HENRIQUE YOSHIKAZU SHISHIDO

Paralelização de algoritmo de processamento de imagens digitais

MARINGÁ
2010

HENRIQUE YOSHIKAZU SHISHIDO

Paralelização de algoritmo de processamento de imagens digitais

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Ronaldo Augusto de Lara Gonçalves

MARINGÁ
2010

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

S558p

Shishido, Henrique Yoshikazu
Paralelização de algoritmo de processamento de imagens digitais. / Henrique Yoshikazu Shishido. -- Maringá, 2010.
xvi, 84 f. : il., figs., tabs.

Orientador : Prof. Dr. Ronaldo Augusto de Lara Gonçalves.
Dissertação (mestrado) - Universidade Estadual de Maringá, Programa de Pós-Graduação em Ciência da Computação, 2010.

1. Paralelização - Algoritmo - Imagem digital. 2. Imagem digital - Sensoriamento remoto. 3. Filtragem digital - Convolução. 4. Bibliotecas de paralelização - MPI e HLRC. 5. Arquitetura paralela - Cluster - Memória distribuída. 6. Arquitetura paralela - Cluster - Memória compartilhada e distribuída. 7. Processamento de imagem digital. 8. Computação paralela. I. Gonçalves, Ronaldo Augusto de Lara, orient. II. Universidade Estadual de Maringá. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 21.ed. 004.3684

HENRIQUE YOSHIKAZU SHISHIDO

PARALELIZAÇÃO DE ALGORITMO DE PROCESSAMENTO DE IMAGENS DIGITAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em 30/07/2010.

BANCA EXAMINADORA

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Anderson Faustino da Silva
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Alan Salvany Felinto
Universidade Estadual de Londrina – DC/UEL

Dedico este trabalho aos meus pais Américo e Rosa, e a todos que, direta ou indiretamente, contribuíram para o sucesso desta pesquisa.

Agradecimentos

Agradeço primeiramente a Deus por sua constante proteção durante esta luta;

Aos meus pais, Américo e Rosa, sempre presentes nos bastidores das maiores realizações da minha vida, pelo exemplo de vida que mostram na incessante luta para a minha formação profissional e pessoal; à minha irmã, Évelin, pelo contínuo incentivo e pela companhia nos momentos da minha vida;

Ao meu orientador, Prof. Dr. Ronaldo Augusto de Lara Gonçalves, pelas valiosas orientações neste trabalho; e por sua amizade com as conversas e discussões que levarei como lições em minha vida;

À Ligia Flávia Batista que colaborou imensamente com o meu objeto de pesquisa;

Ao Prof. Dr. Anderson Faustino da Silva, pelas diversas contribuições a este trabalho e experiências compartilhadas;

Ao programa de Pós-graduação em Ciência da Computação da Universidade Estadual de Maringá, pela infra-estrutura e outros serviços prestados; Em especial, à Maria Inês Davanço pela sua amizade, simpatia, extrema eficiência e por ter salvo a minha vida literalmente. Obrigado Inês!;

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio durante todo o curso;

Agradeço ao meu amigo Francisco Pereira Junior (Thesko) pela confiança, incentivo e grande apoio durante o transcorrer da minha carreira acadêmica e profissional;

Ao pessoal do mestrado: Alberto Biasão, Alexandre Huff, Ana Paula Chaves, Camila Leal, Gustavo Sato, Marcelo Borth, Wellington Cardador pelo companheirismo nos bons e, mais ainda, nos piores momentos;

Aos companheiros da UTFPR: Antônio Carlos, Alexandre Feitosa, Alexandre L'erario, Eidy Tanaka e Rogério Pozza que sempre conversaram nas épocas de desespero;

À toda galera do Departamento de Tênis de Mesa da ACEMA, em especial às famílias Inoue, Matsumura, Miura e Tomita sempre presentes como uma segunda família para

mim nesta cidade;

E a todos aqueles que influenciaram de alguma forma para que eu chegasse até este momento!

Resumo

Esta dissertação discute aspectos inerentes à paralelização de aplicações sequenciais empregados para reduzir o tempo de execução de experimentos e simulações científicas. Muitas dessas aplicações foram escritas em linguagens como FORTRAN e C, em uma época que não existia as facilidades de software e hardware como as que temos atualmente. Uma grande parcela dessas aplicações demanda um tempo de execução relativamente longo. As mais diversas áreas das ciências podem tirar vantagens da paralelização e execução de suas aplicações de interesse em um *cluster* de computadores, que pode ser adquirido por um custo relativamente baixo se comparado aos supercomputadores. É possível processar um volume maior de dados e executar um número maior de tarefas que, anteriormente, eram inviáveis devido ao custo computacional envolvido. Nesse contexto se insere o presente trabalho, que tem como objetivo principal a proposição e avaliação de 2 modelos de paralelização sobre os modelos MPI e HLRC, de um algoritmo sequencial de processamento de imagem aplicado a imagens geográficas. Tal algoritmo utiliza uma convolução específica de *pixels* para realçar bordas e padrões de textura, permitindo analisar o índice de fragmentação de imagem. A execução da versão sequencial em C do algoritmo de convolução sobre uma única imagem pode consumir até 25 minutos de processamento com 15 pixels em uma imagem de 8460x9530 pixels, entretanto, imagens maiores compostas por dezenas destas podem levar dias. Os modelos de paralelização propostos são baseados na metodologia PCAM. A partir dos modelos propostos, 4 versões paralelas foram implementadas, as quais foram executadas em um tempo de execução menor em aproximadamente 73,9%, 85,5%, 86% e 92,2% do que a versão sequencial do algoritmo.

Palavras-chave: Processamento Paralelo, Cluster, MPI, HLRC, Processamento de Imagem, Convolução.

Abstract

This dissertation discuss the related issues of the parallelization of sequencial applications parallelization used to reduce the runtime of experiments and scientific simulations. Much this applications has been written on FORTRAN or C languages, in a period which has not the software and hardware facilities that we have in the present. A large slice of this applications requires a long runtime. The various ciencitifc areas can take the advantages of the parallelization and execution of this algorithms on a cluster, that can be acquired for a low cost than supercomputers. It's possible to process a larger data and execute more tasks that, previously, was impracticable in a unique execution because the computacional cost. In this context it self insert in this present work, that has the main objective the purpose and parallelization of 2 parallel models on MPI and HLRC platforms, of a image processing algorithm applied to geographic images for analysis of the fragmentation index. That algorithm uses the convolution technique to enhance the borders and texture standards allowing analyse the fragmentation index of image. The execution of sequencial version of the convolution algorithm on a unique image can takes up to 25 minutes of processing, however, a set of larger images can be take a lot days. The purposed parallel models are based on PCAM methodology. From this purposed models, 4 parallel versions were developed, that were executed in a less execution time in about 73,9%, 85,5%, 86% e 92,2% than the sequencial algorithm version.

Palavras-chave: Parallel Processing, Cluster, MPI, HLRC, Image Processing, Convolution.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

DNA: *Deoxyribonucleic Acid*
DSM: *Distributed Shared Memory*
E/S: *Entrada e Saída*
IFM: *Índice de Fragmentação Multidimensional*
LECAD: *Laboratório de Computação de Alto Desempenho*
MCA: *Modular Component Architecture*
MIMD: *Multiple Instruction Multiple Data*
MISD: *Multiple Instruction Single Data*
MPI: *Message Passing Interface*
MPMD: *Multiple Program Multiple Data*
NASA: *National Aeronautics and Space Administration*
NOW: *Network of Computers*
ORTE: *Open Run Time Environment*
PC: *Personal Computer*
PVM: *Parallel Virtual Machine*
SIMD: *Single Instruction Multiple Data*
SISD: *Single Instruction Single Data*
SMP: *Symetric Multi-Processing*
SPMD: *Single Program Multiple Data*
SR: *Sensoriamento Remoto*
TCP: *Transmission Control Protocol*

Capítulo

1

Introdução

Os avanços da ciência e tecnologia aumentaram a demanda por plataformas de hardware e software que possam oferecer maior capacidade de processamento para a execução de aplicações associadas a problemas complexos do mundo real. Nesse contexto, a programação paralela é uma técnica que tem sido utilizada para implementar tais aplicações nos casos em que um único processador requeira muito tempo para processá-las sequencialmente (Ivanov, 2006).

