Ministério da
Ciência e Tecnologia

UM PAÍS DE TODOS
GOVERNO FEDERAL

INPE-15670-TDI/1445

PARALELISMO E IMAGENS: UM EXPERIMENTO DE PARALELIZAÇÃO PARA A SEGMENTAÇÃO DE IMAGENS COM APLICAÇÕES PARA A CLASSIFICAÇÃO AUTOMÁTICA DE CENAS GERADAS POR PLATAFORMAS ORBITAIS

Silene de Freitas Fernandes

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Ds. Antônio Miguel Vieira Monteiro e Celso Luiz Mendes, aprovada em 23 de dezembro de 1999

Registro do documento original:
<http://urlib.net/sid.inpe.br/jeferson/2004/06.07.13.21>

INPE
São José dos Campos
2009

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6911/6923
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

CONSELHO DE EDITORAÇÃO:
Presidente:
Dr. Gerald Jean Francis Banon - Coordenação Observação da Terra (OBT)
Membros:
Drª Maria do Carmo de Andrade Nono - Conselho de Pós-Graduação
Dr. Haroldo Fraga de Campos Velho - Centro de Tecnologias Especiais (CTE)
Drª Inez Staciarini Batista - Coordenação Ciências Espaciais e Atmosféricas (CEA)
Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)
Dr. Ralf Gielow - Centro de Previsão de Tempo e Estudos Climáticos (CPT)
Dr. Wilson Yamaguti - Coordenação Engenharia e Tecnologia Espacial (ETE)
BIBLIOTECA DIGITAL:
Dr. Gerald Jean Francis Banon - Coordenação de Observação da Terra (OBT)
Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)
Jefferson Andrade Ancelmo - Serviço de Informação e Documentação (SID)
Simone A. Del-Ducca Barbedo - Serviço de Informação e Documentação (SID)
REVISÃO E NORMALIZAÇÃO DOCUMENTÁRIA:
Marciana Leite Ribeiro - Serviço de Informação e Documentação (SID)
Marilúcia Santos Melo Cid - Serviço de Informação e Documentação (SID)
Yolanda Ribeiro da Silva Souza - Serviço de Informação e Documentação (SID)
EDITORAÇÃO ELETRÔNICA:
Viveca Sant´Ana Lemos - Serviço de Informação e Documentação (SID)

Ministério da
Ciência e Tecnologia

UM PAÍS DE TODOS
GOVERNO FEDERAL

INPE-15670-TDI/1445

PARALELISMO E IMAGENS: UM EXPERIMENTO DE PARALELIZAÇÃO PARA A SEGMENTAÇÃO DE IMAGENS COM APLICAÇÕES PARA A CLASSIFICAÇÃO AUTOMÁTICA DE CENAS GERADAS POR PLATAFORMAS ORBITAIS

Silene de Freitas Fernandes

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelos Ds. Antônio Miguel Vieira Monteiro e Celso Luiz Mendes, aprovada em 23 de dezembro de 1999

Registro do documento original:
<http://urlib.net/sid.inpe.br/jeferson/2004/06.07.13.21>

INPE
São José dos Campos
2009

Dados Internacionais de Catalogação na Publicação (CIP)

F391p Fernandes, Silene de Freitas.
Paralelismo e imagens: um experimento de paralelização para a segmentação de imagens com aplicações para a classificação automática de cenas geradas por plataformas orbitais / Silene de Freitas Fernandes. – São José dos Campos: INPE, 2009.
199p. ; (INPE-15670-TDI/1445)

Dissertação (Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 1999.

1. Processamento paralelo. 2. Processamento de imagem. 3. Segmentação de imagem. 4. Crescimento de regiões. 5. Sensoriamento Remoto. I.Título.

CDU 004.932)

Copyright © 2009 do MCT/INPE. Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação, ou transmitida sob qualquer forma ou por qualquer meio, eletrônico, mecánico, fotográfico, microfílmico, reprográfico ou outros, sem a permissão escrita da Editora, com exceção de qualquer material fornecido especificamente no propósito de ser entrado e executado num sistema computacional, para o uso exclusivo do leitor da obra.

Copyright © 2009 by MCT/INPE. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, eletronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use of the reader of the work.

Aprovada pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre** em **Computação Aplicada.**

Dr. João Argemiro de Carvalho Paiva _____
Presidente

Dr. Antônio Miguel Vieira Monteiro _____ Orientador

Dr. Celso Luis Mendes _____
Orientador

Dr. Luiz Alberto Vieira Dias _____
Membro da Banca

Dr. Alderico Rodrigues de Paula Júnior _____
Membro da Banca
Convidado

Candidata: Silene de Freitas Fernandes

São José dos Campos, 23 de dezembro de 1999.

“Se não houve frutos,
valeu a beleza das flores;
Se não houve flores,
valeu a sombra das folhas;
Mas se não houve folhas,
valeu a intenção da semente...”

Henfil

A meus pais, Márcia e Valdemar

AGRADECIMENTOS

Aos Drs. Antônio Miguel Vieira Monteiro e Celso Mendes meu sincero agradecimento, pela orientação, pelo apoio ao desenvolvimento deste trabalho, que muitas vezes os levaram a deixar suas famílias e suas horas de descanso em segundo plano e principalmente pela eterna compreensão.

Ao amigo Leonardo S. Bins, pela inesgotável fonte de conhecimento, sempre se mostrando mestre e acima de tudo amigo.

Aos monitores da Faculdade de Ciência da Computação da Universidade do Vale do Paraíba, especialmente a Rosana Aparecida de Freitas, o meu muito obrigada, pois sem a ajuda de vocês não seria possível conseguir tempo para conclusão desta dissertação.

Aos membros da banca examinadora pela predisposição em analisar este trabalho, pelas sugestões recebidas e por terem aceito participar de minha defesa.

A todos que, direta ou indiretamente, contribuíram para o desenvolvimento deste trabalho, meu reconhecimento.

E enfim a DEUS, que é a causa, o efeito e a explicação absoluta de tudo....

RESUMO

A segmentação é um importante passo na análise de imagens de sensoriamento remoto, sendo responsável pela divisão das mesmas em suas partes constituintes. Os métodos de segmentação aplicado em imagens desta natureza normalmente se baseiam na análise dos valores digitais dos pixels nas diversas bandas espectrais. Tal análise pode se tornar extremamente custosa computacionalmente. Visando reduzir o tempo de processamento do processo de segmentação de imagens, este trabalho objetiva a paralelização do algoritmo de segmentação por crescimento de regiões proposto por Bins et al. (1992), que está implementado como parte do Sistema de Processamento de Informações Georeferenciadas (SPRING) desenvolvido pela Divisão de Processamento de Imagens (DPI) do Instituto Nacional de Pesquisas Espaciais (INPE). Esta paralelização é feita empregando-se o paradigma Single Process Multiple Data (SPMD) com o modelo de troca de mensagem suportado pela biblioteca Message Passing Interface (MPI). Os testes são executados num clusters de PC´s com sistema operacional Linux. Os resultados são comparados aos resultados obtidos na versão seqüencial, a qual é executada no mesmo ambiente. A paralelização do algoritmo oferece uma redução significativa do tempo total de processamento, aliada a um produto cartográfico de qualidade, bem como oferece um modelo de implementação portável a diversas plataformas e sistemas operacionais. A principal aplicação deste trabalho é no Projeto de Desflorestamento da Amazônia (PRODES), em sua fase digital, visando a redução de custos através da otimização dos processos de classificação automática de cenas de satélites.

PARALLELISM AND IMAGES: A PARALLELIZATION EXPERIMENT FOR IMAGE SEGMENTATION WITH A APPLICATION FOR AUTOMATIC CLASSIFICATION OF SCENES OBTAINED FROM ORBITAL PLATAFORMS

ABSTRACT

Segmentation is an important step in image analysis for remote sensing; it is responsible for spliting the image into two distinct part so. Current segmentation methods used in images of this type are usually based on analysis of the pixel values in all image bands. This type of analysis can be computationally expensive. In order to reduce the processing time in image segmentation process this work introduces a parallel approach to image segmentation. This method is based on region growing technique as shown in Bins et all (1992), and used in the Images Processing System (SPRING) developed by the Image Processing Division (DPI) at the Brazilian Institute for Space Research (INPE). The parallelism is done using the Single Process Multiple Data (SPMD) paradigm and the message passing model under the Message Passing Interface (MPI). All testes have been executed in a PC cluster under the Linux Operating System. The results have been compared with output of a sequential system executing the same environment. The parallelization offers a significant reduction in the processing time combined with a high quality segmentation. The parallelization model can be applied in other types of machine architectures under different operating systems.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

Pág.

CAPÍTULO 1 - INTRODUÇÃO 23
1.1 - Caracterização do Problema 23
1.1.1 - Imagens na Solução de Problemas 23
1.1.2 - Solução Automática: um desafio computacional 24
1.1.3 - Os Problemas da Classificação Automática 25
1.1.4 - O Módulo de Classificação do SPRING 27
1.2 - Proposta de Paralelização do Segmentador 30
1.3 - Organização da Dissertação 31

CAPÍTULO 2 - REVISÃO BIBLIOGRÁFICA 33
2.1 – Imagem Digital 33
2.2 - O Algoritmo de Segmentação por Crescimento de Regiões 34
2.3 - Paralelismo e Imagens 38
2.3.1 - Fundamentos de Computação de Alto Desempenho 38
2.3.1.1 - O que é Computação Paralela? 39
2.3.1.2 - Ambiente Paralelo 43
2.3.1.3 - Paradigmas da Computação Paralela 44
2.3.1.3.1 - O Modelo MIMD 45
2.3.1.3.2 - Os Modelos SIMD e SPMD 46
2.3.2 - Comunicação entre os Processadores 48
2.3.2.1- Memória Compartilhada 48
2.3.2.2 - Memória Distribuída 49
2.3.3 - Análise de Desempenho 50
2.3.3.1 - Potencial de Aumento de Velocidade (ganho e eficiência) 52
2.3.3.2 - Balanceamento de Carga 55
2.3.3.3- Granularidade 56
2.3.3.3.1 - Grânulos Finos 56
2.3.3.3.2 - Grânulos Grosseiros 57
2.3.3.4 - Dependência dos Dados 58
2.3.3.5 - Comunicação e Largura da Banda 59
2.3.3.6 - Entrada e Saída (*Input/Output*) 59
2.3.3.7 - Configuração do Equipamento 60
2.3.3.8 - *Dead Lock* 60
2.3.4 - Computação de Alto Desempenho para Processamento de Imagens 61
2.3.4.1 - Explorando o Paralelismo dos Laços 62
2.3.4.2 - Formas de Paralelismo em Processamento de Imagem 63
2.4 – Bibliotecas de Paralelização 65

2.4.1 - Os Modelos de Troca de Mensagem 65
2.4.1.1 - Message Passing Interface 66
2.4.1.2 - Conceitos Básicos de MPI 75

CAPÍTULO 3 - ESTRATÉGIA DA SOLUÇÃO DO PROBLEMA 79
3.1 – Alvo Principal: As Iterações 79
3.2 – Metodologia 79
3.2.1 - Estudo Detalhado de Algoritmos de Segmentação 79
3.2.2 - Compilação e Execução do Fonte Serial 80
3.2.3 - Tomada de Tempo para Área Piloto 82
3.2.4 - Estudo do Programa Fonte 83
3.2.4.1 - Análise dos Dados e Tarefas 84
3.2.4.2 - Análise dos Trechos Paralelizáveis 84
3.2.4.3 – Pseudo-Código do Fonte Serial 85
3.2.5 - Proposta de Implementação Usando MPI 86
3.2.5.1 – Pseudo_Código da Versão Paralela 89
3.2.6 - Tomadas de Tempo Variando o Número de Processadores MPI 91
3.2.7 - Comparação dos Resultados com Área Piloto 92
3.2.8 – Análise de Desempenho 92

CAPÍTULO 4- TESTES E RESULTADOS 95
4.1 – Plano Geral de Testes 95
4.2 – Descrição dos Testes 96
4.2.1 – Execução do Código Serial 96
4.2.1.1 – Resumo dos Testes da Versão Serial 102
4.2.2 – Execução do Código Paralelo 102
4.2.2.1 – Testes para Limiar de Área 15 e Limiar de Similaridade 30 104
4.2.2.2 – Testes para Limiar de Área 20 e Limiar de Similaridade 30 110
4.2.2.3 – Testes para Limiar de Área 25 e Limiar de Similaridade 30 115
4.2.2.4 – Resumo dos Testes da Versão Paralela 120
4.3 – Validação dos Dados de Saída 124
4.4 – Análise de Desempenho 129
4.4.1 – Desempenho das Etapas de Segmentação e Costura 129
4.4.2 – Desempenho da Etapa de Segmentação 132
4.4.3 – Ganho Operacional 134

CAPÍTULO 5- CONCLUSÕES E RECOMENDAÇÕES 137
5.1 – Considerações Gerais 137
5.2 – Perspectivas 139

REFERÊNCIAS BIBLIOGRÁFICAS 141

APÊNDICE A - LISTAGEM DO ARQUIVO PRINCIPAL 145

APÊNDICE B - LISTAGEM DOS ARQUIVOS SECUNDÁRIOS 151

LISTA DE FIGURAS

Pág.

1.1 – Esquema do Processo 28
2.1. – Imagem de uma Cruz (a) com sua Representação na Forma Digital (b).....34
2.2 – Exemplo de aplicação do método de crescimento de regiões usando dois pixels "sementes": (a) imagem original; (b) resultado da segmentação usando uma diferença absoluta menor que 3 entre os níveis digitais; (c) resultado usando uma diferença absoluta menor que 8 36
2.3 - Comunicação entre os Processadores.....40
2.4 - Paralelismo de Dados 41
2.5 – Paralelismo Funcional 42
2.6 – Paralelismo de Objetos 42
2.7 - Ambiente Paralelo 44
2.8 - Modelo MIMD 46
2.9 - Modelo Paralelo SIMD 47
2.10 - Ambiente de Memória Compartilhada 49
2.11 - Ambiente de Memória Distribuída.....50
2.12 - Lei de Amdahl 52
2.13 - Comunicação x Computação para Grânulos Finos 57
2.14 - Comunicação x Computação para Grânulos Grosseiros 58
3.1 - Estrutura do SP2.....81
3.2 – Recorte das Imagens Testes (Imagem Landsat-TM 5 Órbita/Ponto 231/68 Bandas 3,4 e 5) 83
3.3 – Esquema de Execução Paralela 86
3.4 – Ilustração do Particionamento da Imagem 86
3.5 – Troca de Mensagem Assíncrona 88
3.6 – Esquema da Versão Paralela.....91
4.1. – Imagem Rotulada C256serial 97
4.2. – Imagem Rotulada C512serial 98
4.3 – Imagem Rotulada C1024serial 99
4.4 - Imagem Rotulada C2048serial 100
4.5 - Imagem Rotulada C4096serial 101
4.6 – Esboço da Divisão de Tarefas para uma Imagem 512 X 512 com 4 Processadores Clientes 104
4.7 - Imagem Rotulada c256_a15_rot 105
4.8 - Imagem Rotulada c512_a15_rot 106
4.9 - Imagem Rotulada c1024_a15_rot 107
4.10 - Imagem Rotulada c2048_a15_rot 108
4.11 - Imagem Rotulada c4096_a15_rot 109
4.12 - Imagem Rotulada c256_a20_rot 110
4.13 - Imagem Rotulada c512_a20_rot 111
4.14 - Imagem Rotulada c1024_a20_rot 112
4.15 - Imagem Rotulada c2048_a20_rot 113

4.16 - Imagem Rotulada c4096_a20_rot 114
4.17 - Imagem Rotulada c4256_a25_rot 115
4.18 - Imagem Rotulada c512_a25_rot 116
4.19 - Imagem Rotulada c1024_a25_rot 117
4.20 - Imagem Rotulada c2048_a25_rot 118
4.21 - Imagem Rotulada c4096_a25_rot 119
4.22 – Número de Regiões – Versão Serial x Versão Paralela.....127
4.23 - Comparação Visual – Resultado Serial e Resultado Paralelo - Imagem 256x256.....128
4.24 - Comparação Visual – Resultado Serial e Resultado Paralelo - Imagem 512x512.....128
4.25 - Speedup e Eficiência Imagem C256 – Limiar de Área 25 - Tempo de Segmentação e Costura.....130
4.26 - Speedup e Eficiência Imagem C512 – Limiar de Área 25 - Tempo de Segmentação e Costura 130
4.27 - Speedup e Eficiência Imagem C1024 – Limiar de Área 25 - Tempo de Segmentação e Costura 131
4.28 - Speedup e Eficiência Imagem C2048 – Limiar de Área 25 - Tempo de Segmentação e Costura 131
4.29 - Speedup e Eficiência Imagem C4096 – Limiar de Área 25 - Tempo de Segmentação e Costura 131
4.30 - Speedup e Eficiência Imagem C256 – Limiar de Área 25 - Tempo de Segmentação.....132
4.31 - Speedup e Eficiência Imagem C512 – Limiar de Área 25 - Tempo de Segmentação.....132
4.32 - Speedup e Eficiência Imagem C1024 – Limiar de Área 25 - Tempo de Segmentação.....132
4.33 - Speedup e Eficiência Imagem C2048 – Limiar de Área 25 - Tempo de Segmentação.....132
4.34 - Speedup e Eficiência Imagem C4096 – Limiar de Área 25 - Tempo de Segmentação 133
4.35 – Possível Paralelização da Função de Costura Janelas 134
4.36 - Ganho Operacional Imagem C256 – Serial X Paralelo 135
4.37 - Ganho Operacional Imagem C512 – Serial X Paralelo 135
4.38 - Ganho Operacional Imagem C1024 – Serial X Paralelo 135
4.39 - Ganho Operacional Imagem C2048 – Serial X Paralelo 135
4.40 - Ganho Operacional Imagem C4096 – Serial X Paralelo 136

LISTA DE TABELAS

Pág.

1.1 - Comparação entre Tempos de Processamento para a Fase de Segmentação.....29
2.1 - Limites para Escalabilidade 53
2.2 - Exemplo de *Deadlock* 61
4.1 – Configuração do Hardware Utilizado nos Testes da Versão Serial.....96
4.2 – Dados de Entrada do Experimento Número 1.....97
4.3 – Dados de Entrada do Experimento Número 2.....98
4.4 – Dados de Entrada do Experimento Número 3.....99
4.5 – Dados de Entrada do Experimento Número 4.....100
4.6 – Dados de Entrada do Experimento Número 5.....101
4.7 – Resultados do Experimento Serial.....102
4.7 – Configuração do Hardware das Dez Máquinas Utilizadas nos Testes da Versão Paralela.....103
4.9 – Resultados do Experimento 6.....105
4.10 – Resultados do Experimento 7.....106
4.11 - Resultados do Experimento 8.....107
4.12 – Resultados do Experimento 9.....108
4.13 – Resultados do Experimento 10 119
4.14 – Resultados do Experimento 11.....110
4.15 – Resultados do Experimento 12.....111
4.16 – Resultados do Experimento 13.....112
4.17 – Resultados do Experimento 14.....113
4.18 – Resultados do Experimento 15.....114
4.19 – Resultados do Experimento 16.....115
4.20 – Resultados do Experimento 17.....116
4.21 – Resultados do Experimento 18.....117
4.22 – Resultados do Experimento 19.....118
4.23 – Resultados do Experimento 20.....119
4.24 – Resultados do Experimento Paralelo - Limiar de Área = 15.....121
4.25 – Resultados do Experimento Paralelo - Limiar de Área = 20.....122
4.26 – Resultados do Experimento Paralelo - Limiar de Área = 25.....123
4.27 – Versão Serial (Limiar Área 5/25) X Versão Paralela (Limiar de Área 25).....126
4.28 – Comparação dos Resultados Serial X Paralelo – Limiar de Área = 5/25.....129

LISTA DE SIGLAS E ABREVIATURAS

DPI		Divisão de Processamento de Imagens
HPF		High Performance FORTRAN
INPE		Instituto Nacional de Pesquisas Espaciais
MIMD		Multiple Instruction, Multiple Data
MPI		Message Passing Interface
MPP		Massive Parallel Processing
PC		Personal Computer
PRODES		Projeto de Desflorestamento da Amazônia
PVM		Parallel Virtual Machine
SIMD		Single Instruction, Multiple Data
SISD		Single Instruction, Single Data
SMP		Symmetric Multi Processing
SPMD		Single Process Multiple Data
SPRING	-	Sistema de Processamento de Informações Georeferenciadas

CAPÍTULO 1

INTRODUÇÃO

1.1 - Caracterização do Problema

Esta dissertação tem por objetivo a caracterização das modificações empregadas no módulo de classificação de imagens, parte integrante do Sistema de Processamento de Informações Georeferenciadas (SPRING), visando a viabilização de seu uso através da paralelização de seu código. O SPRING foi proposto e desenvolvido pela Divisão de Processamento de Imagens (DPI) do Instituto Nacional de Pesquisas Espaciais (INPE) e utiliza metodologias contextuais de classificação operando em sistemas computacionais do tipo seqüencial.

1.1.1 - Imagens na Solução de Problemas

A tecnologia aeroespacial tem sido cada dia mais explorada na busca de aquisição das informações necessárias ao planejamento estratégico e a formulação de políticas de desenvolvimento racionais. Mais especificamente os imageadores orbitais, satélites artificiais projetados para adquirir imagens da Terra a partir do espaço, e os imageadores embarcados, dispositivos sensores a bordo de aviões ou ônibus espaciais, têm sido de grande auxílio nesta tarefa (MONTEIRO, 1997).

As muitas informações presentes nas imagens capturadas de uma cena, armazenadas em meio digital, precisam ser reconhecidas em função das necessidades do problema a ser resolvido. Imagens geradas por imageadores orbitais ou embarcados representam, em geral, um grande volume de dados. Por exemplo, uma cena típica imageada pelo satélite americano da série Landsat TM possui 6100 x 6100 pixels (elementos de imagem) em cada uma de suas bandas espectrais.

Recuperar destas imagens a informação que nos interessa requer capacidades de processar, analisar e interpretar esta massa de dados digitais. Mais que isso, os problemas enfrentados, em geral, são críticos com relação a prazos e orçamento. A viabilização da relação tempo-de-processamento x periodicidade-do-projeto x custo, no caso de planejamentos e metodologias que envolvam grandes regiões usando sensoriamento remoto, passa pela exploração das muitas técnicas empregadas no processamento, análise e interpretação automática de imagens digitais. O tratamento automático pode reduzir custos, e a modificação algorítmica de alguns procedimentos, pode reduzir o tempo de processamento para estas etapas viabilizando a periodicidade necessária à solução dos problemas.

1.1.2 - Solução Automática: um desafio computacional

Metodologias que fazem uso de imageamento remoto, freqüentemente envolvem um procedimento básico de análise da cena representada, a fotointerpretação, onde o especialista interpreta a cena com base num produto fotográfico.

Este procedimento de classificação depende exclusivamente do fotointérprete. Nesta fase sua experiência e o seu conhecimento da região em análise são muito importantes para a correta interpretação da cena. As informações extraídas são transferidas para o computador, em geral, através de uma mesa digitalizadora ou scanners.

O procedimento de classificação automática objetiva automatizar a etapa de fotointerpretação, fazendo com que a interpretação de toda cena se baseie nas propriedades de cada pixel e/ou de seus vizinhos com a mínima intervenção possível para o operador.

O desafio computacional está em gerar soluções, usando imagens de satélites e/ou imagens de sensores aerotransportados, com a eficiência necessária para o suporte à implantação de programas operacionais, com metodologias que façam uso efetivo de imageamento remoto e processamento digital.

1.1.3 - Os Problemas da Classificação Automática

O método mais tradicional de extração de informação a partir de imagens geradas por plataformas orbitais ou embarcadas em satélites é a fotointerpretação. Fotointerpretação, é a atividade executada por um ou mais especialistas, que usando uma mesa de luz, uma foto, na verdade um produto fotográfico gerado da imagem capturada pelos sensores, e um papel transparente sobre a foto (*overlay)*, identifica e rotula regiões, ou seja classifica, os elementos encontrados na cena ali representada. Esta informação é então georreferenciada, digitalizada, e armazenada em uma base digital, constituindo-se em um mapa temático que pode ser então usado onde for necessário.

O processo de classificação consiste na identificação de áreas com padrões similares em uma determinada imagem. Cada área identificada pode ser classificada como uma classe homogênea. A classificação se baseia nos valores dos pixels de cada banda espectral da imagem.

Os processos de classificação podem ser divididos em dois grandes grupos:

- *Pixel a Pixel*: a classificação de um certo *pixel* baseia-se apenas na informação espectral do *pixel* em questão;

- Contextual: neste caso, a classificação de um *pixel* leva em conta não somente o valor espectral do *pixel* em questão, mas também os valores dos seus vizinhos.

A imagem resultante é uma imagem dita temática, com seus pixels associados a classes (temas) pré-definidas pelo operador.

Pode-se, ainda, definir os processos de classificação como supervisionados ou não-automáticos (o operador deve treinar os sistemas, identificando na imagem amostras de cada classe) e ainda como não-supervisionados ou automáticos, onde o sistema é responsável por identificar automaticamente todas as classes e padrões (BATISTA et al., 1996).

Neste trabalho aborda-se um método de classificação que se baseia na mistura dos dois tipos de classificação anteriormente apresentados. Para esta miscelânea dá-se o nome de classificação semi-automática do tipo contextual.

A classificação (semi) automática de imagens de sensoriamento remoto visa gerar estes mapas temáticos a partir das imagens digitais com intervenção mínima. Este problema tem se constituído ao longo dos vários anos de pesquisa e desenvolvimento na área de processamento digital de imagens e correlatas como um desafio científico e tecnológico permanente.

O principal ponto a favor do fotointérprete, é que ele pode usar vários outros tipos de informações, tais como área e formato das regiões; informações contextuais, tais como as inter-relações entre pixels e/ou regiões; informações topográficas e resultados de classificações anteriores, para auxiliar no processo de classificação da imagem.

Dentre as diversas abordagens para a criação de um procedimento automatizado de classificação, encontra-se uma que divide o problema em duas partes. A primeira parte engloba os procedimentos que buscam identificar regiões homogêneas nas imagens. A segunda parte engloba as estratégias que buscam rotular, estabelecer o significado de cada uma das regiões já identificadas.

Na fase da identificação das regiões homogêneas, existe uma operação cuja finalidade é particionar a imagem, gerando partições que tenham uma correlação muito forte com os objetos ou áreas reais presentes na cena representada pelas imagens. Este procedimento para o particionamento da imagem é conhecido como segmentação de imagem (HEIJDEN, 1994).

Com outras palavras, a segmentação é uma etapa da análise de imagens que consiste na divisão desta em diferentes regiões, com base em alguma propriedade da mesma. As regiões resultantes da segmentação têm sempre duas características básicas: (1) exibem alguma uniformidade interna com relação à propriedade selecionada da imagem, e (2) diferem de sua vizinhança com base nesta mesma propriedade. As principais abordagens relativas às características do tipo (2) são baseadas na limiarização

(*thresholding*), no crescimento de regiões e na divisão e fusão de regiões. Já com relação às características do tipo (1), estas se baseiam nas mudanças abruptas no nível de cinza, sendo a detecção de pontos isolados e detecção de linhas e bordas na imagem as principais áreas de interesse dentro desta categoria.

Resumindo, uma série de algoritmos para classificação de imagens existe, e a cada dia outros são desenvolvidos. Estes algoritmos estão em sua grande maioria baseados em dois critérios:

- Descontinuidade;
- Similaridade.

O procedimento de buscar um particionamento da imagem em regiões, as quais devem ser compostas por pixels que apresentam conformidade com algum critério de similaridade, gera a operação conhecida como segmentação por regiões.

1.1.4 - O Módulo de Classificação do SPRING

A metodologia descrita em Batista et al. (1996) tem por objetivo automatizar o processo de interpretação, minimizando as etapas de análise e interpretação visual, apresentando uma alternativa de menor custo de processamento e de melhor precisão. Tal metodologia tem como entrada, imagens Landsat-TM, utilizando 3 bandas com a resolução de 30 x 30m, como mostra a Figura 1.1.

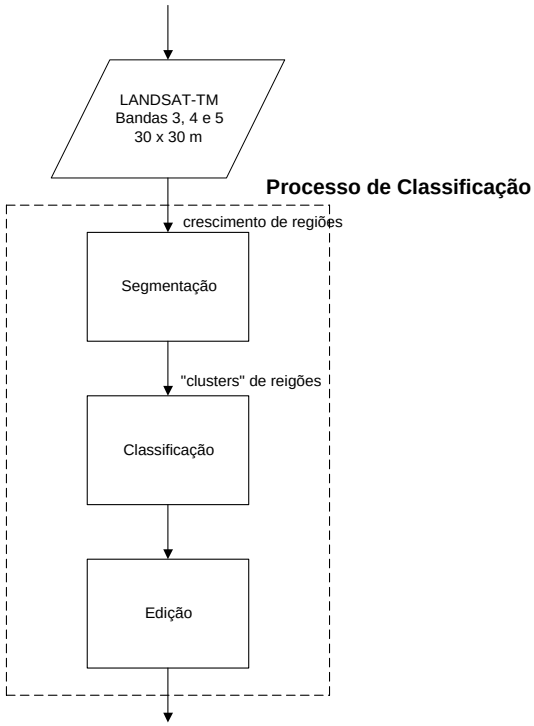


Fig. 1.1 – Esquema do Processo.

Fonte: Batista et al. (1996)

A fase de segmentação se baseia no algoritmo de crescimento de regiões, o qual agrega pixels com propriedades similares (BROWN e BALLARD, 1982).

A fase de classificação, descrita em Batista et al. (1996), baseia-se no algoritmo de agrupamento aplicado a um conjunto de regiões caracterizadas por atributos estatísticos que testa a média entre regiões (clusters de regiões).

Já a fase de edição baseia-se no contexto visual, dependendo exclusivamente do operador.

Baseado na metodologia descrita em Batista et al. (1996), a Tabela 1.1 apresenta os tempos de segmentação utilizando-se imagens Landsat-TM de diferentes tamanhos.

Tabela 1.1 - Comparação entre Tempos de Processamento para a Fase de Segmentação.

Imagem	**Número de Pixels**	**Número de Bandas**	**Equipamento**	**Tempo de Segmentação (horas)**
Landsat-TM	2700x1700	3	IBM/RISC 600-530, 96 Mb de RAM	24
Landsat-TM	6100x6100	3	SUN SPARCstation - 20, 96 Mb RAM, 270 Mb de memória virtual	35

Fonte: Adaptada de Shimabukuro et al. (1997).

Analisando os resultados expostos na Tabela 1.1, pode-se notar que o tempo gasto durante o processo de segmentação das imagens citadas ultrapassa a carga horária de trabalho diário de qualquer pessoa. Portanto, se um operador iniciar o processo de classificação hoje, ele somente terá o resultado depois de alguns dias de trabalho e isso não é viável, pois muitas vezes ele tem que definir parâmetros incertos de entrada, como limiar de área e limiar de similaridade, que ao final do processamento podem não gerar um resultado interessante, tendo o operador que iniciar o processo novamente com outros parâmetros e aguardar mais uma vez todo processamento.

Portanto, este trabalho objetiva a paralelização do código desenvolvido em C++, apresentado em Bins et al. (1992), visando a melhora significativa no tempo de processamento envolvido na fase de segmentação do processo de classificação digital.

O paralelismo, neste contexto, é explorado pela execução simultânea de tarefas em processadores distintos, através da biblioteca de troca de mensagens Message Passing Interface (MPI).

1.2 - Proposta de Paralelização do Segmentador

O tratamento automático pode reduzir custos, e a modificação algorítmica de alguns procedimentos pode reduzir o tempo de processamento para as etapas de tratamento de imagens, viabilizando a periodicidade necessária à solução de alguns problemas.

O processamento paralelo de imagens aparece como um caminho atraente, pois, pode-se dizer que a necessidade cada vez maior que se tem de potência em processamento computacional pode não ser satisfeita somente por um processador que se torna mais rápido e eficiente após determinados intervalos de tempo. Para satisfazer algumas das necessidades imediatas que ainda não podem ser resolvidas em tempo hábil por somente um processador, é necessário um conjunto dos mesmos processadores trabalhando por um objetivo comum.

O processo de segmentação de imagens não é diferente. Com mais recursos de aquisição e armazenamento, quanto mais informação puder ser processada ao mesmo tempo em intervalos aceitáveis, melhor.

Baseado na metodologia de operação proposta em Batista et al. (1996) nota-se que a fase de segmentação é componente necessária e merece uma atenção especial, pois é uma fase que consome muito tempo de processamento.

O objetivo principal deste trabalho, é reduzir este tempo, explorando o paralelismo inerente ao algoritmo de crescimento de regiões, visando a potencialização da metodologia descrita em Batista et al. (1996).

A intenção é potencializar o uso de tal metodologia, reduzindo o tempo necessário à fase de segmentação, dentro do procedimento de classificação ali proposto, sem a preocupação com otimização do algoritmo de crescimento de regiões bem como com técnicas e/ou bibliotecas de paralelização que apresentam melhor desempenho, pois o objetivo é desenvolver um módulo viável, e portável.

1.3 - Organização da Dissertação

Além desse Capítulo 1 – Introdução - esta dissertação é dividida em outros quatro capítulos, que serão organizados como segue:

No Capítulo 2 – Revisão Bibliográfica – será apresentada uma visão geral sobre a paralelização de algoritmos, processamento de imagens e a principal biblioteca baseada em troca de mensagens utilizada na atualidade. Em primeiro lugar, é apresentado o algoritmo de segmentação por regiões. Em seguida são apresentados, de forma sucinta, os principais conceitos de computação de alto desempenho, seus paradigmas, tipos de comunicação, análise de dependência entre os dados, conceitos sobre paralelização de imagens e finalmente apresenta-se rapidamente as principais característica do padrão de troca de mensagens utilizado no desenvolvimento deste trabalho, o MPI.

No Capítulo 3 – Estratégia da Solução - será discutida sucintamente a forma de abordagem do problema e o esquema de paralelização do algoritmo de segmentação mencionado.

No Capítulo 4 – Testes e Resultados – será detalhado o plano de testes realizados com as imagens pilotos, bem como os seus resultados obtidos durante o processo de segmentação de tais imagens, as formas de avaliação dos resultados e uma análise sobre o desempenho do sistema ora proposto.

No Capítulo 5 – Conclusões e Recomendações – apresentar-se-á as conclusões tiradas no decorrer do desenvolvimento dessa dissertação. Também serão apresentadas algumas sugestões para estender e otimizar o sistema aqui desenvolvido na forma de trabalhos futuros.

CAPÍTULO 2

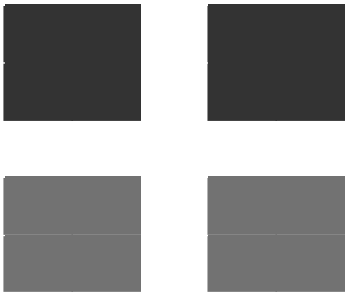
REVISÃO BIBLIOGRÁFICA

2.1 – Imagem Digital

Uma imagem digital pode ser definida como uma função de intensidade bidimensional, f(x,y), onde x e y são as coordenadas espaciais do ponto e o valor de f em qualquer ponto (x,y) é proporcional ao brilho da cena naquela localização. Em uma imagem digital de sensoriamento remoto, este último valor está associado aos valores de radiância dos alvos da realidade terrestre que sensibilizaram o sistema sensor, com uma dada condição de iluminação e de geometria (GONZALEZ E WOODS, 1992). Essa definição é a mais simples e se aplica mais apropriadamente a imagens monocromáticas. No caso de imagens coloridas existem na verdade, para cada ponto da imagem, 3 valores associados: um ao canal vermelho (R), outro ao verde (G) e o último ao azul (B), respectivamente. Pode-se representar então a imagem por uma função Fcor (Equação 1.1), dependente de três outras funções, referentes aos canais R, G e B, respectivamente, ou seja:

$$Fcor\ (x,y) = f\ (fR(x,y);fG(x,y);fB(x,y)) \qquad (1.1)$$

As imagens de sensoriamento remoto podem então ser entendidas como uma matriz numérica bidimensional, onde cada valor se refere a uma área da superfície imageada (Figura 2.1). Cada pequenina célula da matriz é denominada pixel, que é considerado como a menor porção de uma imagem digital. A dimensão da área física representada por um pixel é denominada tamanho do pixel, podendo variar de poucos nanômetros (imagens de microscópio) até dezenas de quilômetros (imagens de alguns satélites). O menor valor digital que um pixel pode ter é zero e o maior depende de como a imagem foi quantizada (número de bits) pelo equipamento que a produziu. Esta idéia dá noção de resolução radiométrica



(a)

5	**5**	**254**	**5**	**5**
5	**5**	**254**	**5**	**5**
254	**254**	**254**	**254**	**254**
120	**120**	**254**	**120**	**120**
120	**120**	**254**	**120**	**120**

(b)

Fig. 2.1. – Imagem de uma Cruz (a) com sua Representação na Forma Digital (b).

Fonte: Adaptada de Crosta (1993)

Com relação à visualização das imagens digitais, esta pode ser realizada em monitores de vídeo ou através de impressão em papel. Tal visualização pode ainda ser realizada em modo monocromático ou multiespectral. A imagem monocromática é, na verdade, representada em tonalidades de cinza que variam do preto ao branco. Já as imagens multiespectrais são normalmente visualizadas através de composições coloridas.

2.2 - O Algoritmo de Segmentação por Crescimento de Regiões

Os algoritmos de segmentação por crescimento de regiões agrupam pixels dentro de regiões homogêneas. Crescimento de Regiões é uma classe de técnicas usadas nos algoritmos de segmentação de imagens em que tipicamente, regiões são construídas por um processo de aglomeração que une pixels a regiões adjacentes quando estes têm propriedades similares as daquela região. Cada pixel da imagem recebe um rótulo através do processo de crescimento de regiões. Pixels terão o mesmo rótulo se e somente se eles pertencerem à mesma região (BADER et al., 1995).

Supondo-se que RT represente uma imagem completa, pode-se considerar o processo de segmentação por crescimento de regiões como a divisão de RT em n subregiões, R1, R2, ..., Rn, tal que:

$$\bigcup_{i=1}^{n} R_i = R_T; \qquad (2.2)$$

$$R_i \text{ é uma região conexa, } i=1, 2, 3, ..., n;; \quad (2.3)$$

$$R_i \cap R_j = \varnothing \quad \text{para todo i e j, } i \neq j; \quad (2.4)$$

$$H(R_i) = \text{Verdadeira para } i=1, 2, 3, ..., n; \quad (2.5)$$

$$H(R_i \cup R_j) = \text{Falso para } R_i \text{ e } R_j \text{ adjacentes, } i \neq j \quad (2.6)$$

onde H (Ri) é o critério de homogeneidade para os pixels da região Ri e $\varnothing$ é o conjunto vazio (GONZALEZ e WOODS, 1992).

A Equação 2.2 indica que a segmentação precisa ser completa, ou seja, cada pixel tem que pertencer a uma região. A Equação 2.3 especifica que os pixels de uma mesma região precisam ser conexos (para se estabelecer se dois pixels são conexos ou não precisa determinar se eles são adjacentes de algum modo, ou seja, vizinhança 4 ou 8, e se seus níveis digitais satisfazem ao critério de similaridade especificado). A Equação 2.4 determina que as regiões precisam ser disjuntas. A Equação 2.5 significa que todos os pixels que pertencem a uma mesma região precisam satisfazer ao critério de homogeneidade. Finalmente, a Equação 2.6 indica que uma região formada pela união de duas regiões Ri e Rj adjacentes não poderá satisfazer ao critério de homogeneidade H.

Conforme pode ser notado no parágrafo anterior, uma região é um conjunto de pixels conectados que satisfazem a algum critério de homogeneidade. Os algoritmos de crescimento de regiões tipicamente iniciam a partir de certos pixels “sementes” na imagem, assinalando os pixels da vizinhança, que satisfaçam a tal critério, como pixels de mesma região da semente em questão. A homogeneidade é a propriedade mais importante das regiões, e pode ser definida em termos do nível de cinza, da cor, textura, etc. Um exemplo de critério usado para análise pode ser a verificação da diferença absoluta entre o valor do pixel e a média dos níveis de cinza da região em questão. Tal pixel pode ser dito como pertencente à região caso esta diferença seja menor que um determinado limiar. Esta função pode ser descrita como (Fórmula 2.7):

$$H(p,R) = \begin{cases} \text{VERDADEIRO SE} & |p - x| \leq \text{Limiar} \\ \text{FALSO CASO CONTRÁRIO} & \end{cases} \qquad (2.7)$$

onde “p” é o valor do pixel sendo testado com relação à homogeneidade da região “R”. A variável “x” é o nível de cinza médio dos pixels da região R e o “Limiar” é um valor que pode ser dado pelo usuário para controlar a aglutinação de pixels.

