436,53	7928,60	7691,47	7873,07	8158,67	8179,33
Desvio Padrão	86,93	582,93	497,15	187,85	408,84	468,08	380,07	341,86	535,52
Variância	7557,02	339804,24	247159,16	35287,45	167146,51	219099,05	144450,06	116865,16	286778,09
Hipótese α = 0,01		-442,50	-592,46	-958,90	-628,29	-632,80	-657,31	-632,56	-521,80
Speedup		1,34	1,47	1,55	1,46	1,50	1,47	1,41	1,41
Eficiência		0,45	0,37	0,31	0,24	0,21	0,18	0,16	0,14

Tabela A.15 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 11 MB.

11 MB	Máquinas								
11x11	1	3	4	5	6	7	8	9	10
Média (ms)	17399,73	10019,47	9135,13	5553,20	8692,27	8538,73	8521,33	8882,20	8673,67
Desvio Padrão	27,01	335,58	301,23	140,07	456,58	533,61	454,66	570,49	514,06
Variância	729,66	112613,58	90738,38	19620,29	208461,26	284736,06	206711,56	325453,49	264253,82
Hipótese α = 0,01		-1501,10	-1766,74	-3549,51	-1533,56	-1449,42	-1566,77	-1349,56	-1452,91
Speedup		1,74	1,90	3,13	2,00	2,04	2,04	1,96	2,01
Eficiência		0,58	0,48	0,63	0,33	0,29	0,26	0,22	0,20

Tabela A.16 - Valores calculados para o desempenho obtido com a execução de máscaras 3x3 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.

21 MB	Máquinas								
3x3	1	3	4	5	6	7	8	9	10
Média (ms)	5130,30	9594,40	10121,30	10589,00	11250,10	11386,80	11679,00	11833,60	11977,10
Desvio Padrão	26,71	619,07	688,97	607,64	457,61	407,19	279,10	216,26	180,11
Variância	713,61	383246,84	474677,41	369220,60	209409,89	165806,36	77898,40	46767,24	32440,09
Hipótese α = 0,01		555,51	589,97	685,37	879,36	949,80	1184,20	1359,91	1505,52
Speedup		0,53	0,51	0,48	0,46	0,45	0,44	0,43	0,43
Eficiência		0,18	0,13	0,10	0,08	0,06	0,05	0,05	0,04

Tabela A.17 - Valores calculados para o desempenho obtido com a execução de máscaras 5x5 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.

21 MB	Máquinas								
5x5	1	3	4	5	6	7	8	9	10
Média (ms)	15943,90	12971,60	12270,00	11847,90	12704,30	12127,80	13355,00	13812,00	13572,90
Desvio Padrão	64,79	406,55	375,33	409,50	860,14	1108,66	552,17	467,67	595,19
Variância	4197,89	165281,24	140872,00	167686,29	739833,41	1229125,56	304894,00	218718,80	354250,89
Hipótese α = 0,01		-432,94	-553,79	-594,76	-336,85	-352,28	-329,60	-292,16	-291,85
Speedup		1,23	1,30	1,35	1,26	1,31	1,19	1,15	1,17
Eficiência		0,41	0,32	0,27	0,21	0,19	0,15	0,13	0,12

Tabela A.18 - Valores calculados para o desempenho obtido com a execução de máscaras 7x7 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.

21 MB	Máquinas								
7x7	1	3	4	5	6	7	8	9	10
Média (ms)	35529,80	19749,50	16819,70	16331,10	15391,00	15399,20	15203,30	15294,10	14931,60
Desvio Padrão	107,06	716,68	296,90	736,92	532,71	413,05	714,92	313,31	835,51
Variância	11462,16	513625,85	88147,81	543051,09	283782,80	170606,56	511105,81	98163,69	698070,64
Hipótese α = 0,01		-1738,68	-2943,80	-2089,80	-2517,79	-2791,32	-2241,98	-3121,05	-2121,65
Speedup		1,80	2,11	2,18	2,31	2,31	2,34	2,32	2,38
Eficiência		0,60	0,53	0,44	0,38	0,33	0,29	0,26	0,24

Tabela A.19 - Valores calculados para o desempenho obtido com a execução de máscaras 9x9 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.

21 MB	Máquinas								
9x9	1	3	4	5	6	7	8	9	10
Média (ms)	22426,10	15402,90	14557,40	14434,90	13644,20	13352,00	12760,20	14082,00	13938,70
Desvio Padrão	797,41	452,95	848,57	602,41	633,53	895,08	451,20	667,81	961,28
Variância	635869,89	205166,89	720071,64	362903,09	401355,76	801169,60	203581,96	445971,20	924066,01
Hipótese α = 0,01		-628,08	-613,32	-675,42	-734,14	-697,49	-865,02	-689,33	-640,00
Speedup		1,46	1,54	1,55	1,64	1,68	1,76	1,59	1,61
Eficiência		0,49	0,39	0,31	0,27	0,24	0,22	0,18	0,16

Tabela A.20 - Valores calculados para o desempenho obtido com a execução de máscaras 11x11 das aplicações seqüencial e paralela com a utilização de imagens de 21 MB.

21 MB	Máquinas								
11x11	1	3	4	5	6	7	8	9	10
Média (ms)	32821,60	18456,00	16224,70	15368,90	14936,20	15499,20	14474,90	14875,00	14218,30
Desvio Padrão	1899,79	333,78	536,83	369,86	487,08	759,77	720,02	771,79	575,67
Variância	3609185,44	111406,00	288191,41	136793,09	237242,96	577254,36	518433,89	595659,40	331390,81
Hipótese α = 0,01		-961,22	-1063,24	-1158,47	-1157,67	-1062,19	-1133,50	-1097,99	-1182,40
Speedup		1,78	2,02	2,14	2,20	2,12	2,27	2,21	2,31
Eficiência		0,59	0,51	0,43	0,37	0,30	0,28	0,25	0,23