A crescente popularização dos computadores aliada aos avanços tecnológicos na comunicação de dados têm permitido a criação de novas plataformas com um alto poder de processamento. Computadores *off-the-shelf* em conjunto com softwares de código aberto permitem a construção de uma plataforma de custo relativamente baixo denominada *cluster* que pode oferecer uma capacidade de processamento similar àquela oferecida por supercomputadores encontrados nos grandes centros de pesquisa e desenvolvimento de anos atrás. Um *cluster* de 32 computadores interligados por uma rede *Ethernet* pode ser adquirido por 75 mil dolares. Um supercomputador com poder computacional equivalente custava em torno de 1 milhão de dolares (Brodkin, 2007)

Experimentos computacionais associados a fenômenos físico-químicos, reconhecimento de padrões biológicos, previsões sísmicas e climatológicas, identificação de transformações na superfície do globo, dinâmica de atração entre corpos espaciais, entre outros, têm sido amplamente investigados por grupos em diferentes áreas de pesquisa. Problemas desta natureza tipicamente envolvem grande quantidade de processamento e podem levar dias para serem resolvidos se executados sequencialmente. Este tempo elevado retarda o

avanço das pesquisas em muitas áreas, principalmente em áreas que exigem experimentos exaustivos e diversificados.

Nesta mesma direção, o processamento de imagens digitais tem sido usado para identificar e reconhecer padrões e facilitar a interpretação visual (Crósta, 1992), sendo a convolução uma técnica de filtragem de frequência que atua no domínio espacial da imagem a fim de se produzir o efeito que se deseja, conforme descrita em (Gonzalez e Woods, 2000). Entretanto, dependendo do número e do tamanho das imagens, o processamento sequencial de várias imagens com 8460x9530 pixels de dimensão adotando janelas acima de 11 *pixels* já não atende mais os requisitos de tempo de execução.

Para reduzir este tempo, computadores paralelos, ferramentas e bibliotecas de programação paralela, compiladores otimizadores e algoritmos de particionamento de domínio estão disponíveis atualmente. Baseados nestes recursos, aplicações científicas desenvolvidas há anos atrás, muitas das quais implementadas em linguagens como FORTRAN e C, podem ser adaptadas com o objetivo de aproveitar as possibilidades de paralelização das plataformas atuais. Um fator de complexidade nos códigos dessas aplicações está no fato dos mesmos terem sofrido diversas modificações no decorrer do tempo por diferentes pessoas e, embora sejam utilizados até os dias de hoje, demandam muito tempo de processamento, quase sempre devido a problemas de otimização e sequencialidade rígida estabelecida pelo reaproveitamento de código.

Inserido neste contexto, o presente trabalho propõe e avalia dois modelos de paralelização de um algoritmo de processamento de imagens digitais baseado em convoluções, com o objetivo principal de reduzir o tempo de execução deste algoritmo originalmente sequencial. Os modelos são identificados como Modelo Básico e Modelo Seletivo. Como estudo de caso, usamos a convolução proposta por (Gonzalez e Woods, 2000) para gerar o índice de fragmentação de uma imagem de sensoriamento remoto por satélite descrito em (Turner, 1989). Esses modelos foram propostos para sistemas de memória distribuída e distribuída compartilhada e foram implementados em MPI e HLRC, respectivamente. Além disso, duas métricas de avaliação qualitativa foram propostas para o Modelo Seletivo. Foram experimentados os modelos em diferentes situações com o intuito de determinar onde cada modelo possui maior aplicabilidade e eficiência.

Experimentos exaustivos foram realizados em um cluster de 6 nós com 2 núcleos cada e os desempenhos foram analisados. Pôde-se observar que quanto maior o tamanho da máscara de convolução, maior foi o desempenho alcançado. As análises a partir dos dados de execução mostram os benefícios da paralelização. Constatou-se também que o Modelo Seletivo pode reduzir o tempo de execução paralela de forma significativa se a qualidade do resultado estiver dentro de limites aceitáveis.

O presente trabalho está organizado em 7 capítulos conforme segue. O Capítulo 2 apresenta uma breve revisão bibliográfica referente ao tema deste trabalho. O Capítulo 3 apresenta algumas metodologias de paralelização e os trabalhos que empregaram *clusters* para viabilizar a execução de seus experimentos. O Capítulo 4 descreve o algoritmo alvo para o desenvolvimento das versões paralelas. O Capítulo 5 discorre sobre os modelos propostos e as respectivas versões paralelas implementadas. A avaliação do desempenho das versões paralelas aparece no Capítulo 6. Por fim, o Capítulo 7 tece as considerações finais deste trabalho.

Capítulo

2

Processamento Paralelo

A evolução da capacidade computacional ao decorrer da evolução dos computadores tornou a computação uma ferramenta multidisciplinar para diversas áreas da ciência para equacionar seus problemas, que há décadas eram inviáveis devido o grande volume de cálculos processados. A necessidade das aplicações em busca de resultados cada vez mais precisos e fiéis resultam na elevação da complexidade e quantidade dos cálculos processados. Assim, o tempo gasto para se obter estes resultados aumentam, mostrando que é preciso um poder computacional cada vez maior e, diante destas circunstâncias, a computação paralela passa a ser uma alternativa para muitas áreas relacionadas à pesquisa.

A arquitetura proposta por Von Neumann, em meados da década de 50, determinou os conceitos básicos da estruturação de computadores e, por muitos, anos foi o modelo seguido pela indústria. Neste modelo, o computador é constituído essencialmente pelo processador, memória e dispositivos de E/S (entrada e saída) que executam sequencialmente as tarefas na ordem determinada por uma unidade de controle.

O desenvolvimento tecnológico dos elementos básicos da arquitetura de Von Neumann permite a possibilidade desse modelo, sempre buscando maior eficiência. Como cada componente da arquitetura de Von Neumann trabalha em velocidades diferentes, as instruções executadas no processador são muito mais rápidas que as operações de E/S. Por exemplo, numa execução de uma aplicação sequencial, o processador fica ocioso enquanto uma operação de escrita de dados do disco está sendo executada. Consequentemente, o

modelo sequencial sofre com o desperdício do precioso tempo que poderia ser aplicado para o processamento.

Para reduzir o desperdício com a ociosidade dos processadores, surgiu a ideia de aproveitar o tempo que um processador usa para fazer E/S, por exemplo, para executar outro processo. Então, os projetistas de sistemas operacionais passaram a dividir o uso do processador entre os processos em estado de pronto, determinando uma fatia de tempo para cada um e executar outro processo enquanto o atual está solicitando uma operação de E/S. Esta troca de contexto é realizada de maneira tão rápida que, em sistemas monoprocessados, o usuário tem a impressão de que os processos estejam sendo executados simultaneamente. Porém, este paralelismo não é real, uma vez que somente um único processo está sendo executado pelo processador em um determinado momento.

Para conseguir atingir um paralelismo real é preciso, ao menos, duas unidades de processamento. As unidades poderiam executar simultanemanete os processos, mas não obrigatoriamente. Cada uma pode estar executando o mesmo processo, contribuindo com a redução do tempo total de execução. Entretanto, as arquiteturas paralelas apresentam complexidades adicionais como o controle de processos em execução, controle de acesso à memória e, dependendo do modelo paralelo, a comunicação entre os processos.

2.1 Arquiteturas Paralelas

A preocupação em se obter resultados precisos, oriundos de um grande volume de instruções e de algoritmos complexos, demandam de recursos computacionais cada vez mais eficazes. A computação paralela é uma alternativa aos caríssimos supercomputadores encontrados em laboratórios de grandes instituições científicas, militares e industriais anos atrás.

De acordo com (Hwang, 1993), o processamento ou computação paralela pode ser definida como: “Forma eficiente do processamento de dados com ênfase na exploração de eventos concorrentes no processo computacional”.

A principal ideia da computação paralela é reduzir o tempo computacional para a resolução de um problema (Grama et al., 2003), (Hariri e Parashar, 2004). É fundamentada no fato de que a execução das instruções de um problema podem ser divididas em um conjunto de instruções menores, que podem ser executadas simultaneamente por meio de algum tipo de coordenação. De forma análoga, podemos imaginar a construção de um edifício, por exemplo. Quanto maior o número de trabalhadores executando as tarefas do processo de construção, menor o tempo gasto para a sua conclusão. Entretanto, sempre existem pontos de dependência (gargalos) que podem atrasar o projeto como um todo

e que precisam ser considerados. No caso desta analogia, se houver 40 trabalhadores, mas 39 dependeram da conclusão da tarefa de 1 trabalhador para que possam prosseguir com os seus trabalhos, então o ganho de tempo será fortemente influenciado pelo tempo gasto por aquele único trabalhador. No caso dos computadores esta situação não é diferente. Quanto maior o número de processadores, e cada um executando tarefas menores simulataneamente aos outros, menor será o tempo de resolução de um problema. Porém, igualmente na analogia, os gargalos (dependência de dados, *bandwidth*[1] de rede, falha em alguma máquina) influenciam diretamente no tempo total de execução e precisam ser considerados.