A Figura a seguir ilustra um exemplo de tal procedimento.

	1	2	3	4	5
1	0	0	5	6	7
2	1	1	5	8	7
3	0	1	6	7	7
4	2	0	7	6	6
5	0	1	5	6	5

(a)

a	a	b	b	b
a	a	b	b	b
a	a	b	b	b
a	a	b	b	b
a	a	b	b	b

(b)

a	a	a	a	a
a	a	a	a	a
a	a	a	a	a
a	a	a	a	a
a	a	a	a	a

(c)

Fig.2.2 – Exemplo de aplicação do método de crescimento de regiões usando dois *pixels* “sementes”: (a) imagem original; (b) resultado da segmentação usando uma diferença absoluta menor que 3 entre os níveis digitais; (c) resultado usando uma diferença absoluta menor que 8.

Fonte: Adaptada de Gonzalez e Woods (1992).

Considerando-se a Figura 2.2 (a), os números no interior das células representam os valores dos níveis digitais da imagem. Os pixels com coordenadas (3,2) e (3,4) são usados como “sementes”. Usando dois pixels sementes iniciais o resultado da segmentação consistiu de duas regiões: (a) associada com a semente (3,2) e (b) associada com a semente (3,4). A propriedade H usada para incluir os pixels vizinhos em cada região foi que a diferença absoluta entre os níveis digitais destes pixels e o da semente fosse menor que um valor limite T. Qualquer pixel que satisfizesse esta condição simultaneamente para as duas sementes foi arbitrariamente marcado como região (a). A Figura 2.2 (b) mostra o resultado obtido usando um limiar T=3. Entretanto, se fosse escolhido um valor limite igual a 8, o resultado seria o mostrado na Figura 2.2 (c).

Ainda na Figura 2.2 pode-se observar, apesar de sua simplicidade, que existem algumas dificuldades associadas com o crescimento de regiões. Dois problemas imediatos são a seleção dos pixels sementes que representam apropriadamente as regiões de interesse e a seleção da propriedade adequada para a inclusão de pixels nas regiões durante o processo de crescimento.

Segundo Gonzalez e Woods (1992), a seleção das "sementes" pode ser baseada na natureza do problema. Por exemplo, em aplicações militares de imageamento infravermelho, os alvos de interesse geralmente são mais quentes que o resto da imagem (pontos usualmente representados mais claros que os outros). Neste caso, é ideal a escolha de pixels mais claros como "sementes" para o algoritmo de crescimento de regiões. Já a seleção da propriedade não depende somente do problema em consideração mas, também do tipo de dados disponíveis da imagem. Por exemplo, uma análise de uso do solo baseada em imageamento orbital depende fortemente da utilização da cor. Este problema será significantemente mais difícil de resolver se forem utilizadas imagens monocromáticas de forma isolada.

Concluindo, o algoritmo de segmentação tem os seguintes parâmetros:

- Similaridade: menor limiar a partir do qual duas regiões são consideradas similares e, portanto, devem ser agrupadas. Na implementação do SPRING, o critério de similaridade é a distância Euclidiana entre as médias espectrais das regiões envolvidas no teste;

- Área: a menor área a ser considerada como uma região, definida em número de pixels.

Várias técnicas de segmentação de imagens podem ser vistas em Pal e Pal (1993).

2.3 - Paralelismo e Imagens

2.3.1 - Fundamentos de Computação de Alto Desempenho

A cada ano, a indústria de hardware investe alguns bilhões de dólares em busca de computadores mais poderosos. Basicamente, o investimento em novas tecnologias atua em duas frentes:

- Expansão dos limites físicos que determinam a velocidade máxima em que os circuitos elétricos conseguem operar;
- Desenvolvimento de novas arquiteturas que permitam obter maior desempenho, dentro dos limites físicos atuais.

As máquinas lançadas a cada ano sempre trazem avanços nestes dois aspectos. No primeiro, nota-se comumente o lançamento de processadores já conhecidos, porém com um clock um pouco mais elevado.

O segundo aspecto refere-se a diferenças estruturais, que podem ir desde mudanças na estrutura interna do processador, passando por mudanças na montagem e conexão deste aos demais dispositivos do computador, ou até mesmo em relação ao uso de vários computadores.

Uma das principais estratégias utilizadas para obtenção de maior desempenho é o paralelismo. Este termo pode ser aplicado em vários níveis:

Vários computadores trabalhando em conjunto para realizar uma determinada tarefa;

- Dentro de um mesmo computador, que pode ter vários processadores;
- Dentro de cada processador, que pode ter várias unidades de execução para executar instruções em paralelo.

Quando se refere ao hardware, existem dois paradigmas de paralelismo: *Symmetric Multi Processing* (SMP) e *Massive Parallel Processing* (MPP).

Sistemas SMP possuem mais de um processador em um mesmo computador. Todos eles compartilham os recursos de memória e disco existentes, segundo uma política de controle de concorrência adotada pelo sistema operacional. Esta complexa arquitetura, entretanto, é bastante transparente para o desenvolvimento de aplicações, pois a maior parte da complexidade fica a cargo do sistema operacional.

Em sistemas MPP os processadores possuem maior independência entre si, havendo pouco ou nenhum compartilhamento de recursos. Tipicamente, cada nó (processador) de um sistema MPP é um computador independente, com memória e disco próprios. O controle do paralelismo é feito pela aplicação, que deve coordenar a distribuição de tarefas e a coerência entre os diversos nós.

Cada alternativa apresenta vantagens e desvantagens. A complexidade de ambos cresce à medida que aumenta o número de nós/processadores. Em SMP, um elevado número de processadores resulta em menor desempenho por processador, pois os mesmos recursos precisam ser compartilhados por todos eles. Em MPP, cada nó trabalha com memória e disco próprios, permitindo que sejam utilizados muitos deles. Por isso mesmo, a aplicação tem que manter a coerência entre todos e garantir a distribuição eficiente do trabalho, caso contrário, pode haver desperdício do poder de processamento total (HWANG, 1993).

2.3.1.1 - O que é Computação Paralela?

Paralelismo é o processo de realizar tarefas concorrentemente, objetivando a obtenção de resultados mais rápidos de tarefas grandes e geralmente complexas (LEWIS e EL-REWINS, 1992). O paralelismo é obtido em geral:

- Dividindo uma tarefa em várias pequenas tarefas e distribuindo estas pequenas tarefas entre vários processadores que irão executá-las simultaneamente;
- Coordenando as atuações desses processadores.

A Figura abaixo mostra a comunicação entre os processadores para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo.

Fig. 2.3 - Comunicação entre os Processadores
Fonte: CENAPAD (1999)

Pode-se pensar em paralelismo em tarefas que possam ser compreendidas como partes independentes. Por exemplo, considere o seguinte laço em FORTRAN:

```
do 10   I = 1,10
        a(I) = 1.0
10  continue
```

Todas as operações realizadas dentro do laço são independentes e podem ser feitas simultaneamente, portanto "paralelizar" aqui seria natural. Contudo, deve-se evitar tarefas, nas quais, coordená-las leve mais tempo que executá-las, pois tem-se que contabilizar o tempo gasto para se dividir uma tarefa e o gerenciamento de todas as suas partes (overhead). Dependendo do caso, o tempo de sincronização e coordenação das pequenas tarefas (overhead) é tão alto que não compensa "paralelizar" a tarefa (LEWIS e EL-REWINS, 1992).

Dentre as várias formas de classificar o paralelismo, leva-se em consideração o objeto paralelizado, como segue abaixo:

Paralelismo de Dados

O processador executa as mesmas instruções sobre dados diferentes. É aplicado, por exemplo, em programas que geralmente utilizam matrizes imensas e para cálculos de elementos finitos (Figura 2.4).

Fig. 2.4 - Paralelismo de Dados.
Fonte: CENAPAD (1999)

Exemplos:
Resolução de sistemas de equações;

- Multiplicação de matrizes;
- Integração numérica.

O paralelismo de dados é o mais fácil de se estudar, escrever, e depurar. Linguagens que usam este tipo de paralelismo são uma simples extensão de sua linguagem seqüencial correspondente onde a responsabilidade de paralelização é do programador. Alguns exemplos são C*, Dataparallel C, Marpar´s MPL como extensões C, PC++ com extensão do C++; e a operação com matrizes no HPF e no FORTRAN 90.

Paralelismo Funcional

O processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados. É aplicado em programas dinâmicos e modulares onde cada tarefa será um programa diferente (Figura 2.5).

Fig. 2.5 – Paralelismo Funcional.

Fonte: CENAPAD (1999)

Exemplos:

Paradigma Produtor-Consumidor;

- Simulação;
- Rotinas específicas para tratamento de dados (imagens).

<u>Paralelismo de Objetos</u>

Este é o modelo mais recente, que se utiliza o conceito de objetos distribuídos por uma rede, como a Internet, capazes de serem acessados por métodos (funções) em diferentes processadores para uma determinada finalidade (Figura 2.6).

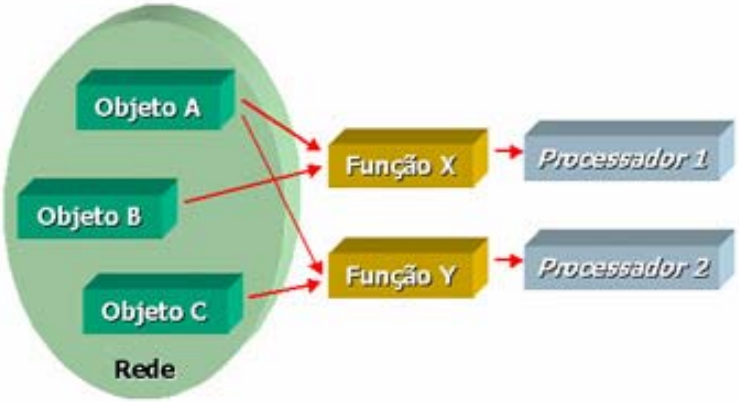


Fig. 2.6 – Paralelismo de Objetos.

Fonte: CENAPAD (1999)

Exemplo:

MPP - *Massive Parallel Processing* (Processamento Massivamente Paralelo), compreendendo o conceito de massivamente distribuído.

2.3.1.2 - Ambiente Paralelo

Um ambiente paralelo (Figura 2.7) define-se da seguinte forma:
Vários processadores interligados em rede;

a) Plataforma para manipulação de processos paralelos;

b) Sistema Operacional;

c) Linguagem de Programação;

d) Modelos de Programação Paralela.

Message Passing: é o método de comunicação baseado no envio e recebimento de mensagens através da rede seguindo as regras do protocolo de comunicação entre vários processadores que possuam memória própria. O programador é responsável pela sincronização das tarefas. Exemplos de bibliotecas paralelas:

- PVM - *Parallel Virtual Machine*
- MPI - *Message Passing Interface*
- MPL - *Message Passing Library*

Data Parallel: é a técnica de paralelismo de dados, normalmente automática ou semi-automática, ou seja, é o compilador que se encarrega de efetuar toda a comunicação necessária entre os processos de forma que o programador não necessita entender os métodos de comunicação. Exemplo:

- HPF - *High Performance* FORTRAN

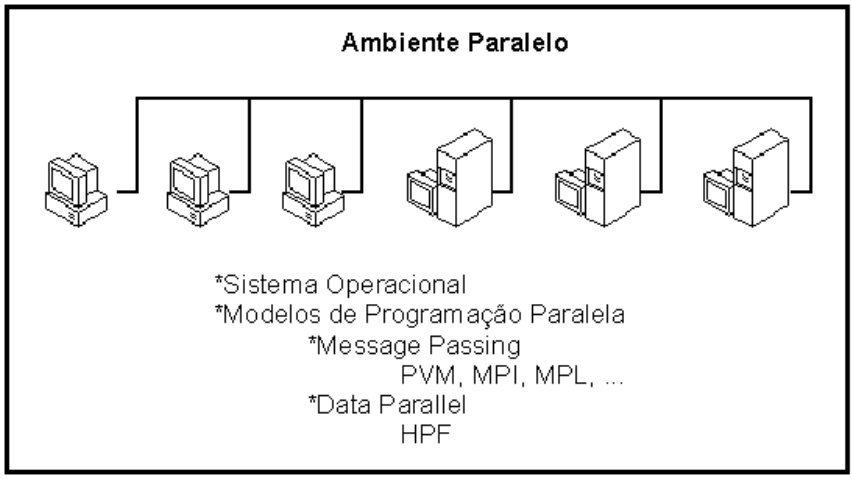


Fig. 2.7 - Ambiente Paralelo
Fonte: CENAPAD (1999)

Especialistas em computação paralela definiram paradigmas de programação baseados nas arquiteturas dos computadores com o objetivo de propiciar aos programadores uma base de abstração para os detalhes de hardware.

2.3.1.3 - Paradigmas da Computação Paralela

Um paradigma é um modelo, usado para formular uma solução computacional de classes de problemas.

Por volta de 1966 foi estabelecido o Método Flynn, o qual avalia a relação entre o número de instruções de um programa com os dados, baseando-se na arquitetura dos computadores (LEWIS e EL-REWINS, 1992).

De acordo com Hwang (1993) e Lewis e El-Rewins (1992) a taxonomia Flynn distingue os modelos *Single Instruction, Single Data* (SISD), *Single Instruction, Multiple Data* (SIMD) e *Multiple Instruction, Multiple Data* (MIMD). No caso deste trabalho serão abordados apenas os modelos MIMD e SIMD.

2.3.1.3.1 - O Modelo MIMD

Multiple Instruction, Multiple Data é o modelo mais comum de paralelismo. A sincronização é alcançada explicitamente e localmente através de um mecanismo de sincronização global. É um modelo flexível, mas de difícil controle (LEWIS e EL-REWINS, 1992). O MIMD, como na Figura 2.8, é usado quando o problema admite várias tarefas heterogêneas serem realizadas ao mesmo tempo.

Segundo Lewis e El-Rewins (1992) as características fundamentais do modelo MIMD são:

- O paralelismo é alcançado pela conexão de vários processadores;
- Inclui todas as formas de configuração de processadores;
- Cada processador executa seu fluxo de instrução independente dos outros processadores em um único fluxo de dados.

As vantagens do modelo MIMD são:

- Processadores podem executar vários fluxos de trabalho simultaneamente;
- Cada processador pode realizar qualquer operação independentemente do que os outros processadores estão fazendo.

A principal desvantagem do modelo MIMD é:

- Overhead no balanceamento de carga - tempo necessário para que todos os processadores terminem a tarefa que possivelmente estejam realizando e possam ser sincronizados.

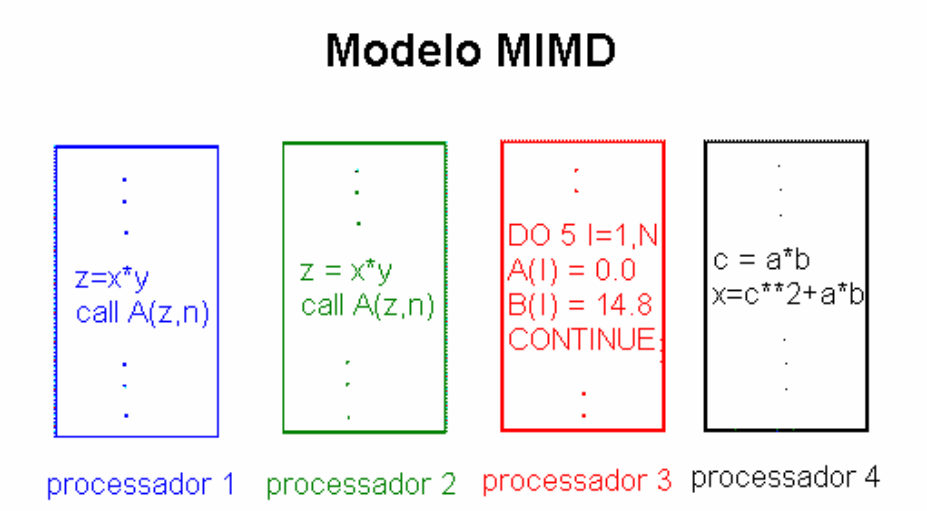


Fig. 2.8 - Modelo MIMD.

Fonte: Singh (1993).

2.3.1.3.2 - Os Modelos SIMD e SPMD

Já o modelo SIMD parece restrito a primeira vista, mas é talvez o paradigma mais usado para computação paralela científica.

Em um computador SIMD como na Figura 2.9, uma única instrução corrente está agindo sobre muitos elementos que estão sendo processados. Uma instrução contadora é usada para controlar a seqüência através de uma única cópia do programa. O dado que é processado por cada elemento de processamento difere de processador para processador. Portanto, um único programa e uma única unidade de controle agem simultaneamente em muitas coleções de dados diferentes, (LEWIS e EL-REWINS, 1992).

SIMD é um exemplo de computação de dados paralelos sincronizados. Essa forma de paralelismo não deve ser confundida com SPMD (*Single Program - Multiple Data*) que é uma forma de paralelismo assíncrona. SPMD permite o processamento simultâneo de dados diferentes, mas sem coordenação (*lockstep*). SPMD não é parte da Taxonomia Flynn.

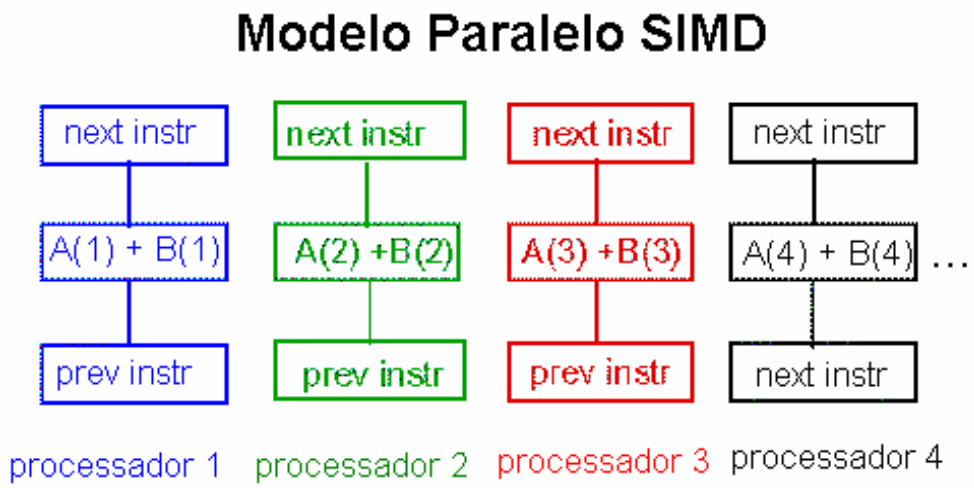


Fig. 2.9 - Modelo Paralelo SIMD.

Fonte: Singh (1993).

No modelo SPMD, um programa inteiro é executado em dados separados, com isso é possível que diferentes ramos sejam criados, precedendo o paralelismo sincronizado. Os processadores não estão mais fazendo a mesma coisa, ou não executando nenhuma tarefa. Agora, eles estão ocupados executando instruções diferentes dentro do mesmo programa. SIMD envolve uma instrução contadora, SPMD envolve várias instruções contadoras.

As vantagens do modelo SIMD são:

- Laço condizente para o paralelismo SIMD

```
do 100     I=1, 100

           c(I) = a ( i ) + b ( i )

100        continue
```

- Sincronização não é um problema, todos os processadores operam de forma amarrada (l*ock-step*);

A principal desvantagem do modelo SIMD é que as decisões dentro de laços podem resultar numa execução deficiente, devido a necessidade de todos os processadores realizarem a operação controlada pela decisão sem saber se os resultados serão usados ou não.

Aplicações SPMD são admitidas trivialmente paralelas. SPMD é diferente do SIMD, porque os processadores não estão firmemente sincronizados, eles estão sincronizados somente no inicio e no final de um procedimento ou seção de código que é duplicada em todos os processadores. Os processadores executam sincronismo entre cada procedimento ou seção idêntica do código para ceder uma forma de operação pseudo - SIMD (LEWIS e EL-REWINS, 1992);

Portanto, durante o projeto de desenvolvimento de um código, deve-se determinar o modelo de programação a ser usado, baseando-se na arquitetura do equipamento e no software a ser empregado. O modelo de programação é um item importante quando se fala em análise de desempenho, mas outros critérios devem ser analisados conjuntamente.

2.3.2 - Comunicação entre os Processadores

A maneira como os processadores se comunicam é dependente da arquitetura de memória utilizada no ambiente, que por sua vez irá influenciar na maneira de se escrever um programa paralelo.

2.3.2.1- Memória Compartilhada

Num ambiente de memória compartilhada, como na Figura 2.10, vários processadores operam independentemente, mas compartilham o mesmo recurso de memória, somente um processador pode acessar a memória de cada vez e a sincronização é alcançada através do controle das tarefas.

As principais vantagens de um ambiente de memória compartilhada são:

Facilidade em se alcançar eficiência;

- Rápido compartilhamento dos dados através das tarefas.

As principais desvantagens são:

Memória limitada;

- Usuário é responsável pela especificação da sincronização.

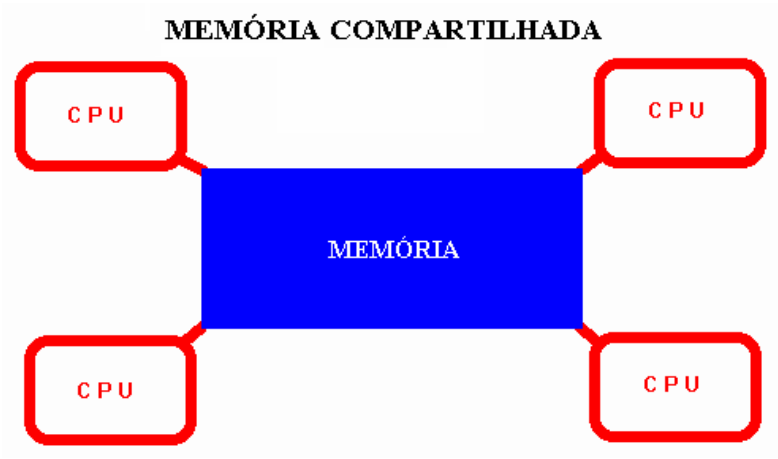


Fig. 2.10 - Ambiente de Memória Compartilhada.

Fonte: MPCC (1997).

2.3.2.2 - Memória Distribuída

Num ambiente de memória distribuída, como mostra a Figura 2.11, vários processadores operam independentemente, mas cada um tem sua própria memória, o dado é compartilhado através da rede de comunicação utilizando-se troca de mensagem e o usuário é responsável pelo sincronismo.

As principais vantagens são:

- Memória escalonável;

- Cada processador pode acessar rapidamente sua própria memória sem interferência.

As principais desvantagens de um ambiente de memória distribuída são:

- Dificuldade em mapear a estrutura de dados para a memória;
- O usuário é responsável pelo envio e recebimento dos dados através dos processadores;
- Para minimizar o overhead, os dados devem ser empacotados e enviados antes que outro nó necessite dele.

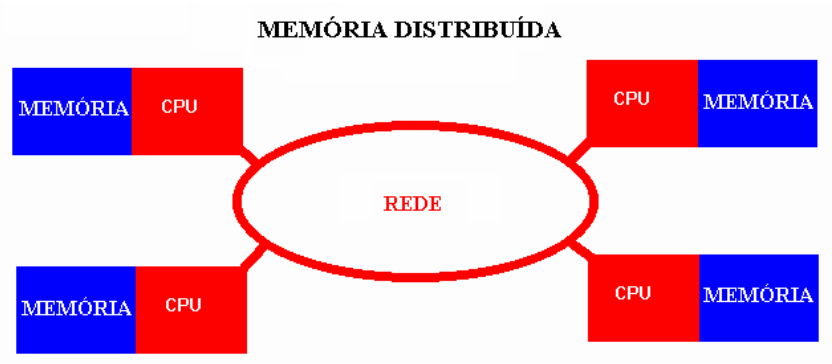


Fig. 2.11 - Ambiente de Memória Distribuída.
Fonte: MPCC (1997).

2.3.3 - Análise de Desempenho

Na computação seqüencial, um algoritmo é bem caracterizado em termos do trabalho que realiza, podendo ser avaliado pela contagem das operações envolvidas e da quantidade de memória necessária. Neste contexto, o desempenho de um computador é melhor do que de outro quando a mesma aplicação é executada mais rapidamente no primeiro que no segundo.

Uma medida muito utilizada para medir o desempenho de um computador é o MFlops (MFlop/s), que define o desempenho através da taxa de milhões de operações de ponto

flutuante por segundo que este é capaz de realizar. Este valor é conhecido como *peak performace*.

Com relação às máquinas paralelas, a dificuldade está na definição de medidas adequadas para avaliação do desempenho deste tipo de processamento. Medidas adequadas são essenciais para o estabelecimento de diretrizes no projeto de algoritmos, de novas arquiteturas, na identificação de gargalos que prejudicam a paralelização de algoritmo e até mesmo na escolha de uma arquitetura mais apropriada ao problema a ser resolvido.

Ainda que a velocidade de processamento e as exigências de memória continuem sendo fatores essenciais, deve-se também levar em conta outros dois fatores que afetam significativamente o desempenho neste tipo de computação: gastos de tempo para comunicação e perdas de tempo devido à sincronização. A influência destes fatores pode reduzir sensivelmente a eficiência de um algoritmo paralelo, resultando em tempos de processamento distantes do razoável quando aumenta-se o número de processadores e as dimensões do problema a ser resolvido.

Dentre as principais medidas que buscam avaliar o desempenho da computação paralela, estão o ganho (ou *speedup*) (S) e a eficiência (E). O ganho é um fator que compara o tempo total consumido por um algoritmo quando executado em uma máquina seqüencial em relação ao tempo requerido pelo mesmo algoritmo quando executado em uma máquina paralela com P processadores.

Quando fala-se em análise de desempenho, além do potencial de aumento de velocidade (ganho e eficiência), tem-se que considerar os seguintes itens:

- Balanceamento de carga;
- Granularidade;
- Dependência dos dados;
- Comunicação e tamanho da banda;

- Entrada/saída (input/output);
- Configuração do equipamento;
- *Dead lock*.

2.3.3.1 - Potencial de Aumento de Velocidade (ganho e eficiência)

As principais metodologias que fazem considerações ao desempenho de um programa baseiam-se na Lei de Amdahl (HWANG,1993). A Lei de Amdahl (Figura 2.12) determina o potencial de aumento de velocidade a partir da porcentagem paralelizável de um programa (f). Ver Equação 2.8:

$$\text{Speedup} = \frac{1}{1-f} \tag{2.8}$$

Num programa, no qual não ocorra paralelização, f=0, logo, speedup = 1. Não existe aumento na velocidade de processamento. Já, no programa no qual ocorra paralelização total, f=1, teoricamente o speedup é infinito se o número de processadores for infinito.

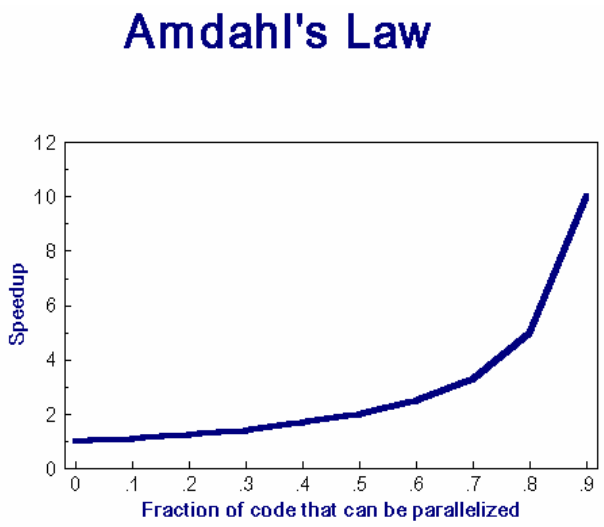


Fig. 2.12 - Lei de Amdahl.

Fonte: Hwang (1993).

De acordo com a Equação 2.9, se introduzir o número de processadores na porção paralelizável a relação passará a ser modelada por:

$$\text{Speedup} = \frac{1}{\frac{p}{N} + S} \tag{2.9}$$

onde: P = porcentagem paralelizável

N = número de processadores

S = porcentagem serial

Existem limites para a escalabilidade do paralelismo, pois, tem um ponto que o aumento do número de processadores envolvidos não necessariamente acarreta o aumento da porcentagem paralelizável de um programa.

Para compreender melhor o significado da Lei de Ahmdahl, veja como se comportariam os casos hipotéticos da Tabela 2.1.

Tabela 2.1 - Limites para Escalabilidade

N	**P=0,50**	**P=0,90**	**P=0,99**
10	1,82	5,26	9,17
100	1,98	9,17	50,25
1000	1,99	9,91	90,99
10000	1,99	9,91	99,02

Fonte: CENAPAD (1999).

A expressão do *Speedup* poderá ser empregada na sua forma simplificada devido a ausência dos parâmetros porcentagem paralelizável e porcentagem serial e/ou supondo-se que o programa em questão é 100% paralelizável. Portanto simplificando-se a

expressão de ganho, baseado num programa 100% paralelizável tem-se (Equação 2.10 e 2.11):

$$\text{speedup} = \frac{1}{\frac{p}{N} + S} \quad \text{se o programa é 100\% paralelo } P = 1 \text{ e } S = 0 \tag{2.10}$$

$$\text{speedup} = \frac{1}{\frac{1}{N}} \quad \therefore \quad \text{speedup} = N \tag{2.11}$$

Observando as Equações 2.12 e 2.13, ao se introduzir as variáveis de tempo, sabe-se que:

$$\text{Tpar} = \frac{Tseq}{N} \quad \therefore \quad N = \frac{Tseq}{\frac{Tseq}{N}} \tag{2.12}$$

Onde:

Tseq - tempo consumido por uma máquina seqüência

Tp - tempo consumido por uma máquina paralela

$$\text{se } N = \text{speedup} \qquad \text{speedup} = \frac{Tseq}{\frac{Tseq}{N}} \quad \therefore \quad \text{speedup} = \frac{Tseq}{Tpar} \tag{2.13}$$

No caso deste trabalho, será empregada a fórmula simplificada do speedup.

O fator ganho nos dá uma medida de como uma aplicação paralela é executada em comparação a um programa seqüencial equivalente. Teoricamente ele deve ser sempre menor ou igual ao número de processadores P. Entretanto, dependendo de como o tempo seqüencial seja medido, o valor máximo de *Speedup* (Sp) pode variar, uma vez que o processador usado para medir Tseq pode diferir muito em velocidade daqueles processadores usados na máquina paralela, podendo mascarar a eficiência do algoritmo paralelo.

A especificação do algoritmo usado é muito importante, já que algoritmos distintos para um mesmo problema apresentam tempos de processamento diferentes. Ou ainda, a versão paralela de um algoritmo pode não ser a melhor opção quando executada em um único processador. Uma alternativa é definir um algoritmo como sendo seqüencial ótimo para um problema particular e usá-lo para medir o tempo Tseq na expressão do ganho. A definição deste algoritmo ótimo, no entanto, é muito vaga.

Da definição de ganho, nota-se que seu valor ideal é igual ao número de processadores. (Sp=P). Isto leva ao conceito de eficiência de um sistema paralelo como apresenta a Equação 2.14:

$$E = \frac{Speedup}{P} \tag{2.14}$$

A eficiência (E) dá uma indicação da porcentagem do tempo total realmente dispendido na aplicação por cada um dos processadores.

2.3.3.2 - Balanceamento de Carga

Balanceamento de carga se refere à distribuição das tarefas entre os processadores. Esta distribuição deverá ser, de maneira que o tempo da execução paralela seja eficiente.

Para uma melhor distribuição deve-se considerar que as tarefas devem ser distribuídas de maneira balanceada. É possível que ocorra espera pelo término do processamento de uma única tarefa, para dar prosseguimento ao programa. O desempenho pode aumentar se o trabalho for igualmente distribuído. Deve-se também, considerar um ambiente heterogêneo onde existam máquinas com extrema variação de poder e carga do usuário versus um ambiente homogêneo com processadores idênticos executando uma tarefa por processador.

2.3.3.3- Granularidade

Granularidade é a medida da razão entre a quantidade de tarefas computacionais realizadas numa tarefa paralela, pela quantidade de comunicação efetuada.

Existem duas classificações para granularidade: grânulos finos e grânulos grosseiros.

2.3.3.3.1 - Grânulos Finos

Grânulos finos (*fine-grained*) - pouca tarefa computacional e alta comunicação como mostrado na Figura 2.13.

- Todas as tarefas executam um pequeno número de instruções entre os ciclos de comunicação;
- Facilidade para o balanceamento de carga;
- Pouca computação em razão da comunicação;
- Grande tempo gasto com o gerenciamento do paralelismo (overhead);
- Se a granularidade é muito fina, é possível que o overhead exija mais comunicação e sincronização entre as tarefas do que computação.

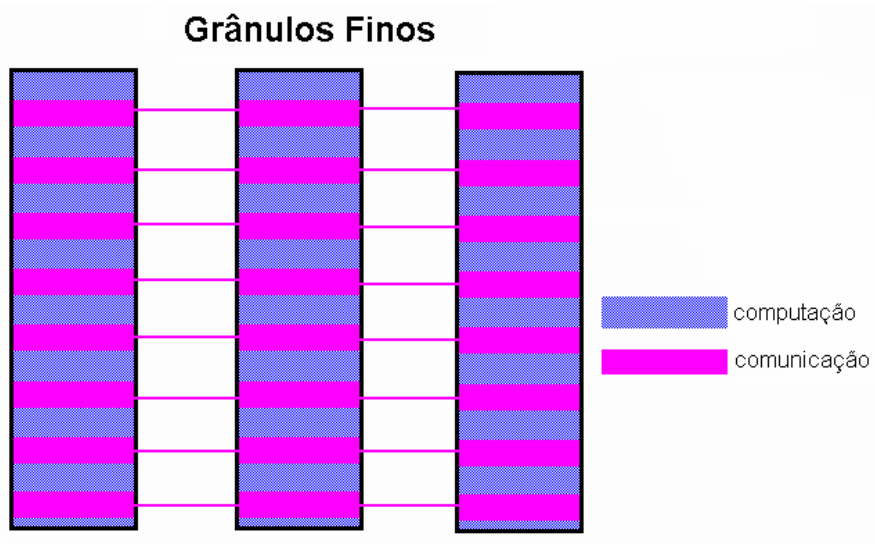


Figura 2.13 - Comunicação x Computação para Grânulos Finos.
Fonte: MPCC (1997).

2.3.3.3.2 - Grânulos Grosseiros

Grânulos grosseiros (*coarse-grained*) - alta tarefa computacional e pouca comunicação. A Figura 2.14 exemplifica a discussão.

- Alta computação constituída por um grande número de instruções entre os pontos de sincronização e comunicação;
- Alta computação em razão da comunicação;
- Mais oportunidade para melhorar o desempenho;
- Difícil para efetivar o balanceamento da carga.

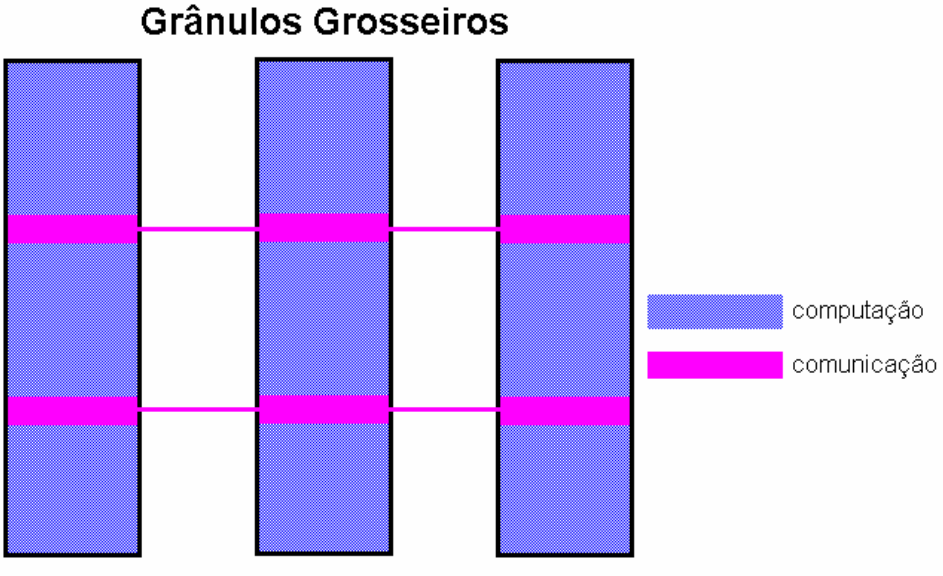


Fig.2.14 - Comunicação x Computação para Grânulos Grosseiros.
Fonte: MPCC (1997).

Uma granularidade eficiente, na maioria das vezes, depende do algoritmo e do ambiente de hardware. Na maioria dos casos o overhead associado com comunicação e sincronização é alto com relação à velocidade de execução. Portanto é vantajoso ter grânulos grosseiros. No entanto, a análise da aplicação é que vai dizer como "granularizar" o problema.

2.3.3.4 - Dependência dos Dados

Uma dependência existe, quando existe mais de um emprego de uma variável para mesma posição de armazenamento. Freqüentemente a dependência de dados inibe a execução paralela (LILJA, 1994).

2.3.3.5 - Comunicação e Largura da Banda

Para alguns problemas, aumentar o número de processadores não significa somente reduzir o tempo de execução de uma tarefa, mas também pode significar aumento no tempo de comunicação.

O tempo necessário para comunicação é dependente de dados parâmetros do sistema de comunicação (Equação 2.15). Por exemplo, o tempo (t) requerido para enviar W palavras entre 2 processadores quaisquer pode ser modelado por:

$$t = L + W/B \qquad (2.15)$$

onde L = latência e B= *hardware bitstream* por segundo

O modelo de comunicação também afeta a computação com relação à comunicação.

2.3.3.6 - Entrada e Saída (*Input/Output*)

Operações de Entrada/Saída (*Input/Output*) são geralmente conhecidas como inibidoras de paralelismo. Sistemas paralelos de Entrada/Saída são até agora objeto de pesquisa.

Num ambiente onde todos os processadores vêem o mesmo espaço de arquivo, operações de escrita irão resultar em arquivos reescritos. Já as operações de leitura serão afetadas pela habilidade do servidor de arquivo para manipular várias leituras ao mesmo tempo. Entrada/Saída, se conduzido sobre a rede (não local), pode causar um perigoso gargalo.

Como opções para tentar amenizar os problemas relacionados com Entrada/Saída tem-se:

Reduzir Entrada/Saída total o máximo possível;

- Confinar Entrada/Saída para uma parte serial específica do trabalho. Por exemplo, a tarefa 1 pode ler um arquivo de entrada e então comunicar o

requerido dado para outras tarefas. Do mesmo modo, a tarefa 1 pode realizar operações de escrita após o recebimento dos dados requeridos de todas outras tarefas;

- Criar único nome de arquivo para cada arquivo de Entrada/Saída da tarefa;
- Para sistemas de memória distribuída com arquivos de espaço compartilhados, realizar Entrada/Saída em área local, não em arquivo de espaço compartilhado. Por exemplo, cada processador pode ter um arquivo de espaço temporário que possa usar. Isso é muito mais eficiente que realizar Entrada/Saída sobre a rede para um diretório "*home*".

2.3.3.7 - Configuração do Equipamento

Para uma ótima abordagem de desempenho paralelo, o algoritmo deve ser elaborado para a arquitetura e configuração do equipamento que irá executá-lo. O conhecimento de itens como o número de processadores, tamanho da memória, cache e sistema de carga são imprescindíveis. Falhas no equipamento, falhas nas tarefas, ponto de verificação e reinicio do equipamento são de inteira responsabilidade do programador.

2.3.3.8 - *Dead Lock*

O *deadlock* descreve uma condição onde dois ou mais processos estão esperando por um evento ou comunicação de um dos outros processos. Um exemplo é demonstrado para dois processos que são programados para ler/receber de outro antes de escrever/enviar.

Exemplo: A Tabela 2.2 esboça um exemplo.

Tabela 2.2 - Exemplo de *Deadlock*

TAREFA 1	TAREFA 2
X = 4 SOURCE = TAREFA 2 RECEIVE (SOURCE,Y) DEST = TAREFA 2 SEND (DEST,X) Z = X + Y	Y = 8 SOURCE = TAREFA 1 RECEIVE (SOURCE,X) DEST = TAREFA 1 SEND (DEST,Y) Z = X + Y

Cabe também ao usuário depurar, monitorar e analisar a execução dos programas paralelos com o objetivo de melhorar o desempenho. Mas, essas tarefas são mais difíceis em programas paralelos do que em programas seriais. O usuário pode usar ferramentas auxiliares, mas muito trabalho ainda tem que ser feito na mão.