O ideal para uma execução com n processadores seria a redução do tempo para $1/n$ em relação ao total gasto com a execução de um único processador. Em contrapartida, diversos fatores como a *bandwidth*, a latência de rede e a memória, bem como algoritmos com poucos pontos de paralelização não contribuem para que este desempenho ideal seja alcançado.

2.2 Classificação de Arquiteturas Paralelas

As arquiteturas paralelas possuem várias formas de organização de fluxos em sistemas computacionais. (Flynn e Rudd, 1996) definiu a relação entre fluxos de instruções e fluxos de dados dentro do processo computacional. Um fluxo de instruções corresponde a uma sequência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas (Tanenbaum, 2001), (Flynn e Rudd, 1996).

- **SISD** (*Single Instruction Single Data*) - É a classe onde é executada um único fluxo de instruções em um único conjunto de dados, e oferece o mínimo de concorrência em relação aos demais modelos. Utiliza *pipelining* para alcançar a concorrência de processamento atualmente presente na maioria dos processadores. O modelo de Von Neumann é regido por esta estrutura (Figura 2.1).

[1] largura de banda

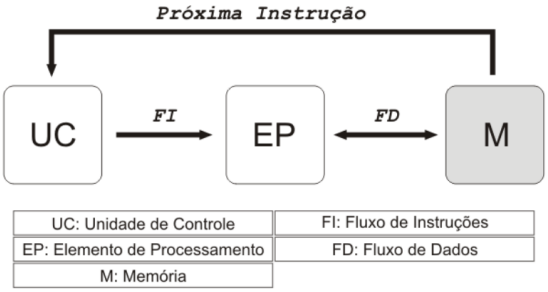

Figura 2.1: Modelo computacional SISD

- **SIMD** (*Single Instruction Multiple Data*) - Este modelo é concebido pelo fluxo único de instruções sobre múltiplos conjuntos de dados. Múltiplos processadores (escravos) são controlados por uma única unidade de controle (mestre), sendo a mesma instrução executada simultaneamente sobre diversos conjuntos de dados (Figura 2.2). Este modelo pode ser aplicado na manipulação de matrizes e processamento de imagens.

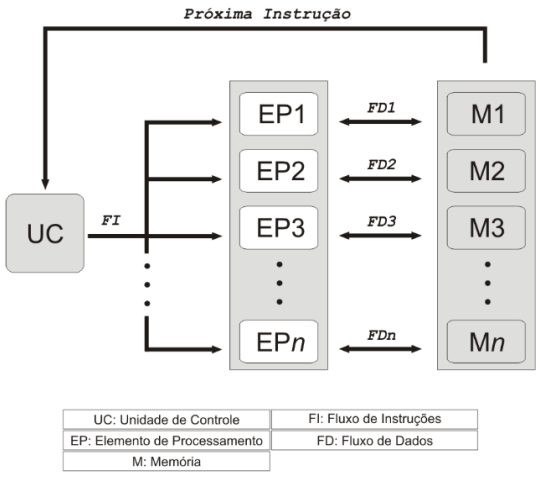

Figura 2.2: Modelo computacional SIMD

- **MISD** (*Multiple Instruction Single Data*) - Fluxo múltiplo de instruções sobre um único conjunto de dados. É um tipo de arquitetura em que múltiplos processadores executam diferentes instruções em um único conjunto de dados (Figura 2.3). Apesar de ser uma arquitetura de fácil entendimento, há pouco interesse nesta arquitetura pela dificuldade no mapeamento dos programas neste modelo. Pode-se dizer que a arquitetura *pipeline* se enquadra parcialmente neste sistema, desde que seja considerado que os dados são diferentes após o processamento de cada fase do *pipeline*.

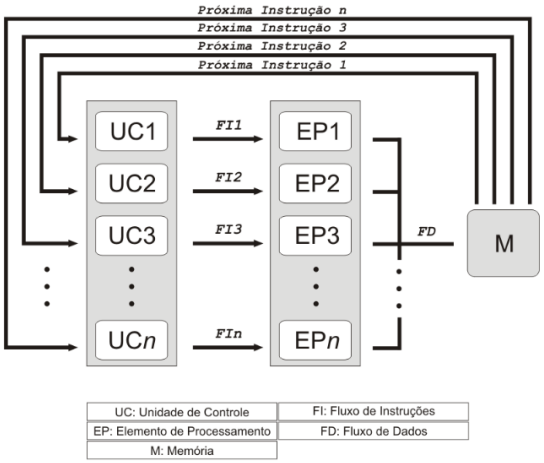

Figura 2.3: Modelo computacional MISD

- **MIMD** (*Multiple Instruction Multiple Data*) - Esta classe é definida por fluxo múltiplos de instruções e múltiplos conjunto de dados. Múltiplos processaores executam diferentes instruções em diferentes conjuntos de dados, de forma independente (Figura 2.4). É a arquitetura mais familiar de processamento paralelo e tem vários processadores interconectados. A maioria dos elementos MIMD é homogênea com todos os processadores idênticos. A distribuição dos elementos de processamento é dada em uma rede local ou até mesmo em uma rede aberta. A comunicação entre os processadores pode ser realizada por meio do endereçamento de um espaço de memória compartilhada ou por passagem de mensagem. Porém, alguns problemas devem ser considerados como: manter a consistência da memória, coerência da *cache* e a coordenação da mensagens.

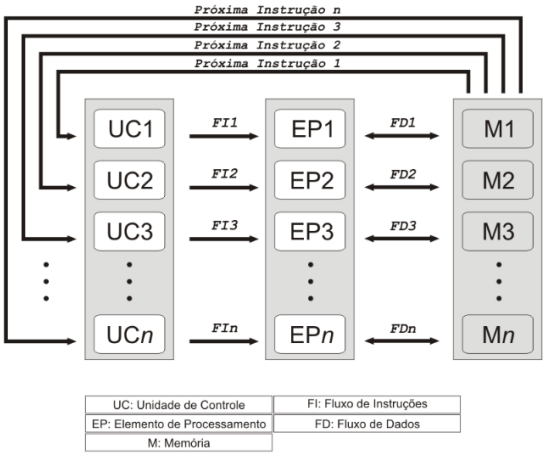

Figura 2.4: Modelo computacional MIMD

Segundo Tanenbaum (2007), a classe MIMD, proposta por (Flynn e Rudd, 1996), pode ser subdividida em multiprocessadores e multi-computadores.

Os multiprocessadores (arquitetura de memória compartilhada ou centralizada) usam uma única memória global, que é compartilhada entre os processadores para realizar a comunicação entre os processos, característica de sistemas fortemente acoplados. A adoção de técnicas propostas por (Yu et al., 2005) e (Radović e Hagersten, 2001) tentam reduzir o efeito gargalo ao manter os dados da cache coerentes. Os multicomputadores possuem memória distribuída e os processos comunicam-se por meio de passagem de mensagens sobre uma rede de interconexão dedicada.

Apesar de antiga, a taxonomia de Flynn ainda é muito utilizada, mas apresenta algumas deficiências para representar as novas arquiteturas computacionais. Muitas vezes é difícil classificar um computador dentro de um modelo arquitetural ou outro, uma vez que a evolução das tecnologias acabaram se combinando ao decorrer dos anos. Entretanto, (Duncan, 1990) propôs uma nova classificação que estende a taxonomia de Flynn, que permite melhor enquadrar os novos computadores paralelos da atualidade, conforme ilustra a Figura 2.5.

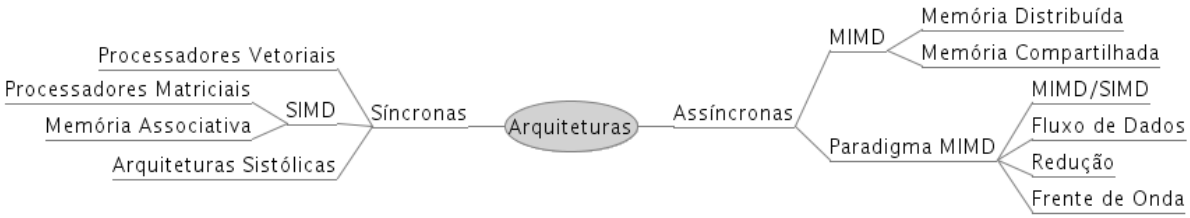

Figura 2.5: Classificação de Duncan

Algumas arquiteturas foram extintas da classificação proposta por (Duncan, 1990), uma vez que determinados modelos já se tornaram comuns nos computadores modernos e que possuem apenas mecanismos de paralelismo de baixo nível, como unidades funcionais múltiplas em uma único processador e o *pipeline* dos estágios de execução de uma instrução.

Tanenbaum (2007) apresenta uma organização composto pelas quatro categorias definidas por Flynn conforme o modelo esquemático ilustrado na Figura 2.6. A arquitetura usada para a realização dos experimentos deste trabalho é a *Cluster of Workstations* (COW) delineada na representação da classificação de Tanenbaum.

O desenvolvimento de redes comerciais de alta velocidade permitiu agrupar vários computadores e interligá-los com dispositivos comuns e a preços acessíveis, concebendo, então, uma nova categoria de computadores paralelos e distribuídos denominada *cluster*,

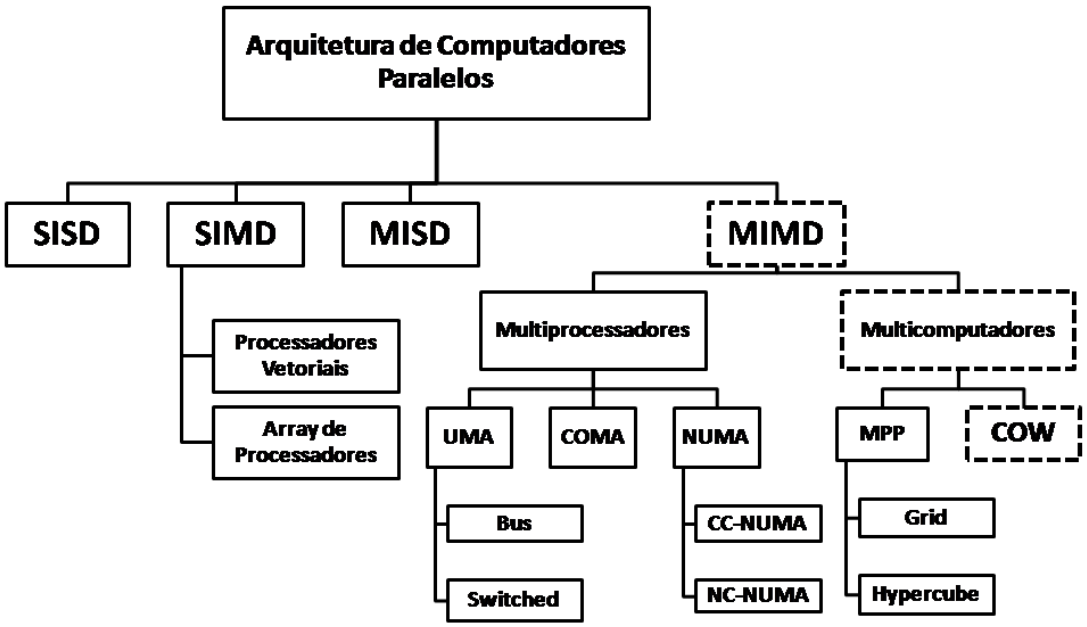

Figura 2.6: Classificação de arquiteturas paralelas proposto por Tanenbaum

que é uma arquitetura formada por vários computadores interconectados por meio de um barramento de comunicação de alta velocidade que pode apresentar um desempenho comparável aos supercomputadores encontrados somente em grandes instituições anos atrás.

Assim, um *cluster* pode ser uma alternativa ao processamento paralelo de muitas instituições que não dispõem de recursos financeiros e de espaço físico para alocar um supercomputador. E ainda, pode ser construído utilizando computadores comuns *off-the-shelf* ou até mesmo com computadores que se tornaram obsoletos com o decorrer do tempo (Meredith et al., 2003).

A NASA idealizou, em meados da década de 90, o primeiro projeto de um típico *cluster* chamado de *Beowulf*, com a objetivo de processar as informações espaciais. Além de oferecer baixo custo de implementação, apresenta alta disponibilidade e é tolerante a falhas, pois os recursos podem ser duplicados de maneira redundante, cosiderados fatores importantes em sistemas de alto desempenho. Também oferece alta disponibilidade, sendo necessário apenas adicionar uma nova máquina para aumentar o poder de processamento. Porém, devem ser considerados alguns aspectos quanto ao *overhead* causado pela latência de comunicação de rede entre os processos.

Um *cluster* pode ser homogêneo ou heterogêneo: homogêneo quando todos os equipamentos em todos os nós são iguais e, heterogêneo, caso contrário. Além dos *clusters*, existe outra categoria denominada por *Network of Workstation* (NOW) em que computadores

que não são exclusivamente dedicados à computação paralela também são aproveitados para efetuar o processamento de aplicações paralelas. Contudo, os *clusters* não são utilizados apenas em sistemas em que se busca o desempenho. Baker (2000) estende sua aplicabilidade em outras áreas, de acordo com a sua funcionalidade em quatro categorias básicas: alta disponibilidade, balanceamento de carga, computação distribuída e computação de alto desempenho, sendo possível a combinação entre esses, conforme a necessidade. Para se ter uma idéia da crescente popularidade dessa arquitetura, no ano de 2008 81.80% (409) dos TOP500 supercomputadores mais rápidos do mundo eram formados por *clusters*. Em 2009, esse percentual subiu para 83.4% (417) totalizando 2.556.728 processadores capazes de atingir um pico de processamento de 2.722.084 gigaflops.

Com o constante avanço das pesquisas em busca de novas tecnologias, a mais nova arquitetura paralela denominada *Grid Computing*, consiste no agrupamento de vários computadores e/ou *clusters* geograficamente distribuídos seja em uma rede local ou Internet. Como em qualquer outra arquitetura paralela, onde a comunicação entre os processadores é um dos principais vilões, os *Grids* também levam em consideração este aspecto e, geralmente, pelo motivo da alta latência das redes em que opera, só se torna viável com determinadas aplicações que façam o baixo uso da comunicação constante de dados. O projeto FOLDING@HOME (Larson et al., 2009) lançado em 2000, no qual milhares de computadores conectados à Internet, realizam intensivas simulações da síntese de proteínas e outros elementos químicos enquanto estão ociosos como se fossem um grande supercomputador virtual.

O Laboratório de Computação de Alto Desempenho (LECAD) em parceria com o Núcleo de Processamento de Dados da Universidade Estadual de Maringá dispõe de um *cluster* que foi utilizado para a realização dos experimentos deste trabalho. Este *cluster* é formado por seis nós da Sun MicroSystems interconectados por meio de uma rede Ethernet Gigabit. A configuração dos nós é detalhada no Capítulo 6.

Neste trabalho não serão explanados todos os tipos propostos na taxonomia de (Tanenbaum, 2007). Devido ao escopo, discutiremos o modelo de memória compartilhada, memória distribuída e memória compartilhada e distribuída.

2.3 Modelos de Programação

Conforme visto anteriormente, na arquitetura MIMD, cada processador executa instruções independentes sobre um conjunto de dados independentes.Analogamente, podemos visualizar vários trabalhadores em uma construção. Um trabalhador (processador) pode

realizar tarefas (instruções) diferentes como pintar uma parede, enquanto outro realiza a pintura da parede externa de maneira independentente e simultânea.

O intercâmbio de dados entre processos é baseado em dois paradigmas de comunicação: memória compartilhada e passagem de mensagem. Cada um desses modelos empregam diferentes métodos de gerenciamento do acesso aos dados da memória. (Fleisch, 1989) adiciona uma categoria híbrida, denominada por *distributed shared memory* (DSM) cuja comunicação é realizada por meio do mapeamento das memórias compartilhadas de cada máquina dando a visão de uma única memória para todo o ambiente de execução, ou seja, as memórias fisicamente separadas podem ser endereçadas como um espaço único de endereço logicamente compartilhado.

2.3.1 Memória Compartilhada

Em arquiteturas de memória compartilhada a comunicação é implícita, pois a memória é acessível diretamente por todos os processadores. O paradigma de programação para computadores de memória compartilhada foca a concorrência e sincronização com técnicas para minimização de *overhead* (Grama et al., 2003).