Referências bibliográficas como Bader et al. (1995), Baglietto et al. (1996), Barder e Jaja (1997) e Singh (1993) descrevem exemplos de análise de desempenho, as quais empregam metodologias diferenciadas.

2.3.4 - Computação de Alto Desempenho para Processamento de Imagens

Operações de processamento de imagens são usualmente de uma extensiva computação, devido ao grande número de dados a serem processados e a complexidade das operações elementares. A complexidade geralmente é da ordem $O(N^2)$ para imagem de dimensão N x N e para operações relativamente simples.

Baseado na velocidade necessária, a complexidade computacional e o volume de dados, podem alcançar facilmente os limites dos computadores quando se fala em processamento digital de imagens. Processamento paralelo é visto como uma das possíveis soluções para um rápido processamento digital de imagens, e o crescimento

acentuado do número de máquinas dedicadas a tal filosofia estão expandindo seu emprego (PITAS, 1993).

O processamento digital de imagens envolve tarefas naturalmente paralelas, devido ao tamanho e a regularidade das estruturas dos dados envolvidos, as quais tipicamente consistem de grandes matrizes bidimensionais de elementos (*pixels*) e por causa das características dos algoritmos usados, tipicamente consistindo de execuções repetidas (laços) do mesmo conjunto de operações em todos os pixels e da combinação de cada pixel com um pequeno conjunto de pixels localizados a uma curta distância (BAGLIETTO, 1996).

A execução repetida do mesmo conjunto de operações (laço) é o ponto mais crítico quando se trata da fase de segmentação de uma imagem. Portanto, os laços podem ser o ponto de partida para se tentar reduzir o tempo total de processamento digital de uma imagem, pois pode-se optar pelo particionamento da cena e a conseqüente distribuição das partes em diferentes processadores, os quais analisariam suas partes concorrentemente.

2.3.4.1 - Explorando o Paralelismo dos Laços

Limitando a extração do paralelismo para um bloco básico consegue-se acelerar o limite máximo da velocidade somente de duas a quatro vezes em todos os tipos de programas. Contudo, se os limites dos blocos básicos podem ser ignorados, o paralelismo de todo programa ficará disponível para exploração. Experiências simulando este caso ideal mostram que programas aplicados na engenharia e na área científica têm alto nível de paralelismo inerente (PITAS, 1993).

Uma eficiente abordagem para extrair este potencial paralelo é concentrar-se na exploração do paralelismo nos laços do código. O corpo de um laço pode ser executado várias vezes, os laços freqüentemente compreendem uma grande parte do paralelismo nos programas.

Pode-se explorar dois tipos de paralelismo nos laços, os com ciclos de dependência gráfica e os sem dependência no cruzamento das interações. Ambos podem ser executados em arquiteturas que exploram o paralelismo de grânulos finos (*fine-grained*) ou as que exploram grânulos grosseiros (*coarse-grained*) (LILJA, 1994).

2.3.4.2 - Formas de Paralelismo em Processamento de Imagem

Segundo (PITAS, 1993), os seguintes tipos de paralelismo podem ser identificados em processamento de imagem:

- Paralelismo geométrico;
- Paralelismo de vizinhança;
- Paralelismo de parte do elemento (*pixel-bit*);
- Paralelismo de operador.

As imagens digitais são usualmente amostradas em uma grade retangular e são armazenadas em uma matriz bidimensional. Portanto, elas possuem um inerente paralelismo geométrico. O paralelismo pode ser explorado usando uma grande matriz bidimensional de processadores, talvez um por elemento de imagem. O principal problema encontrado em cada solução é o efeito de borda que é criado devido a divisão da imagem em muitos processadores (PITAS, 1993).

Muitos algoritmos de processamento de imagem são essencialmente operações de vizinhança local da forma apresentada na Equação 2.16:

$$Yij = F\,(Xi+r,\ j+s) \quad (r,s) \in A \tag{2.16}$$

onde Xij, Yij são a imagem de entrada e a imagem de saída respectivamente. F é um operador (linear ou não linear) e A é a janela de processamento. A janela mais usada é a quadrada de tamanho 3x3. Paralelismo de vizinhança denota a execução paralela de

operações de vizinhança local. Neste caso o processador local pode ter acesso ou comunicação com o dado ou processador vizinho.

O paralelismo explora o fato de que uma imagem pode ser decomposta em b partes, onde b é o número de partes no elemento da imagem (usualmente b=1 ou 8). Vários operadores de processamento de imagem, particularmente os operadores lineares, podem ser realizados em cada parte independentemente. A aritmética que é realizada nas partes é chamada aritmética distribuída.

Dois tipos de operadores de paralelismo existem em operações de processamento digital de imagem: linha de produção paralela *(pipelining*) e a decomposição paralela. A *pipelining* é a mais usada e pode se expressada de acordo com a Equação 2.17:

$$Y = F(X) = Fn\ (Fn\text{-}1(...F2(F1(X))...)) \qquad (2.17)$$

onde X é a imagem inicial ou uma subregião, Y é a imagem de saída, F é um operador e Fi, i=1,...,n é a decomposição.

Uma variedade de arquitetura de computadores paralelos e técnicas de compilação têm sido propostas para explorar o paralelismo em diferentes granularidades, bem como um grande número de estratégias de planejamento diferentes podem ser usadas para determinar quais interações podem ser executadas por cada processador.

As arquiteturas paralelas diferem-se no tempo gasto para sincronização (*overhead*), limitação no planejamento de instruções, latência da memória, detalhes e técnicas de implementação. Tudo isso é importante para se determinar uma melhor abordagem para se explorar o paralelismo.

Os laços podem abranger uma grande parte do paralelismo dos programas de aplicação, pois as iterações do laço podem ser executadas várias vezes. Para explorar este paralelismo tem-se que ver além de um simples bloco básico ou uma simples iteração para operações independentes. A escolha de técnicas depende da arquitetura da máquina paralela e das características individuais de cada laço.

2.4 – Bibliotecas de Paralelização

As bibliotecas de paralelização são bibliotecas, tais como o PVM, P4, Linda, e MPI, as quais podem ser chamadas a partir de diversas linguagens. O programador deve paralelizar explicitamente o seu código e tratar os problemas de sincronização. Este grupo está dividido entre duas classes, os de memória compartilhada (*shared memory*) e os de troca de mensagem (*message passing*). Ambas as classes são implementadas acrescentando-se rotinas de uma biblioteca específica de paralelismo à linguagem seqüencial já existente, para a criação e coordenação de tarefas em paralelo.

Novas linguagens de alto nível com paralelismo implícito estão sendo desenvolvidas. Este enfoque requer que os programadores entendam um novo paradigma, não apenas uma nova sintaxe de uma linguagem. No próximo item será dado enfoque aos modelos de troca de mensagem, pois como já mencionado, este modelo é o usado nesta dissertação.

2.4.1 - Os Modelos de Troca de Mensagem

Troca de mensagem é um paradigma extremamente usado em certas classes de máquinas paralelas, especialmente aquelas com memória distribuída. Embora, existam muitas variações, o conceito básico do processamento da comunicação através de troca mensagens é muito conhecido.

Nos últimos dez anos, progressos substanciais têm sido feitos no sentido de se aplicar este paradigma. Mais recentemente, vários sistemas têm demonstrado que o paradigma de troca de mensagens pode ser eficientemente e portavelmente implementado.

As bibliotecas paralelas, baseadas em troca de mensagem, mais conhecidas e utilizadas são a P*arallel Virtual Machine* (PVM) e *Message Passing Interface* (MPI). Uma sucinta comparação entre as duas bibliotecas pode ser vista em Geist (1996) e em Fagg e Dongarra (1996).

Parallel Virtual Machine (PVM) - Possui como característica o conceito de uma "máquina virtual paralela", dentro da qual processadores recebem e enviam mensagens, com finalidades de obter um processamento global.

Message Passing Interface (MPI) – É uma biblioteca de troca de mensagem desenvolvida para ser o padrão em ambientes de memória distribuída. As plataformas alvo para o MPI, são ambientes de memória distribuída, máquinas paralelas massivas, clusters de estações de trabalho, redes heterogêneas.

Nosso trabalho se detém em explicar apenas o MPI, pois será a biblioteca empregada no desenvolvimento do mesmo.

2.4.1.1 - *Message Passing Interface*

Message Passing Interface (MPI) é um padrão de domínio público, implementado em diversas arquiteturas, que permite ao usuário integrar várias CPUs e ter as mesmas flexibilidades de se estar trabalhando em um grande computador paralelo.

O MPI foi projetado mediante a necessidade de se criar uma biblioteca padrão baseada na troca de mensagem e computação. O grupo de idealização do projeto era formado por mais de 80 pessoas de 40 organizações, representando vendedores de sistemas paralelos, usuários industriais, laboratórios de pesquisa nacionais e industriais, bem como universidades (SNIR, 1996).

O projeto foi baseado nas características mais atrativas dos diversos sistemas de troca de mensagem existentes na época. Estas características foram adaptadas uma a uma para se criar um sistema padrão. O grupo identificou também muitas deficiências nos sistemas existentes, em áreas como layout de dados complexos e suporte para modularidade e comunicação segura (SNIR, 1996).

As plataformas alvo para o MPI são os ambientes de memória distribuída, máquinas paralelas massivas, clusters de estações de trabalho (NOWs – *network of workstations*) e redes heterogêneas (GROPP,1998).

O MPI define um conjunto de rotinas para facilitar a comunicação (troca de dados e sincronização) entre processos paralelos e tem aproximadamente 125 funções para programação e ferramentas para se analisar o desempenho.

Ele possui rotinas para programas em FORTRAN 77 e ANSI C e pode ser usado também para FORTRAN 90 e C++. Os programas são compilados e linkados à biblioteca MPI. Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar o algoritmo utilizando chamadas aos comandos da biblioteca MPI. Existem duas divisões para MPI:

- MPI Básico: O MPI básico contém 6 funções básicas indispensáveis para o uso do programador, permitindo escrever um vasto número de programas em MPI.

- MPI Avançado: O MPI avançado contém cerca de 125 funções adicionais que acrescentam às funções básicas flexibilidade (permitindo tipos diferentes de dados), robustez (modo de comunicação *non-blocking*), eficiência (modo *ready*), modularidade através de grupos e comunicadores (*group* e *communicator*) e conveniência (comunicações coletivas, topologias). As funções avançadas fornecem uma funcionalidade que pode ser ignorada até serem necessárias.

A seguir são apresentadas as 6 funções básicas do MPI:

MPI_Init - Iniciando um processo MPI

Definição: Inicializa um processo MPI. Portanto, deve ser a primeira rotina a ser chamada por cada processo, pois estabelece o ambiente necessário para executar o MPI. Ela também sincroniza todos os processos na inicialização de uma aplicação MPI.

Sintaxe:

C

```
int MPI_Init (int *argc, char *argv[])
```

Fortran

```
call MPI_INIT (mpierr)
```

Parâmetros:

argc - Apontador para a quantidade. de parâmetros da linha de comando;

argv - Apontador para um vetor de *strings*

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
        int ret;
        ret = MPI_Init(&argc, &argv);
        if (ret < 0)
        {
                printf ("Nao foi possivel inicializar o processo MPI!\n");
                return;
        }
        else
        ...
}
```

MPI_Comm_rank - Identificando um processo no MPI

Definição: Identifica um processo MPI dentro de um determinado grupo. Retorna sempre um valor inteiro entre 0 e n-1, onde n é o número de processos.

Sintaxe:

C

int MPI_Comm_rank (MPI_Comm comm, int *rank)

Fortran

call MPI_COMM_RANK (comm, rank, mpierr)

Parâmetros:

comm - Comunicador do MPI;

rank - Variável inteira com o numero de identificação do processo.

Exemplo:

```
 #include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
      int mpierr, rank;
       mpierr = MPI_Init(&argc, &argv);
        if (mpierr < 0)
      {
              printf ("Nao foi possivel inicializar o processo MPI!\n");
               return;
      }
      else
      {
              MPI_Comm_rank(MPI_COMM_WORLD, &rank);
              printf ("Minha identificação no MPI e':%d\n",rank);
               ...
      }
 }
```

MPI_Comm_size - Contando processos no MPI

Definição: Retorna o número de processos dentro de um grupo.

Sintaxe:

C

int MPI_Comm_size (MPI_Comm comm, int *size)

Fortran

call MPI_COMM_SIZE (comm, size, mpierr)

Parâmetros:

comm

- Comunicador do MPI;

size

- Variável interna que retorna o numero de processos iniciados pelo MPI.

Exemplo:

```
 #include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
                int mpierr, rank, size;
                mpierr = MPI_Init(&argc, &argv);
                if (mpierr < 0)
       {
                printf ("Nao inicializou o processo MPI!\n");
                return;
       }
                else
       {
                MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                printf ("Minha identificação no MPI e':%d\n",rank);
                MPI_Comm_size(MPI_COMM_WORLD, &size);
                printf("O numero de processos e': %d\n",size);
                ...
       }
 }
```

MPI_Send: Enviando mensagens no MPI

Definição: Rotina básica para envio de mensagens no MPI, utiliza o modo de comunicação *blocking send* (envio bloqueante), o que traz maior segurança na transmissão da mensagem. Após o retorno, libera o *system buffer* e permite o acesso ao *application buffer*.

Sintaxe:

C

int MPI_Send (void *sndbuf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)

Fortran

call MPI_SEND (sndbuf, count, dtype, dest, tag, comm, mpierr)

Parâmetros:

Sndbuf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados serão enviados);

count - Número de elementos a serem enviados;

dtype - Tipo de dado

dest - Identificação do processo destino;

tag - Rótulo (label) da mensagem;

comm - MPI Communicator

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
 int mpierr, rank, size, i;
 char message[20];
 mpierr = MPI_Init(&argc, &argv);
 if (mpierr < 0)
  {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
```

```
  }
 else
  {
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      printf ("Minha identificação no MPI e':%d\n",rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      printf("O numero de processos e': %d\n",size);
      if (rank == 0)
      {
              strcpy(message,"Ola', Mundo!\n");
              for (i=1; i<size; i++)
              MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      }
      else
              ...
  }
}
```

MPI_Recv - Recebendo mensagens no MPI:

Definição: É uma rotina básica para recepção de mensagens no MPI, que utiliza o modo de comunicação *blocking receive* (recepção bloqueante), de forma que o processo espera até que a mensagem tenha sido recebida. Após o retorno, libera o *system buffer*, que pode ser então, novamente utilizado.

Sintaxe:

C

int MPI_Recv (void *recvbuf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status status)

Fortran

call MPI_RECV (recvbuf, count, dtype, source, tag, comm, status, mpierr)

Parâmetros:

sndbuf - Identificação do *buffer* (endereço inicial do *application buffer* - de onde os dados estão sendo enviados);

count - Número de elementos a serem recebidos;

dtype - Tipo de dado

source - Identificação do processo emissor;

tag - Rótulo (label) da mensagem;

comm - MPI Communicator

status - Vetor de informações envolvendo os parâmetros *source e tag*.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, tag, i;
  MPI_Status status;
  char message[20];
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr < 0)
   {
        printf ("Nao foi possivel inicializar o processo MPI!\n");
        return;
   }
  else
   {
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        printf ("Minha identificação no MPI e':%d\n",rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("O numero de processos e': %d\n",size);
        if (rank == 0)
        {
                strcpy(message,"Ola', Mundo!\n");
                for (i=1; i<size; i++)
                MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
        }
        else
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
        &status);
        printf("Mensagem do no' %d : %s\n", rank, message);
    ...
   }
 }
```

MPI_Finalize - Encerrando um processo no MPI:

Definição: Finaliza um processo MPI. Portanto deve ser a última rotina a ser chamada por cada processo. Sincroniza todos os processos na finalização de uma aplicação MPI.

Sintaxe:

C

int MPI_Finalize (void)

Fortran

call MPI_FINALIZE (mpierr)

Parâmetros:

(Nenhum)

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
        int mpierr, rank, size, tag, i;
        MPI_Status status;
        char message[20];
        mpierr = MPI_Init(&argc, &argv);
         if (mpierr < 0)
        {
                printf ("Nao foi possivel inicializar o processo MPI!\n");
         }
        else
        {
                MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                printf ("Minha identificação no MPI e':%d\n",rank);
                MPI_Comm_size(MPI_COMM_WORLD, &size);
                printf("O numero de processos e': %d\n",size);
                if (rank == 0)
                {
                        strcpy(message,"Ola', Mundo!\n");
                        for (i=1; i<size; i++)
                        MPI_Send(message, 13, MPI_CHAR, i, tag,
                        MPI_COMM_WORLD);
```

```
                }
                else
                MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
                &status);
                printf("Mensagem do no' %d : %s\n", rank, message);
                MPI_Finalize();
        }
}
```

Uma implementação de MPI pode variar a quantidade de funções, a forma de inicialização de processos, a quantidade de cada sistema de buffering, os códigos de retorno e erro, de acordo com os desenvolvedores de cada implementação. O MPI padrão não especifica todos os aspectos de implementação.

2.4.1.2 - Conceitos Básicos de MPI

A familiarização com algumas nomenclaturas particulares às bibliotecas de troca de mensagem é necessária para um melhor entendimento:

Processo: Por definição, cada programa em execução constitui um processo. Considerando um ambiente multi-processado, pode-se ter processos em inúmeros processadores.

Mensagem (*message*): É o conteúdo de uma comunicação, formado de duas partes:

a) Envelope: endereço (origem ou destino) e rota dos dados. O envelope é composto de três parâmetros:

   - Identificação dos processos (transmissor e receptor);

   - Rótulo da mensagem;

   - Grupo de processos (*Communicator*).

b) Dado: Informação que se deseja enviar ou receber. É representado por três argumentos:

- Endereço onde o dado se localiza;

- Número de elementos do dado na mensagem;

- Tipo do dado.

Rank: Todo o processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado. Essa identificação é representada por um número inteiro, começando de zero, até N-1, onde N é o número de processos. *Rank* é o identificador único do processo, utilizado para identificar o processo no envio (*send*) ou recebimento (*receive*) de uma mensagem.

Grupo: é um conjunto ordenado de N processos. Todo e qualquer *group* é associado a um *communicator* muitas vezes já predefinido como MPI_COMM_WORLD. Inicialmente, todos os processos são membros de um *group* com um *communicator*.

***Communicator*:** é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser contactados). O MPI utiliza esta combinação de grupo e contexto para garantir segurança na comunicação e evitar problemas no envio de mensagens entre os processos.

A maioria das rotinas de MPI exige que seja especificado um *communicator* como argumento. O MPI_COMM_WORLD é o comunicador predefinido que inclui todos os processos definidos pelo usuário numa aplicação MPI.

Application Buffer: é o endereço de memória, gerenciado pela aplicação, que armazena um dado que o processo necessita enviar ou receber.

System Buffer: é um endereço de memória reservado pelo sistema para armazenar mensagens. Dependendo do tipo de operação de envio/recebimento (*send/receive*), o dado no *buffer* da aplicação (*application buffer*) pode necessitar ser copiado de/para o

buffer do sistema (*system buffer*) através das rotinas *send buffer/receive buffer*. Neste caso a comunicação é assíncrona.

Blocking Communnication: em uma rotina de comunicação *blocking*, a finalização da chamada depende de certos eventos. Em uma rotina de envio de mensagem, o dado tem que ter sido enviado com sucesso, ou ter sido salvo no *buffer* do sistema (*system buffer*), indicando que o endereço do *buffer* da aplicação (*application buffer*) pode ser reutilizado.

Em uma rotina de recebimento de mensagem, o dado tem que ser armazenado no *buffer* da aplicação (*application buffer*), indicando que o dado pode ser utilizado.

Non-Blocking Communnication: em uma rotina de comunicação *non-blocking*, a finalização da chamada não espera qualquer evento que indique o fim ou sucesso da rotina. As rotinas *non-blocking communnication* não esperam pela cópia de mensagens do *buffe*r da aplicação (*application buffer*) para o *buffer* do sistema (*system buffer*), ou a indicação do recebimento de uma mensagem.

Overhead: é um desperdício de tempo que ocorre durante a execução dos processos. Os fatores que levam ao overhead são: comunicação, tempo em que a máquina fica ociosa (tempo *idle*) e computação extra. Existem os seguintes tipos de *overhead*:

(a) *System Overhead*: o tempo gasto pelo sistema na transferência dos dados para o processo destino.

(b) *Synchronization Overhead*: é o tempo gasto para a sincronização dos processos.

O MPI sempre apresenta um *overhead* de sincronização inicial, quando espera até que todas as máquinas estejam disponíveis para iniciar o processamento e inicializar os processos. O mesmo ocorre quando da conclusão: existe um *delay* até que todos os processos possam encerrar adequadamente e terminarem a execução.

CAPÍTULO 3

ESTRATÉGIA DA SOLUÇÃO DO PROBLEMA

3.1 – Alvo Principal: As Iterações

A principal meta deste trabalho é a redução do tempo de processamento da fase de segmentação apresentada no Capítulo 1, modificando o algoritmo, adaptando-o para sistemas paralelos com um padrão de comunicação entre processadores, visando otimizar e avaliar o processo, além de permitir a sua execução em diferentes arquiteturas paralelas, sem que isso implique em grandes modificações no código original. Isso É viável desenvolvendo-se as características paralelas inerentes ao algoritmo de crescimento de regiões, uma vez que tais características não estão exploradas na versão seqüencial, hoje implementada no SPRING.

Seja qual for a técnica automática empregada no tratamento de uma imagem, as iterações geralmente representam o ponto crítico quando se fala em tempo de processamento.

A opção pelo uso do modelo de troca de mensagens objetiva uma solução mais barata e menos dependente de hardware.

3.2 – Metodologia

3.2.1 - Estudo Detalhado de Algoritmos de Segmentação

Inicialmente, foi feito um estudo de alguns algoritmos de segmentação de imagens baseado em crescimento de regiões, objetivando conhecer suas fases e aprofundar os conhecimentos. O algoritmo de segmentação utilizado em Batista et al. (1996), esboçado no Capítulo 1 e ora implementado no SPRING, foi o principal alvo deste estudo.

3.2.2 - Compilação e Execução do Fonte Serial

Primeiramente, os códigos fonte da solução serial utilizada em Bins et al. (1996), desenvolvidos em linguagem C++, os programas fontes foram obtidos junto a Divisão de Processamento de Imagens (DPI) do INPE.

Devido à complexidade do sistema SPRING, software do qual o procedimento de segmentação em estudo é parte integrante, foi necessária a separação do módulo, para tornar mais simples e rápida a verificação da proposta de paralelização, pois nosso interesse é exclusivo na rotina de segmentação para este trabalho.

Tradicionalmente os programas de um computador são elaborados para serem executados em máquinas seriais, ou seja, para máquinas com somente um processador, executam uma instrução por vez e o tempo de execução depende de quão rápido a informação se movimenta pelo hardware.

Antes de iniciar uma minuciosa análise destes códigos, visando sua paralelização propriamente dita, o programa original foi compilado para a plataforma SP2 da Universidade do Vale do Paraíba (Figura 3.1), a qual utiliza sistema operacional AIX 4.1.4, compilador GCC 2.7.2, e também compilado para plataforma SUN do INPE a qual utiliza sistema operacional SunOs 5.5 e Solaris 2.7. A intenção disso é executar o programa em plataformas que além de suportarem programas seqüenciais, suportam também programas paralelos, possibilitando uma comparação de desempenho entre os sistemas.

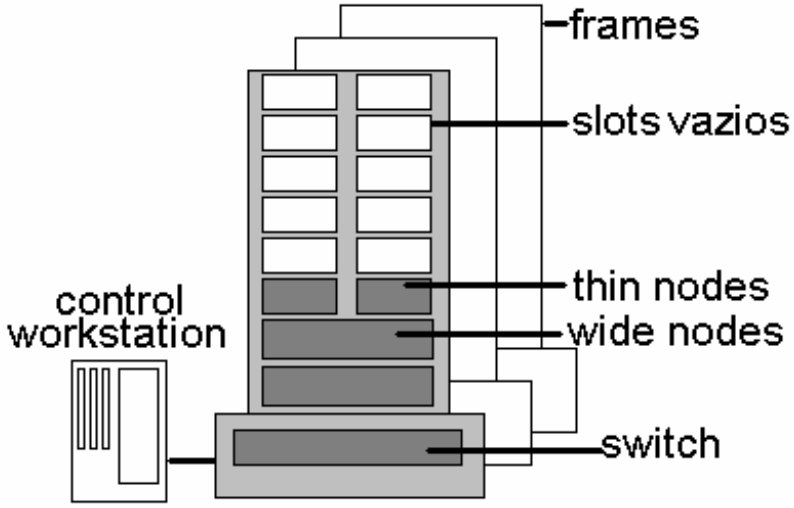


Fig. 3.1 - Estrutura do SP2

Já o ambiente paralelo do INPE é formado por cinco estações de trabalho Sun Microsystem, denominadas soyuz.dpi.inpe.br, sputnik.dpi.inpe.br, meteosat.dpi.inpe.br, cygnus.dpi.inpe.br, prodes.dpi.inpe.br.

Alguns ajustes foram necessários para que o código fosse compilado e ligado, pois existem alguns comandos/funções não padronizados nas diferentes versões dos compiladores C++ das plataformas usadas.

Apesar do esforço despedido no ajuste do código para executar no ambiente SP2, ao longo do desenvolvimento de uma versão paralela, abandonou-se os teste em tal ambiente, pois a biblioteca de paralelização PE 2.1, versão do MPI da IBM, disponível, não suporta código C++ e este projeto não conseguiu recursos financeiros para a aquisição desta versão.

Ao final do trabalho, optou-se por executar o programa serial em um PC InfoWay – Itautec, 333 MHz, 64 Mb de memória RAM, baseado em sistema operacional Linux 4.2, compilador GNU, pois com isso Ter-se-ia uma uniformidade dos processadores para se comparar coerentemente a versão serial e a paralela, além de se proporcionar um código C++ aderente ao padrão ANSI, o que facilita muito o porte para qualquer outro ambiente que suporte MPI. A plataforma PC também está mais próxima do uso efetivo

dos usuários do SPRING, visto que 90% desses usuários trabalham em tal plataforma baseada em sistema operacional, Microsoft Windows, NT ou Linux.

A migração do código fonte da plataforma SUN para PC foi direta, não necessitando de nenhum ajuste na programação, isso prova a boa portabilidade do MPI.

3.2.3 - Tomada de Tempo para Área Piloto

Após a compilação do programa seqüencial, o mesmo foi executado na plataforma PC, visando à tomada de tempo. Isto significa que foram medidos detalhadamente os tempos de execução do programa seqüencial, utilizando cinco imagens testes da região amazônica. As imagens utilizadas para testes, são imagens tipo *raw* obtidas pelo satélite TM-Landsat, são compostas pelas bandas 3, 4 e 5 e possuem tamanhos que variam de 256 linhas por 256 colunas até 4096 linhas por 4096 colunas, como mostra a Figura 3.2.

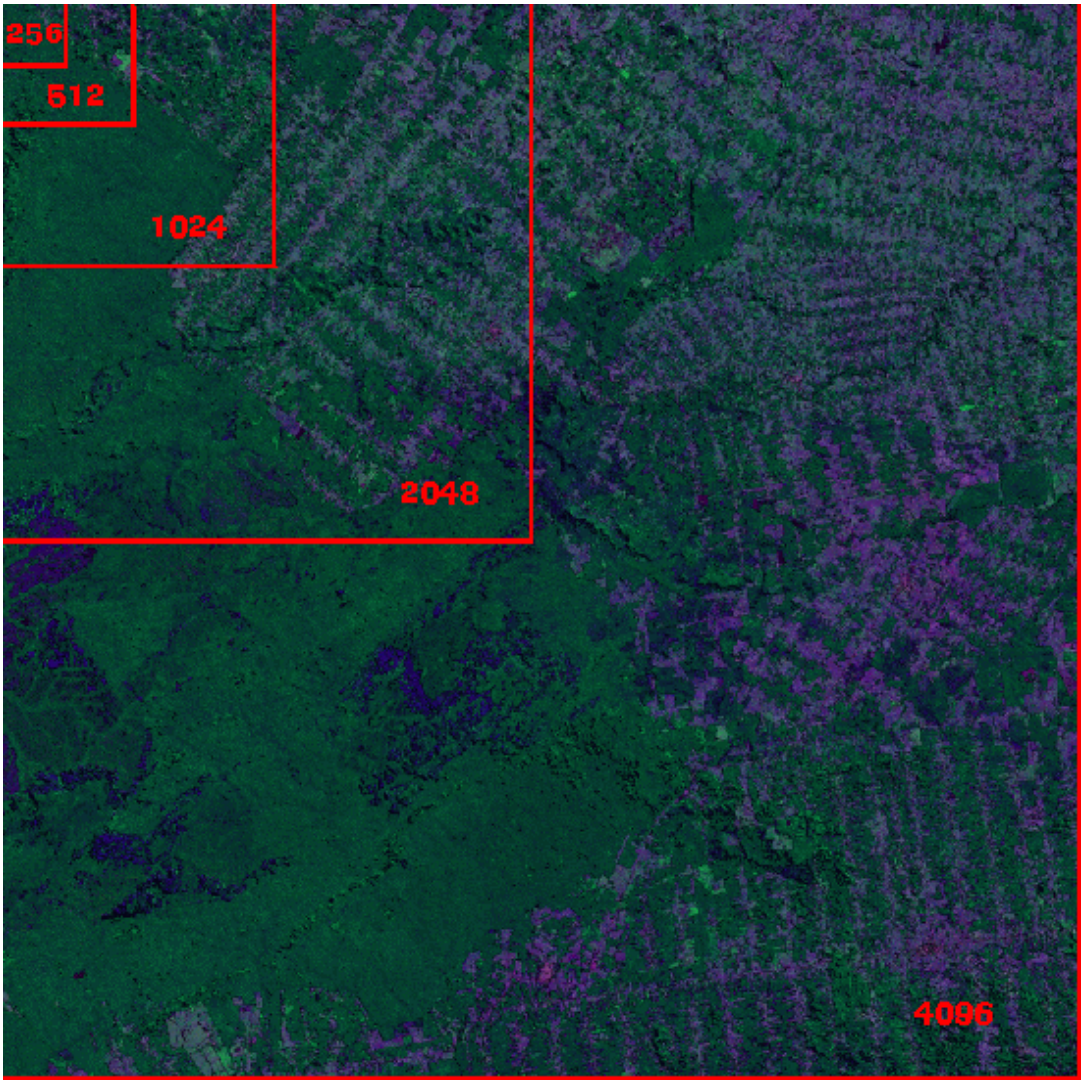


Fig. 3.2 – Recorte das Imagens Testes (Imagem Landsat-TM 5 Órbita/Ponto 231/68 Bandas 3,4 e 5).

Os resultados destes testes estão analisados no próximo Capítulo.

3.2.4 - Estudo do Programa Fonte

O programa fonte na versão seqüencial é composto por sete arquivos (rgrow.cc, cell.cc, image.cc, dinvect.cc, image.hpp, cell.hpp, dinvect.hpp). Estes arquivos foram minuciosamente estudados para que se entender a lógica empregada pelos

programadores desta versão. Nesta fase, esbarrou-se num defeito comum a programadores, a falta de documentação do código. A maior dificuldade foi entender as rotinas e a função atribuída a cada variável. O programa é baseado numa lista de células, que por sua vez armazena características importantes de cada pixel que compõe a imagem e associadas a esta lista existem outras listas que apontam para a vizinhança de cada pixel.

Durante a fase se segmentação mais uma lista de células é gerada, só que desta vez ela é composta pelas regiões rotuladas baseadas na imagem segmentada final.

3.2.4.1 - Análise dos Dados e Tarefas

Para se decompor um programa em pequenas tarefas a serem executadas em paralelo é necessária uma prévia e minuciosa análise do código fonte. Neste caso foi projetada a decomposição dos dados ou decomposição de domínio, onde os dados (neste caso a imagem) são decompostos em grupos que serão distribuídos por entre múltiplos processadores que executarão, simultaneamente, um mesmo programa e essa técnica é perfeita para resolução de imensas matrizes, como é o caso do algoritmo de segmentação (modelo SPMD).

Portanto, a matriz que compõe a imagem teste é particionada, e cada processador irá receber N partes de acordo com o número de processadores disponíveis.

3.2.4.2 - Análise dos Trechos Paralelizáveis

Definida a clara necessidade da divisão da imagem em porções menores visando agilizar o tempo de segmentação da mesma, localizou-se no código, trechos que deveriam ser modificados para atenderem corretamente à versão paralela. O principal alvo foi um laço do arquivo cell.cc, o qual aplicava o algoritmo de crescimento de regiões pixel a pixel de acordo com o número de linhas e colunas que compunham a imagem. Muitas mudanças foram necessárias para quebrar a dependência entre os métodos e funções da versão seqüencial.

3.2.4.3 – Pseudo-Código do Fonte Serial

```
Inicio
    Leia número de linhas e número de colunas
    Leia limiar de área
    Leia limiar de similaridade
    Leia número de bandas
    Faça J de 1 até  número de bandas
        Leia imagem (J)
    Fim Faça
    Nlin = número de linhas /128
    Ncol = número de colunas/128
    Número de janelas = Nlin x Ncol
    Faça I de 1 até Número de Janelas
        Faça p de 1 até 128*128
            Se média espectral do pixel p for semelhante a média espectral do
            pixel p +1 então
                Agregue os pixels em questão
            Fim Se
            Se área < 5  então
                Agregue esta região a região vizinha espectralmente
                mais próxima
            Fim Se
        Fim Faça
    Fim Faça
    Faça do inicio até o final da lista de regiões
        Se área < limiar de área  então
            Agregue esta região à região vizinha espectralmente
            mais próxima
        Fim Se
    Fim Faça
Fim
```

3.2.5 - Proposta de Implementação Usando MPI

Esta dissertação limita-se ao estudo da paralelização de laços mais internos, em nível de granularidade grosseira. Nesta granularidade, o paralelismo é explorado através da execução simultânea de iterações distintas em processadores distintos, onde cada iteração é executada seqüencialmente em um processador. A Figura abaixo esboça o esquema de paralelização de um programa:

Trecho paralelo

<table>
<tr><td rowspan="4">Trecho seqüencial</td><td>P1 seqüencial</td><td rowspan="3">Trecho seqüencial</td><td rowspan="3">. . .</td></tr>
<tr><td>P2 seqüencial</td></tr>
<tr><td>. . .</td></tr>
<tr><td>P_{nproc} seqüencial</td><td></td><td></td></tr>
</table>

Fluxo de execução TM

Fig. 3.3 – Esquema de Execução Paralela.

O trecho paralelo representa um laço mais interno. Este laço é dividido em partes seqüenciais, que são executadas simultaneamente por processadores distintos.

A Figura 3.4 ilustra o modelo de divisão da imagem.

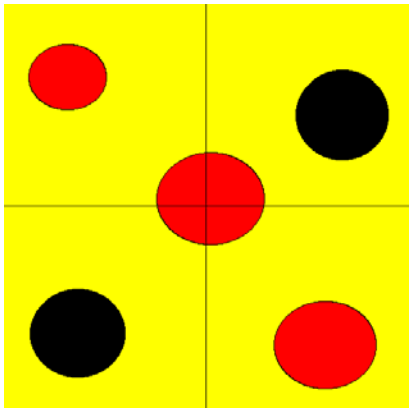


Fig.3.4 – Ilustração do Particionamento da Imagem

Baseado no programa fonte serial optou-se por dividir a imagem em porções de 128 linhas por 128 colunas. Dependendo do número de processadores invocados pelo usuário cada processador irá receber N processos. Por exemplo, para a imagem acima, 256 x 256, o programa particionará a imagem em 4 partes iguais de 128x128. Estas quatro porções deverão ser processadas pelos processadores disponíveis. Um desses processadores (denominado “mestre”) se encarregará de realizar a divisão do trabalho e coordenar a execução dos demais processadores (denominados “clientes” ou “escravos”). Portanto se o usuário tiver disponível apenas 2 processadores, um deles será o mestre e o outro será o escravo. O processador escravo irá receber a janela 0 da imagem, processá-la e devolve-la ao mestre junto com uma lista de regiões rotuladas. Logo após receber a janela 0 processada, o mestre irá enviar a janela 1 para o escravo e isto deverá ocorrer até que a janela 3 tenha sido processada e devolvida ao mestre. Ao receber cada janela processada e sua lista de regiões, o mestre deverá posicioná-la corretamente na matriz resultante e atualizar a lista de regiões adicionando a lista ora recebida. No caso do emprego de apenas um processador escravo, fica simples a ordenação dos dados processados, mas quando se trata de vários processadores escravos o processador mestre é o responsável pela ordenação dos dados e a concatenação das janelas.

Supondo que a mesma imagem seja empregada, mas agora o usuário tenha cinco processadores disponíveis, a redução do tempo de execução é facilmente obtida, pois quatro processadores serão empregados como escravos e um será o mestre. Cada processador escravo receberá uma janela diferente da imagem, irá processá-la e a devolverá para o processador mestre que se encarregará de concatená-la à janela vizinha.

Portanto, o número de janelas enviadas para cada processador depende diretamente do número de processadores disponíveis e do tamanho da imagem. Por exemplo, se existe cinco processadores disponíveis e uma imagem 2048 x 2048, totalizando 256 janelas. Têm-se 1 processador mestre e quatro escravos e cada escravo irá processar 64 janelas, uma a uma. Se existem oito processadores escravos, cada um irá receber 32 janelas.

A fase de concatenação das janelas recebidas e ordenadas pelo processador mestre baseia-se na lista de regiões criada durante a execução do algoritmo de crescimento de regiões por cada processador escravo. Esta lista é composta pelos endereços das regiões, endereços de seus vizinhos, número de bandas, média espectral, retângulo envolvente e número de pixels que compõem a região.

Associada aos vizinhos tem-se uma outra lista composta pelos vizinhos mais próximos de cada região, ou seja, vizinhos que além de terem fronteira também tem características muito semelhantes à da região em questão. O processo de concatenação é responsável pela união das janelas e a junção de regiões consideradas vizinhas mais próximas.

A biblioteca de paralelização MPI é a responsável pelo envio e recebimento dos dados entre os diferentes processadores. Para tanto, empregou-se a comunicação ponto a ponto assíncrona, onde a mensagem é enviada para um processador escravo, mas o mestre não espera a confirmação de recebimento para enviar a próxima mensagem a outro processador, como ilustrado na Figura 3.5.

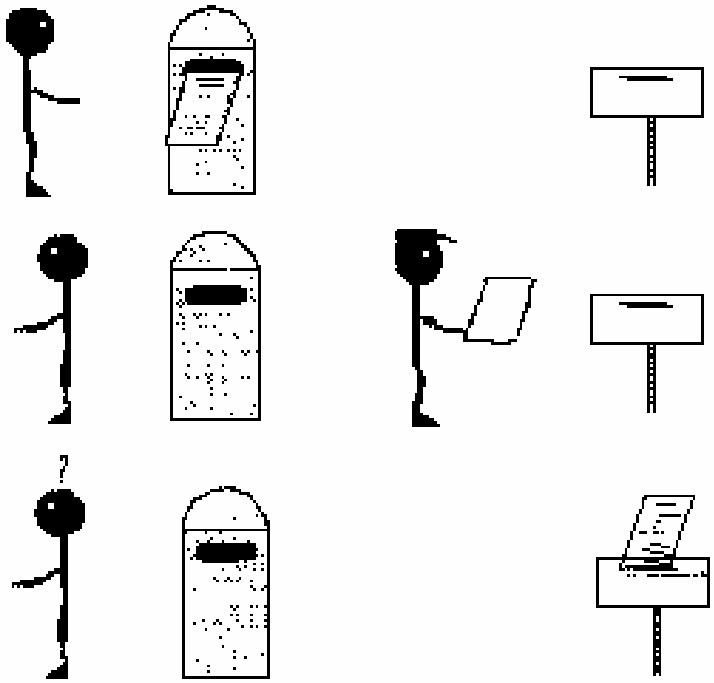


Fig. 3.5 – Troca de Mensagem Assíncrona

3.2.5.1 – Pseudo_Código da Versão Paralela

```
// ************ PROCESSADOR MESTRE  ************//

Inicio
        Inicializa Número de Processos
        Size = número de processos inicializados
        Rank = número do processo corrente

        Se rank = processador mestre  então
                Leia número de linhas,
                Leia número de colunas
                Leia limiar de área
                Leia limiar de similaridade
                Leia número de bandas
                Leia imagem
                Calcule total de janelas de 128x128
                Envie o número de janelas para os processadores clientes
                Calcule quantas janelas cada processador cliente processará
                Recorte a imagem
                Envie as 3 bandas das porções recortadas para o processador cliente
                Aguarde resposta dos clientes
                // Desvie a execução para os processadores clientes
                Receba lista dos clientes
                Receba imagem rotulada dos clientes
                Posicione imagem rotulada na matriz
                Concatene as listas
                Costure as  imagens
```

```
// *************** PROCESSADORES CLIENTES **************//
        Senão
                Inicialize imagem na memória
                Receba do processador mestre  as 3 bandas da imagem
                Faça para todos os pixels que compõem a porção da imagem recebida
                        Se média espectral do pixel p >= média espectral  do pixel
                        p+1  ou média espectral do pixel 1 <= média espectral  do
                        pixel  p +1 então
                                Agregue os pixels
                        Fim Se
                        Se área < limiar de área  então
                                Agregue esta região a região vizinha
                                espectralmente mais próxima
                        Fim Se
                        Envia imagem rotulada e lista para Mestre
                Fim Faça
        Fim Se
Fim
```

A Figura a seguir ilustra o pseudo-código da versão paralela.