Esta arquitetura pode variar de acordo com os mecanismos de compartilhamento de dados, modelos de concorrência e sincronização. Modelos baseados em processos assumem que todos os dados associados a um processo é privado. Isso é importante para garantir a proteção em sistemas multi-usuário, porém não é necessário quando diversos processos estão cooperando para solucionar o mesmo problema. O modelo de memória compartilhada é ilustrado pela Figura 2.7.

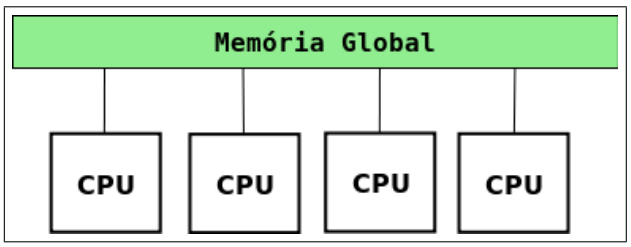

Figura 2.7: Modelo de memória compartilhada

A vantagem dos sistemas de memória compartilhada é o modelo que permite o simples compartilhamento de dados por meio de um mecanismo uniforme de leitura e escrita das estruturas em uma memória comum. Devido a facilidade de programação, o custo do desenvolvimento de um software paralelo é reduzido, mas adicionam uma complexidade extra ao requerer uma sincronização explícita por meio do uso de barreiras e *locks* que podem gerar erros não determinísticos, muitas vezes difíceis de detectar e corrigir. Além

disso, multiprocessadores com memória compartilhada normalmente sofrem com o aumento da contenção e a alta latência no acesso da memória, muitas vezes resultando em baixa escalabilidade e desempenho (Dongarra e Dunigan, 1997).

2.3.2 Memória Distribuída

O paradigma de passagem de mensagem é uma abordagem usada para a programação de computadores paralelos distribuídos. A passagem de mensagens pode ser realizada através de bibliotecas como o MPI (Burns et al., 1994), (Graham et al., 2006), (Gropp e Lusk, 1996) e PVM (Al Geist et al., 1994). A visão lógica de uma máquina que oferece suporte a este modelo consiste em p processos, em que cada um possui a sua memória local. Os dados de inicialização do problema bem como os resultados parciais e finais, são transmitidos por meio de subrotinas explícitas de passagem de mensagem (Grama et al., 2003). A visão lógica do ambiente de passagem de mensagem é composto por um conjunto de p processadores. Cada processador possui a sua memória local como pode o modelo representado pela Figura 2.8.

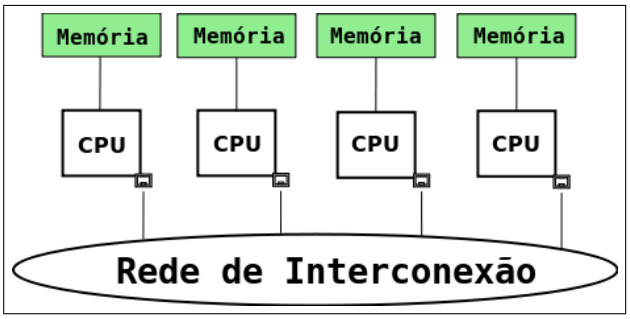

Figura 2.8: Modelo de memória distribuída

Para um processo emissor enviar uma cópia do dado a outro processo é utilizado uma função. Esta função necessita de parâmetros como o endereço de destino, a mensagem e o tamanho da mensagem. O processo receptor deve especificar o tamanho do *buffer* de entrada e a identificação do endereço do processo remetente. Este modelo pode ter a comunicação síncrona e assíncrona.

O desempenho do modelo de passagem de mensagem é usualmente medido pelo tempo ou tráfego de dados. Diversos fatores afetam o desempenho na passagem de mensagem: o número de vezes que uma mensagem precisa ser copiada e o tamanho da mensagem. Além disso, a banda de rede, o volume de concorrência, o gerenciamento das passagens de mensagens podem influenciar no desempenho da aplicação (Chandra et al., 1994), (Lu et al., 1995). A latência do meio de interconexão pode afetar o desempenho de aplicações

de memória distribuída. Para diminuir ou otimizar as comunicações nestes ambientes, existem diversas pesquisas relacionadas que buscam reduzir o gargalo da comunicação como (Tang e Yang, 2001), (Karwande et al., 2003), (Small e Yuan, 2009), (Kühnemann et al., 2004).

2.3.3 Memória Compartilhada Distribuída

O modelo de passagem de mensagens exige que o programador faça chamadas explícitas no código para enviar ou receber dados de um computador a outro. Esse processo abre uma maior possibilidade a erros de programação e torna difícil a depuração de código. Entretanto, a passagem de mensagem não precisa de mecanismos de sincronização para controlar o acesso simultâneo. Esse controle de sincronização pode aumentar significantemente o tempo de execução de aplicações paralelas (Wilkinson e Allen, 2005).

No ponto de vista do programador, o paradigma de memória compartilhada é atraente devido a facilidade de abstração. Assim, várias pesquisas têm sido alavancadas para o desenvolvimento de ambientes de memória compartilhada distribuída. Neste paradigma, cada memória encontra-se respectivamente distribuída com o seu processador, mas cada processador tem uma visão de um espaço único e global da memória como pode ser visto na Figura 2.9. Para que cada processador acesse um endereço que não esteja mapeado à sua memória local, é realizada a passagem de mensagem implícita entre os computadores.

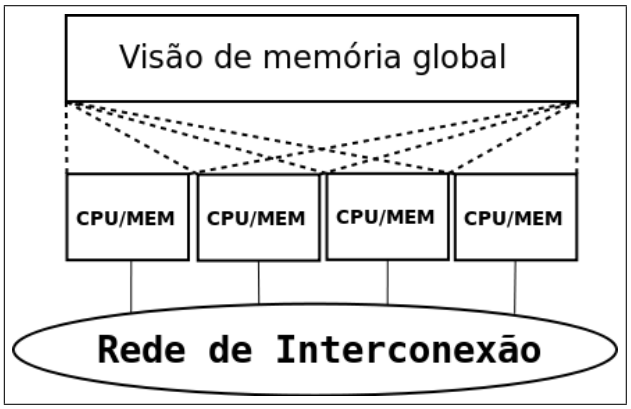

Figura 2.9: Modelo de memória compartilhada distribuída

O conceito de memória compartilhada distribuída é atraente para os programadores. Porém, esses sistemas tem a característica de difícil implementação, pois esses sistemas precisam garantir que todos os processos tenham imagens idênticas do espaço de endereço compartilhado e devem tratar os atrasos de comunicação (Freisleben e Kielmann, 1995).

2.4 O Protocolo MPI

Um dos principais protocolos utilizados na programação de aplicações paralelas com base no modelo de passagem de mensagens é o *Message Passing Interface* (MPI) formado por uma biblioteca de funções (em C) ou sub-rotinas (em FORTRAN) que permite os programadores desenvolver programas paralelos, portáveis e escaláveis em arquiteturas de memória distribuída para executar a comunicação de dados entre processos (Lusk, 2001). O padrão MPI foi criado em 1993 pelo MPI Forum composto por um grupo de fabricantes de computadores paralelos, cientistas da computação e programadores. O principal objetivo deste grupo foi padronizar os diversos sistemas de passagem de mensagens existentes que não eram compatíveis entre si.

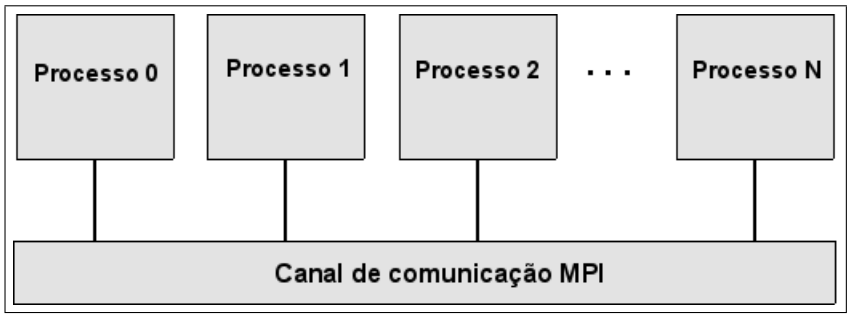

Figura 2.10: Processos sobre a plataforma MPI

Geralmente, os programas baseados em passagem de mensagens executam múltiplos processos de aplicações seriais que utilizam chamadas de bibliotecas para realizar a comunicação, conforme ilustrado na Figura 2.10. A execução de programas por meio de passagem de mensagens pode conter um conjunto fixo de processos, em que cada processo pode executar programas diferentes. Deste modo, o MPI oferece suporte tanto ao paradigma *Single Program, Multiple Data* (SPMD) no qual é executado um único programa sobre o mesmo conjunto de dados, quanto o paradigma *Multiple Program, Multiple Data* (MPMD) em que podem ser executadas diferentes programas ainda que no mesmo conjunto de dados. O MPI permite a execução de mais uma instância de processo no mesmo processador, isto permite explorar melhor os recursos dos processadores com tecnologia *multicore*, apesar da redução do desempenho em função do gargalo de comunicação. O MPI oferece funções que permitem realizar comunicações globais ou ponto-a-ponto, sincronização dos dados e definição de tipos de dados básicos ou derivados, sendo atributos fundamentais para que não ocorram erros durante a comunicação entre os processos (Geist et al., 1996).

O MPI possibilitou o desenvolvimento e a execução de aplicações paralelas em *clusters*, computadores paralelos ou heterogêneos, permitindo a resolução de problemas suficientemente complexos, e não leva em consideração aspectos relacionados à segurança do barramento de comunicação (Gropp et al., 1999). Além disso, as implementações MPI permitem a execução em diversas tecnologias de rede como a InfiniBand, Myrinet e, a mais usada até hoje, a Ethernet que são comparadas em (Majumder e Rixner, 2004). E para cada uma dessas tecnologias, diversas pesquisas estão constantemente tentando otimizar a eficiência na comunicação, analisando os aspectos que reduzem o seu desempenho como a latência e a largura de banda de rede, como o trabalho de (Nupairoj e Ni, 1994) e (Grove e Coddington, 2005). Outras pesquisas são (Roy et al., 2000) que propõe a implementação de qualidade de serviço (QoS) em redes não dedicadas; na otimização de compiladores (Danalis et al., 2009); no aperfeiçoamento entre comunicação e computação (*overlapping*) em (Patrick et al., 2008).

Além disso, outros trabalhos com maior nível de abstração têm sido pesquisados, como o uso da orientação a objetos (McCandless et al., 1996), uso de novas linguagens como Java (Getov et al., 1999) e C# (Gregor e Lumsdaine, 2008), e, inclusive, metodologias de desenvolvimento de aplicações paralelas (Ojima et al., 2005) e (Bartoli et al., 1995). Deste modo, procuram oferecer subsídios que facilitem o desenvolvimento de aplicações paralelas e explorar o potencial das tecnologias já consolidadas.

2.5 Protocolo HLRC

Os ambientes de memória compartilhada distribuída usam técnicas para implementar a ideia de sistemas de memória compartilhada. As implementações em *hardware* encarecem o custo da produção do sistema tanto em aspectos financeiros quanto na complexidade da arquitetura do *hardware* (Faustino, 2003). Uma alternativa aos sistemas em *hardware*, podem ser desenvolvidas implementações em *software* que permitem utilizar arquiteturas como *clusters*. Porém, esse tipo de implementação sofre com a performance limitada devido ao *overhead* gerado pelo algoritmo de coerência de dados implementado.

Para desenvolver um sistema de memória compartilhada distribuída é preciso analisar o modelo de consistência de memória, o protocolo de coerência e a localidade dos dados.

2.5.1 Modelo de Consistência de Memória

O modelo de consistência de memória define o momento em que os outros processadores poderão ver os dados modificados por um processador.

Existem diferentes modelos de consistência para serem aplicados a diferentes tipos de aplicações paralelas. A escolha do modelo afetará diretamente o desempenho do sistema na execução das aplicações paralelas. Pode-se elencar dois grupos de consistência: forte e fraco:

- Modelo forte: aumentam a latência dos acessos à memória além do uso mais expressivo de *bandwidth*, porém reduz a complexidade de programação. O modelo de consistência sequencial é um exemplo que trata os acessos à memória como instruções de escrita e leitura. Além disso, determina que todos os processadores observem a sequência das instruções de leitura e escrita na mesma ordem em que seriam realizadas por um único processador.

- Modelo fraco: neste grupo os dados compartilhados apenas podem ser considerados como consistentes após uma sincronização. O acesso às variáveis de sincronização é sequencialmente consistente; e não se permite o acesso a uma variável de sincronzição enquanto os acessos anteriores não forem completados.

2.5.2 Protocolos de Coerência

Um protocolo de coerência de memória deve definir quais dados poderão ser vistos por outros processadores em pontos de sincronização (Shi et al., 1998). Um protocolo é constituído por uma estrutura de dados e algoritmos que implementam um determinado modelo de consistência. Essas implementações são realizados para gerenciar as falhas de acesso e especificar as primitivas de sincronzição.

O tratamento de falhas de acesso especifica quais rotinas devem ser executadas quando o acesso a uma unidade de compartilhamento é inválido. Quando ocorre a falha de acesso, o protocolo deve buscar uma cópia atualizado da unidade referenciada.

As primitivas de sincronismo especificam os pontos de sincronismo entre dois processos. Tradicionalmente, como em muitos sistemas de memória compartilhada, o protocolo deve oferecer a primitiva *lock* e *barrier*. Os *locks* são empregados para eliminar possíveis condições de disputa em modelos fracos. As *barriers* (barreiras) determinam os pontos de sincronismo entre todos os processadores do ambiente de execução.

2.5.3 Localidade de Dados

Para resolver o problema de manter a coerência dos dados pode-se estabelecer uma residência para cada unidade de compartilhamento de dados como , por exemplo, páginas. Cada residência possuirá uma cópia da página para escrita e uma relação de todos os

processadores que contém uma réplica da unidade. Determinar a residência fornece um método simples para o gerenciamento da consistência das páginas em memória.

Caso exista apenas uma cópia para escrita no sistema e depois processadores executarem uma instrução de escrita, ocorrerá um problema de sincronismo. Este problema pode ser solucionado por um dos seguintes mecanismos (Stenstrom, 1990) *write-broadcast* ou invalidação.

A técnica de atualização envia, após a execução de uma instrução de escrita, uma cópia atualizada da página de compartilhamento para todos os processadores que possuírem uma réplica da unidade.

Na técnica de invalidação, uma escrita em um dado compartilhado repercute na invalidação de todas as suas cópias, que passam a ser inacessíveis. Neste momento, somente uma cópia válida passa a existir no sistema. As próximas instruções de escrita na cópia válida não irão produzir efeito algum sobre as outras cópias. Na próxima operação em uma cópia invalidada, ocorrerá uma falha de acesso, sendo necessário que uma cópia válida seja buscada para a memória local.

2.5.4 A Implementação do Protocolo HLRC

O *home-based lazy release consistency* (Iftode, 1998) é uma implementação de um modelo de memória compartilhada distribuída. Uma versão do protoclo HLRC com suporte ao protocolo TCP/IP foi desenvolvida pela COPPE - UFRJ que permite a programação de aplicações paralelas na linguagem C/C++. Diferentemente do MPI, a comunicação entre processos no HLRC é implícita o que facilita a programação tirando a responsabilidade do desenvolvedor no controle do envio e recebimento de mensagens.

O protocolo HLRC pode ser implementado totalmente utilizando *diffs* como no protocolo LRC tradicional (Keleher et al., 1992). Em protocolos *home-based*, um nó é nomeado para manter a página mais atualizada. Para obter a página mais atualizada, um nó requisita a página ao processador *home*. Quando um nó realiza uma operação de escrita na página, este deve realizar um *broadcast* para os demais processadores, atualizando o nó *home* do dado garantindo a consistência da página.

No HLRC, no final de cada intervalo são computados os *diffs* para todas as páginas modificadas. Após criados, estes são enviados para os seus respectivos *home* onde são atualizados. Os *homes* de páginas aplicam *diffs* a medida que estes chegam e, em seguida, os discarta.

Para manter o sincronismo, o HLRC dispõe ao desenvolvedor as primitivas de *lock* e *barrier* descritos em (Faustino, 2003).

O protocolo HLRC apresenta vantagens sobre o modelo LRC por manter um controle mais simples, enviar poucas mensagens e pelo menor *overhead* e consumo de memória.

2.6 Desempenho e Eficiência

A computação paralela tem como um de seus principais objetivos o aumento do desempenho de programas, em busca da redução do tempo computacional em relação à versão sequencial. Para medir o ganho de desempenho de programas paralelos, foram criadas diversas métricas que expressam numericamente os fatores da redução do tempo de execução de um programa paralelizado em P processadores. Entre essas métricas podemos citar o *speedup* e a eficiência.