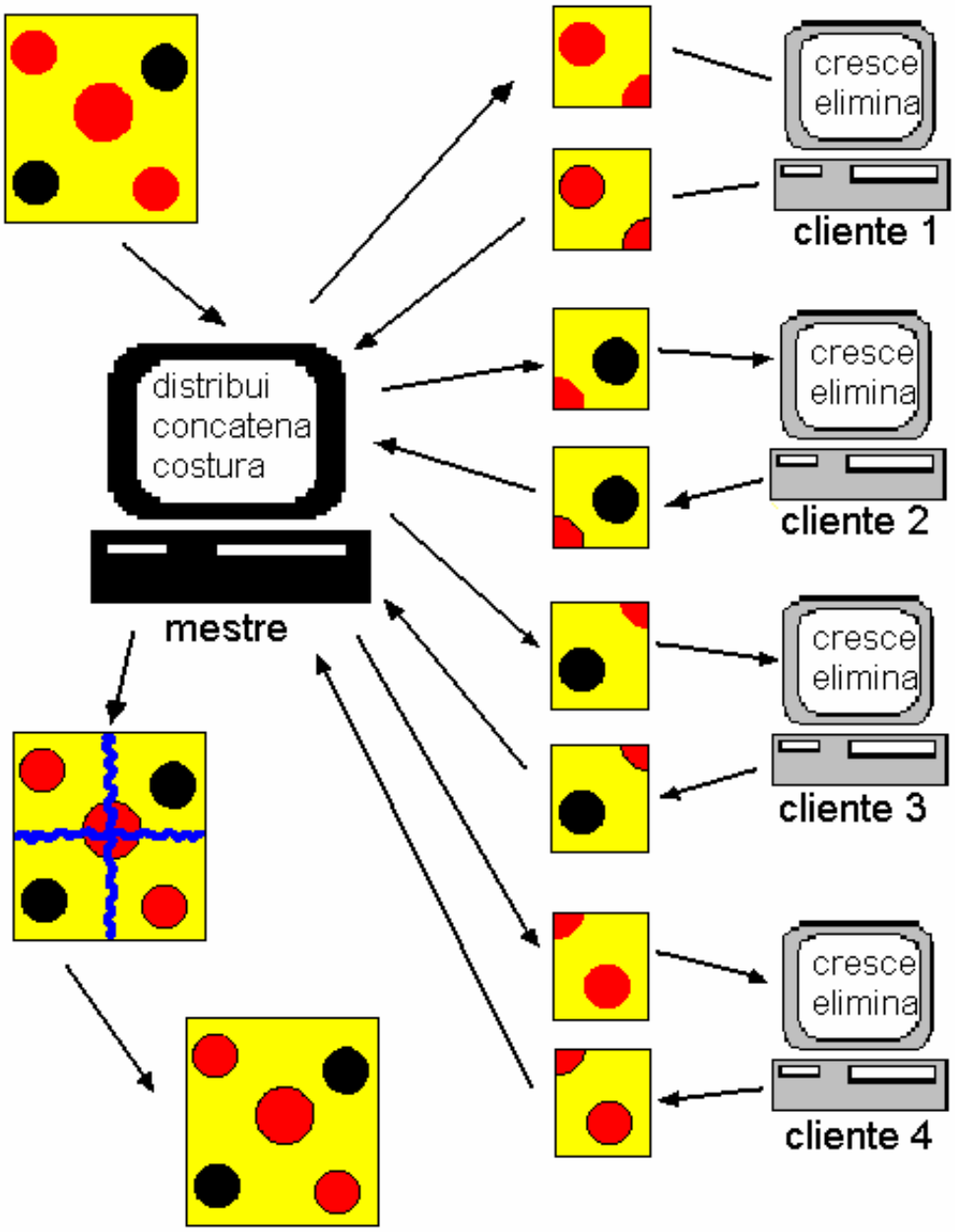


Fig. 3.6 – Esquema da Versão Paralela

3.2.6 - Tomadas de Tempo Variando o Número de Processadores MPI

A versão paralela do código foi desenvolvida num ambiente formado por cinco estações de trabalho SUN da Divisão de Processamento de Imagens – DPI/INPE, baseadas em

sistemas operacionais SunOS e Solaris, mas devido a discrepâncias na configuração de tais estações e a impossível exclusividade na utilização de tais máquinas, optou-se por realizar os testes num cluster de dez PCs baseados no sistema operacional LINUX com a biblioteca de paralelização mpich 1.1.2 do laboratório da Univap. Estes PC´s InfoWay Itautec tem processador Pentium II 333 MHz, 64 Mb RAM, 128 Mb de memória virtual. Os testes variam o tamanho da imagem e o número de processadores empregados. O número de processadores varia de dois até nove processadores, onde sempre um deles é empregado como mestre e os demais como escravos.

Os resultados dos tempos dos testes realizados poderão ser analisados no próximo Capítulo.

3.2.7 - Comparação dos Resultados com Área Piloto

Evidentemente que além da preocupação com a redução do tempo de processamento através da paralelização do código, tem-se que se preocupar com a veracidade dos dados gerados. Para tanto, além da imagem segmentada, o sistema apresenta o número de regiões contabilizada pelo processo de segmentação. A diferença entre o número de regiões gerado pela versão seqüencial e a paralela é facilmente explicada, pois o processo de crescimento de regiões baseia-se na comparação da média espectral do pixel semente com a média espectral de seus vizinhos. Como no caso paralelo fez-se uma pequena mudança algorítmica, pode-se encontrar uma classificação final diferente.

3.2.8 – Análise de Desempenho

Como citado no Capítulo 2, a análise de desempenho da versão paralela é baseada nas fórmulas de ganho (*speedup* – Equação 2.18) e eficiência (Equação 2.19).

$$Sp = Tseq/Tp \qquad (2.18)$$

Onde:

Tseq - tempo consumido por uma máquina seqüencial

Tp - tempo consumido por uma máquina paralela

$$E = Sp/P \tag{2.19}$$

Onde:

P – número de processadores empregados no processamento.

Os resultados da análise de desempenho são apresentados no Capítulo 4.

CAPÍTULO 4

TESTES E RESULTADOS

4.1 – Plano Geral de Testes

As imagens selecionadas para o teste, citadas e apresentadas no capítulo anterior, tiveram sua seleção baseada na diversificação de classes apresentadas e na aplicação, já que o desflorestamento da Amazônia foi a principal motivação para este trabalho, buscando gerar mapas confiáveis e com maior agilidade. Também foram selecionadas as bandas espectrais das referidas imagens que proporcionam, na composição colorida, uma maior separabilidade de classes de desflorestamento.

Iniciou-se os experimentos pela execução da versão serial do programa em um PC com a mesma configuração dos PC´s que formam o cluster de PC´s onde foi executada a versão paralela. Visando comparar a versão paralela com a versão serial, após a execução do programa foi anotado o tempo total de execução, o número de regiões segmentadas e foi impressa a imagem rotulada gerada no processo.

Para a execução da versão serial variou-se o tamanho da imagem, utilizando imagens de 256 linhas por 256 colunas até 4096 linhas x 4096 colunas.

Constitui a segunda fase dos experimentos a execução da versão paralela variando o número de processadores e o tamanho da imagem teste. As imagens utilizadas são as mesmas utilizadas na versão serial. O programa foi executado em um ambiente Linux, formado por dez PC´s com a biblioteca de paralelização Mpich 1.1.2.

Como no experimento serial, foi anotado o tempo total de processamento, o número de regiões segmentadas e foi gerada a impressão da imagem rotulada.

4.2 – Descrição dos Testes

A primeira providência antes de iniciar o teste foi garantir que não havia nenhum outro usuário compartilhando os recursos das máquinas utilizadas durante o processamento.

Como apresentado na Figura 3.2 as imagens empregadas nos testes fazem parte de uma mesma cena e quanto maior o corte, maior é a complexidade do processamento.

4.2.1 – Execução do Código Serial

O código serial foi executado na máquina tico.fcc.univap.br (200.136.180.11), sistema operacional Conectiva Linux 3.0 (guarani), Kernel 2.0.36 do laboratório 6 da Universidade do Vale do Paraíba. A Tabela abaixo descreve as características principais da máquina empregada no experimento serial.

Tabela 4.1 – Configuração do Hardware Utilizado nos Testes da Versão Serial.

Marca	**Modelo**	**Clock**	**RAM**	**Memória Virtual**	**Processador**
Itautec	InfoWay	333 MHz	64 Mb	128 Mb	Pentium II

O programa foi disparado, mas antes de iniciar a contagem do tempo, alguns parâmetros foram passados via teclado para o programa. Os parâmetros necessários são o número de linhas e colunas que compõem a imagem, número de bandas, nome das imagens, limiar de similaridade, limiar de área e o nome que será dado a imagem rotulada gerada no final do processamento.

É importante salientar que para todos os experimentos o limiar de similaridade foi igual a 30 e o número de bandas igual a 3. Outro parâmetro instanciado pelo usuário é o limiar de área. Nos testes para o código serial os desenvolvedores do sistema utilizavam

2 limiares diferentes. O primeiro acionado dentro do laço principal é igual a 5 e o limiar empregado na rotina de junção final das regiões é igual a 25.

O programa serial foi executado para as imagens Landsat-TM, denominadas C256, C512, C1024, C2048 e C4096, pertencentes a órbita/ponto 231/68 da região Amazônica, bandas 3, 4 e 5.

Os parâmetros de entrada são apresentados nas Tabelas 4.2, 4.3, 4.4, 4.5 e 4.6 e as imagens resultantes nas Figuras 4.1, 4.2, 4.3, 4.4 e 4.5.

Tabela 4.2 – Dados de Entrada do Experimento Número 1.

Número de		**Banda**			**Liminar Similaridade**	**Limiar Área**	**Imagem Rotulada**
Lin	**Col**	**3**	**4**	**5**			
256	256	C256r	C256g	C256b	30	5 e 25	C256serial

Após **56 segundos** de processamento, o programa finalizou totalizando 74 regiões e gerando a imagem rotulada abaixo.

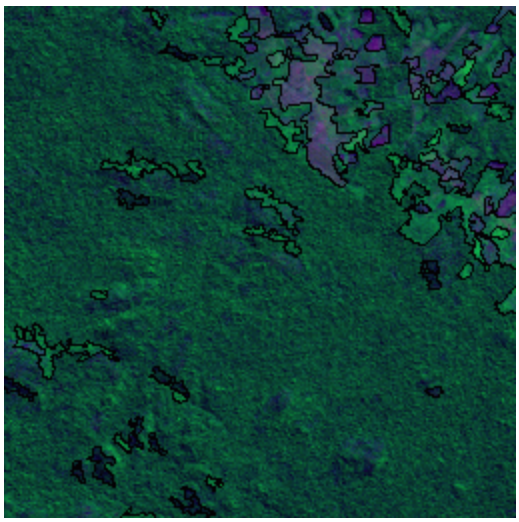


Fig. 4.1. – Imagem Rotulada C256serial

Tabela 4.3 – Dados de Entrada do Experimento Número 2

Número de		**Banda**			**Liminar Similaridade**	**Limiar Área**	**Imagem Rotulada**
Lin	**Col**	**3**	**4**	**5**			
512	512	C512r	C512g	C512b	30	5 e 25	C512serial

Após **4 minutos e 3 segundos** de processamento, o programa finalizou totalizando **379 regiões** e gerando a imagem rotulada abaixo.

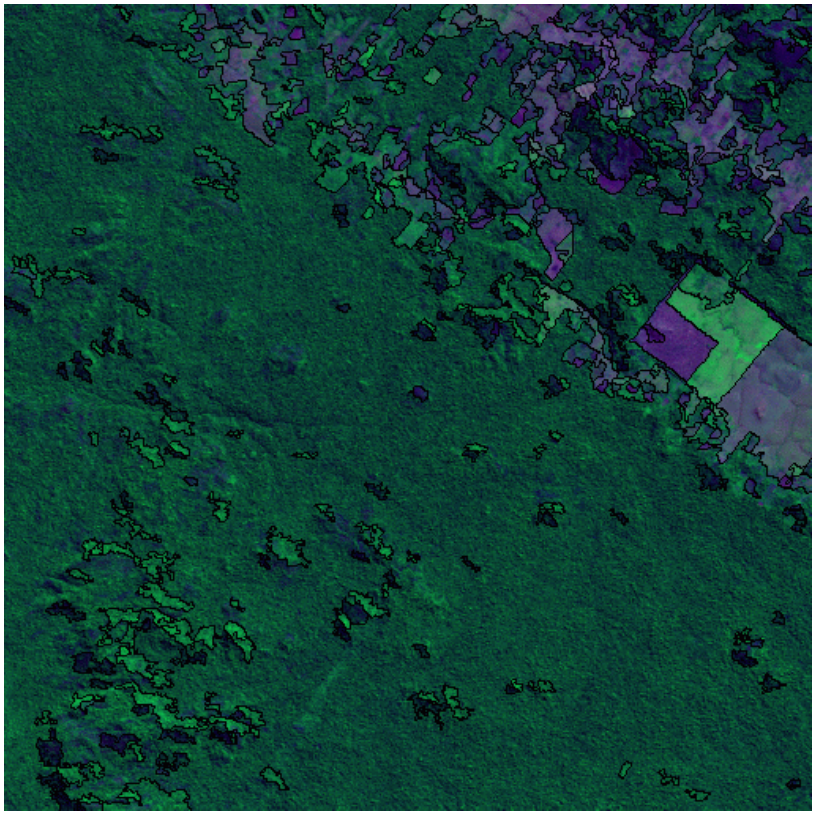


Fig. 4.2. – Imagem Rotulada C512serial

Tabela 4.4 – Dados de Entrada do Experimento Número 3

Número de		**Banda**			**Liminar Similaridade**	**Limiar Área**	**Imagem Rotulada**
Lin	**Col**	**3**	**4**	**5**			
1024	1024	C1024r	C1024g	C1024b	30	5 e 25	C1024serial

Após **30 minutos** de processamento, o programa finalizou totalizando **1494 regiões** e gerando a imagem rotulada abaixo.

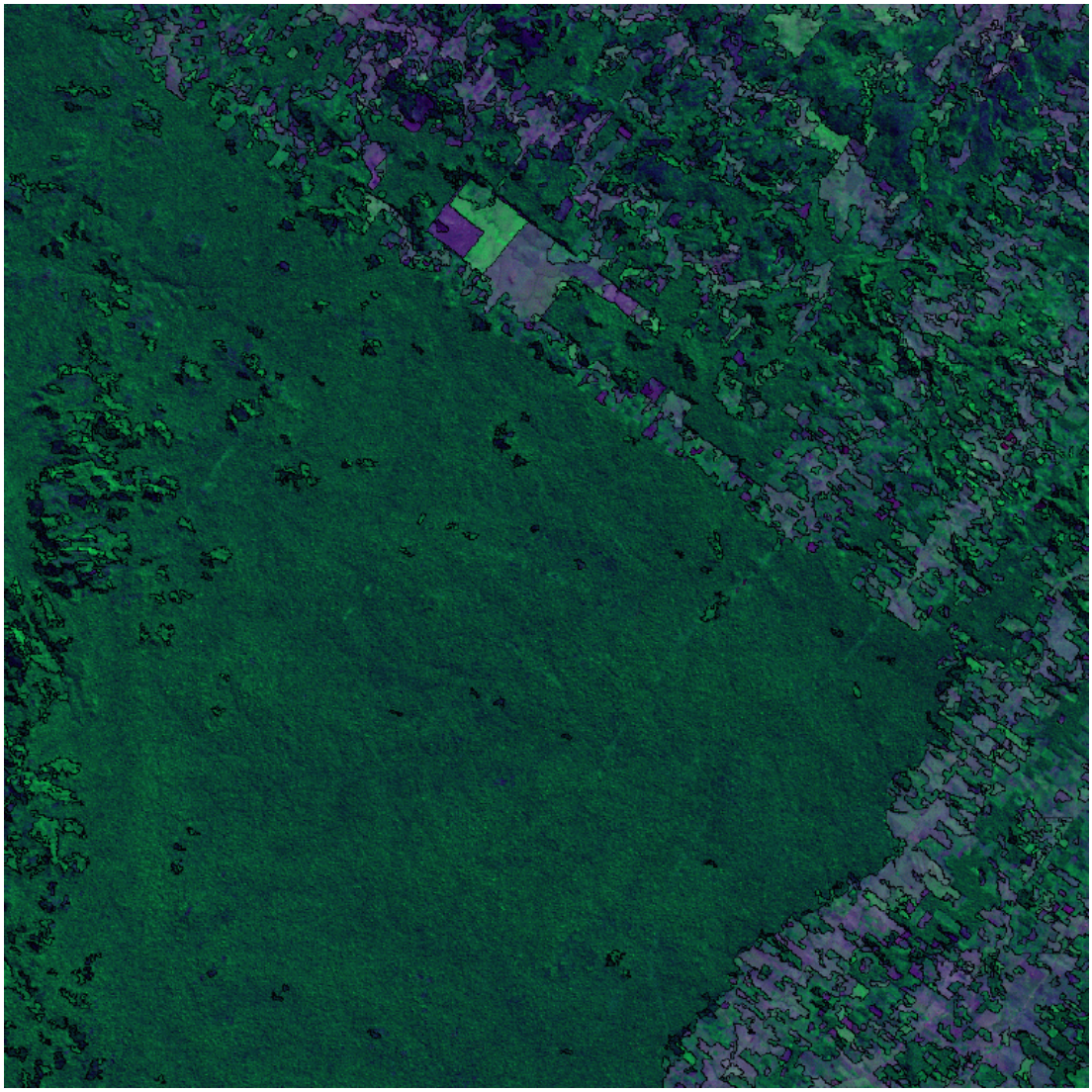


Fig. 4.3. – Imagem Rotulada C1024serial

Tabela 4.5 – Dados de Entrada do Experimento Número 4

Número de		**Banda**			**Liminar Similaridade**	**Limiar Área**	**Imagem Rotulada**
Lin	**Col**	**3**	**4**	**5**			
2048	2048	C2048r	C2048g	C2048b	30	5 e 25	C2048serial

Após **3 horas e 10 minutos** de processamento, o programa finalizou totalizando **6.215 regiões** e gerando a imagem rotulada abaixo.

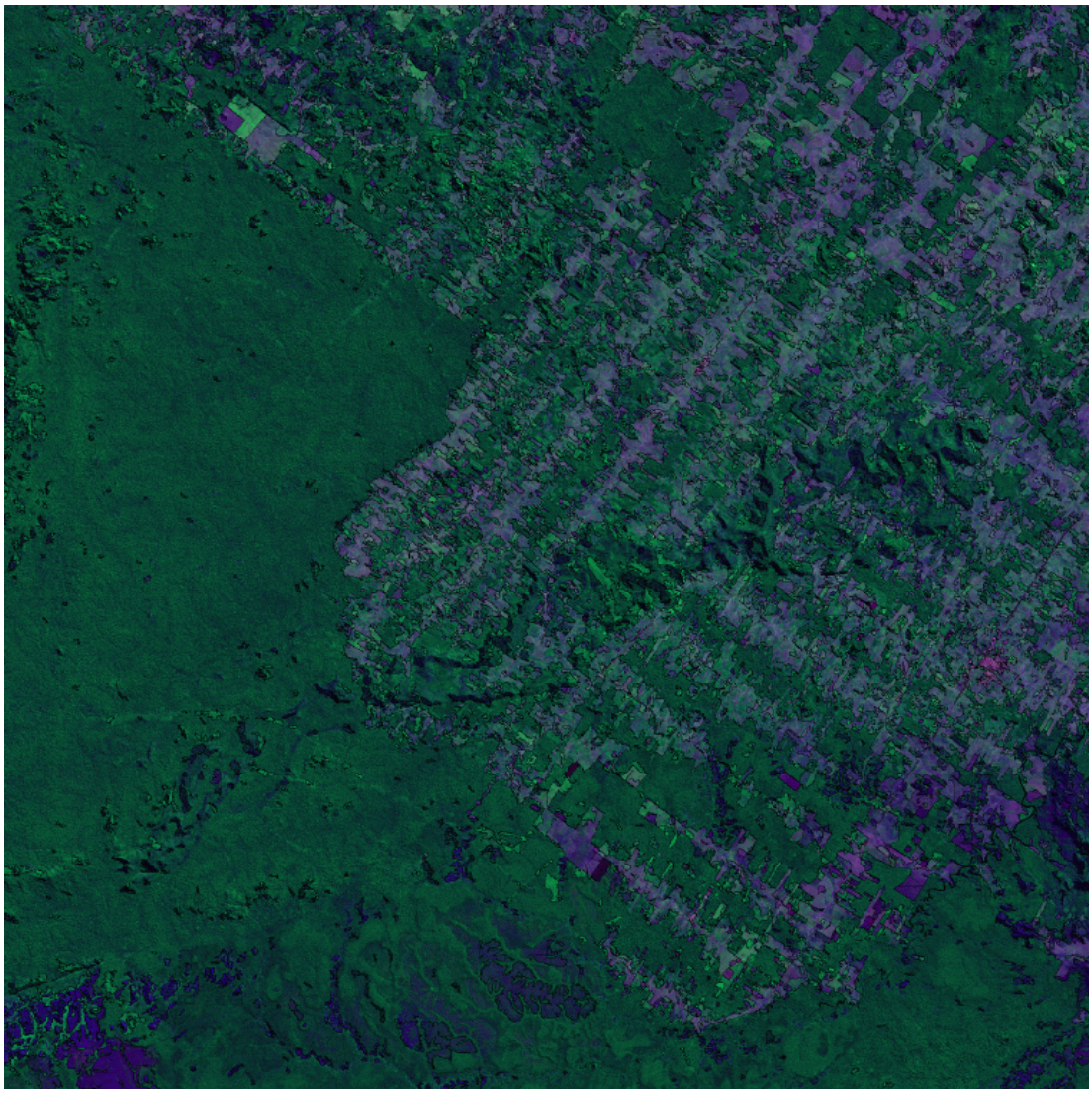


Fig. 4.4. – Imagem Rotulada C2048serial

Tabela 4.6 – Dados de Entrada do Experimento Número 5

Número de		**Banda**			**Liminar Similaridade**	**Limiar Área**	**Imagem Rotulada**
Lin	**Col**	**3**	**4**	**5**			
4096	4096	C4096r	C4096g	C4096b	30	5 e 25	C4096serial

Após **28 horas e 10 minutos** de processamento, o programa finalizou totalizando **27.737 regiões** e gerando a imagem rotulada abaixo.

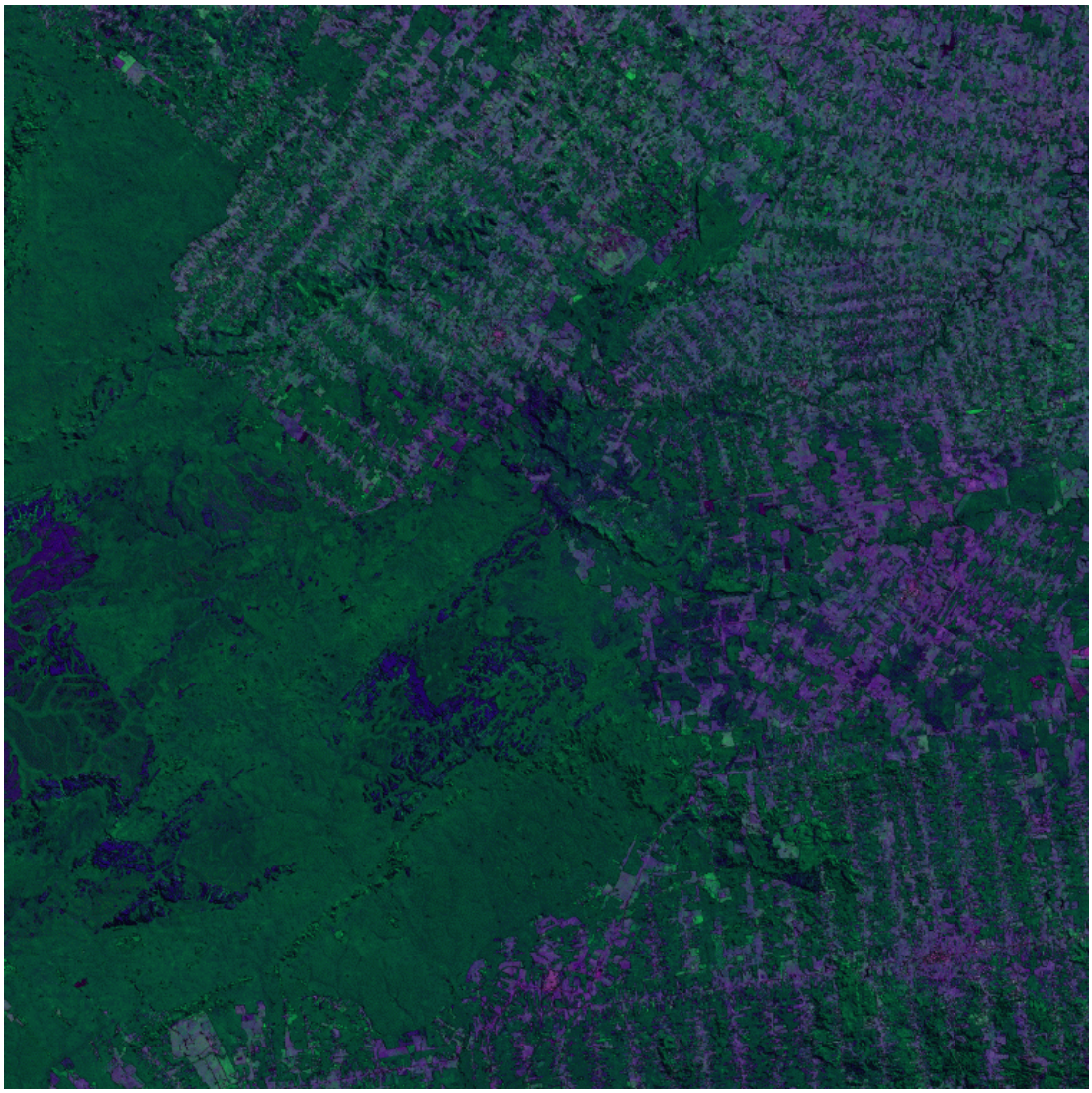


Fig. 4.5. – Imagem Rotulada C4096serial

4.2.1.1 – Resumo dos Testes da Versão Serial

A Tabela a seguir expõe de forma resumida os resultados obtidos nos testes da versão serial do programa. Pode-se notar que o tempo de execução para a imagem 4096 X 4096 ultrapassa 28h, o que dificulta a operacionalização da metodologia proposta em Batista et al (1996), uma vez que a Amazônia Legal é coberta com 229 cenas LANDSAT de 6100 X 6100 pontos, que devem ser classificados em 12 meses.

Tabela 4.7 – Resultados do Experimento Serial

Imagem	Tempo	Tempo (Segundos)	Número de Regiões
C256	56 seg	56	74
C512	4 min 3 seg	243	379
C1024	30 min	1800	1494
C2048	3h 10 min	11400	6215
C4096	28h 10 min	101400	27737

4.2.2 – Execução do Código Paralelo

O código paralelo foi executado num cluster formado por dez PC´s, com sistema operacional Conectiva Linux 3.0 (guarani), Kernel 2.0.36, Mpich 1.1.2, do laboratório 6 da Universidade do Vale do Paraíba. A máquina empregada como mestre foi a tico.fcc.univap.br (200.136.180.11), mesma máquina usada no processamento da versão serial. As demais máquinas possuem a mesma configuração de hardware. A tabela abaixo descreve as características principais do hardware das máquinas que compõem o cluster de PC´s.

Tabela 4.7 – Configuração do Hardware das Dez Máquinas Utilizadas nos Testes da Versão Paralela.

Marca	**Modelo**	**Clock**	**RAM**	**Memória Virtual**	**Processador**
Itautec	InfoWay	333 MHz	64 Mb	128 Mb	Pentium II

Como na versão serial, o programa foi disparado, mas antes de iniciar a contagem do tempo, alguns parâmetros foram passados via teclado para o programa. Os parâmetros necessários são o número de linhas e colunas que compõem a imagem, número de bandas, nome das imagens, limiar de similaridade, limiar de área e o nome que será dado a imagem rotulada gerada no final do processamento. É importante salientar que para todos os experimentos o limiar de similaridade foi igual a 30 e o número de bandas igual a 3.

Como parte dos testes, o limiar de área foi variado, foram empregados limiares iguais a 15, 20 e 25. É necessário observar que quando se variou o número de processadores a imagem rotulada gerada ao final do processamento é a mesma para um mesmo limiar de área. Portanto este documento limita-se a apresentar somente uma imagem resultante para cada limiar, independente do número de processadores empregados na execução.

Na linha de comando para executar o programa é necessário que o usuário defina o número de processadores que participarão do processamento. Por exemplo, se o usuário quiser utilizar 5 processadores a sintaxe é a seguinte (Equação 2.19):

$$\text{mpirun –np 5 <programa executável>} \qquad 2.19$$

onde, 5 equivale ao número de processadores.

Se a imagem processada tem 512 linhas x 512 colunas e se tem 5 processadores disponíveis, um desses processadores será destinado como mestre e os demais (quatro) serão os escravos. Como o programa foi construído para particionar a imagem em

janelas de 128 x 128, caberá a cada processador escravo processar 4 janelas. A Figura 4.6 representa este exemplo.

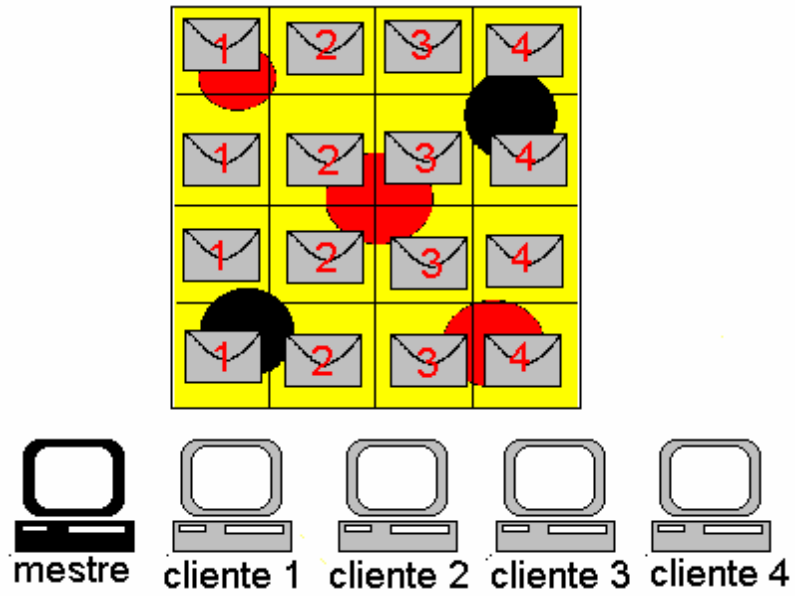


Fig. 4.6 – Esboço da Divisão de Tarefas para uma Imagem 512 X 512 com 4 Processadores Clientes

4.2.2.1 – Testes para Limiar de Área 15 e Limiar de Similaridade 30

O programa paralelo foi executado para as imagens Landsat-TM, denominadas C256, C512, C1024, C2048 e C4096, pertencentes a órbita/ponto 231/68 da região Amazônica, bandas 3, 4 e 5, limiar de área 15 e limiar de similaridade 30

As Tabelas 4.9, 4.10, 4.11, 4.12 e 4.13 apresentam os dados de entrada e os resultados de cada teste para limiar de área 15 e limiar de similaridade 30.

Tabela 4.9 – Resultados do Experimento 6

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	**Col**	2	4	4	103	25,64
256	256	3		2		13,98
		5		1		8,19

A Figura 4.7 representa as imagens geradas.

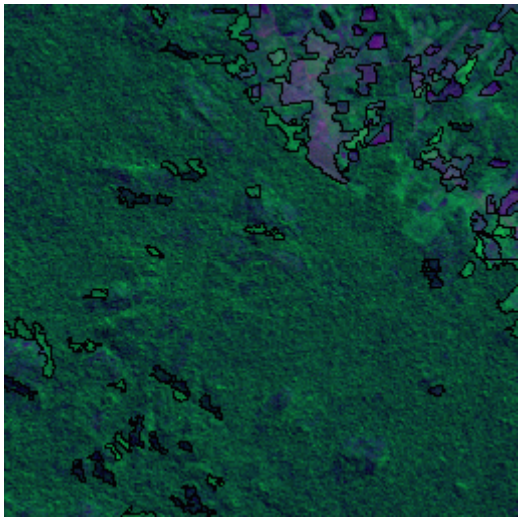


Fig. 4.7 - Imagem Rotulada c256_a15_rot

Tabela 4.10 – Resultados do Experimento 7

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	16	16	489	102,15
512	512	3		8		55,71
		5		4		30,56
		9		2		17,98

A Figura 4.8 representa as imagens geradas.

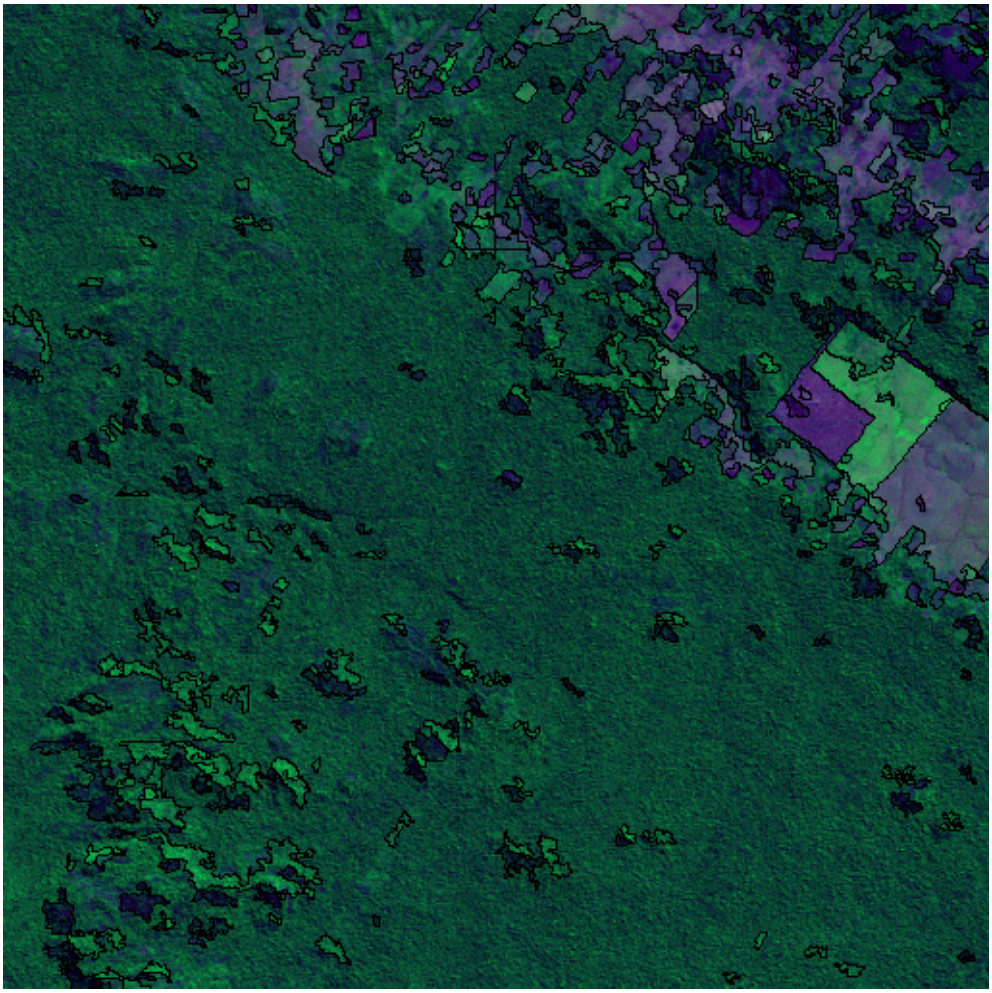


Fig. 4.8 - Imagem Rotulada c512_a15_rot

Tabela 4.11 – Resultados do Experimento 8

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	64	64	2033	415,89
1024	1024	3		32		220,83
		5		16		138,05
		9		8		87,35

A Figura 4.9 representa as imagens geradas.

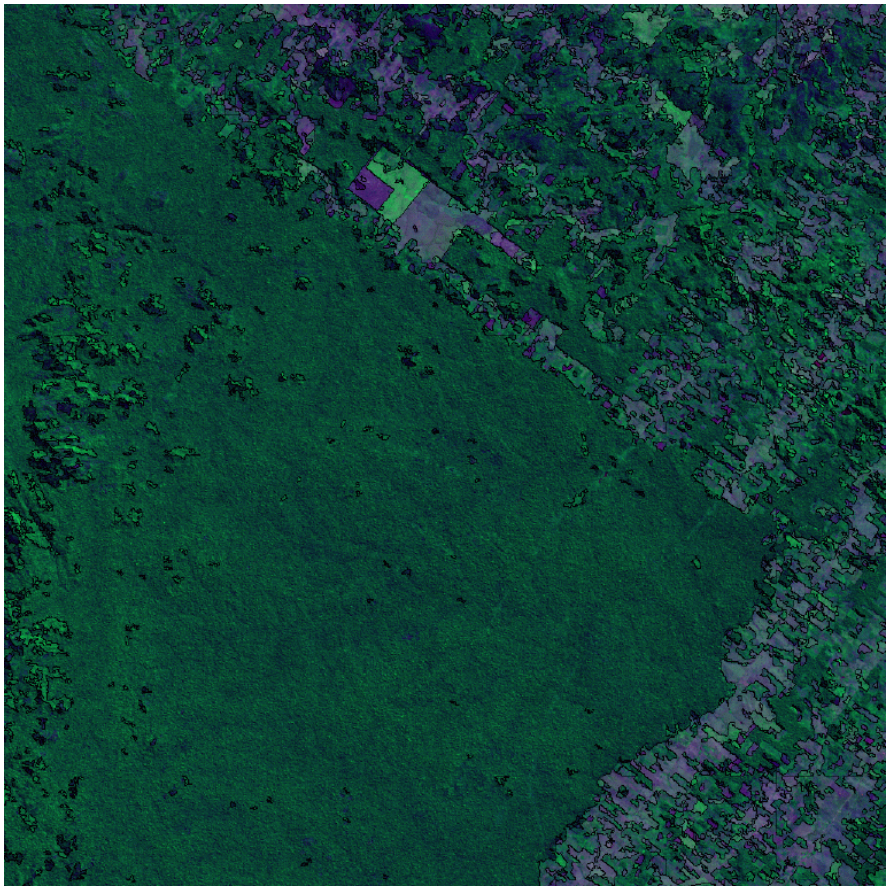


Fig. 4.9 - Imagem Rotulada c1024_a15_rot

Tabela 4.12 – Resultados do Experimento 9

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	**Col**	2	256	256	8885	2004,24
2048	2048	3		128		1268,99
		5		64		875,61
		9		32		566,56

A Figura 4.10 representa as imagens geradas.