O *speedup* é uma equação que possibilita medir a razão entre o tempo gasto para executar uma aplicação em um único processador, pelo tempo de execução em paralelo, conforme a seguir:

$$speedup_{(n)} = \frac{tempoExecucao_{(serial)}}{tempoExecucacao_{(paralelo)}}$$

Sun e Ni (1990), Hwang (1993), Sahni e Thanvantri (1996) definem duas conotações para o *speedup*: absoluto e relativo. O *speedup* absoluto é a relação entre o tempo do melhor algoritmo sequencial pelo tempo gasto com uma versão do algoritmo paralelo em P processadores. O *speedup* absoluto pode levar em consideração os recursos da máquina ou não. No primeiro caso, o *speedup* é definido como a relação entre o tempo gasto com o melhor algoritmo sequencial em um processador e o tempo gasto pelo algoritmo paralelo. No segundo caso, o *speedup* é definido pelo tempo do melhor algoritmo sequencial executado na máquina mais rápida em relação ao tempo do algoritmo paralelo na máquina paralela.

O *speedup* relativo considera o paralelismo de maneira igualitária. É definido pela relação entre o tempo gasto pelo algoritmo paralelo em 1 processador e o tempo gasto pelo algoritmo paralelo em P processadores. O *speedup* relativo permite avaliar como o desempenho do algoritmo varia de acordo com o número de processadores, já que é comparado consigo mesmo.

Ao utilizar dez processadores em um ambiente paralelo, por exemplo, logo vem a ideia de que as aplicações executadas neste ambiente serão processadas dez vezes mais rápidas. Este desempenho considerado linear, raramente é alcançado na prática (Patterson e

Hennessy, 2005). Amdahl (1967) observa que se a parcela sequencial de uma aplicação consome $1/s$ do tempo de execução total, logo o *speeedup* máximo possível por um computador paralelo é $1/s$. Por exemplo, se um programa precisa de 20 horas para ser executada em um ambiente com apenas 1 processador, e contém uma parcela de 1 hora que não pode ser paralelizada, restando 19 horas (95%) de porção paralelizável, então mesmo utilizando centenas de processadores na execução desta aplicação, o tempo mínimo de execução não pode ser menor do que 1 hora. Assim, o *speedup* máximo é 20x.

Segundo Sun (1991), os aspectos inerentes a fração sequencial dos programas que podem degradar o desempenho, podem ser:

- Latência de comunicação: o tempo da transmissão e a recepção das mensagens no barramento de interconexão da rede, bem como o tempo gasto no particionamento dos dados e a espera nas filas de transmissão;
- Distribuição e balanceamento de carga: em ambientes paralelos heterogêneos, uma parcela do trabalho pode terminar antes que as demais devido ao mal balanceamento da carga, o que torna uma unidade de processamento ociosa gerando um desperdício computacional;
- Granularidade e dependência: algoritmos que tem como natureza a comunicação intensa entre os nós, muitas vezes, dependem do progresso parcial do processamento de um outro nó para que então se possa dar continuidade ao processamento.

Algumas vezes um *speedup* pode atingir um coeficiente maior que o número total de elementos de processamento, caracterizado como *speedup superlinear*. O *speedup superlinear* dificilmente pode acontecer e, às vezes, sua ocorrência é de difícil compreensão. As possíveis razões para se obter um coeficiente *super linear* pode possuir diversos motivos como a variação de diferentes hierarquias de memória; o melhor aproveitamento dos dados retidos em cache reduzem dramaticamente o acesso à memória favorecendo para a obtenção de um desempenho computacional extra; e a sobreposição de tarefas (*overlapping*) *cpu-bound* e *i/o-bound* busca eliminiar o tempo gasto com instruções de entrada e saída conforme analisado nos trabalhos de (Brightwell e Underwood, 2004), (Patrick et al., 2008) e (Marjanović et al., 2010).

A eficiência é outra métrica adotada para se medir o ganho de desempenho em ambientes paralelos. É determinada pela fração entre o *speedup* de P processadores sobre P conforme representado na equação a seguir:

$$eficiencia_{(p)} = \frac{speedup_{(p)}}{p}$$

Capítulo

3

Metodologias, Granularidade e Trabalhos Relacionados a Clusters

Este capítulo apresenta uma visão geral sobre alguns dos principais conceitos associados aos *clusters* de computadores e as aplicações paralelas e distribuídas, abrangendo metodologias de paralelização, incluindo a PCAM, escalabilidade em ambientes paralelos e a granularidade do paralelismo. Diversos trabalhos relacionados também são apresentados.

3.1 Metodologias de Paralelização de Algoritmos

Desenvolver um algoritmo sequencial para solucionar um problema computacional é o modo convencional de ensino de programação e comumente adotada na prática. A forma de abstrair o processo de um programa tendo um início, meio e fim, no qual uma etapa é realizada somente após o termino da anterior, ocorre entre o início do seu processo até o seu término (Evans e Goscinski, 1995).

De acordo com (Foster, 1995), um algoritmo paralelo deve considerar quatro aspectos explanados a seguir:

- Concorrência: capacidade de duas ou mais instruções serem executadas simultaneamente;

- Escalabilidade: capacidade de oferecer suporte ao aumento do número de processadores;
- Modularidade: habilidade de decompor componentes complexos em entidades mais simples permitindo até mesmo a reusabilidade pregada pela engenharia de software, tanto no desenvolvimento de aplicações sequenciais quanto nas paralelas;
- Localidade: é a razão entre a comunicação da memória local e remota. Esta característica deve ser fortemente considerada em ambientes multicomputadores em que o barramento de comunicação entre processos é uma rede de interconexão cuja latência é inúmeras vezes maior que o barramento memória-processador.

Normalmente, o algoritmo paralelo é desenvolvido a partir de uma implementação sequencial. Durante a análise do código sequencial, o programador deve encontrar os blocos que não possuem dependência entre sí. Se um resultado não tem dependência de outros anteriores, então a paralelização é possível, otimizando o desempenho da aplicação. O desenvolvedor precisa ter ciência dos quatro tipos de paralelismo: de dados, funcional, objetos e domínio.

No paralelismo de dados (lado direito da Figura 3.1), os dados são divididos entre os elementos de processamento, podendo cada parte ser manipulada por uma réplica de um mesmo código ou por um código distinto. Por exemplo, na multiplicação matricial, o mesmo tipo de operação de multiplicação pode ser executada sobre diferentes linhas e colunas e cada uma independe do resultado de operações anteriormente realizadas. Os resultados dessas õperações parciais podem ser enviados ao processo que gerencia os dados, para que sejam agrupados na matriz resultante, conforme o exemplo mostrado a seguir.

$$A = \begin{bmatrix} 1 & 4 & 0 \\ -2 & 4 & 3 \end{bmatrix} \times B = \begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix}$$

$$AxB = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + ... + a_{in}b_{nj}$$

A paralelização funcional divide o código entre os elementos de processamento, podendo estes atuarem sobre o mesmo conjunto de dados ou sobre dados distintos. O problema "leitores e escritores"é um exemplo em que o processo "escritor"atua sobre os dados gerando alguma informação que será "lida"pelo processo "leitor". O paralelismo funcional pode ser representado pelo modelo esquemático no lado esquerdo da Figura 3.1.

Um conceito mais recente, o paralelismo de objetos é formado por elementos distribuídos por diversas redes. O paralelismo de objetos pode ser usado por procedimentos em