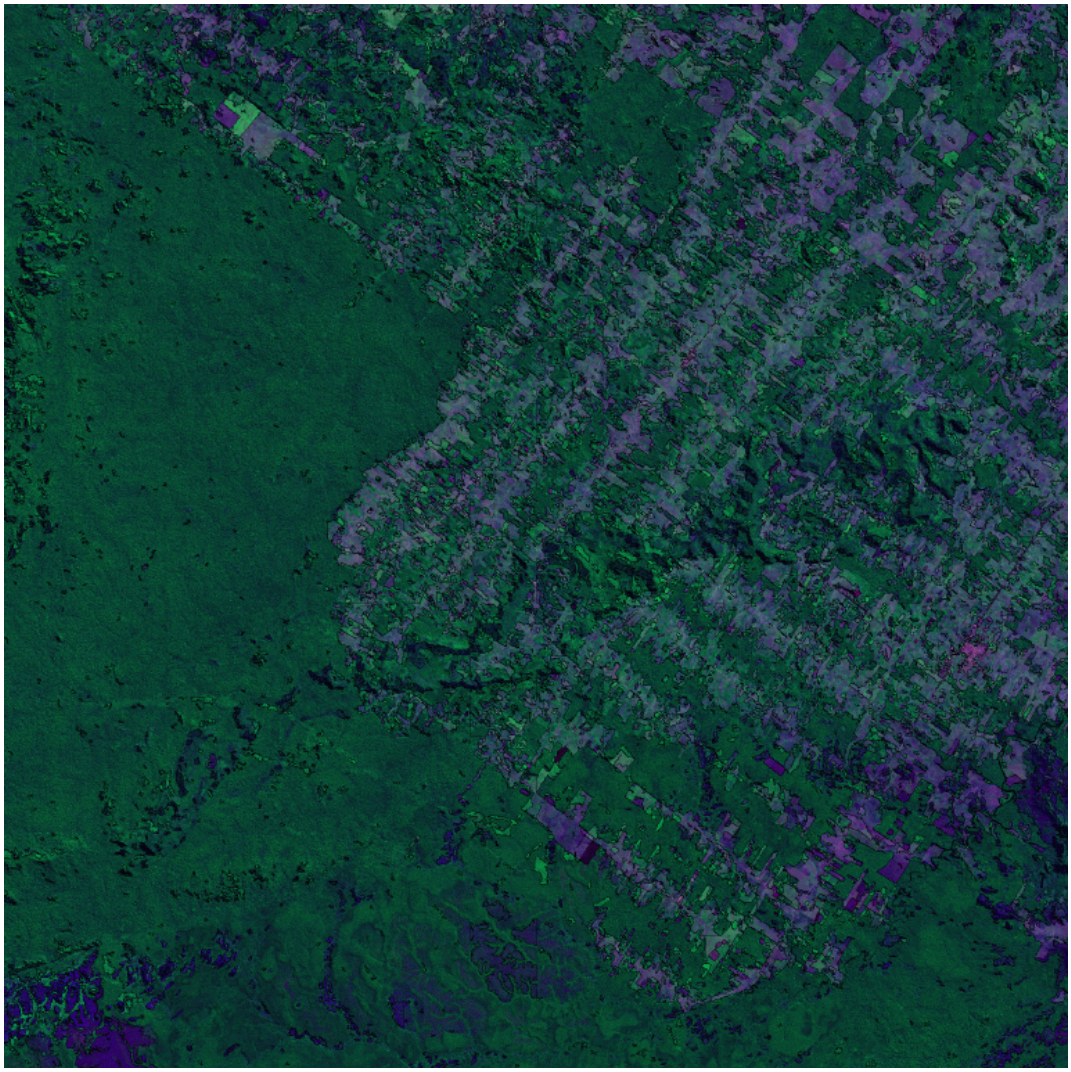


Fig. 4.10 - Imagem Rotulada c2048_a15_rot

Tabela 4.13 – Resultados do Experimento 10

Numero		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	1024	1024	40243	15634,86
4096	4096	3		512		12363,54
		5		256		5194,12
		9		128		5881,87

A Figura 4.11 representa as imagens geradas.

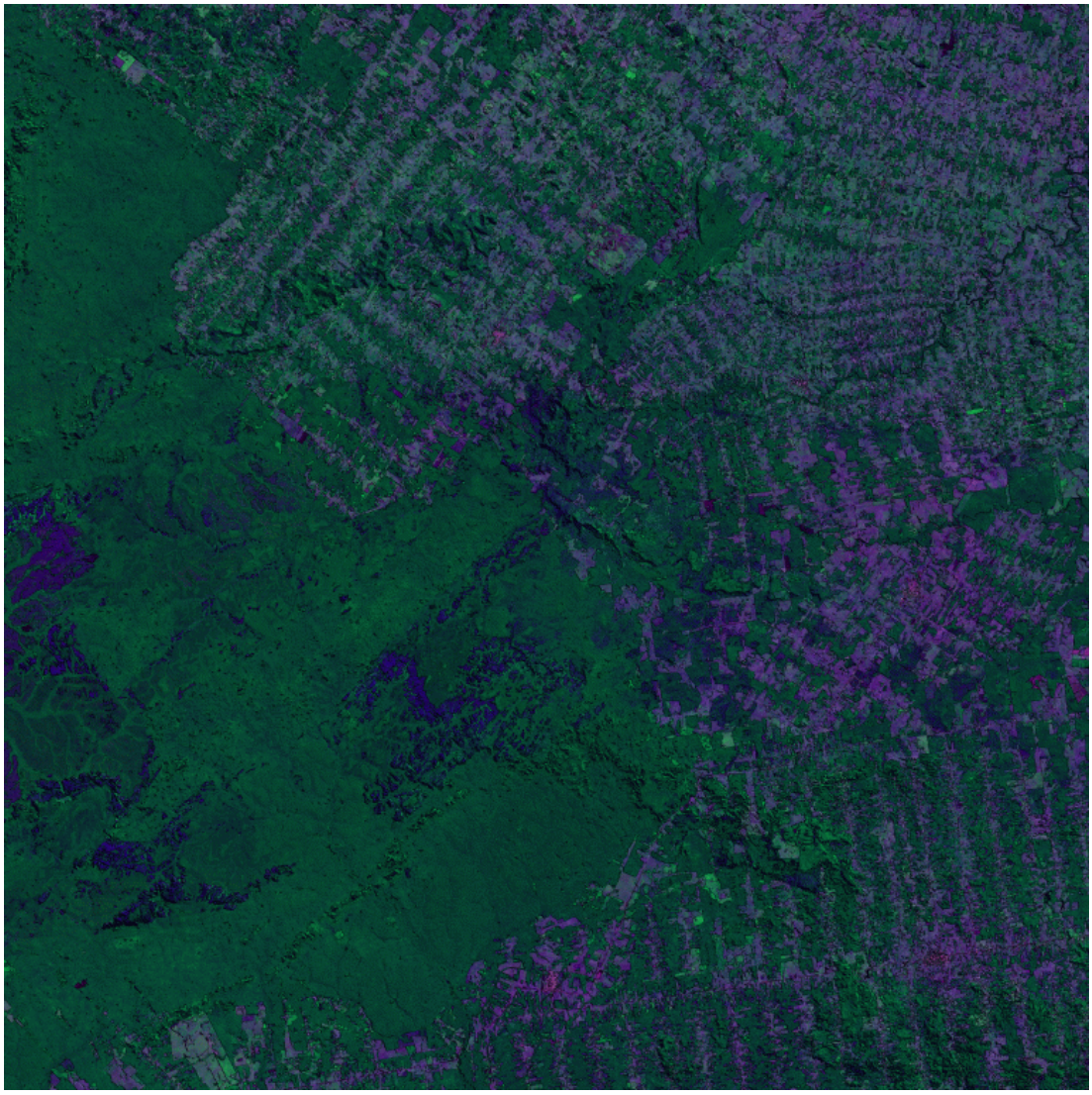


Fig. 4.11 - Imagem Rotulada c4096_a15_rot

4.2.2.2 – Testes para Limiar de Área 20 e Limiar de Similaridade 30

O programa paralelo foi executado para a imagem Landsat-TM, denominada C256, C512, C1024, C2048 e C4096, pertencentes a órbita/ponto 231/68 da região Amazônica, bandas 3, 4 e 5, limiar de área 20 e limiar de similaridade 30

As Tabelas 4.14, 4.15, 4.16, 4.17 e 4.18 apresentam os dados de entrada e os resultados de cada teste.

Tabela 4.14 – Resultados do Experimento 11

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	**Col**	2	4	4	85	25,81
256	256	3		2		13,76
		5		1		8,00

A Figura 4.12 representa as imagens geradas.

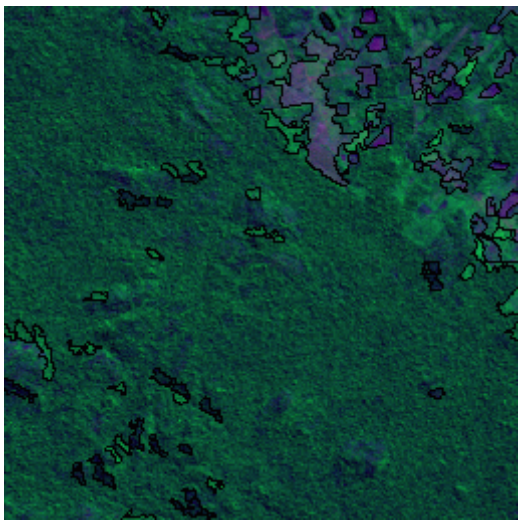


Fig. 4.12 - Imagem Rotulada c256_a20_rot

Tabela 4.15 – Resultados do Experimento 12

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	16	16	388	102,59
512	512	3		8		55,06
		5		4		31,46
		9		2		18,35

A Figura 4.13 representa as imagens geradas.

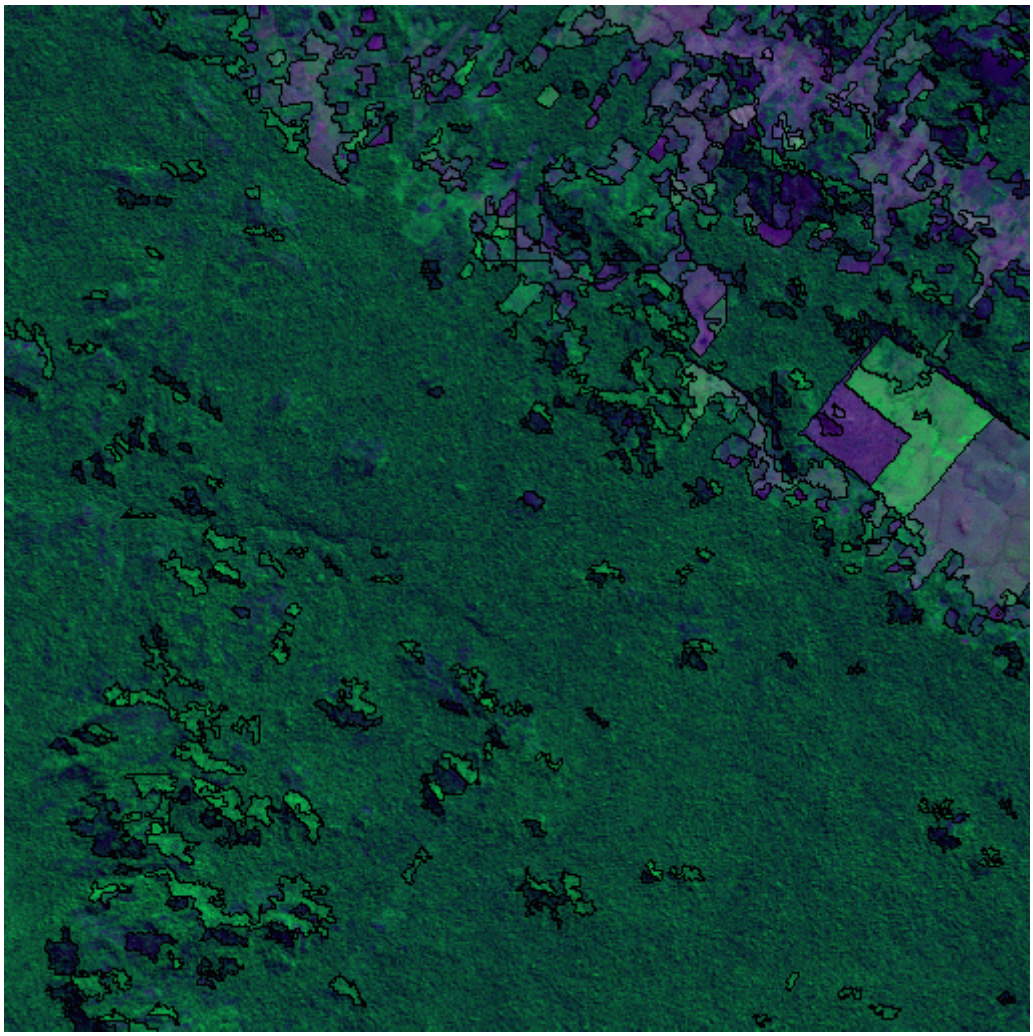


Fig. 4.13 - Imagem Rotulada c512_a20_rot

Tabela 4.16 – Resultados do Experimento 13

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	64	64	1615	420,71
1024	1024	3		32		230,23
		5		16		122,85
		9		8		70,48

A Figura 4.14 representa as imagens geradas.

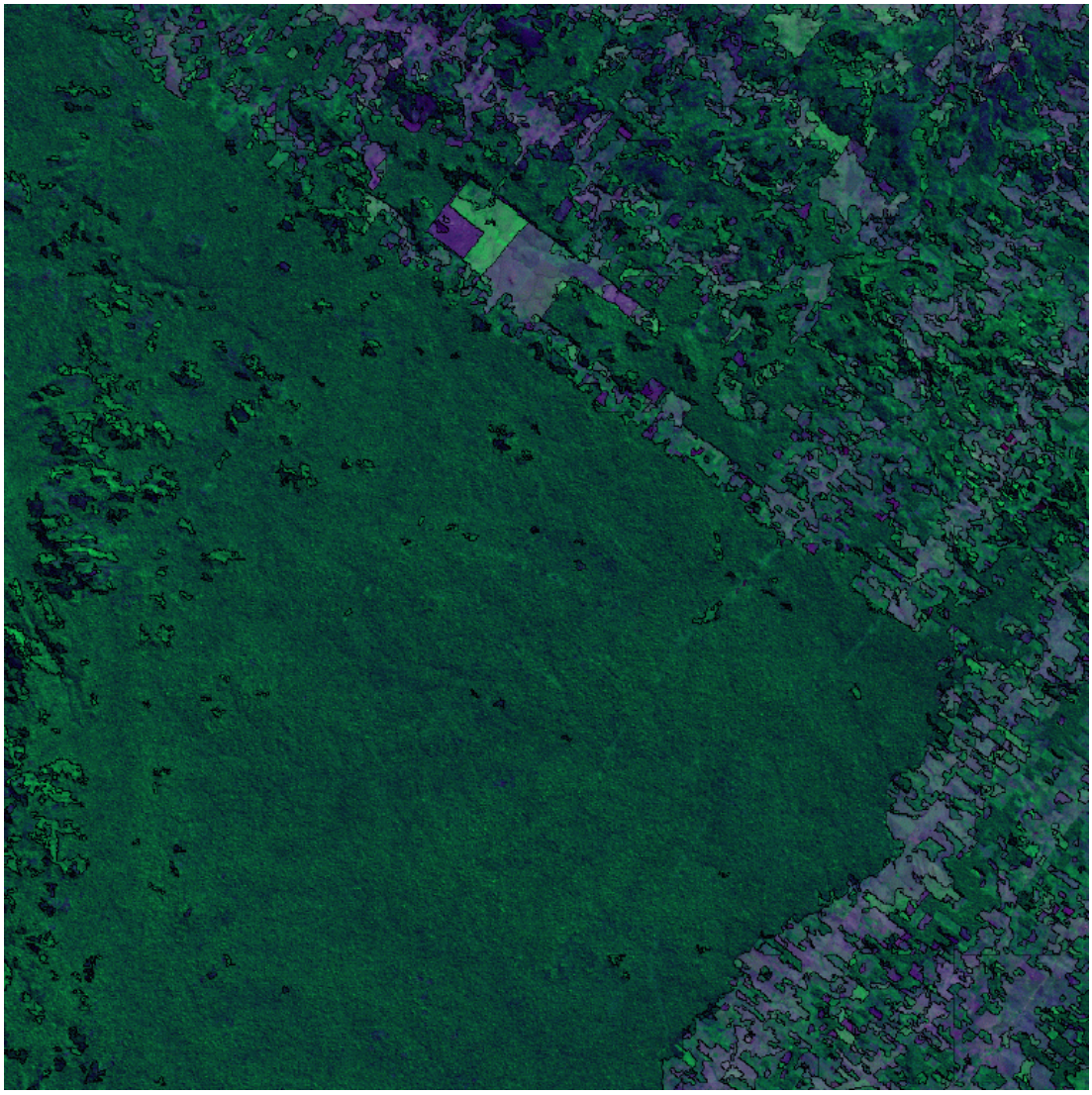


Fig. 4.14 - Imagem Rotulada c1024_a20_rot

Tabela 4.17 – Resultados do Experimento 14

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	256	256	7069	2069,40
2048	2048	3		128		1059,10
		5		64		716,69
		9		32		690,98

A Figura 4.15 representa as imagens geradas.

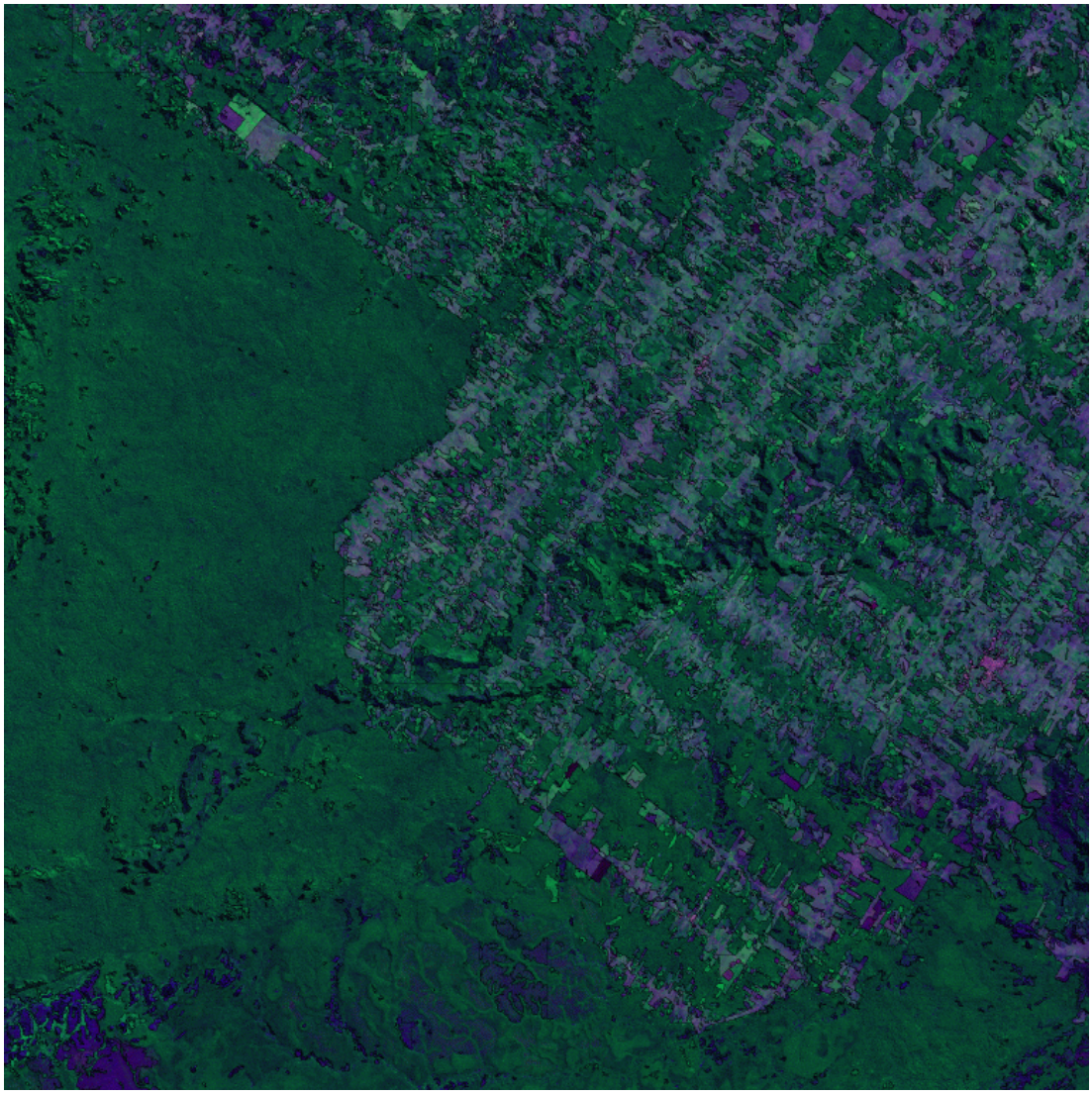


Fig. 4.15 - Imagem Rotulada c2048_a20_rot

Tabela 4.18 – Resultados do Experimento 15

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	1024	1024	32071	11092,14
4096	4096	3		512		7857,41
		5		256		6644,75
		9		128		6217,84

A Figura 4.16 representa as imagens geradas.

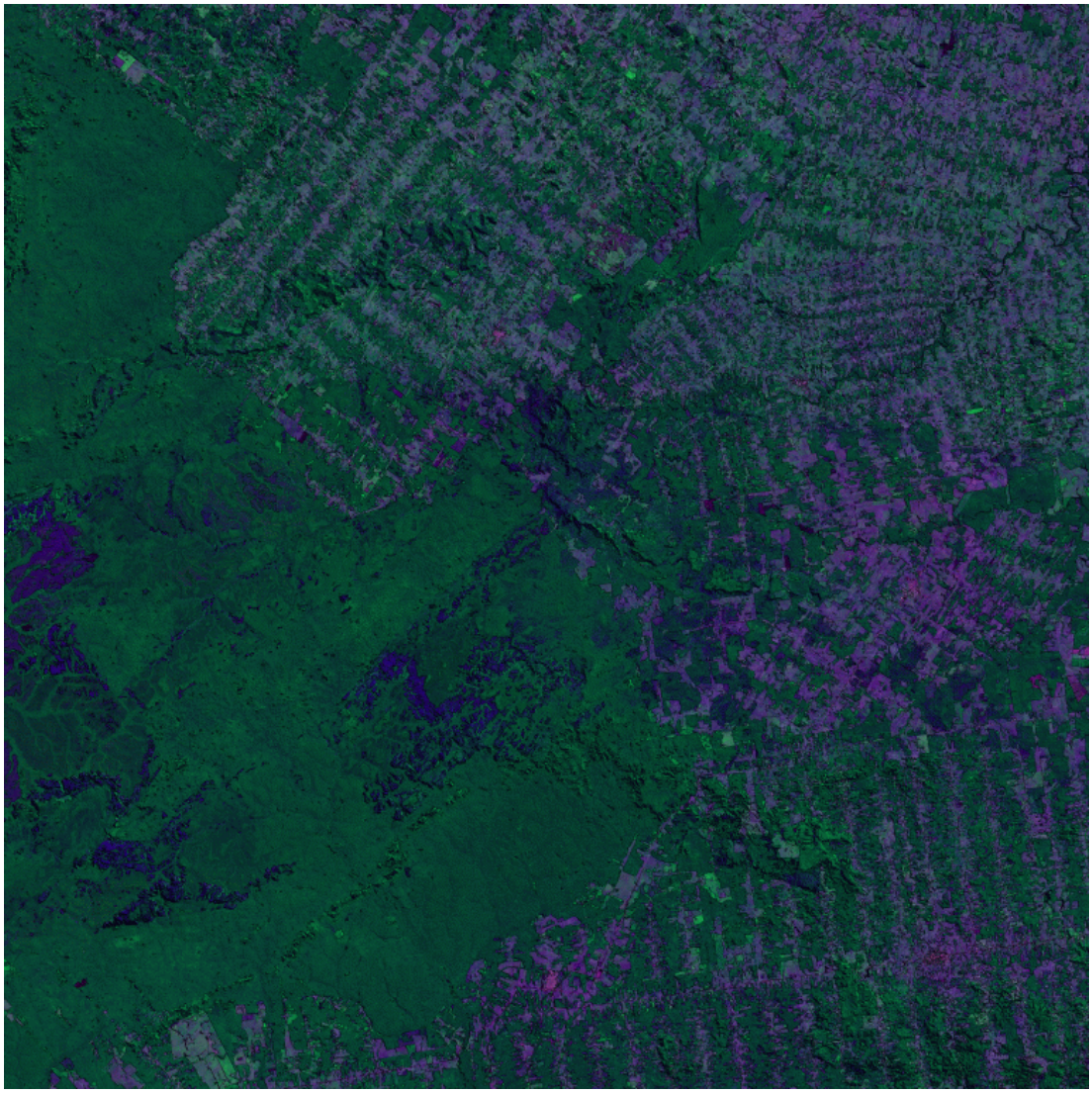


Fig. 4.16 - Imagem Rotulada c4096_a20_rot

4.2.2.3 – Testes para Limiar de Área 25 e Limiar de Similaridade 30

O programa paralelo foi executado para a imagem Landsat-TM, denominada C256, C512, C1024, C2048 e C4096, pertencentes a órbita/ponto 231/68 da região Amazônica, bandas 3, 4 e 5, limiar de área 25 e limiar de similaridade 30

As Tabelas 4.19, 4.20, 4.21, 4.22 e 4.23 apresentam os dados de entrada e os resultados de cada teste.

Tabela 4.19 – Resultados do Experimento 16

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	**Col**	2	4	4	68	25,48
256	256	3		2		13,69
		5		1		7,71

A Figura 4.17 representa as imagens geradas.

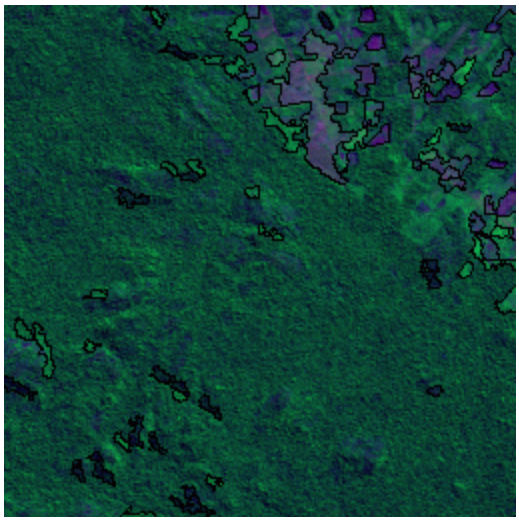


Fig. 4.17 - Imagem Rotulada c256_a25_rot

Tabela 4.20 – Resultados do Experimento 17

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	**Col**	2	16	16	328	103,01
512	512	3		8		54,25
		5		4		29,24
		9		2		17,41

A Figura 4.18 representa as imagens geradas.

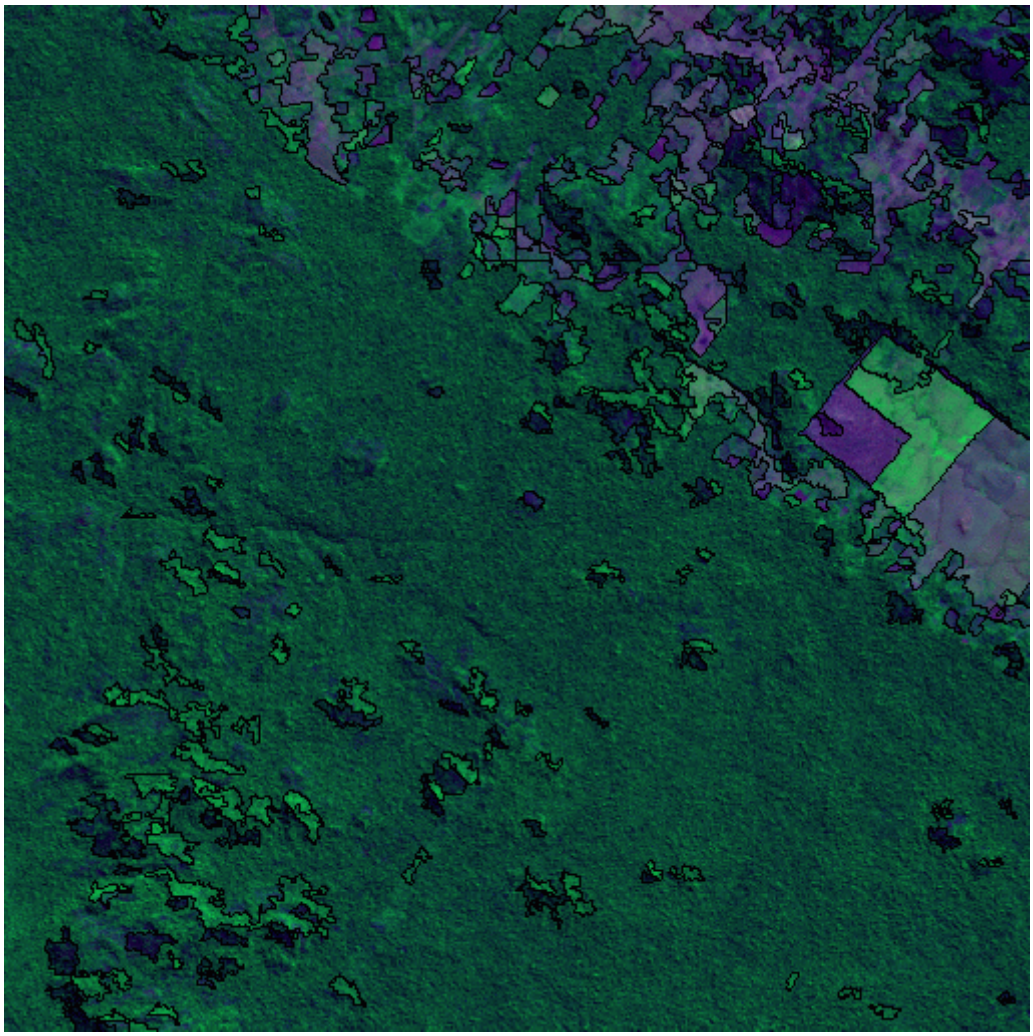


Fig. 4.18 - Imagem Rotulada c512_a25_rot

Tabela 4.21 – Resultados do Experimento 18

Número		**Número de Processadores**	**Total de Janelas**	**Janela por Processador**	**Número de Regiões**	**Tempo (segundos)**
Lin	Col	2	64	64	1373	420,21
1024	1024	3		32		222,13
		5		16		122,13
		9		8		68,39

A Figura 4.19 representa as imagens geradas.

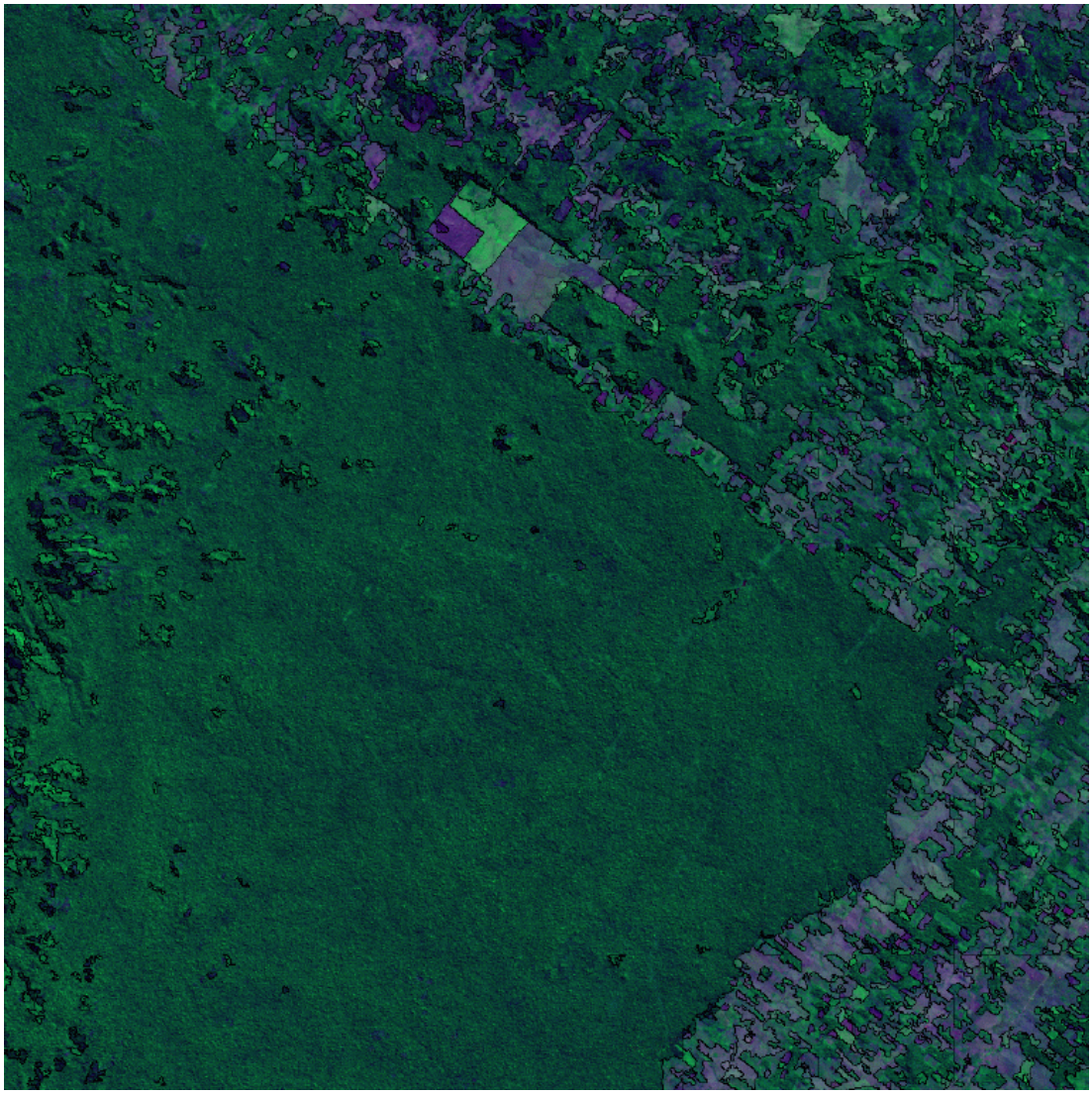


Fig. 4.19 - Imagem Rotulada c1024_a25_rot

Tabela 4.22 – Resultados do Experimento 19

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	256	256	5743	1724,21
2048	2048	3		128		959,89
		5		64		579,86
		9		32		382,94

A Figura 4.20 representa as imagens geradas.

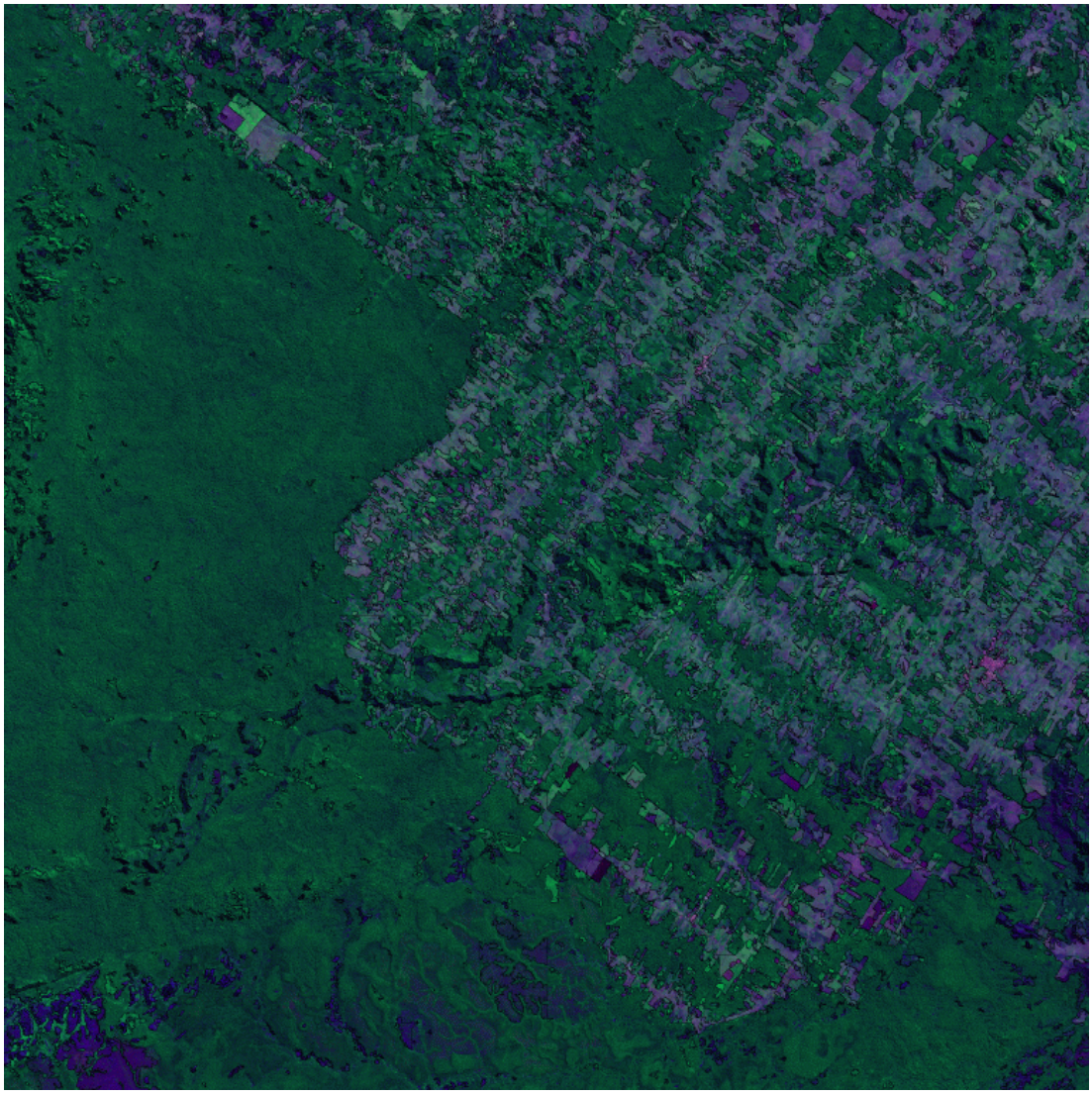


Fig. 4.20 - Imagem Rotulada c2048_a25_rot

Tabela 4.23 – Resultados do Experimento 20

Número		Número de Processadores	Total de Janelas	Janela por Processador	Número de Regiões	Tempo (segundos)
Lin	**Col**	2	1024	1024	26734	9628,86
4096	4096	3		512		5103,29
		5		256		3716,94
		9		128		2996,57

A Figura 4.21 representa as imagens geradas.

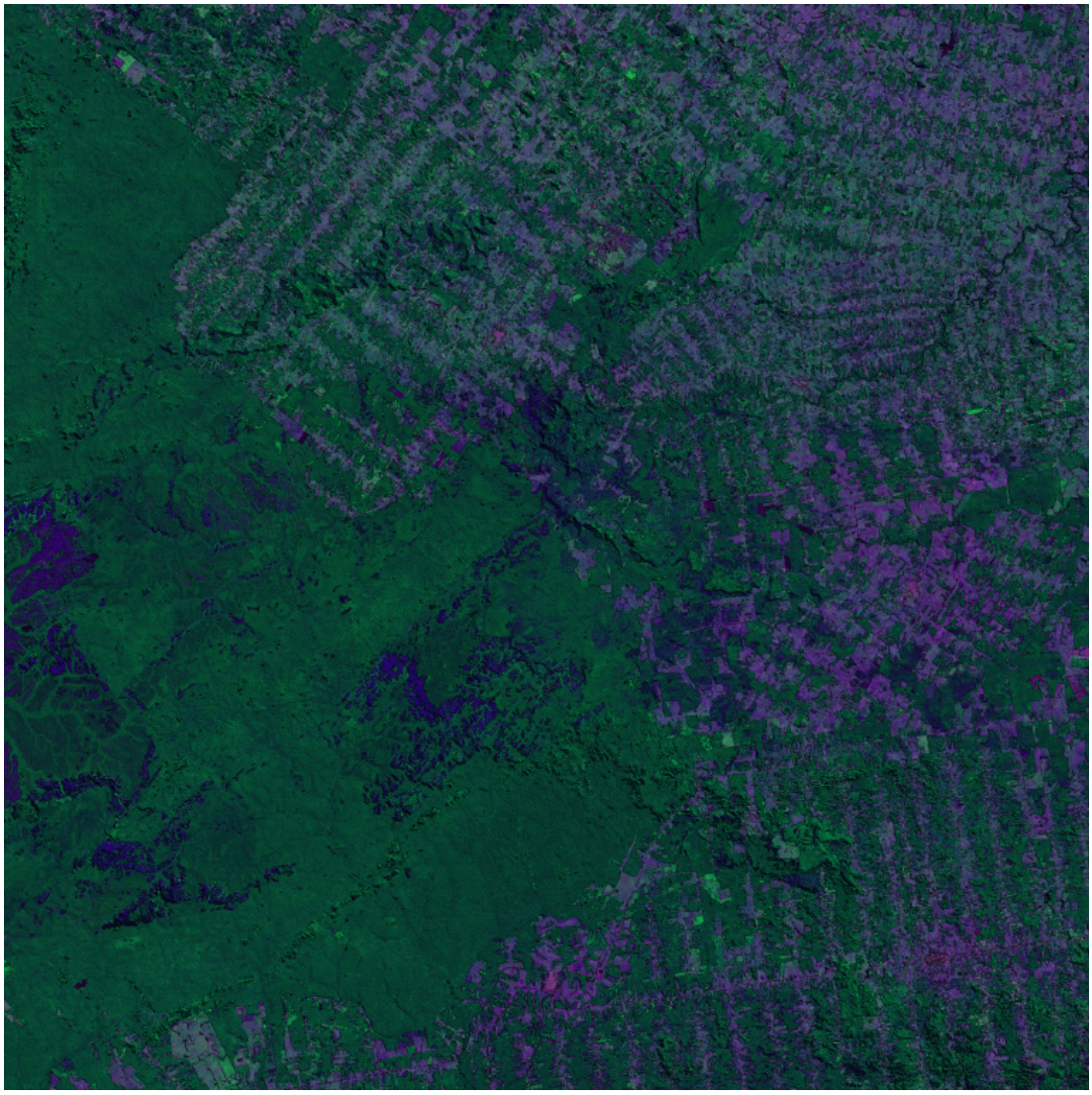


Fig. 4.21 - Imagem Rotulada c4096_a25_rot

4.2.2.4 – Resumo dos Testes da Versão Paralela

As Tabelas 4.24, 4.25 e 4.26 expõem de forma resumida os resultados obtidos nos testes da versão paralela do programa. Nota-se que o tempo de execução para a imagem C4096 foi reduzido consideravelmente quando comparado ao da versão serial, tornando o possível o operador tentar encontrar o limiar que melhor se ajuste ao problema, através da possibilidade de diversas execuções do programa durante um dia de trabalho.

Nestas tabelas existe uma divisão da coluna tempo, onde são expressados os valores de tempo para o processo de segmentação (função cresce e elimina regiões, exercida pelos processadores clientes), costura (função de concatenação das lista e costura das janelas, exercida pelo processador mestre) e total que apresenta a soma do tempo de segmentação e costura

Nota-se nestas tabelas que o tempo de costura é praticamente o mesmo, independente do número de processadores empregados na execução, pois este processo é realizado no processador mestre, não havendo a divisão do trabalho.

Outra característica importante é que quanto maior a imagem, maior será o tempo gasto com a costura da mesma e este tempo pode ser maior que o tempo gasto com a segmentação. Uma solução para este problema seria a paralelização do processo de costura da imagem.

Tabela 4.24 – Resultados do Experimento Paralelo - Limiar de Área = 15

Imagem	Processadores Clientes	Número de Regiões	Tempo (segundos)			Tempo
			Segmentação	Costura	Total	
C256	1	103	25,55	0,07	25,64	26 seg
C256	2	103	13,91	0,07	13,98	14 seg
C256	4	103	8,12	0,07	8,19	8 seg
C512	1	489	101,47	0,67	102,15	1 min e 42 seg
C512	2	489	53,32	0,80	54,71	55 seg
C512	4	489	29,88	0,67	30,56	31 seg
C512	8	489	17,32	0,65	17,98	18 seg
C1024	1	2033	410,40	5,49	415,89	6 min e 55 seg
C1024	2	2033	215,41	5,41	220,83	3 min e 40 seg
C1024	4	2033	132,41	5,63	138,05	2 min e 18 seg
C1024	8	2033	81,97	5,37	87,35	1 min e 27 seg
C2048	1	8885	1626,05	378,19	2004,24	33 min e 24 seg
C2048	2	8885	876,24	392,74	1268,99	21 min e 8 seg
C2048	4	8885	482,98	392,63	875,61	14 min e 35 seg
C2048	8	8885	285,44	281,11	566,56	9 min e 26 seg
C4096	1	40243	7174,23	8459,99	15634,86	4 h e 20 min
C4096	2	40243	3831,78	8531,71	12363,54	3 h e 26 min
C4096	4	40243	1891,65	3302,42	5194,12	1 h e 26 min
C4096	8	40243	2415,05	3466,16	5881,87	1 h e 38 min

Tabela 4.25 – Resultados do Experimento Paralelo - Limiar de Área = 20

Imagem	Processadores Clientes	Número de Regiões	Tempo (segundos)			Tempo
			Segmentação	Costura	Total	
C256	1	85	25,71	0,07	25,81	26 seg
C256	2	85	13,69	0,06	13,76	14 seg
C256	4	85	7,92	0,07	8,00	8 seg
C512	1	388	101,94	0,64	102,59	1 min e 42 seg
C512	2	388	54,40	0,66	55,06	55 seg
C512	4	388	30,81	0,64	31,46	31 seg
C512	8	388	17,70	0,64	18,35	18 seg
C1024	1	1615	415,55	5,16	420,71	7 min
C1024	2	1615	225,00	5,22	230,23	3 min e 50 seg
C1024	4	1615	117,69	5,15	122,85	2 min e 2 seg
C1024	8	1615	64,45	6,03	70,48	1 min e 10 seg
C2048	1	7069	1593,03	145,82	1739,68	29 min
C2048	2	7069	858,23	200,87	1059,10	17 min e 39 seg
C2048	4	7069	484,58	232,11	716,69	11 min e 56 seg
C2048	8	7069	610,69	77,47	688,17	11 min e 28 seg
C4096	1	32071	6664,53	4427,58	11092,14	3 h e 5 min
C4096	2	32071	3511,43	4345,80	7857,41	2 h e 11 min
C4096	4	32071	1962,83	4681,86	6644,75	1 h e 51 min
C4096	8	32071	1162,67	5054,98	6217,84	1 h e 43 seg

Tabela 4.26 – Resultados do Experimento Paralelo - Limiar de Área = 25

Imagem	Processadores Clientes	Número de Regiões	Tempo (segundos)			Tempo
			Segmentação	Costura	Total	
C256	1	68	25,41	0,068	25,48	25 seg
C256	2	68	13,62	0,067	13,69	14 seg
C256	4	68	7,64	0,067	7,71	8 seg
C512	1	328	102,36	0,65	103,01	1 min e 43 seg
C512	2	328	53,56	0,68	54,25	54 seg
C512	4	328	28,58	0,66	29,24	29 seg
C512	8	328	16,76	0,65	17,41	17 seg
C1024	1	1373	415,22	4,99	420,21	7 min
C1024	2	1373	217,21	4,91	222,13	3 min e 42 seg
C1024	4	1373	117,58	5,06	122,64	2 min e 3 seg
C1024	8	1373	63,46	4,93	68,39	1 min e 8 seg
C2048	1	5743	1597,72	126,48	1724,21	28 min e 44 seg
C2048	2	5743	835,04	124,84	959,89	16 min
C2048	4	5743	453,77	126,09	579,86	9 min e 40 seg
C2048	8	5743	258,55	124,39	382,94	6 min e 23 seg
C4096	1	26734	6502,17	3126,30	9628,86	2h e 40 min
C4096	2	26734	3403,38	1699,87	5103,29	1 h e 25 min
C4096	4	26734	1846,18	1870,39	3716,94	1 h e 2 min
C4096	8	26734	1065,49	1931,03	2996,57	49 min e 56 seg

4.3 – Validação dos Dados de Saída

A segmentação é apenas uma das fases da classificação de imagens. Ela é responsável por separar as regiões que se assemelhem espectralmente, com base num limiar de similaridade imposto pelo operador. O operador também é responsável pela imposição do limiar de área, com agregação das regiões que possuem área menor que a estipulada pelo limiar à região mais próxima espectralmente, ou seja, agrega por força e não por similaridade. O limiar de área, juntamente com o limiar de similaridade são os grandes responsáveis pelo aspecto da imagem rotulada final; eles representam a restrição especificada pelo operador, que estipula valores de acordo com a percepção que tem da complexidade da imagem.

O importante nesta fase é o operador ter tempo hábil para testar limiares, até gerar uma imagem rotulada adequada à solução do problema.

Após a fase de segmentação o operador deverá classificar as regiões e editá-las. Nestas fases ele poderá alterar algumas características perdidas ou erroneamente separadas no processo de segmentação. Hoje em dia, ainda não se tem um processo totalmente automático de classificação, todos os meios conhecidos necessitam da intervenção do operador para os mapas gerados sejam satisfatórios.

A validação dos dados de saída pode ser obtida através da comparação entre o número de regiões segmentadas apontadas pela versão serial e pela versão paralela e também, através da subtração das imagens geradas em ambos os casos ou simplesmente através da comparação visual das mesmas.

A Tabela 4.27 esboça a comparação dos resultados obtidos a partir da versão serial e versão paralela. É importante ressaltar que em ambos os casos o limiar de similaridade usado é 30, mas devido a diferenças algorítmicas o limiar de área não é o mesmo, o que acarreta a diferença entre o número de regiões segmentadas.

Como dito anteriormente, visando a economia de memória e agilizar o processamento, a versão serial emprega a função “junção de áreas” (responsável pela junção das regiões

similares ou que tenham área menor que o limiar de área imposto pelo operador) duas vezes durante o processo de segmentação. Na primeira passagem utiliza um limiar fixo igual 5 e processa pequenos blocos da imagem por vez e ao final de todo processo de segmentação chama a função “eliminação de áreas” novamente, agora se baseando no limiar de área estipulado pelo operador, que no caso do exemplo da Tabela 4.26 é igual a 25. Já na versão paralela todo o processo de junção de células é realizado nos processadores escravos, objetivando a distribuição do trabalho e consequentemente a redução do tempo de processamento, sem grande prejuízos para o mapa final.

Com relação a tempos de processamento, pode-se notar, na próxima tabela, que houve redução e essa redução se acentuada a medida que se emprega mais processadores. Mas, não se deve esquecer que a redução do tempo não está somente relacionada a distribuição das tarefas, mas também a uma ligeira mudança no algoritmo como citado anteriormente. Análises mais detalhadas sobre os tempos de execução podem ser vistas no próximo item.

Tabela 4.27 – Versão Serial (Limiar Área 5/25) X Versão Paralela (Limiar De Área 25)

	Versão serial		**Versão paralela**				
Imagem	**Tempo (seg)**	**Número regiões**	**Processadores. Clientes**	**Tempo (seg)**	**Número regiões**	**Relação tempo**	**Diferença número regiões (%)**
C256	56	74	1	25,48	68	2,2	-8,1
			2	13,69		4,1	
			4	7,71		7,3	
C512	243	379	1	103,01	328	2,4	-13,5
			2	54,25		4,5	
			4	29,24		8,3	
			8	17,41		14,0	
C1024	1800	1494	1	420,21	1373	4,3	-8,1
			2	222,13		8,1	
			4	122,64		14,7	
			8	68,39		26,3	
C2048	11400	6215	1	1724,21	5743	6,6	-7,6
			2	959,89		11,9	
			4	579,86		19,7	
			8	382,94		29,8	
C4096	101400	27737	1	9328,07	26734	10,9	-3,6
			2	5103,29		19,9	
			4	3716,94		27,3	
			8	2996,57		33,8	

A Figura 4.22 esboça a diferença entre o número de regiões segmentadas pela versão serial e o número de regiões segmentadas pela versão paralela. nota-se que não existe nenhuma correlação entre o aumento do tamanho da imagem e a diferença nos resultados do número de regiões segmentadas pelas duas versões.

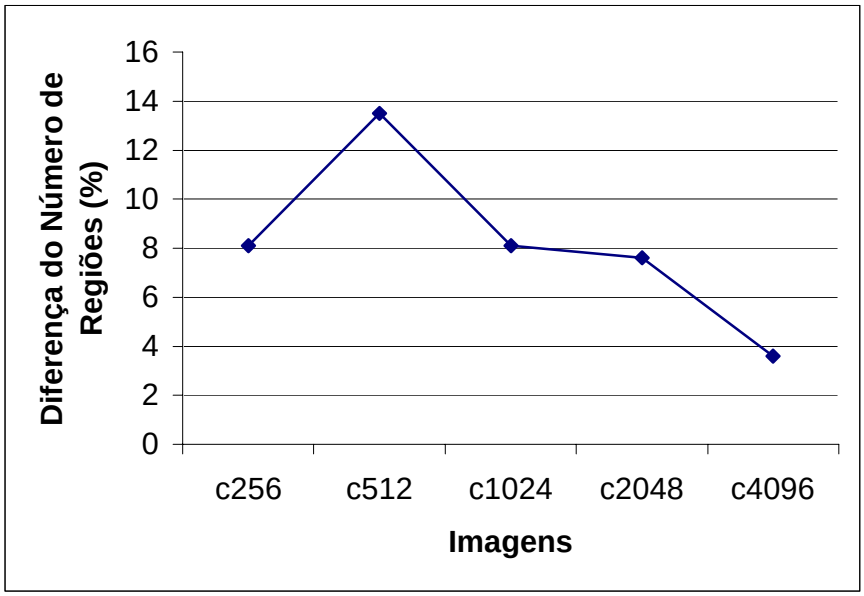


Fig. 4.22 – Número de Regiões – Versão Serial x Versão Paralela

Abaixo, nas Figuras 4.23 e 4.24, são apresentadas as imagens resultantes da versão serial limiar de área 5/25 e as imagens resultantes da versão paralela limiar 25, para a simples comparação visual. Dependendo da complexidade da imagem e da área das regiões que a compõem esse método de validação pode não apresentar um resultado expressivo.

Devido ao tamanho das imagens testes, não é possível, nesta documentação, colocar as imagens resultantes lado a lado para a comparação visual. Portanto, este trabalho apresenta apenas a comparação para as imagens 256x256 (Figura 4.23) e 512x512 (Figura 4.24).

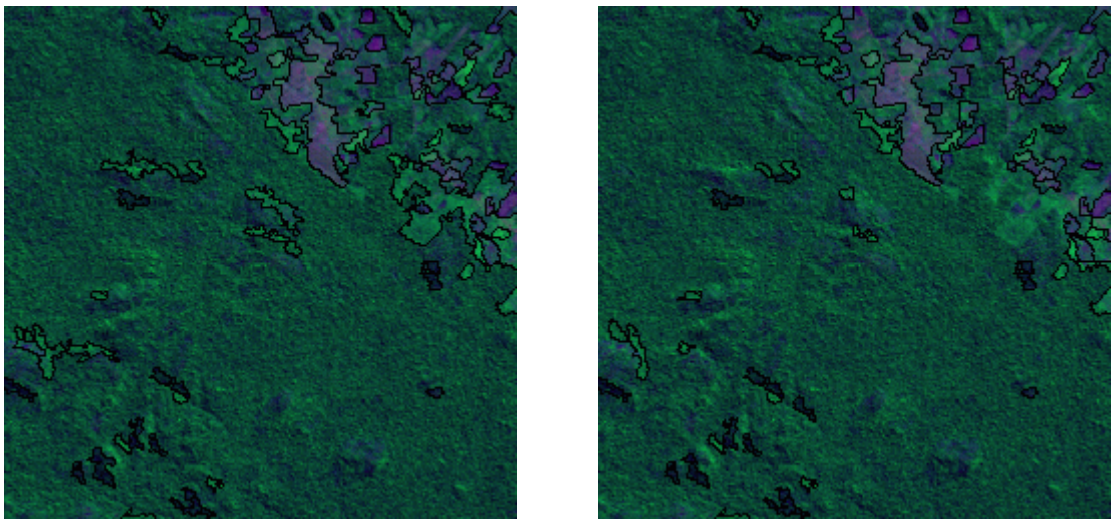


Fig. 4.23 - Comparação Visual – Resultado Serial e Resultado Paralelo - Imagem 256x256

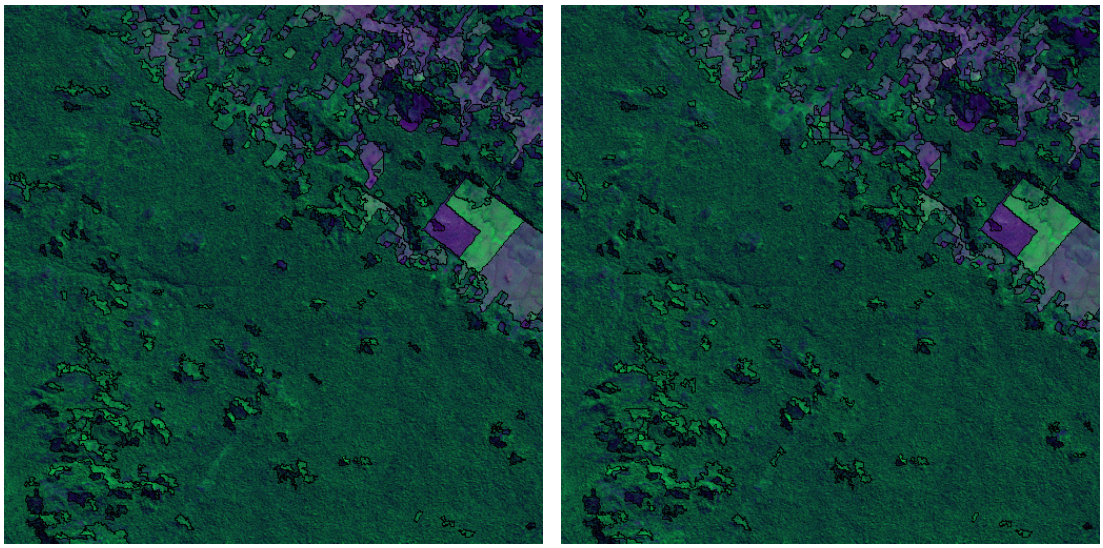


Fig. 4.24 - Comparação Visual – Resultado Serial e Resultado Paralelo - Imagem 512x512

A título de teste, executou-se a versão paralela mantendo as duas chamadas a função de junção de áreas visando uma comparação mais fiel com a versão serial, ou seja, o limiar de similaridade foi mantido em 30 e o limiar de área empregado na primeira chamada foi 5 e na segunda chamada foi 25.

Os resultados são apresentados na Tabela 4.28.

Tabela 4.28 – Comparação dos Resultados Serial X Paralelo – Limiar de Área = 5/25

Imagem	**Serial** **Número regiões**	**Paralelo** **Número regiões**	**Diferença número regiões (%)**
C256	74	75	1,4
C512	379	383	1,1
C1024	1494	1502	0,5
C2048	6215	6294	1,3
C4096	27737	27869	0,5

Nota-se uma pequena diferença entre o número de regiões de ambas as versões, devido à independência das janelas, o que ocasiona valores de média espectral diferente para cada janela. Já com relação ao tempo de processamento nota-se que o mesmo teve um aumento entre 40 e 60%, pois executando duas vezes a função de eliminação de regiões o processador mestre terá que dispender mais tempo de processamento. Na versão paralela se for executada apenas uma vez a função de eliminação de regiões (dentro dos processadores clientes) nota-se que a partição das regiões apresentam uma diferença aceitável, mas existe uma redução significativa no tempo de processamento, por estes motivos optou-se pela mudança algorítmica.

4.4 – Análise de Desempenho

4.4.1 – Desempenho das Etapas de Segmentação e Costura

Como citado no Capítulo 3, a análise de desempenho será baseada na derivação simplificada da Lei de Amdahl, visto que não se tem como contabilizar separadamente as porcentagens paralelizável e serial do programa.

As Figuras 4.25, 4.26, 4.27, 4.28 e 4.29 apresentam os resultados dos cálculos do speedup (ganho) e da eficiência do programa, de acordo com as Equações 2.20 e 2.21:

$$Sp = Tseq/Tp \tag{2.20}$$

Onde:

Tseq - tempo consumido por uma máquina seqüencial

Tp - tempo consumido por uma máquina paralela

$$E = Sp/P \tag{2.21}$$

Onde:

P – número de processadores empregados no processamento

Baseado na Tabela 4.27 as Figuras 4.25, 4.26, 4.27, 4.28 e 4.29, apresentam os valores de speedup e eficiência mediante a comparação dos tempos obtidos a partir do aumento do número de processadores clientes na execução total do processo, que inclui segmentação e costura.

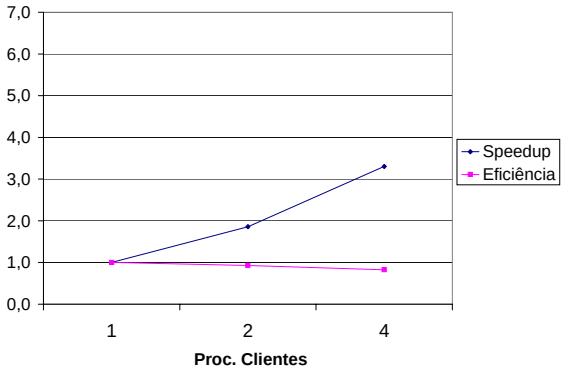


Fig. 4.25- Speedup e Eficiência
Imagem C256 – Limiar de Área 25
Tempo de Segmentação e Costura

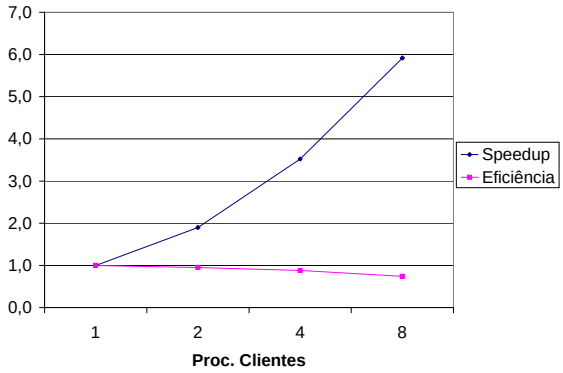


Fig. 4.26- Speedup e Eficiência
Imagem C512 – Limiar de Área 25
Tempo de Segmentação e Costura

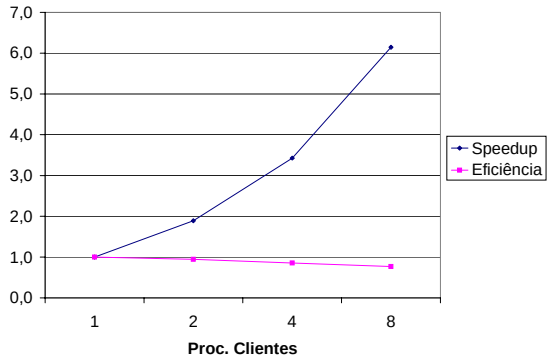


Fig. 4.27- Speedup e Eficiência Imagem C1024 – Limiar de Área 25 Tempo de Segmentação e Costura

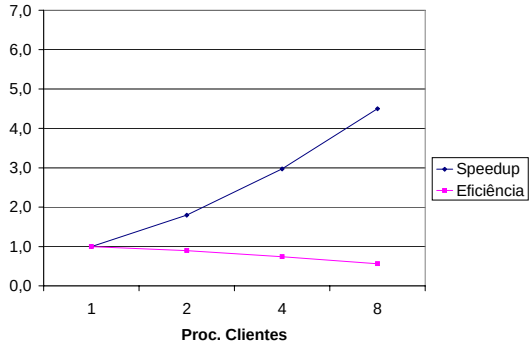


Fig. 4.28- Speedup e Eficiência Imagem C2048 – Limiar de Área 25 Tempo de Segmentação e Costura

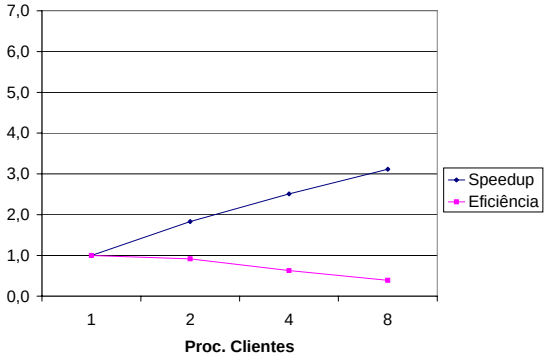


Fig. 4.29- Speedup e Eficiência Imagem C4096 – Limiar de Área 25 - Tempo de Segmentação e Costura

Observando as Tabelas 4.24, 4.25 e 4.26, nota-se que a medida que aumenta-se o número de processadores, o tempo total de processamento diminui consideravelmente. Pode-se notar também que a melhor situação, com relação ao tempo de processamento, é quando cada processador escravo tem que processar apenas uma janela, ou seja, o melhor caso é quando se tem número de janelas igual ao número de processadores escravos, mas este é um cenário ilusório, pois nas aplicações reais dificilmente ter-se-á número de processadores escravos igual ao número de janelas da imagem. Portanto na realidade o melhor caso é quando se tem o mínimo de comunicação. A diferença apresentada entre o número de regiões segmentadas será alvo de discussão no próximo item

4.4.2 – Desempenho da Etapa de Segmentação

Neste item será comparado o desempenho somente da etapa de segmentação através do cálculo do speedup e eficiência, baseados na Tabela 4.40 (tempo de segmentação X número de processadores). Vide Figuras 4.30, 4.31, 4.32, 4.33 e 4.34.

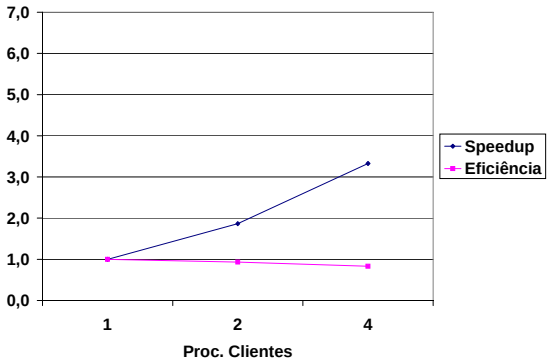


Fig. 4.30- Speedup e Eficiência
Imagem C256 – Limiar de Área 25
Tempo de Segmentação

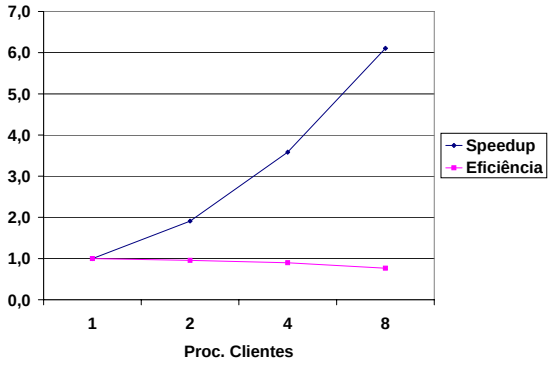


Fig. 4.31- Speedup e Eficiência
Imagem C512 – Limiar de Área 25
Tempo de Segmentação

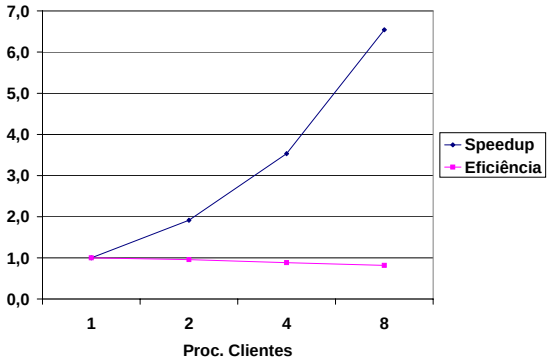


Fig. 4.32- Speedup e Eficiência
Imagem C1024 – Limiar de Área 25
Tempo de Segmentação

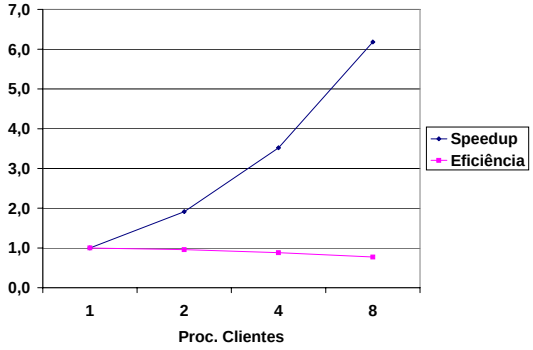


Fig. 4.33- Speedup e Eficiência
Imagem C2048 – Limiar de Área 25
Tempo de Segmentação

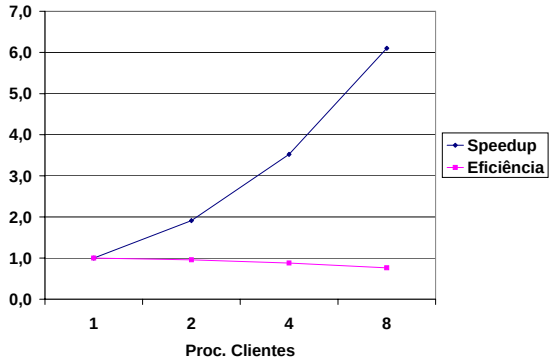


Fig. 4.34- Speedup e Eficiência Imagem C4096 – Limiar de Área 25 - Tempo de Segmentação

Comparando os gráficos de speedup e eficiência do processo de segmentação e costura das janelas com os gráficos de speedup e eficiência somente do processo de segmentação, pode-se notar que há um ganho maior no segundo caso. Isso porque todo processo de concatenação das janelas, atualização das listas de vizinhanças e alteração da imagem rotulada é feito no processador mestre logo após o mesmo receber todas as janelas segmentadas pelos processadores clientes.

Esse ganho poderia ser ampliado se o processo de costura das janelas fosse também paralelizado, sobrando com isso, para o processador mestre, uma tarefa pequena de costura final.

Analisando as Figuras 4.30, 4.31, 4.32, 4.33 e 4.34 nota-se a perda de eficiência do algoritmo paralelizado, devido ao desbalanceamento de trabalho entre os processadores, causada pela variação nos tempos de cálculo em cada janela. Isso poderia ser minimizado se fosse implementado um controle independente de cada janela onde assim que os processadores clientes devolvessem a janela processada, imediatamente receberiam outras e não necessitariam esperar que os demais processadores clientes também terminassem suas tarefas.

A Figura a seguir representa um possível esquema para a paralelização da função de costura janelas.

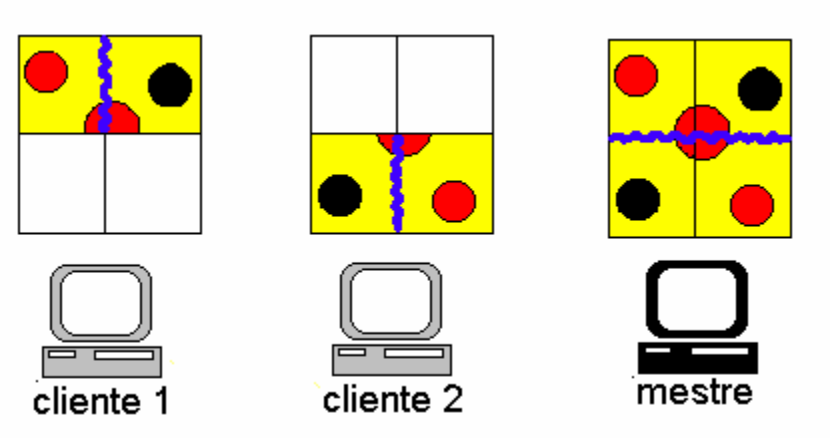


Fig. 4.35 – Possível Paralelização da Função de Costura Janelas

4.4.3 – Ganho Operacional

O ganho operacional pode ser analisado a partir das Figuras, 4.36, 4.37, 4.38, 4.39 e 4.40. As mesmas apresentam a relação entre o tempo de processamento da versão serial com a versão paralela.

Os dados utilizados nos gráficos correspondem as colunas tempo da versão serial e tempo da versão paralela da Tabela 4.27. Esses dados não devem ser usados como resultado da eficiência total do programa, pois existe uma diferença no algoritmo. Nota-se nas ilustrações a seguir que houve um ganho significativo em desempenho do sistema, tornando-o operacional e que este ganho aumenta a medida que aumenta-se o número de processadores clientes envolvidos no processamento.

Outra característica a ser observada é que quanto mais homogêneas forem as regiões vizinhas das janelas concatenadas, maior será o tempo de concatenação, pois ter-se-á mais pixels pertencentes a região em questão e todos os pixels serão analisados até que a região seja totalmente explorada e unida a região vizinha com características similares.

Nota-se também a saturação do ganho (Figura 4.40), pois à medida que se aumenta o número de processadores envolvidos, a porcentagem de ganho é reduzida. Isto

mostra que à medida que se aumenta o número de processadores aumenta-se o overhead de comunicação.

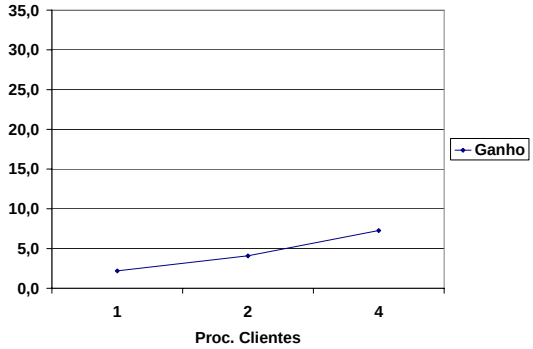


Fig. 4.36- Ganho Operacional
Imagem C256 – Serial X Paralelo

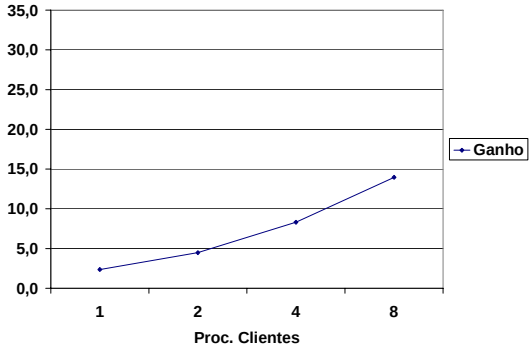


Fig. 4.37- Ganho Operacional
Imagem C512 – Serial X Paralelo

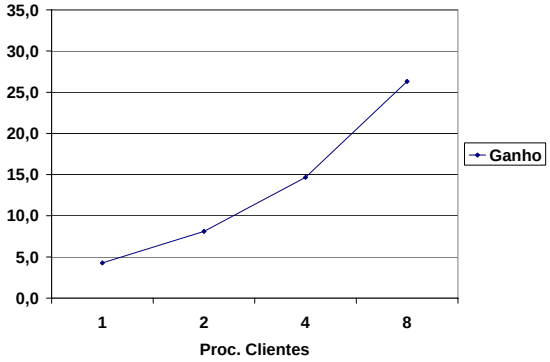


Fig. 4.38- Ganho Operacional
Imagem C1024 – Serial X Paralelo

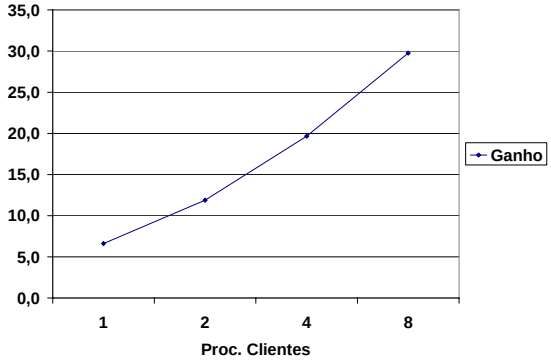


Fig. 4.39- Ganho Operacional
Imagem C2048 – Serial X Paralelo

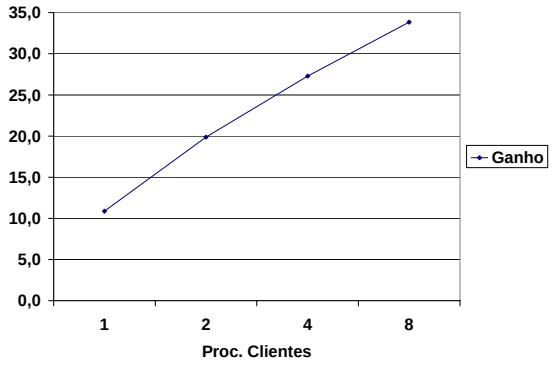


Fig. 4.40- Ganho Operacional Imagem C4096 – Serial X Paralelo

CAPÍTULO 5

CONCLUSÕES E RECOMENDAÇÕES

5.1 – Considerações Gerais

Conforme foi dito na introdução deste trabalho, sua finalidade foi de paralelizar o módulo de segmentação de imagens que é parte integrante do Sistema de Processamento de Informações Georeferenciadas – SPRING, desenvolvido pela Divisão de Processamento de Imagens – DPI/INPE, na tentativa de reduzir o tempo total de processamento de uma imagem de satélite, gerando um produto cartográfico de qualidade e gerando um código portável para diversas plataformas a um custo de implantação baixo.

Realmente verificou-se que este tipo de problema pode ser minimizado através da metodologia descrita neste trabalho, além de não necessitar de grandes mudanças se portado para outras plataformas.

Optou-se pela utilização do padrão MPI como biblioteca de paralelização por ser uma ferramenta que está em evidência entre os pesquisadores da área de processamento de alto desempenho, por ser de domínio público e por dispor de versões para diversas plataformas e sistemas operacionais.

Os testes foram desenvolvidos em plataforma PC, por disponibilizarem uma rede com dez máquinas de mesma configuração de hardware e ser a plataforma mais utilizada pelos usuários do SPRING.

Como o objetivo era paralelizar o código existente, a linguagem original, C++, foi mantida, o que dificultou o desenvolvimento, pois ainda se trata de uma abordagem nova, visto que a grande maioria das publicações e manuais de MPI existem para dar suporte a aplicações escritas em linguagem C e FORTRAN.

Com relação aos resultados obtidos na paralelização do código serial, foi observado que:

- A redução do tempo total de processamento é em média 22 vezes tornando o método operacional, pois o usuário poderá ter a resposta do seu trabalho em um tempo muito menor, o que agilizará consideravelmente a tomada de decisões;

- Aliado à redução do tempo, notou-se neste caso que a diferença entre o número de regiões segmentadas pelo programa serial e a versão paralela foi em média 8% menor, no entanto, em função da redução significativa do tempo o usuário poderá fazer testes com diferentes limiares e poderá fazer uma escolha;

- Falhas visíveis do processo de segmentação poderão ser ajustadas manualmente durante a classificação e edição das regiões;

- O número de regiões segmentadas está mais fortemente ligado ao valor do limiar de área no caso desta proposta paralela;

- Quanto maior for a área das regiões similares de contato entre as janelas , maior será o tempo empregado na fase de costura das mesmas;

- Dependendo do tamanho da imagem o tempo gasto no processo de costura pode ser maior que o tempo gasto no processo de segmentação da imagem;

- A um certo ponto o ganho com o aumento do número de processadores envolvidos sofre uma saturação e começa a reduzir;

- O custo de projetos como o PRODES poderia ser reduzido, pois o tempo hora/máquina e hora/homem despedido no processo de segmentação seria menor;

- A biblioteca de paralelização MPI é eficiente, de fácil aplicação e está disponível ao público em diversas versões que possibilitam sua implantação em várias plataformas e sistemas operacionais.

5.2 – Perspectivas

Algumas sugestões puderam ser anotadas pela autora ao longo da elaboração deste trabalho, para propostas de trabalhos futuros relacionados com a paralelização do processo de segmentação baseado no algoritmo de crescimento de regiões. São elas:

- Modificar o algoritmo original (seqüencial), pois o mesmo apresenta problemas na alocação de memória, o que dificulta otimizar ainda mais sua paralelização;

- Criar uma função de controle das janelas enviadas e recebidas pelo processador mestre, não necessitando aguardar que todos os processadores clientes acabem suas tarefas para distribuir mais dados para os processadores ociosos;

- No caso de ambientes formados por máquinas com configurações heterogêneas é aconselhado criar um módulo de balanceamento de carga, onde as janelas seriam previamente analisadas e distribuídas de acordo com o peso de seu processamento, ou seja, as janelas que exigem mais processamento deveriam ser enviadas para os processadores mais rápidos;

- Atribuir a responsabilidade de concatenação e costura das janelas aos processadores clientes, reduzindo ao máximo a carga de trabalho do processador mestre, através de uma estrutura de árvore para as costuras parciais.

- Prover uma modificação na estrutura do algoritmo paralelo, empregando funções complexas da biblioteca MPI, visando a otimização do processo;

- Portar o código para ambiente Windows NT e comparar os resultados aos obtidos no ambiente Linux;

- Integrar mais fortemente este trabalho ao SPRING, possibilitando aos usuários optarem por rodar seu código em vários processadores ou não.

REFERÊNCIAS BIBLIOGRÁFICAS

BADER, D.A.; JAJA, J.; HARWOOD, D.; DAVIS, S. L. **Parallel algorithms for image enhancement and segmentation by region growing with an experimental study.** Maryland: University of Maryland, May, 1995. 293p.

BAGLIETTO, P.; MARESCA, M.; MIGLIARDI, M.; ZINGIRIAN, N. Image processing on high-performace risc systems. **Proceedings of the I.E.E.E.**, v.84, p.917-929, July, 1996.

BANERJEE, U. **Dependence analysis for supercomputing.** New York: Kluwer Academic Publishers Norwell, 1988. 228p.

BARDER, D.A.; JAJA, J. **Simple: a methodology for programming high performace algoritms on clusters of symmetric multiprocessors (smps).** Maryland: Institute for Advanced Computer Studies, University of Maryland., May, 1997. 531p.

BATISTA, G. T.; MEDEIROS, J.S.; MELLO, E.M.K.; MOREIRA, J.C.; BINS, L.S. A new approach for deflorestation assessment. In: International Geoscience and Remotely-Sensing Symposium (IGARSS´96), Jun, 1996, Philadelphia. **Proceedings…** Philadelphia: IGARSS, 1996. p. 345-349.

BINS, L. S.; ERTHAL, G.J.; FONSECA, L.M.G.. Um método de classificação não supervisionado por regiões. In: Simpósio Brasileiro de Computação Gráfica - SIBGRAPI V, Jul, 1998, Florianópolis, Santa Catarina. **Anais...**.Florianópolis: SIBGRAPI´98. p.65-68, 1998.

BROWN, C. M., BALLARD, D. H. **Region Growing.** 2ª ed. New York: Prentice Hall Inc, 1982. 212p.

Centro Nacional de Processamento de Alto Desempenho (CENAPAD). **Introdução ao processamento paralelo.** Fortaleza, 1999. 45p.

CROSTA, A P. **Processamento digital de imagens de sensoriamento remoto**. Campinas: IG/UNICAMP, 1993. 203p.

FAGG, G. E., DONGARRA, J. J. PVMPI: an integration of the PVM and MPI systems. **Calculateurs Paralleles**, v. 2, p.42-51, Feb 1996.

FERRANTE, J. The program dependence graph and its use in optimization. **ACM Transactions Programming Languages and Systems**, v. 9, nº 3, 197p, July, 1987.

GEIST, G. A.; KOHL, J.A; PAPADOPOULOS, P.M. **PVM and MPI: a comparison of features.** USA: U.S. Department of Energy, ,May, 1996. 34p. (DE-AC05-96OR22464).

GONZALEZ R. C., WINTZ, P. **Digital image processing.** New York: Addison-Wesley, 2ª ed., 1987. 321p.

GONZALEZ, R. C., WOODS, R. E. **Digital image processing.** New York: Addison-Wesley, 1992. 270p.

GROPP, W.; LUSK, E.; SKJELEUM, A. **MPI: the complete reference- the mpi extensions.** Massachusetts: The MIT Press, v. 2, 1998. 523p.

HEIJDEN, F. V. **Image based measurement.** New York: John Wiley & Sons., 1994. 196p.

HWANG, K. **Advanced computer architecture. Parallelism, scalability, programmability.** New York: McGraw Hill Inc., 1993. P.3-69.

LEWIS, T. G., EL-REWINS, H. **Introdution to parallel computing.** New York: Prentice -Hall Inc., 1992. p. 1-23.

LILJA, D. J. **Exploiting the parallelism available in loops**. Minnessota: Computer Inc., v. 27, nº2, Feb. 1994. p.13-26.

MONTEIRO, A. M. V. **PARIMA: paralelismo e imagens. paralelização de alguns algoritmos básicos para a classificação automática de imagens digitais geradas por satélites de recursos naturais, radares orbitais e radares embarcados.** São José dos Campos: INPE, 1997.

PAL, N. R., PAL K. S. **A review on image segmentation techniques.** India: Machine Intelligence Unit, Indian Statistical Institute, v. 26, Sept., 1993. p.1277-1294.

PITAS, I. **Parallel algorithms for digital image processing, computer vision and neural networks.** New York: John Wiley, 1993. p. 1-24.

SHIMABUKURO, Y. E.; MELLO, E. M. K.; MOREIRA, J.C.; DUARTE, V.. **Segmentação e classificação da imagem sombra do modelo de mistura para mapear desflorestamento na amazônia**. São José dos Campos: Instituto Nacional de Pesquisas Espaciais (INPE), Maio, 1997. (INPE-6147-PUD/029).

SINGH, J. P.; HEMESSY, J. L., GUPTA, A. scaling parallel programs for multiprocessors: methodology and examples. **IEE Computer**, v. 18, p.42-50, July 1993.

SNIR, M. ; OTTO, S. W.; LEDERMAN, S. H.; WALKER, D. W.; DONGARRA, J. **MPI: the complete reference.** Massachusetts: The MIT Press, 1996. 643p.

Maui High Performance Computing Center (MPCC). **SP parallel programming workshop: introduction to parallel programming.** , 1997. Disponível em: <http://www.mhpcc.edu/training/workshop/html/parallel-intro/ParallelIntro.html>. . 1999, 16:30:15.

STALLMAN, R. M. **Using and porting GNU CC for version 2.7.2.** Iowa, 1995. 245p.

APÊNDICE A

LISTAGEM DO ARQUIVO PRINCIPAL

```
// inicialização das bibliotecas
#include        <stdio.h>
#include        <string.h>
#include        <math.h>
#include        <unistd.h>
#include        <sys/param.h>
#include        <cell11.hpp>
#include    <mpi++.h>

int
main (int argc, char *argv[])
  {
        // tipando variáveis
        double starttime,endtime;
        double Wtime();
        double jan_starttime, jan_endtime;
        char    dirima[33] = "/home/tico/parima/IMAGENS",
        dirout[32] = ".",
        arquivo [1000],
        nomima[1000];
        int     nban, nlin, ncol, sim, area;
        int num_janelas, num_passos;

        // inicialização do padrão MPI
        MPI::Init (argc, argv);

        // computando número de processos inicializados e processo corrente
         int rank = MPI::COMM_WORLD.Get_rank();
        int size = MPI::COMM_WORLD.Get_size();

        // testa possivel erro na inicialização do padrão MPI
         if (size <= 0)
        {
                printf("nao foi possivel inicializar o processo MPI %d ! \n", rank);
                return(0);
        }

        // verifica se o processo corrente corresponde ao processador  Mestre
        if(rank == 0)
```

```
{
        Image           *pima = NULL;
        pima = new Image[nban];

        printf("\n\nProcesso 0 iniciou!!!!!\n");

        // Solicita parâmetros de entrada
        printf("\nEntre com o numero de linhas : ");
        scanf("%d",&nlin);
        printf(" ",nlin);
        printf("\nEntre com o numero de colunas: ");
        scanf("%d",&ncol);
        printf(" ",ncol);

        // calcula quantidade de janelas
        int numWindowLines = ( nlin % TAMJAN_LIN ) ? ( nlin /
        TAMJAN_LIN + 1 ) : ( nlin / TAMJAN_LIN );
        int numWindowCols = ( ncol % TAMJAN_COL ) ? ( ncol /
        TAMJAN_LIN + 1 ) : ( ncol / TAMJAN_LIN );
        num_janelas = numWindowLines * numWindowCols ;

        printf("\nSERVER: num_janelas = %d\n",num_janelas);
        printf("SERVER: num_passos = %d\n",num_janelas/(size-1));

        // Solicita nome das imagens
        for( int i = 0; i < nban; i++ ){
                printf("\nEntre nome da imagem        : ");
                 strcpy (arquivo, dirima);
                strcat (arquivo, "/");
                scanf("%s", nomima);
                strcat (arquivo, nomima);

                // Verifica possível erro de inicialização das imagens
                if( pima[i].Init(arquivo, nlin, ncol, 1 ) == FALSE ){
                        printf("\nErro Image %s.Init()\n", pima[i].Name() );
                        exit(0);
                }
        }

        // Solicita parâmetros de entrada
        printf("\nEntre nome imagem rotulada   : ");
        scanf("%s", nomima );
        printf("\nEntre Limiar de area : ");
        scanf("%d", &area);
        printf("\n Limiar de area                        : ", area);
        printf("\nLimiar de Similaridade : ", sim);
```

```
            // Inicializa imagem resultante
            Image           irot;
        if( irot.Create(nomima, nlin, ncol, 4 ) == FALSE ){
                    printf("\nErro Image %s.Init()\n", irot.Name() );
                    exit(0);
            }
            // Inicia  contagem de tempo
            starttime=MPI::Wtime();

            //  Envia numero de janelas para cada Client:
            for (int dest=1; dest<size; dest++)

        MPI::COMM_WORLD.Send(&num_janelas,1,MPI::INT,dest,1001);

                    RGrowrg;

                    // chama função Server_Apply do arquivo CELL.CC onde irá
                    distribuir para o processadores clientes as janelas
                    particionadas
                    // rebece dos clientes imagem rotulada e lista s
                    rg.Server_Apply(pima, irot, nban, sim, area);

                    // computa tempo total de processamento
                    endtime= MPI::Wtime();

                    // imprime resultados
                    printf("\nSERVER: Tempo Total = %lf seg\n",endtime-
starttime);
                     int segundos= ((int)(endtime - starttime)) % 60;
    printf("\n\nTempo total = %d minutos  %d segundos\n\n", ((int)(endtime - starttime)) / 60, segundos);



        }

        // início das tarefas do processadores clientes
        else
        {
            // inicializa imagem na memória
             ImageMemory *pima;

            // para saber de quem recebeu a mensagem
            MPI::Status stat;
            int len = 0;
            int resp = 2;
```

```
// inicia contagem de tempo para os processadores clientes
starttime = MPI::Wtime();

//Recebe numero de janelas do mestre
MPI::COMM_WORLD.Recv(&num_janelas,1,MPI_INT,0,1001,stat);
num_passos = num_janelas / (size-1) ;

 pima = new ImageMemory[nban];
ImageMemory irot;
char* errmsg = new char[MPI::MAX_ERROR_STRING];

// verifica possível erro no recebimento da mensagem
if(!errmsg)
 {
        printf("\n Erro no new do errmsg");
        return FALSE;
}

RGrow rg[num_passos];

// inicia laço para controlar o número de passos
for (int passo=0 ; passo<num_passos; passo++)
{
        int bufsize2;
        jan_starttime = MPI::Wtime();

        // controla número de bandas que compõe a imagem
        for(int i = 0; i < nban; i++)
        {

                // recebe porção da imagem enviada pelo mestre
                MPI::COMM_WORLD.Recv(pima[i].buf,
                TAMJAN_LIN*TAMJAN_COL, MPI::LONG, 0, i,
                stat);
                bufsize2 = stat.Get_count(MPI::LONG);

                // verifica possível erro no recebimento
                if(stat.Get_error() != MPI::SUCCESS)
                {
                        MPI::Get_error_string(stat.Get_error(), errmsg,
len);
printf("\n[%d] erro no recebimento da matriz %s\n",rank,errmsg);

                }
        }
```

```
                    // inicializa variável que armazena lista
                    char *buffer = new char[50000];
                    memset(buffer,0,50000);

                     int currentwindow = passo * (size-1) + rank - 1 ;

                    // chama a função Client_Apply do arquido CELL.CC
                    // executa a segmentação da imagem
len = rg[passo].Client_Apply(pima, irot, nban, sim, area, currentwindow, buffer);


// envia a lista  para o mestre
                    MPI::COMM_WORLD.Send(buffer, len, MPI::CHAR, 0, 3);

                    // envia a iamgem rotulada para o mestre
MPI::COMM_WORLD.Send(irot.buf, TAMJAN_LIN*TAMJAN_COL, MPI::LONG, 0, 4);


// finaliza variáveis
                    delete buffer;
                    jan_endtime = MPI::Wtime();

                    // calcula e imprime os  tempos
printf("Client[%d] (Final do passo %d len=%d) Tempo=%lf seg\n",rank,passo,len,
                    jan_endtime-jan_starttime);

                }
                endtime = MPI::Wtime();
                printf("Client[%d] Tempo Client Total = %lf seg \n",rank,endtime-
starttime);
        }

 // finaliza o padrão MPI
MPI::Finalize();

}
```

APÊNDICE B

LISTAGEM DOS ARQUIVOS SECUNDÁRIOS

```
//     RGROW10.CC       //
//                     //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //


#include    <stdio.h>
#include    <string.h>
#include    <math.h>
#include    <unistd.h>
#include    <sys/param.h>
#include    <cell10.hpp>
#include <mpi++.h>


int
main (int argc, char *argv[])
  {
   double starttime,endtime;
   double Wtime();
      double jan_starttime, jan_endtime;

      MPI::Init (argc, argv);

   int rank = MPI::COMM_WORLD.Get_rank();
      int size = MPI::COMM_WORLD.Get_size();


   if (size <= 0)
      {
            printf("nao foi possivel inicializar o processo MPI %d !
\n", rank);
            return(0);
      }

char  dirima[33] = "/home/tico/parima",
            dirout[32] = ".",
            arquivo [1000],
            nomima[1000];

int   nban, nlin, ncol, sim, area;

int num_janelas, num_passos;

      nban=3;
      sim=30;
      area=25;
```

```
  if(rank == 0)
  {
            Image       *pima = NULL;
            pima = new Image[nban];

      printf("\n\nProcesso 0 INICIOU!!!!!\n");
      printf("\nEntre com o numero de linhas : ");
      scanf("%d",&nlin);
      printf("\nEntre com o numero de colunas: ");
      scanf("%d",&ncol);

      int numWindowLines = ( nlin % TAMJAN_LIN ) ?( nlin /
TAMJAN_LIN + 1 ) : ( nlin / TAMJAN_LIN );
      int numWindowCols = ( ncol % TAMJAN_COL ) ?( ncol / TAMJAN_LIN
+ 1 ) : ( ncol / TAMJAN_LIN );
      num_janelas = numWindowLines * numWindowCols ;
      printf("\nSERVER: num_janelas = %d\n",num_janelas);
      printf("SERVER: num_passos = %d\n",num_janelas/(size-1));

      for( int i = 0; i < nban; i++ ){
            printf("\nEntre nome da imagem         : ");
                strcpy (arquivo, dirima);
                strcat (arquivo, "/");
            scanf("%s", nomima);
                strcat (arquivo, nomima);
            if( pima[i].Init(arquivo, nlin, ncol, 1 ) == FALSE ){
                  //nbytes=1 aquisicao=1
                  printf("\nErro Image %s.Init()\n", pima[i].Name()
);
                  exit(0);
            }
      }
      printf("\nEntre nome imagem rotulada   : ");
      scanf("%s", nomima );

      Image       irot;

      if( irot.Create(nomima, nlin, ncol, 4 ) == FALSE ){
            printf("\nErro Image %s.Init()\n", irot.Name() );
            exit(0);
      }

   // **** iniciando contagem de tempo *****
   starttime=MPI::Wtime();

// - Envia numero de janelas para cada Client:
      fflush(stdout);
      for (int dest=1; dest<size; dest++)
        MPI::COMM_WORLD.Send(&num_janelas,1,MPI::INT,dest,1001);
      RGrow rg;

      rg.Server_Apply(pima, irot, nban, sim, area);

   endtime= MPI::Wtime();
printf("\nSERVER: Tempo Total = %lf seg\n",endtime-starttime);
```

```
   int segundos= ((int)(endtime - starttime)) % 60;
   printf("\n\nTempo total = %d minutos  %d segundos\n\n",
((int)(endtime - starttime)) / 60, segundos);

  }
  else
  {
   ImageMemory *pima;
   MPI::Status stat;           // para saber de quem recebeu a
mensagem
   int len = 0;
   int resp = 2;

   starttime = MPI::Wtime();
//Celso - Recebe numero de janelas
      MPI::COMM_WORLD.Recv(&num_janelas,1,MPI_INT,0,1001,stat);
      num_passos = num_janelas / (size-1) ;
//DEBUG     printf("Client[%d]: num_passos= %d\n",rank,num_passos);

   pima = new ImageMemory[nban];
   ImageMemory irot;
   char* errmsg = new char[MPI::MAX_ERROR_STRING];

   if(!errmsg)
   {
      printf("\n Erro no new do errmsg");
      return FALSE;
   }

   RGrow rg[num_passos];
  for (int passo=0 ; passo<num_passos; passo++)
  {
//DEBUG   printf("\nClient[%d]: Iniciando passo %d\n",rank,passo);
//DEBUG   fflush(stdout);
   int bufsize2;

   jan_starttime = MPI::Wtime();
   for(int i = 0; i < nban; i++)
   {
         MPI::COMM_WORLD.Recv(pima[i].buf, TAMJAN_LIN*TAMJAN_COL,
MPI::LONG, 0, i, stat);

      bufsize2 = stat.Get_count(MPI::LONG);

      if(stat.Get_error() != MPI::SUCCESS)
      {
            MPI::Get_error_string(stat.Get_error(), errmsg, len);
         printf("\n[%d] erro no recebimento da matriz
%s\n",rank,errmsg);
      }

      MPI::COMM_WORLD.Send(&resp, 1, MPI::INT, 0, i);
   }


   char *buffer = new char[50000];
```

```
   memset(buffer,0,50000);
// len = rg[passo].Client_Apply(pima, irot, nban, sim, area, rank,
buffer);
   int currentwindow = passo * (size-1) + rank - 1 ;
   len = rg[passo].Client_Apply(pima, irot, nban, sim, area,
currentwindow, buffer);
   MPI::COMM_WORLD.Send(buffer, len, MPI::CHAR, 0, 3);
   MPI::COMM_WORLD.Send(irot.buf, TAMJAN_LIN*TAMJAN_COL, MPI::LONG,
0, 4);
   jan_endtime = MPI::Wtime();
   printf("Client[%d] (Final do passo %d len=%d) Tempo=%lf
seg\n",rank,passo,len,
                        jan_endtime-jan_starttime);
   fflush(stdout);
  }
   endtime = MPI::Wtime();
   printf("Client[%d] Tempo Client Total = %lf seg \n",rank,endtime-
starttime);
}


MPI::Finalize();

}
```

```
//     CELL10.CC         //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //



#include <stdio.h>
#include <string.h>
#include <math.h>
#include <image10.hpp>
#include <cell10.hpp>
#include <stdlib.h>
#include <mpi++.h>


int cellsort(const void* e1, const void* e2)
{
      Cell *c1 = (Cell*)e1;
   Cell *c2 = (Cell*)e2;

   if(c1->Id() < c2->Id())
      return -1;
   else if(c1->Id() > c2->Id())
      return 1;
   else
      return 0;
}


//------------------------------ REGPROX ---------------------------
--------

CloserCells :: CloserCells()
{
      for(short i = 0; i < MAXCLOSECELLS; i++ ){
            cmin[i] = NULL;
            dmin[i] = 1000000.;
      }
}


void
CloserCells :: Insert( Cell *cell, float dist )
{
Cell  **pci,
      **p;
float *di,
      *d;
      for( pci = &cmin[0], di = &dmin[0]; pci <
&cmin[MAXCLOSECELLS]; pci++, di++ ){
            if( *pci == NULL ){
                  *pci = cell;
                  *di  = dist;
                  return;
            }
```

```
            if( dist <= *di ){
                  if( dist == *di && (*pci)->Id() < cell->Id() )
                        continue;
                  if( pci < &cmin[MAXCLOSECELLS] ){
                        for( p = &cmin[MAXCLOSECELLS-1], d =
&dmin[MAXCLOSECELLS-1]; p > pci; p--, d-- ){
                              *p = *(p-1);
                              *d = *(d-1);
                        }
                  }
                  *pci = cell;
                  *di  = dist;
                  return;
            }
      }
}

void
CloserCells :: Update( Cell *cell, float dist )
{
char  ok = 1;
Cell  **pci,
      **pc0,
      **pc01,
      **pc02;

float *di,
      *d0,
      *d01;
      pc0 = NULL;
      for( pci = &cmin[0], di = &dmin[0]; pci <
&cmin[MAXCLOSECELLS]; pci++, di++ ){
            if( *pci == NULL ){
                  pc0 = pci;
                  d0  = di;
            }else if( (*pci)->Dead() ){
                  *pci = NULL;
                  pc0 = pci;
                  d0  = di;
            }else if( *pci == cell ){
                  *pci = NULL;
                  pc0  = pci;
                  d0   = di;
            }

            if( ok && dist <= *di ){
                  if( dist == *di ){
                        if( *pci == NULL ){
                              if( pci < &cmin[MAXCLOSECELLS-1] )
                                    continue;
                        }else if( (*pci)->Id() < cell->Id() )
                              continue;
                  }
                  if( pc0 != NULL ){
                        for( pc01 = pc0, d01 = d0; pc01 < pci-1;
pc01++, d01++ ){
```

```
                                *pc01 = *(pc01+1);
                                *d01  = *(d01+1);
                        }
                        *pc01 = cell;
                        *d01  = dist;
                }else if( pci < &cmin[MAXCLOSECELLS-1] ){
                        for( pc01 = pci+1, d01 = di+1; pc01 <
&cmin[MAXCLOSECELLS-1]; pc01++, d01++ ){
                                if( (*pc01) == NULL )
                                        break;
                                if( (*pc01)->Dead() )
                                        break;
                                if( *pc01 == cell )
                                        break;
                        }
                        for( pc02 = pc01; pc02 > pci; pc02-- ){
                                *pc02 = *(pc02-1);
                                *d01  = *(d01-1);
                                d01--;
                        }
                        *pci = cell;
                        *di  = dist;
                }else{
                        *pci = cell;
                        *di  = dist;
                }
                ok   = 0;
        }
    }
return;
}

Cell*
CloserCells :: Minimum( float& dist )
{
    if( cmin[0] != NULL ){
        if( !cmin[0]->Dead() ){
            dist = dmin[0];
            return cmin[0];
        }
    }
    for( short i = 1; i < MAXCLOSECELLS; i++ ){
        if( cmin[i] != NULL )
            if( !cmin[i]->Dead() ){
                dist = dmin[i];
                return cmin[i];
            }
    }
return NULL;
}

void
CloserCells :: Adjust()
{
    for( short i = 0; i < MAXCLOSECELLS; i++ )
        if( cmin[i] != NULL )
```

```
                if( cmin[i]->Dead() )
                      cmin[i] = NULL;
}



//------------------------------ LISTAREG --------------------------
--------

long
CellList :: Position( long id )
{
long  pi = 0L;
long  pf = Size()-1;
long  pm;
Cell* c;

      if( Size() <= 0L )
            return 0L;

      do{   pm = (pi+pf)/2;
            c = (Cell*)Get(pm);
            if( id < c->Id() )
                  pf = pm-1;
            else
                  pi = pm+1;

      }while( pi <= pf && c->Id() != id);

return pm;
}

long
CellList :: Search( long id )
{
long pos;
      if( Size() <= 0L )
            return -1L;
      pos = Position(id);
      if( Get(pos)->Id() == id )
            return pos;
return -1L;
}


int
CellList :: AddCell( Cell *c )
{
Cell* cl;
long  pos;

      if( c == NULL )
            return FALSE;
      if( c->Dead() )
            return FALSE;
```

```
	pos = Position( c->Id() );
	cl = (Cell*)Get(pos);
	if( cl == NULL ){
		(*this)[pos] = c;
		return TRUE;
	}
	if( cl->Dead() ){
		(*this)[pos] = c;
		return TRUE;
	}
	if( cl->Id() > c->Id() ){
		if( pos > 0L && ( (Cell*)Get(pos-1) )->Dead() ){
			(*this)[pos-1] = c;
			return TRUE;
		}
		if( Add( c, pos ) == FALSE )
			return FALSE;
	}
	else if( cl->Id() < c->Id() ){
		if( pos < Size()-1 && ( (Cell*)Get(pos+1) )->Dead() ){
			(*this)[pos+1] = c;
			return TRUE;
		}
		if( Add( c, pos+1 ) == FALSE )
			return FALSE;
	}
return TRUE;
}

int
CellList :: InsertCell( Cell *c )
{
	if( c == NULL )
		return FALSE;
	if( c->Dead() )
		return FALSE;
	return Add( c, Size() );
}


void
CellList :: Adjust()
{
Cell  *c;
	for( long i = 0; i < Size(); i++ ){
		if( ( c = (Cell*)Get( i ) ) != NULL )
			if( c->Dead() )
				(*this)[i] = NULL;
	}
	Compress();
	Shrink();
}


void
CellList :: Save ( char* buf, int& len )
```

```
{
      int auxlen = 0;
      len = sprintf(buf, "%ld", Size());
   buf += len;
   for (long i=0; i < Size(); i++)
   {
      Get(i)->Save(buf, auxlen);
      len += auxlen;
      buf += auxlen;
   }
   *buf = 0;
   len++;   // o NULL final
}



void
CellList :: Load ( char* buf, int index, long WLines,long WColumns)
{
      long n = 0;

   sscanf(buf, "%ld", &n);
   strtok(buf, "\n");   // indicador da 1.a Cell
      buf = strtok(NULL, " ");      // consome o \n e posiciona após
   Cell *tempv = new Cell[n];

   for (int i = 0; i < n; i++)
      buf = tempv[i].Load(buf, index, WLines, WColumns);

   qsort(tempv, n, sizeof(Cell), cellsort);
   for (i = 0; i < n; i++)
   {
      Cell* c = &tempv[i];
      for(int j = 0; j < c->nsize; j++)
      {
            Cell t, *tc;
         t.Idnumber = c->tempid[j];
         tc = (Cell*)bsearch(&t, tempv, n, sizeof(Cell), cellsort);
         if(tc)
            c->Neighbors->InsertCell(tc);
      }

      InsertCell(c);
   }
}



//---------------------------------Cell ------------------------------
----


Cell :: Cell( unsigned long* tuple, long id, short lin, short col,
short nban )
{
      Stat          = 0;
```

```
      Idnumber      = id;
      Npix          = 1;
      Nban          = nban;
      LinMax        = lin;
      LinMin        = lin;
      ColMax        = col;
      ColMin        = col;
   nsize                 = 0;
      Neighbors     = new CellList;
      Media         = new float[nban];
      PreviousMedia = new float[nban];
      Cc            = new CloserCells;
      if ( PreviousMedia == NULL || Media == NULL || Neighbors ==
NULL || Cc == NULL )
            return;
      for( int ban = 0; ban < nban; ban++ )
            Media[ban] = PreviousMedia[ban] = (float) tuple[ban];
}

Cell :: Cell()
{
   Stat          = 0;
      Idnumber      = 0;
      Npix          = 1;
      Nban          = 3;
      LinMax        = 0;
      LinMin        = 0;
      ColMax        = 0;
      ColMin        = 0;
   nsize                 = 0;
      Neighbors     = new CellList;
      Media         = new float[Nban];
      PreviousMedia = new float[Nban];
      Cc            = new CloserCells;
      if ( PreviousMedia == NULL || Media == NULL || Neighbors ==
NULL || Cc == NULL )
   {
      int uy=8;
      uy++;
            return;
      }
      for( int ban = 0; ban < Nban; ban++ )
            Media[ban] = PreviousMedia[ban] = 0.0;
}


int
Cell :: AddNeighbor( Cell *c, float dist )
{
      if( c == NULL )
            return FALSE;
      if( c->Dead() )
            return FALSE;
      Cc->Update( c, dist );
return Neighbors->AddCell( c );
}
```

```
int
Cell :: InsertNeighbor( Cell* c, float dist )
{
      if( c == NULL )
            return FALSE;
      if( c->Dead() )
            return FALSE;
      Cc->Insert( c, dist );
return Neighbors->InsertCell( c );
}



void
Cell :: Save( char* buf, int& len )
{
Cell  *c;
len = 0;

   len = sprintf( buf, "\n%ld %d %d %d %d %ld %d %d %g", Idnumber,
ColMin, ColMax, LinMin, LinMax, Npix, Nban, Stat, delta );
      for(  int ban = 0; ban < Nban; ban++ )
            len += sprintf( &buf[len], " %f", Media[ban] );
      len += sprintf( &buf[len], " %ld", Neighbors->Size() );
      for( long i = 0L; i < Neighbors->Size(); i++ ){
            c = Neighbors->Get( i );
            if( !c->Dead() )
                  len += sprintf( &buf[len], " %ld", c->Idnumber );
   }

}

char*
Cell :: Load(char* buf, int janela, long WLines, long WColumns)
{
    Cell  *c;
    int   TStat;

    sscanf( buf, "%ld", &Idnumber);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &ColMin);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &ColMax);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &LinMin);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &LinMax);
    buf = strtok(NULL, " ");

    // Modificar  indexacao para adequar janela a imagelab total
```

```
    LinMin = (LinMin + (janela / WColumns) * TAMJAN_LIN);
    LinMax = (LinMax + (janela / WColumns )* TAMJAN_LIN);
    ColMin = (ColMin + (janela % WColumns) * TAMJAN_COL);
    ColMax = (ColMax + (janela % WColumns) * TAMJAN_COL);

    sscanf( buf, "%ld", &Npix);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &Nban);
    buf = strtok(NULL, " ");

    sscanf( buf, "%d", &TStat);
    buf = strtok(NULL, " ");
    Stat=TStat;

    sscanf( buf, "%f", &delta);
    buf = strtok(NULL, " ");

    for(  int ban = 0; ban < Nban; ban++ )
    {
            sscanf( buf, "%f", &Media[ban] );
      buf = strtok(NULL, " ");
      PreviousMedia[ban]=Media[ban];
    }

    sscanf(buf,"%ld", &nsize);
    tempid = new long[nsize];

    for (long i=0; i < nsize - 1; i++)
    {
      buf = strtok(NULL, " ");
      sscanf(buf, "%ld", &tempid[i]);
    }

    buf = strtok(NULL, "\n"); // o último é \n
    sscanf(buf, "%ld", &tempid[i]);

    buf = strtok(NULL, " ");  // o último é \n

    return buf;   // retorna ponteiro para o início da próxima Cell
ou NULL se for o fim.
}


   int
Cell :: Merge( Cell *c )
{
Cell  *cviz;
float dist;
long  area;
long  i;

      if( c == NULL )
            return FALSE;
      if( LinMax < c->LinMax ) LinMax = c->LinMax;
      if( LinMin > c->LinMin ) LinMin = c->LinMin;
```

```
      if( ColMax < c->ColMax ) ColMax = c->ColMax;
      if( ColMin > c->ColMin ) ColMin = c->ColMin;

      area  = Npix + c->Npix;
      delta = (float)0;

      for( int ban = 0; ban < Nban; ban++ ){
            Media[ban] = (Media[ban] * Npix + c->Media[ban] * c-
>Npix) / area;
            dist = Media[ban] - PreviousMedia[ban];
            delta += dist * dist;
      }
      Npix = area;

      c->Kill();

      if( delta > (float)1 ){                    // atualizando
vizinhanca
            Cc->Reset();
            for( int ban = 0; ban < Nban; ban++ )
                  PreviousMedia[ban] = Media[ban];
            for( i = 0; i < NeighborhoodSize(); i++ ){
                  cviz = Neighbors->Get( i );
                  if( cviz == NULL )
                        return FALSE;
                  if( cviz->Dead() )
                        continue;
                  dist = Distance( cviz );
                  Cc->Insert( cviz, dist );
                  cviz->Cc->Update( this, dist );
            }
      }

      for( i = 0L; i < c->NeighborhoodSize(); i++ ){
            cviz = c->Neighbors->Get( i );
            if( cviz == NULL )
                  return FALSE;
            if( cviz != this && !cviz->Dead()){
                  dist = Distance( cviz );
                  if( cviz->AddNeighbor( this, dist ) == FALSE )
                        return FALSE;
                  if( AddNeighbor( cviz, dist ) == FALSE )
                        return FALSE;
            }
      }
return TRUE;
}


float
Cell :: Distance( Cell *c )
{
float diff;
float dist;

      if( c == NULL )
```

```
            return (float)100000;

      dist = 0.;
      for( short ban = 0; ban < Nban; ban++ ){
            diff  = PreviousMedia[ban] - c->PreviousMedia[ban];
            dist += (diff * diff);
      }
return dist;
}


Cell*
Cell :: ClosestNeighbor( float& dist )
{
Cell  *c;

      if( Neighbors->Size() <= 0L )
            return NULL;

      if( ( c = Cc->Minimum( dist ) ) != NULL )
            return c;

      Cc->Reset();

      for( long i = 0L; i < Neighbors->Size(); i++ ){
            c = Neighbors->Get( i );
            if( !c->Dead() ){
                  dist = Distance( c );
                  Cc->Insert( c, dist );
            }
      }
      c = Cc->Minimum( dist );
return c;
}

//--------------------------------- RGrow --------------------------
--


RGrow :: RGrow()
{
      Imagein       = NULL;
      Imagelab      = NULL;
      ImageinM     = NULL;
      ImagelabM    = NULL;
      ListCell      = NULL;
      tuple         = NULL;
      WindowLines   = 0;
      WindowColumns = 0;
      Nlin          = 0;
      Ncol          = 0;
      Nban          = 0;
      Areamin       = 0;
      Difsim            = (float)0;
}
```

```
void
RGrow :: Clear()
{
Cell  *c;

      if( Imagein    != NULL ) Imagein  = NULL;
      if( Imagelab   != NULL ) Imagelab = NULL;
      if( ImageinM  != NULL ) ImageinM  = NULL;
      if( ImagelabM != NULL ) ImagelabM = NULL;

      if( tuple      != NULL ) { delete tuple;      tuple      =
NULL; }
      Nlin       = 0;
      Ncol       = 0;
      Nban       = 0;
      Areamin    = 0;
      Difsim     = (float)0;

}


CellList*
RGrow :: InitWindow(long CurrentWindow)

{
   int            lin, col;
   long           ind;
   unsigned long  idcell;
   float          dist;
   Cell           *c;
   unsigned long WindowOffset =
CurrentWindow*TAMJAN_LIN*TAMJAN_COL+1L;
   idcell = WindowOffset;
      ilin = 0;
      icol = 0;
      flin = TAMJAN_LIN-1;
      fcol = TAMJAN_COL-1;
      sizelin = TAMJAN_LIN;
      sizecol = TAMJAN_COL;

      //**** criei a WindowCelll aqui *****"
      Cell **WindowCell;
      if( ( WindowCell = new Cell*[TAMJAN_LIN*TAMJAN_COL] ) == NULL
){
            printf("\nRGrow::Apply >>> V\n");
            exit(0);
      }

      for( lin = ilin; lin <= flin; lin++ ){
            for( col = icol; col <= fcol; col++ ){
                        (*ImagelabM)(lin,col) = idcell;

                  for( int ban = 0; ban < Nban; ban++ )
                        tuple[ban] = (*ImageinM[ban])(lin,col);
                  WindowCell[idcell-WindowOffset] = new Cell( tuple,
idcell, lin, col, Nban);
```

```
                  if( WindowCell[idcell-WindowOffset] == NULL ){
                        for( int i = 0; i < idcell-WindowOffset; i++
)
                              delete WindowCell[i];
                        printf("\nRGrow::InitWindow >>> I\n");
                        exit(0);
                  }
                  idcell++;
            }
      }

      // Take the neighborhood of regions lying inside the window
      for( lin = ilin; lin <= flin; lin++ )
            for( col = icol; col <= fcol; col++ ){
                  ind = (lin - ilin) * sizecol + ( col - icol );
                  c = WindowCell[ind];
                  if( col < fcol ){
                        dist = WindowCell[ind]->Distance(
WindowCell[ind+1] );
                        if( WindowCell[ind]-
>InsertNeighbor(WindowCell[ind+1], dist ) == FALSE ){
                              printf("\nRGrow::InitWindow >>>
II\n");
                              exit(0);
                        }
                        if( WindowCell[ind+1]-
>InsertNeighbor(WindowCell[ind], dist ) == FALSE ){
                              printf("\nRGrow::InitWindow >>>
III\n");
                              exit(0);
                        }
                  }
                  if( lin < flin ){
                        dist = WindowCell[ind]->Distance(
WindowCell[ind+sizecol] );
                        if( WindowCell[ind]-
>InsertNeighbor(WindowCell[ind+sizecol], dist ) == FALSE ){
                              printf("\nRGrow::InitWindow >>>
IV\n");
                              exit(0);
                        }
                        if( WindowCell[ind+sizecol]-
>InsertNeighbor(WindowCell[ind], dist ) == FALSE ){
                              printf("\nRGrow::InitWindow >>> V\n");
                              exit(0);
                        }
                  }
            }


// ********* criei aqui ListCellWindow ********
        CellList *ListCellWindow;
      ListCellWindow = new CellList;
      if( ListCellWindow == NULL ){
            printf("\nRGrow::Apply >>> IVii\n");
            exit(0);
```

```
      }

      // Add Regions in WindowCell in the list ListCellWindow;
   for( ind = 0; ind < sizelin * sizecol; ind++ )
            ListCellWindow->InsertCell( WindowCell[ ind ] );

   if( WindowCell != NULL ) { delete WindowCell; WindowCell = NULL;
}

   return ListCellWindow;
}


int
RGrow :: MergeMutuallyClosestCells(CellList* ListCellWindow)
{
float dist;
float diff;
char  status;
Cell  *c,
      *cgo,
      *cback;
long  i;

 for( int step = 5; step > 0 ; step-- ){
      diff = Difsim / ( step * step );
      do {
            status  = 0;
            // ****for( i = 0L; i < ListCell->Size();  i++ ){
***troquei abaixo
            for( i = 0L; i < ListCellWindow->Size();  i++ ){
                  c = ListCellWindow->Get( i );
                  while( !c->Dead() ){
                        if( ( cgo = c->ClosestNeighbor(dist) ) ==
NULL )
                              break;
                        if( ( cback = cgo->ClosestNeighbor(dist) )
== NULL ){

      printf("\nRGrow::MergeMutuallyClosestCells >>> I\n");
                              exit(0);
                        }
                        if( c == cback ){
                              if( dist <= diff  ){
                                    c = MergeCells( c, cgo );
                                    if( c == NULL ){

      printf("\nRGrow::MergeMutuallyClosestCells >>> II\n");
                                          exit(0);
                                    }
                                    status = 1;
                              }
                              else{ break; }
                        }
                        else{  break; }
                  }
```

```
            }
            Adjust(ListCellWindow);
      }while( status );
}
return TRUE;
}


int
RGrow :: ConcatLR(CellList* ListCellLeft, CellList* ListCellRight,
int ilin, int icol)
{
unsigned long     idcell,
            idwindow,
            cellid;
Cell        *cell,
            *cwindow;
float             dist;

            cellid = 0L;
            flin = ilin + TAMJAN_LIN - 1;
            if (flin>=Nlin) flin=Nlin-1;
            for( int lin = ilin; lin <= flin; lin++ ){
                  idcell = (*Imagelab)(lin,icol-1);
                  idwindow = (*Imagelab)(lin,icol);
                  if( cellid != idcell ){
                        if( ( cell = ListCellLeft->Find( idcell ) )
== NULL ){
                        printf("\nRGrow::InitWindow >>> VI\n");
                              exit(0);
                        }
                        cellid = cell->Id();
                  }
                  cwindow = ListCellRight->Find(idwindow);

                  if(cwindow==NULL){
                        printf("\ncwindow nao encontrado!");
                        exit(0);
                  }
                  dist = cell->Distance( cwindow );
                  if( cell->AddNeighbor( cwindow, dist ) == FALSE){
                        printf("\nRGrow::InitWindow >>> VII\n");
                        exit(0);
                  }
                  if( cwindow->AddNeighbor( cell, dist ) == FALSE ){
                        printf("\nRGrow::InitWindow >>> VIII\n");
                        exit(0);
                  }
            }
return TRUE;
}


int
RGrow :: ConcatUL(CellList* ListCellUp, CellList* ListCellLow, int
ilin, int icol)
```

```
{
unsigned long     idcell,
            idwindow,
            cellid;
Cell        *cell,
            *cwindow;
float             dist;

         cellid = 0L;
            fcol = icol + TAMJAN_COL - 1;
            if (fcol>=Ncol) fcol=Ncol-1;
            for( int col = icol; col <= fcol; col++ ){
                  idcell = (*Imagelab)(ilin-1, col);
                  idwindow = (*Imagelab)(ilin, col);

                  if( cellid != idcell ){
                        if( ( cell = ListCellUp->Find( idcell ) ) ==
NULL ){
                              printf("\nRGrow::InitWindow >>>
IX\n");
                              exit(0);
                        }
                        cellid = cell->Id();
                  }
                  cwindow = ListCellLow->Find(idwindow);
                  dist = cell->Distance( cwindow );
                  if (cwindow==NULL){
                  printf("\nRGrow::InitWindow NULL>>> X\n");
                        exit(0);
                  }
                  if( cell->AddNeighbor( cwindow, dist ) == FALSE){
                        printf("\nRGrow::InitWindow >>> X\n");
                        exit(0);
                  }
                  if( cwindow->AddNeighbor( cell, dist ) == FALSE ){
                        printf("\nRGrow::InitWindow >>> XI\n");
                        exit(0);
                  }
            }

      return TRUE;
      }



int
RGrow :: MergeSimilarCellsDisk(CellList* ListCell)
{
float d1;
float diff;
char  status;
Cell  *c,
      *cgo,*cback;
long i;
for( int step = 5; step > 0 ; step-- ){
      diff = Difsim / ( step * step );
```

```
      do {
            status  = 0;
            for( i = 0L; i < ListCell->Size();  i++ ){
                  c = ListCell->Get( i );
                  while( !c->Dead() ){
                        if( ( cgo = c->ClosestNeighbor(d1) ) == NULL
)
                              break;
                        if( ( cback = cgo->ClosestNeighbor(d1) ) ==
NULL )
                        {
                                printf("\n erro no
mergesimilarcelssdisk");
                              exit(0);
                         }
                         if (c==cback){
                              if( d1 <= diff ){
                                    c = MergeCellsDisk( c, cgo );
                                    if( c == NULL ){

      printf("\nRGrow::MergeSimilarClosestCells >>> I\n");
                                          exit(0);
                                    }
                                    status = 1;
                              }
                              else{  break; }
                         }
                         else {break;}
                  }
            }
            Adjust(ListCell);
      }while( status );
}
return TRUE;
}



int
RGrow :: MergeSmallCells(long area,CellList* ListCellWindow)
{
char  status;
Cell  *c,
      *cgo,
      *cback;
float dist;
float diff;
long  i;

if( area > Areamin )area = Areamin;

// Elimina regioes pequenas atraves do merging com a regiao vizinha
mais proxima
for( int step = 1; step <= 6; step++ ){
      diff = (float) sqrt((double) Difsim ) + step * 2;
      diff = diff * diff;
```

```
      do {  status  = 0;
            for( i = 0L; i < ListCellWindow->Size();  i++ ){
                  c = ListCellWindow->Get( i );
                  if( c->Dead() || c->Area() > area )
                        continue;
                  if( ( cgo = c->ClosestNeighbor(dist) ) == NULL )
                        break;
                  if( ( cback = cgo->ClosestNeighbor(dist) ) == NULL
){
                        printf("\nRGrow::MergeSmallCellsCells >>>
I\n");
                        exit(0);
                  }
                  if( c != cback )
                        continue;
                  if( dist <= diff || step > 5 ){
                        if( ( c = MergeCells( c, cgo ) ) == NULL ){
                              printf("\nRGrow::MergeSmallCellsCells
>>> II\n");
                              exit(0);
                        }
                        status = 1;
                  }
            }
            Adjust(ListCellWindow);
      }while( status );
}

for( step = 1; step <= 6; step++ ){
      diff = (float) sqrt((double) Difsim ) + step * 2;
      diff = diff * diff;
      do {  status  = 0;
            for( i = 0L; i < ListCellWindow->Size();  i++ ){
                  c = ListCellWindow->Get( i );
                  if( c->Dead() || c->Area() > area )
                        continue;
                  if( ( cgo = c->ClosestNeighbor(dist) ) == NULL )
                        break;
                  if( dist <= diff || step > 5 ){
                        if( ( c = MergeCells( c, cgo ) ) == NULL ){
                              printf("\nRGrow::MergeSmallCellsCells
>>> III\n");
                              exit(0);
                        }
                        status = 1;
                  }
            }
            Adjust(ListCellWindow);
      }while( status );
}
return TRUE;
}

int
RGrow :: MergeSmallCellsDisk(long area,CellList* ListCellWindow)
{
```

```
char  status;
Cell  *c,
      *cgo,
      *cback;
float dist;
float diff;
long  i;

if( area > Areamin )area = Areamin;

// Elimina regioes pequenas atraves do merging com a regiao vizinha
mais proxima
for( int step = 1; step <= 6; step++ ){
      diff = (float) sqrt((double) Difsim ) + step * 2;
      diff = diff * diff;
      do {  status  = 0;
            for( i = 0L; i < ListCellWindow->Size();  i++ ){
                  c = ListCellWindow->Get( i );
                  if( c->Dead() || c->Area() > area )
                        continue;
                  if( ( cgo = c->ClosestNeighbor(dist) ) == NULL )
                        break;
                  if( ( cback = cgo->ClosestNeighbor(dist) ) == NULL
){
                        printf("\nRGrow::MergeSmallCellsCells >>>
I\n");
                        exit(0);
                  }
                  if( c != cback )
                        continue;
                  if( dist <= diff || step > 5 ){
                        if( ( c = MergeCellsDisk( c, cgo ) ) == NULL
){
                              printf("\nRGrow::MergeSmallCellsCells
>>> II\n");
                              exit(0);
                        }
                        status = 1;
                  }
            }
            Adjust(ListCellWindow);
      }while( status );
}

for( step = 1; step <= 6; step++ ){
      diff = (float) sqrt((double) Difsim ) + step * 2;
      diff = diff * diff;
      do {  status  = 0;
            for( i = 0L; i < ListCellWindow->Size();  i++ ){
                  c = ListCellWindow->Get( i );
                  if( c->Dead() || c->Area() > area )
                        continue;
                  if( ( cgo = c->ClosestNeighbor(dist) ) == NULL )
                        break;
                  if( dist <= diff || step > 5 ){
```

```
                        if( ( c = MergeCellsDisk( c, cgo ) ) == NULL
){
                              printf("\nRGrow::MergeSmallCellsCells
>>> III\n");
                              exit(0);
                        }
                        status = 1;
                  }
            }
            Adjust(ListCellWindow);
      }while( status );
}
return TRUE;
}


void
RGrow :: Adjust(CellList* ListCellWindow)
{
Cell  *c;
      for( long i = 0; i < ListCellWindow->Size(); i++ ){
            c = ListCellWindow->Get(i);
            if( c == NULL )
                  continue;
            if( !c->Dead() )
                  c->AdjustNeighborhood();
      }
      for( i = 0; i < ListCellWindow->Size(); i++ ){
            c = ListCellWindow->Get(i);
            if( c == NULL )
                  continue;
            if( c->Dead() ){
                  ListCellWindow->RemoveCell( i );
       //         delete c;
            }
      }
      ListCellWindow->Compress();
      ListCellWindow->Shrink();
}



Cell*
RGrow :: MergeCells( Cell *c1, Cell *c2 )
{
int   lmin, lmax,
      cmin,cmax;


Cell  *cret = NULL;

      if( c1 == NULL || c2 == NULL )
            return NULL;
      if( c1 == c2 )
            return NULL;;
      if( c1->Area() >= c2->Area() ){
```

```
            if( c1->Merge( c2 ) == FALSE )
                  return NULL;
            c2->BoundingRectangle( lmin, cmin, lmax, cmax );
            for( int lin = lmin; lin <= lmax; lin++ )
                  for( int col = cmin; col <= cmax; col++ )
                        if( (*ImagelabM)(lin,col) == c2->Id()
)(*ImagelabM)(lin,col) = (unsigned long)c1->Id();
                  cret = c1;
      }else{
            if( c2->Merge( c1 ) == FALSE )
                  return NULL;
            c1->BoundingRectangle( lmin, cmin, lmax, cmax );
            for( int lin = lmin; lin <= lmax; lin++ )
                  for( int col = cmin; col <= cmax; col++ )
                        if( (*ImagelabM)(lin,col) == c1->Id()
)(*ImagelabM)(lin,col) = (unsigned long)c2->Id();
            cret = c2;
      }
return cret;
}



Cell*
RGrow :: MergeCellsDisk( Cell *c1, Cell *c2 )
{
int   lmin, lmax,
      cmin,cmax;

Cell  *cret = NULL;

      if( c1 == NULL || c2 == NULL )
            return NULL;
      if( c1 == c2 )
            return NULL;
      if( c1->Area() >= c2->Area() ){
            if( c1->Merge( c2 ) == FALSE )
                  return NULL;
      c2->BoundingRectangle( lmin, cmin, lmax, cmax );

            for( int lin = lmin; lin <= lmax; lin++ )
                  for( int col = cmin; col <= cmax; col++ )
                        if( (*Imagelab)(lin,col) == c2->Id()
)(*Imagelab)(lin,col) = (unsigned long)c1->Id();
                  cret = c1;
      }else{
            if( c2->Merge( c1 ) == FALSE )
                  return NULL;
            c1->BoundingRectangle( lmin, cmin, lmax, cmax );
            // pintando o pixel
            for( int lin = lmin; lin <= lmax; lin++ )
                  for( int col = cmin; col <= cmax; col++ )
                        if( (*Imagelab)(lin,col) == c1->Id()
)(*Imagelab)(lin,col) = (unsigned long)c2->Id();
            cret = c2;
      }
```

```
return cret;
}



int
RGrow :: Resort()
{
long  newid,
      oldid = 0L,
      index;

      ListCell->Adjust();
      for( int lin = 0; lin < ImagelabM->NumLin(); lin++ )
            for( int col = 0; col < ImagelabM->NumCol(); col++ ){
                  newid = (*ImagelabM)(lin,col);
                  if( newid != oldid || oldid == 0L ){
                        index = ListCell->Search( newid );
                        oldid = newid;
                  }
                  (*ImagelabM)(lin,col) = index+1;
            }
      for( int i = 0; i < ListCell->Size(); i++ )
            ListCell->Get( i )->ResetId( i+1 );
return TRUE;
}



int
RGrow :: Client_Apply( ImageMemory* im, ImageMemory& lab, int nban,
int ts, int areamin, int currentwindow, char* buffer)
{
int         ban;
//int             raizint,teste;

// long           CurrentWindow = rank - 1;
long        CurrentWindow = currentwindow;

      CellList* ListCellWindow;

      Clear();

      // Initialize Images

      if( ( Nban = nban ) <= 0 ){
            printf("\nRGrow::Apply >>> I\n");
            exit(0);
      }

      if( ( ImageinM = new ImageMemory*[Nban] ) == NULL ){
            printf("\nRGrow::Apply >>> II\n");
            exit(0);
      }
```

```
      for( ban = 0; ban < Nban; ban++ ){
            ImageinM[ban] = &im[ban];
      }

   ImagelabM = &lab;

      Nlin = ImagelabM->NumLin();
      Ncol = ImagelabM->NumCol();
      // Initialize colection of regions

      ListCell = new CellList;
      if( ListCell == NULL ){
            printf("\nRGrow::Apply >>> IV\n");
            exit(0);
      }

      // Inicializando o array de regioes da janela
      if( ( tuple = new unsigned long[Nban] ) == NULL ){
            printf("\nRGrow::Apply >>> XX\n");
            exit(0);
      }

      WindowLines = ( Nlin % TAMJAN_LIN ) ? ( Nlin / TAMJAN_LIN + 1
) : ( Nlin / TAMJAN_LIN );
      WindowColumns = ( Ncol % TAMJAN_COL ) ? ( Ncol / TAMJAN_COL +
1 ) : ( Ncol / TAMJAN_COL );
      NWindow    = WindowLines * WindowColumns;

      Areamin   = areamin;
      Difsim    = (float)ts * ts;

      int nlin, ncol;
      ImagelabM->y0 = nlin = (CurrentWindow / WindowColumns) *
TAMJAN_COL; // seta origem virtual
      ImagelabM->x0 = ncol = (CurrentWindow % WindowColumns) *
TAMJAN_COL;
      ImageinM[0]->y0 = nlin;
      ImageinM[0]->x0 = ncol;
      ImageinM[1]->y0 = nlin;
      ImageinM[1]->x0 = ncol;
      ImageinM[2]->y0 = nlin;
      ImageinM[2]->x0 = ncol;

      long  perc = -1;

            if(!( ListCellWindow=InitWindow(CurrentWindow))){
                  printf("\nRGrow::Apply >>> VII\n");
                  exit(0);
            }
            if( MergeMutuallyClosestCells(ListCellWindow) == FALSE
){
                  printf("\nRGrow::Apply >>> VIII\n");
                  exit(0);
            }
            if( MergeSmallCells(5,ListCellWindow) == FALSE ){
                  printf("\nRGrow::Apply >>> IX\n");
```

```
                exit(0);
            }
        int len = 0;

        ListCellWindow->Save(buffer, len);

        return len;
}



int RGrow::Server_Apply(Image* im, Image& lab, int nban, int ts, int
areamin)
{
 long CurrentWindow;
 CellList* temp;
 int ban, raizint, teste;
 CellList** ListCellWindowArray;
int num_passos,passo;
double passoinit, passoend;

        // Initialize Images

        if( ( Nban = nban ) <= 0 ){
                printf("\nRGrow::Apply >>> I\n");
                exit(0);
        }
        if( ( Imagein = new Image*[Nban] ) == NULL ){
                printf("\nRGrow::Apply >>> II\n");
                exit(0);
        }
        for( ban = 0; ban < Nban; ban++ ){
                Imagein[ban] = &im[ban];
                Imagein[ban]->SetReadOnly();
        }
        Imagelab = &lab;
        if( Imagelab->NumByte() != 4 ){
                printf("\nRGrow::Apply >>> III\n");
                exit(0);
        }
        Imagelab->SetReadWrite();

        Nlin = Imagelab->NumLin();
        Ncol = Imagelab->NumCol();

        // Initialize colection of regions

        ListCell = new CellList;
        if( ListCell == NULL ){
                printf("\nRGrow::Apply >>> IV\n");
                exit(0);
        }

        if( ( tuple = new unsigned long[Nban] ) == NULL ){
                printf("\nRGrow::Apply >>> XX\n");
                exit(0);
```

```
      }

   int processos;

      WindowLines = ( Nlin % TAMJAN_LIN ) ? ( Nlin / TAMJAN_LIN + 1
) : ( Nlin / TAMJAN_LIN );
      WindowColumns = ( Ncol % TAMJAN_COL ) ? ( Ncol / TAMJAN_COL +
1 ) : ( Ncol / TAMJAN_COL );
      NWindow    = WindowLines * WindowColumns;

   processos= MPI::COMM_WORLD.Get_size() - 1;

   num_passos = NWindow / processos ;
//DEBUG printf("SERVER: NWindow=%d
num_passos=%d\n",NWindow,num_passos);
//DEBUG fflush(stdout);

//Celso - Remove comparacao do numero de janelas com numero de
processos
// if(NWindow != MPI::COMM_WORLD.Get_size() - 1)
// {
//    printf("\n N.o de processos criados diferente de NWindow");
//    printf("\n NWindow = %d    no. processos criados = %d
",NWindow,processos);
//    exit(0);
// }

      ListCellWindowArray= new CellList* [NWindow];
      Areamin   = areamin;
      Difsim    = (float)ts * ts;


   // envia as janelas com as imagens originais
   unsigned long* tempmat= new unsigned long[TAMJAN_LIN*TAMJAN_COL];
   int sync = 0;
   MPI::Status stat;           // para saber de quem recebeu a
mensagem
   char* errmsg = new char[MPI::MAX_ERROR_STRING];
   int len = 0;

  for (passo=0; passo<num_passos; passo++)
  {
   passoinit = MPI::Wtime();
// for(int i = 0; i < NWindow; i++)
   for (int janela=0; janela < processos ; janela++)
{
      int i = passo * processos + janela;
//    int ilin = (i / WindowColumns) * TAMJAN_COL;
//    int flin = (i / WindowColumns) * TAMJAN_COL + TAMJAN_COL;
      int ilin = (i / WindowColumns) * TAMJAN_LIN;
      int flin = (i / WindowColumns) * TAMJAN_LIN + TAMJAN_LIN;
      int icol = (i % WindowColumns) * TAMJAN_COL;
      int fcol = (i % WindowColumns) * TAMJAN_COL + TAMJAN_COL;

      for(int n = 0; n < nban; n++)
      {
```

```
         for(int j = ilin; j < flin; j++)
            for(int k = icol; k < fcol; k++)
            {
                  tempmat[(j % TAMJAN_COL) * TAMJAN_COL + k %
TAMJAN_COL] = (*Imagein[n])(j, k);
             }
                MPI::COMM_WORLD.Send(tempmat, TAMJAN_COL*TAMJAN_COL,
MPI::LONG, (i % processos) + 1, n);
                MPI::COMM_WORLD.Recv(&sync, 1, MPI::INT, (i %
processos) + 1, n, stat);
             int index = stat.Get_source();
                if(stat.Get_error() != MPI::SUCCESS)
             {

MPI::Get_error_string(stat.Get_error(), errmsg, len);
                              printf("\n erro no envio de resp de
sinc do client p/ server %s", errmsg);
                }

      }

   }

//  for(int k = 1; k < NWindow+1; k++)
 for(int k = 0 ; k < processos ; k++)
 {
       temp = new CellList;
    char buffer[50000];
       int bufsize, index;

      // recebe stream do cliente e indica a janela de quem é em
index

    MPI::COMM_WORLD.Recv(buffer, 50000, MPI::CHAR, MPI::ANY_SOURCE,
3, stat);

       if(stat.Get_error() != MPI::SUCCESS)
       {
            MPI::Get_error_string(stat.Get_error(), errmsg, len);
            printf("\n erro no recebimento da matriz %s", errmsg);
       }
       bufsize = stat.Get_count(MPI::CHAR);
//     index = stat.Get_source() - 1;
       int origem = stat.Get_source() - 1;
       index = passo * processos + origem ;

      // recebe imagelab do cliente e coloca-a na posição correta
        MPI::COMM_WORLD.Recv(tempmat, TAMJAN_LIN*TAMJAN_COL,
MPI::LONG, stat.Get_source(), 4, stat);
     int bufsize3 = stat.Get_count(MPI::LONG);

      // colocando janelas recebidos na ordem

      int ilin = (index / WindowLines) * TAMJAN_LIN;
      int flin = (index / WindowLines) * TAMJAN_LIN + TAMJAN_LIN;
      int icol = (index % WindowColumns) * TAMJAN_COL;
```

```
      int fcol = (index % WindowColumns) * TAMJAN_COL + TAMJAN_COL;
      for(int j = ilin; j < flin; j++)
            for(int k = icol; k < fcol; k++)
            (*Imagelab)(j, k) = tempmat[(j % TAMJAN_COL) *
TAMJAN_COL + k % TAMJAN_COL];

      temp->Load(buffer, index, WindowLines, WindowColumns);
      ListCellWindowArray[index] = temp;
 }
  passoend = MPI::Wtime() ;
printf("SERVER: Passo=%d Tempo=%lf seg\n",passo,passoend-passoinit);
fflush(stdout);
 }    //Final do loop em "passo"


      printf("\n\nproc0: iniciando concatenacao...");

       //   iniciando a CONCATENACAO
       for (int i = 0; i<NWindow; i++){
            const long limit=ListCellWindowArray[i]->Size();
            for (long j=0; j<limit; j++)
            {
                  ListCell->AddCell(ListCellWindowArray[i]->Get(j));
            }

            if (i%WindowColumns!=0)
      {
            ConcatLR(ListCellWindowArray[i-1],
ListCellWindowArray[i],
i/WindowColumns*TAMJAN_LIN,i%WindowColumns*TAMJAN_COL);

      }
            if(i>=WindowColumns)
      {
            ConcatUL(ListCellWindowArray[i-
WindowColumns],ListCellWindowArray[i],i/WindowColumns*TAMJAN_LIN,(Wi
ndowColumns<WindowLines)?0:i%WindowColumns*TAMJAN_COL);

      }
      }

   MergeSimilarCellsDisk(ListCell);
   MergeSmallCellsDisk(Areamin, ListCell);
   ListCell->Adjust();

   printf("\n\nNo. de regioes = %d", ListCell->Size());


 return TRUE;
}
```

```
//     IMAGE10.CC       //
//                     //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //

#include <image10.hpp>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

Image :: Image( void )
{
      Inx               = 0;
      Iny               = 0;
      Inbytes           = 0;
      pixel             = 0;
      outsidevalue      = 0;
      rw                = 1;
      curline           = -1;
      fp          = NULL;
      Iname             = NULL;
      tbuf        = NULL;
      line        = NULL;
}

Image :: Init( char *name, short nlin, short ncol, short nbytes )
{
      Clear();

      if( ( fp = fopen( name, "r+b" ) ) == NULL ){
            printf("\nImage >>> Could not find file %s\n", name );
            return FALSE;
      }

      if( nbytes > 4 || nbytes == 3 )
            nbytes = 4;
      if( nbytes < 0 )
            nbytes = 1;
      Inx     = ncol;
      Iny     = nlin;
      Inbytes = nbytes;
      curline = -1;

      if( ( Iname = strdup ( name ) ) == NULL )
            return FALSE;
      rw     = 0;

      if( !AllocateMemoryLine() )
            return FALSE;

return TRUE;
}

Image :: Create( char *name, short nlin, short ncol, short nbytes )
{
```

```
        Clear();

        if( ( fp = fopen( name, "w+b" ) ) == NULL ){
              printf("\nImage >>> Could not find create %s\n", name );
              return FALSE;
        }

        if( nbytes > 4 || nbytes == 3 )
              nbytes = 4;
        if( nbytes < 0 )
              nbytes = 1;
        Inx     = ncol;
        Iny     = nlin;
        Inbytes = nbytes;
        curline = -1;

        if( ( Iname = strdup ( name ) ) == NULL )
              return FALSE;
        rw     = 1;

        if( !AllocateMemoryLine() )
              return FALSE;

        if ( !AllocateDisk() )
              return FALSE;

return TRUE;
}


int
Image :: AllocateMemoryLine()
{
        if( Inx <= 0 || Iny <= 0 ){
              printf("\nImage :: AllocateMemoryLine >>> Invalid number
of lines/columns (%s)\n", Iname );
              DeallocateMemory();
              return FALSE;
        }
        pixel = outsidevalue;
        tbuf  = new unsigned char[ Inx * Inbytes ];
        line  = new unsigned long[ Inx ];
        memset( (void*)tbuf, 0, Inx*Inbytes );
        memset( (void*)line, 0, Inx*Inbytes );
        if( tbuf == NULL || line == NULL) {
              printf("\nImage >>> No memory available\n");
              DeallocateMemory();
              return FALSE;
        }

return TRUE;
}


int
Image :: AllocateDisk()
```

```
{
      rewind( fp );
      memset( tbuf, 0, Inx * Inbytes );
      for( short i = 0; i < Iny; i++ )
            if( fwrite( (char*)tbuf, 1,  Inx*Inbytes, fp ) !=
(unsigned)(Inx*Inbytes) )
                  return FALSE;
      return TRUE;
}

unsigned long&
Image :: operator() ( int lin, int col )
{
      if( lin < 0 || lin >= Iny || col < 0 || col >= Inx )
            return pixel;

      if( curline < 0 ){
            ReadLine(lin);
            curline = lin;
      }else if( curline != lin ){
            if( rw ) WriteLine( curline );
            ReadLine( lin );
            curline = lin;
      }

return line[col];

}

void
Image :: Flush()
{
      if( rw == 0 ) return;
      WriteLine( curline );
}

void
Image :: SetReadWrite()
{
      if( rw == 1) return;
      curline = -1;
      rw = 1;
}

void
Image :: SetReadOnly()
{
      if( rw == 0 ) return;
      Flush();
      rw = 0;
}

int
Image :: WriteLine( int lin )
{
      if( rw == 0 || lin < 0 ) return TRUE;
```

```
      else if( rw != 1 ){
            printf("\nWriteLine >>> Writing mode undefined (%s)\n",
Iname );
            return FALSE;
      }

      fseek( fp, (long)Inbytes*Inx*lin, SEEK_SET );

      int c;
      switch( Inbytes ){
            case 1:     for( c = 0; c < Inx; c++ )
                        tbuf[c] = (unsigned char)line[c];
                  if( fwrite( (void*)tbuf, 1,  Inx*Inbytes, fp ) !=
(unsigned)(Inx*Inbytes) ) {
                        printf("\nImage :: WriteLine >>> Writing
error (%s)\n", Iname);
                        return FALSE;
                  }
                  break;
            case 2:     for(  c = 0; c < Inx; c++ )
                        ( (unsigned short*)tbuf )[c] = (unsigned
short)line[c];
                  if( fwrite( (void*)tbuf, 1,  Inx*Inbytes, fp ) !=
(unsigned)(Inx*Inbytes) ) {
                        printf("\nImage :: WriteLine >>> Writing
error (%s)\n", Iname);
                        return FALSE;
                  }
                  break;
            case 4:     if( fwrite( (void*)line, 1,  Inx*Inbytes, fp
) != (unsigned)(Inx*Inbytes) ) {
                        printf("\nImage :: WriteLine >>> Writing
error (%s)\n", Iname);
                        return FALSE;
                  }
                  break;
            default: printf("\nImage :: WriteLine >>> Invalid number
of bytes per pixel (%s)\n", Iname );
                   return FALSE;
      }
      return TRUE;
}


int
Image :: ReadLine( int lin )
{
unsigned lido;

      fseek( fp, (long)Inbytes*Inx*lin , SEEK_SET );

      int c;
      switch( Inbytes ){
            case 1:     if( (lido = fread( (void*)tbuf, 1,
Inx*Inbytes, fp )) != (unsigned)(Inx*Inbytes) ) {
```

```
                        printf("\nImage :: ReadLine >>> Reading
error (%s)\n",Iname);
                        return FALSE;
                  }
                  for( c = 0; c < Inx; c++ )
                        line[c] = (unsigned long)tbuf[c];
                  break;
            case 2:     if( fread( (void*)tbuf, 1,  Inx*Inbytes, fp
) != (unsigned)(Inx*Inbytes) ) {
                        printf("\nImage :: ReadLine >>> Reading
error (%s)\n",Iname);
                        return FALSE;
                  }
                  for( c = 0; c < Inx; c++ )
                        line[c] = (unsigned long)  ( ((unsigned
short*)tbuf)[c] );
                  break;
            case 4:     if( fread( (void*)line, 1,  Inx*Inbytes, fp
) != (unsigned)(Inx*Inbytes) ) {
                        printf("\nImage :: ReadLine >>> Reading
error (%s)\n",Iname);
                        return FALSE;
                  }
                  break;
            default: printf("\nImage :: ReadLine >>> Invalid number
of bytes per pixel (%s)\n", Iname );
                   return FALSE;
      }
      return TRUE;
}


void
Image :: DeallocateMemory()
{
      if( tbuf ){ delete tbuf; tbuf = NULL; }
      if( line ){ delete line; line = NULL; }
}

void
Image :: Clear()
{

      Flush();

      DeallocateMemory();

      if( pixel != outsidevalue )
            printf("\nImage >>> Warning: Writing outside image
!!!\n");

      Inx            = 0;
      Iny            = 0;
      Inbytes        = 0;
      pixel          = 0;
      outsidevalue   = 0;
```

```
      rw             = 1;
      if( fp ){ fclose(fp); fp = NULL; }
}


void
ImageMemory :: Clear()
{
      if(buf == 0)
      {
        printf("\nErro de alocação de buf!!!");
        exit(0);
        }

    x0=0;
    y0=0;
   memset(buf, 0, 128*128*4);
}



unsigned long& ImageMemory::operator() (int lin, int col)
{
   int index = (lin*128) + col;
   return buf[index];
}
```

```
//     DINVECT10.CC       //
//                       //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //

#include "dinvect10.hpp"
//-------------------------------------------------------------------
//                METHODS FOR DINVECT
//-------------------------------------------------------------------

DinVect&
DinVect :: operator=( const DinVect& dv )
{
      if( NewSize( dv.bodysize ) == FALSE )
            return *this;

      bodysize = dv.bodysize;
      size     = dv.size;
      void **pthis = &body[0], **pdv = &dv.body[0], **pf =
&body[size];
      while( pthis < pf )
            *(pthis++) = *(pdv++);
return *this;
}

int
DinVect :: ShiftLeft( long n )
{

      if( n <= 0L || n > size ) return TRUE;

      void **pthis = &body[n-1], **pshift = &body[n], **pf =
&body[size];
      while( pshift < pf )
            *(pthis++) = *(pshift++);
        size--;
return TRUE;
}

int
DinVect :: ShiftRight( long n )
{
      if( n >= size || n < 0) return FALSE;
        if( size >= bodysize ){
                if( NewSize( bodysize+block ) == FALSE )
                  return FALSE;
      }
      void **pthis = &body[size-1], **pshift = &body[size], **pf =
&body[n];
      //void **pthis = &body[size-1], **pshift = &dv.body[size],
**pf = &body[n];
      while( pshift > pf )
            *(pshift--) = *(pthis--);
      size++;
return TRUE;
```

```
}

int
DinVect :: Add( void* obj, long n )
{
      if( n < 0 ) return FALSE;
      if( n >= size ){
            if( n >= bodysize )
                  if( NewSize( (n-bodysize >= block) ? (n+block) :
(bodysize+block) ) == FALSE )
                        return FALSE;
            size = n+1;
      }
      else
            ShiftRight( n );
      body[n] = obj;
return TRUE;
}

void*&
DinVect :: operator[]( long i )
{
      if( i < 0L ) return dummy;

      if( i >= size ){
            if( i >= bodysize ){
                  if( NewSize( (i-bodysize >= block ) ? ( i+block )
: (bodysize+block) )  == FALSE )
                        return dummy;
            }
            size = i + 1;
      }

return body[i];
}

int
DinVect :: NewSize( long n )
{
      if( n < 0 ) return FALSE;
      if( n == bodysize ) return TRUE;
      if( n == 0L ) {Clear();return TRUE;}

      void **newbody = NULL;
      if( ( newbody = new void*[n] ) == NULL ) return FALSE;

      if( size > n ) size = n;
      bodysize = n;

      void  **pthis = body, **pnew = newbody, **pf = body+size;
      while( pthis < pf )
            *(pnew++) = *(pthis++);
      if( body )
            delete[] body;
      body = newbody;
return TRUE;
```

```
}

int
DinVect :: Reverse()
{
void  **pi = &body[0],
      **pf = &body[size-1],
      **pobj ;

      if( size <= 0L )
            return TRUE;

      while( pi < pf ){
            *pobj   = *pi;
            *(pi++) = *pf;
            *(pf--) = *pobj;
      }

return TRUE;
}

int
DinVect :: Compress()
{
void  **p1,
      **p2,
      **pf;

      if( DinVect::Size() <= 0 )
            return TRUE;
      p1 = &body[0];
      pf = &body[size-1];
      for( p1 = &body[0]; p1 <= pf && *p1 != NULL; p1++ );
      p2 = p1+1;
      while( p2 <= pf ){
            for(; p2 <= pf && *p2 == NULL; p2++);
            for(; p2 <= pf && *p2 != NULL; p2++)
                  *(p1++) = *p2;
      }
      for( p2 = p1; p2 <= pf; *(p2++) = NULL );
      size = p1 - &body[0];

return TRUE;
}
```

```
//     CELL10.HPP       //
//                     //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //


#ifndef     cellDEFINED
#define     cellDEFINED

#include <image10.hpp>
#include <dinvect10.hpp>

#define     MAXCLOSECELLS     5
#define  SQR_NC         65025
#define     TAMJAN_LIN  128
#define     TAMJAN_COL  128
#define     STATDEAD    (char)2

class CellList;
class Cell;

//-------------------------------------------------------------------
---

class CloserCells{
public :
      Cell  *cmin[MAXCLOSECELLS];
      float dmin[MAXCLOSECELLS];
            CloserCells();
      void  Insert( Cell *cell, float dist );
      void  Update( Cell *cell, float dist );
      Cell* Minimum( float& dist );
      void  Adjust();
      void  Reset(){ for(short i = 0; i < MAXCLOSECELLS; i++ ){
cmin[i] = NULL; dmin[i] = 0.0; } }
            ~CloserCells(){}
};

//-------------------------------------------------------------------
--

class CellList : public DinVect
{
      long  Position( long id );

public:
      CellList() {}
      long  Size(){ return DinVect::Size(); }
      long  Search( long id );
      Cell* Get( long i ){ return (Cell*)DinVect::Get( i ); }
      Cell* Find( long id ){ return  ( id >= 0L ) ?  ( (Cell*)Get(
Search( id ) ) ) : NULL ; }
      int   AddCell( Cell *c );
      void  RemoveCell( long i ){ if( i >= 0L && i < Size()
)(*this)[i] = NULL; }
```

```
      int   InsertCell( Cell *c );
      void  Adjust();
      void  Print();
  void      Save( char* buf, int& len );
   void     Load( char* buf, int index, long WindowLines, long
WindowColumns);
            ~CellList(){}
};


//-----------------------------------------------------------------
-

class Cell
{
//public:
      friend class CellList;
      char        Stat;
      long        Idnumber;
      long        Npix;
      int         LinMin,
                  LinMax,
                  ColMin,
                  ColMax,
                  Nban;
   long     *tempid, *tempcc;
   long     nsize;
      float       delta;
      float       *Media;
      float       *PreviousMedia;
      CellList    *Neighbors;
      CloserCells *Cc;

public:
            Cell();
      Cell( unsigned long *tuple, long id, short lin, short col,
short nban );
      void  ResetId( long idc ) { Idnumber = idc; }
      char  Dead()  { return (Stat & STATDEAD); }
      void  Kill() { Stat = STATDEAD; }
      void  BoundingRectangle( int& lmin, int& cmin, int& lmax, int&
cmax )
            { lmin = LinMin; cmin = ColMin; lmax = LinMax; cmax =
ColMax; }
      int   Merge( Cell *c );
      void  AdjustNeighborhood(){ Cc->Adjust(); Neighbors->Adjust();
}
      long  NeighborhoodSize(){ return Neighbors->Size(); }
      int   AddNeighbor( Cell* c, float dist );
      int   InsertNeighbor( Cell* c, float dist );
      Cell* GetNeighbor( long i ){ return Neighbors->Get(i); }
      Cell* ClosestNeighbor( float& dist );
      float Distance( Cell *c );
      long  Id() { return Idnumber; }
      long  Area() { return Npix; }
      int   GetNban() { return Nban; }
```

```
      void  Save( char* buf, int& len );
   char*    Load( char* buf, int janela, long WindowLines,long
WindowColumns);
      void  Print();
            ~Cell(){ if( Media         != NULL ) { delete Media;
Media = NULL; }
                  if( Neighbors     != NULL ) { delete Neighbors;
Neighbors = NULL; }
                  if( PreviousMedia != NULL ) { delete
PreviousMedia; PreviousMedia = NULL; }
                  if( Cc            != NULL )
            { delete Cc;            Cc = NULL; }
            }
};

//-----------------------------------------------------------------
---

class RGrow
{
      ImageMemory **ImageinM; //
      ImageMemory *ImagelabM; //

      Image **Imagein;
      Image *Imagelab;

      CellList    *ListCell;
      unsigned long     *tuple;
      long        NWindow;
      long        WindowLines;
      long        WindowColumns;
      long        Areamin;
      float       Difsim;
      int         Nlin;
      int         Ncol;
      int         ilin;
      int         icol;
      int         flin;
      int         fcol;
      int         Nban;
      int         sizelin, sizecol;

      Cell* MergeCells( Cell *c1, Cell *c2 );
   Cell*    MergeCellsDisk( Cell *c1, Cell *c2 );
      CellList* InitWindow(long CurrentWindow);
      void  Adjust(CellList* ListCellWindow);
      void  Clear();
      int   MergeSmallCells( long area,CellList*);
      int   MergeSimilarCells();
   int      MergeSmallCellsDisk( long area,CellList*);
      int   MergeSimilarCellsDisk(CellList*);
      int   MergeSimilarCells(CellList* );
      int   MergeMutuallyClosestCells(CellList*);
      int   Resort();

      int ConcatLR(CellList*, CellList*, int ilin, int icol);
```

```
      int ConcatUL(CellList*, CellList*, int ilin, int icol);
public:
            RGrow();
      int  Client_Apply( ImageMemory* im, ImageMemory& lab, int
nban, int ts, int areamin, int rank, char* buffer);
   int  Server_Apply( Image* im, Image& lab, int nban, int ts, int
areamin);
      int  Save();
            ~RGrow(){ Clear(); }
};

#endif
```

```
//     IMAGE10.HPP      //
//                     //
//                     //
// VERSAO PRELIMINAR 10 //
// 25-11-99           //


#ifndef imagemDEFINED
#define imagemDEFINED

#include <stdio.h>
#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif


class Image
{
      FILE        *fp;              // File pointer
      char*       Iname;                  // Image name

      unsigned char     *tbuf;                  // buffer for
transfer data
      unsigned long     *line;                  // buffer with data

      int   ReadLine( int lin );          // reads a line from disk
      int   WriteLine( int lin );         // writes a line to disk
      int   AllocateMemoryLine();         // Allocates memory for
lines
      void  DeallocateMemory();           // Frees memory
      int   AllocateDisk();               // Allocates disk space

protected:
      unsigned long   outsidevalue;       // Default value to pixels
outside Image

      short       Inx,              // no. of columns
                  Iny;              // no. of lines
      short       Inbytes;          // No. bytes/pixel

   int            curline;          // Current line.

      char        rw;               // access mode :0 read only
                                              // 1 read/write

      unsigned long     pixel;                  // pixel out of image

      virtual void  Clear();                    // Resets image

public:

      Image( void );
```

```
      // Empty Constructor

      virtual ~Image(){ Clear(); }
      // Destructor.

      short NumCol () { return Inx; }
      // Get the horizontal image size.

      short NumLin () { return Iny; }
      // Get the vertical image size.

      short NumByte(){ return Inbytes; }
      // Get the number of bites/pixel

      const char* Name(){ return Iname; }
      // Get the name of Image

      virtual unsigned long& operator() ( int lin, int col );
      // Access to image pixels.

      virtual void  SetReadWrite();
      // Set permission access to read/write

      virtual void  SetReadOnly();

      // Set permission access to read only

      virtual void  Flush();

      void  SetOutsideValue( long out ) { pixel = out; outsidevalue
= out; }
      // Set default value for pixels lying outside image

      virtual int Init( char *name, short nlin, short ncol, short
nbytes = 1 );
      // Initializes a new image from disk

      virtual int Create( char *name, short nlin, short ncol, short
nbytes = 1 );
      // Creates a new image in disk

};



class ImageMemory
// Implementa um buffer de imagem em memória
{
   public:

   unsigned long* buf;  // buffer de dados da imagem

   int x0, y0;          // posição do canto superior esquerdo da
imagem
   int Inx, Iny;  // no linhas, no de colunas
```

```
   public:

   void Clear();
   ImageMemory()  // constructor
   {
      Inx = 128;
      Iny = 128;
      buf=0;
      buf= new unsigned long [128*128];
      Clear();
   }

        ~ImageMemory()
        {delete[] buf; }

      unsigned long& operator() ( int lin, int col );
      // Access to image pixels.

      short NumCol () { return Inx; }
      // Get the horizontal image size.

      short NumLin () { return Iny; }
      // Get the vertical image size.

};

#endif
```

```
//     DINVECT8.HPP    //
//                     //
//                     //
// VERSAO PRELIMINAR 9 //
// 21-11-99           //

#ifndef dinvectDEFINED
#define dinvectDEFINED

#include <stdlib.h>

#define     TRUE  1
#define FALSE     0

class  DinVect
{
public:

      DinVect( long b = 2 ) : block(b), size(0L), bodysize(0L),
body(NULL) {
                  if( b <= 0L ) block = 10;
      }
      DinVect( const DinVect& dv) : block(10), size(0L),
bodysize(0L), body(NULL){
                  *this = dv;
      }
      virtual           ~DinVect() { Clear(); }
      DinVect&    operator=( const DinVect& );
      void*&            operator[]( long n );
      int         ShiftLeft( long n );
      int         ShiftRight( long n );
      int         Add( void* obj, long n );
      int         Reverse();
      int         Compress();
      long        Size() const { return size; }
      void        Qsort( int compar( const void*, const void* ) ){
                        qsort( (char*)body, (int)size,
sizeof(void*), compar );
                  }
      int         Shrink(void){
                        return (bodysize-size > 2*block) ?
(NewSize(size+block)) : (TRUE);
                  }
       void*&           Get( long n ){
                        return ( n >= 0L && n < size ) ? body[n] :
dummy;
                  }
      int         Find( void* objt ){
                        int i = 0; while( i < size && !(body[i] ==
objt) ){ i++; }
                        return ( i < size ) ? (i) : (-1);
                  }

      int         CheckBounds( long n ){
                        return (n < 0 || n >= size ) ? (0) : (1);
                  }
```

```
protected:

      int         NewSize( long n );

private:

      void**            body;       // Array of Ts
      void*       dummy;            // Dummy object
      long        block;            // Positions to be added to the
vector when it grows.
      long        bodysize;   // Positions available in the vector
      long        size;       // Maximum index occupied by an object
+ 1
      void        Clear(){
                        size = bodysize = 0L;
                        if( body  != NULL ){ delete[] body; body =
NULL; }
                  }

};


#endif
```

PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programa de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. São aceitos tanto programas fonte quanto executáveis.