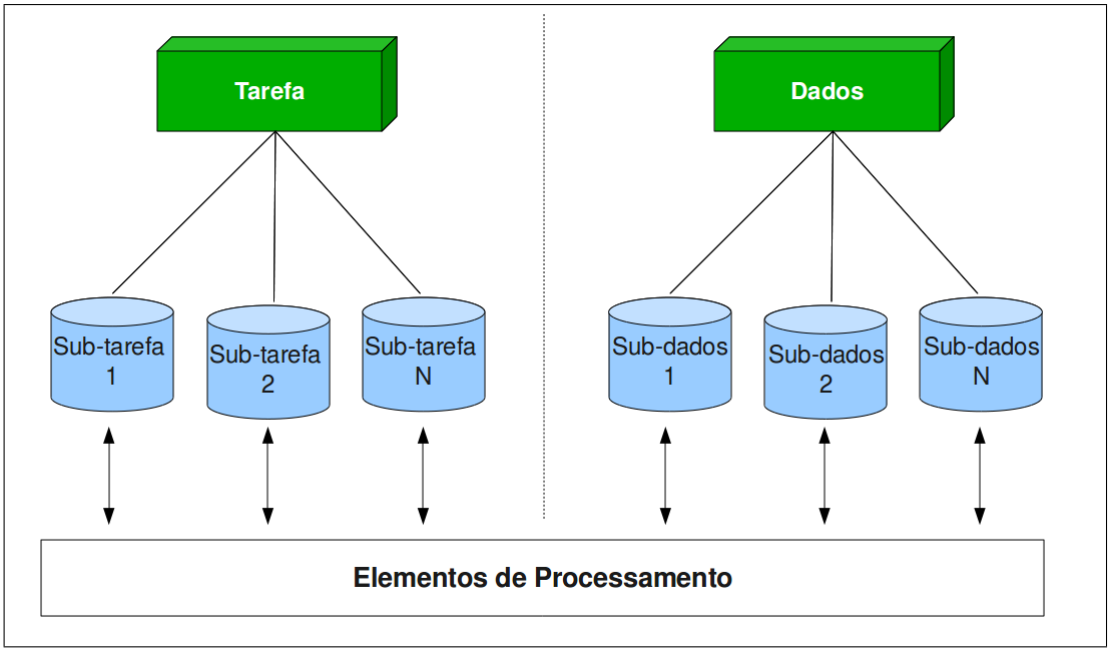

Figura 3.1: Esquema do paralelismo de dados

diferentes processadores para uma determinada finalidade. Por exemplo, um método pode utilizar dados provenientes de vários serviços web simultaneamente para a resolução de um problema em específico em processadores distintos, conforme ilustrado na Figura 3.2.

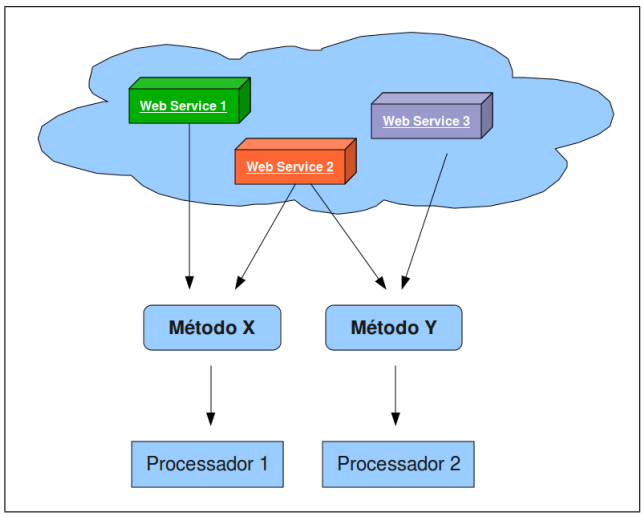

Figura 3.2: Esquema do paralelismo de objetos

O paralelismo de domínio utiliza a técnica de "dividir e conquistar"para gerar sub-domínios menores que podem ser resolvidos paralela e independentemente utilizando métodos comuns para análise linear de sistemas como visto em (Smith et al., 2004) e (Saad, 2003). Este paralelismo surgiu quando os problemas ultrapassavam a capacidade

da memória disponível nos computadores. Agora com os novos arranjos arquiteturais, é aplicado para aumentar a performance de aplicações paralelas, reduzindo o seu tempo de processamento. Em um ambiente paralelo, cada subdomínio é enviado a um processador diferente, cabendo a um algoritmo "automático"dividir o grafo original e identificar os vértices limitantes e os nós internos de cada subdomínio (Nascimento et al., 1996). A Figura 3.3 ilustra a decomposição do grafo da Baia de Guanabara em 8 subdomínios pela heurística *METIS* (Karypis e Kumar, 1999).

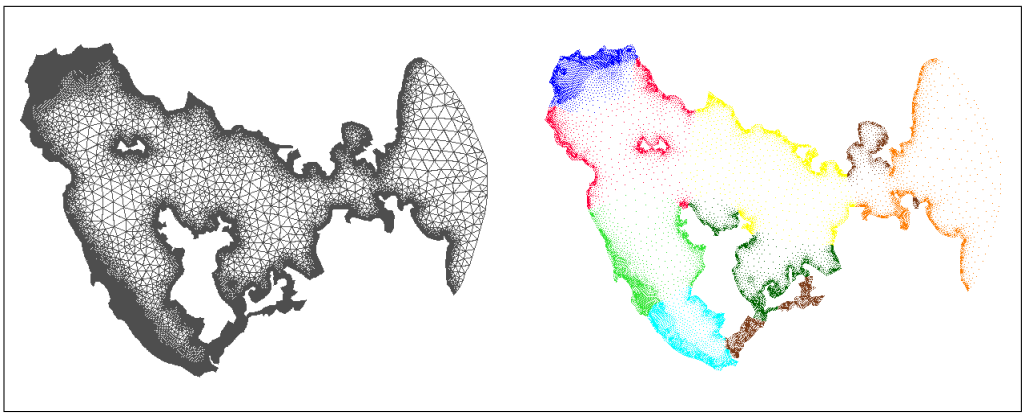

Figura 3.3: Esquema do paralelismo de domínio

Apesar das técnicas de paralelização mencionadas nesta seção, o ponto que mais dificulta a paralelização de um algoritmo é a compreensão total sobre os trechos que podem ser paralelizados. A granularidade do código é outro aspecto que deve ser levado em consideração durante a paralelização de um código. O desempenho de uma aplicação sobre uma determinada arquitetura está diretamente relacionado ao tipo de granularidade do algoritmo, motivo pelo qual detalhamos estes conceitos na próxima seção.

O balanceamento de carga é outro ponto que deve ser considerado, e consiste em distribuir a carga de processamento de modo proporcional ao poder de processamento de cada nó do *cluster*. Assim, o balanceamento precisa ser ponderado principalmente em sistemas heterogêneos em que um computador pode ter o poder computacional diferente que outro, comprometendo o desempenho geral do programa em função da máquina mais lenta. A exemplificar como o balanceamento de carga pode favorecer o processamento em ambientes heterogêneos, podemos imaginar um problema cuja carga varie de acordo com os resultados parciais da aplicação. Enquanto um computador está processando uma carga e observa que outros computadores estão com o nível de processamento mais baixo, este analisa e redistribui a carga na tentativa de equilibrar o processamento em todo o ambiente paralelo. Apesar da sua importancia, não é o foco do presente trabalho se aprofundar neste assunto.

Foster (1995) descreve uma metodologia para auxiliar no processo de desenvolvimento de uma aplicação paralela denominada PCAM . Esta metodologia está organizada em quatro fases distintas: particionamento, comunicação, aglomeração e mapeamento. Os quatro estágios estão ilustrados na Figura 3.4 e são descritos a seguir:

1. Particionamento: nesta primeira etapa, é realizada a decomposição tanto das operações quanto os dados do problema em pequenas tarefas. Não considera as questões práticas como o número de processadores ou a arquitetura em que será executada, mas sim em detectar os potenciais pontos de paralelização no código;

2. Comunicação: Nesta etapa são determinadas as comunicações necessárias para a coordenação da execução das tarefas, definindo os algoritmos e as estruturas de dados necessárias;

3. Aglomeração: Os artefatos gerados nos estágios anteriores são avaliados sob os requesitos de desempenho e custos de implementação. Caso necessário, as tarefas são combinadas em tarefas maiores para melhorar o desempenho e reduzir custos de desenvolvimento;

4. Mapeamento: Nesta última etapa, cada tarefa é distribuída a um processador de maneira que esta aproveite os recursos de processamento e com baixo custo de comunicação. A distribuição das tarefas pode ser estática ou determinada por um algoritmo de balanceamento de carga durante a execução da aplicação.

Assim, observa-se que nos dois primeiros estágios busca-se explorar a concorrência e a escalabilidade. No terceiro e quarto estágio, o foco da atenção muda para a análise da localidade dos dados e dos aspectos relacionados ao desempenho.

3.2 Granularidade de Aplicações

A granularidade pode ser considerada fina quando o paralelismo está presente em nível de instruções ou grossa quando em nível de aplicações (Hayes et al., 1992). Porém, a granularidade de um algoritmo pode ocorrer também em níveis intermediários, podendo envolver outros fluxos de execução como *threads*, funções ou estruturas lógicas do algoritmo, de modo que podemos então definir a granularidade como uma medida relativa. Quanto mais fina, as instruções executadas pelo processador são mais rápidas, mas o tráfego no barramento de comunicação será maior, resultando na saturação do sistema de comunicação e na queda do desempenho da aplicação. Deste modo, quanto maior

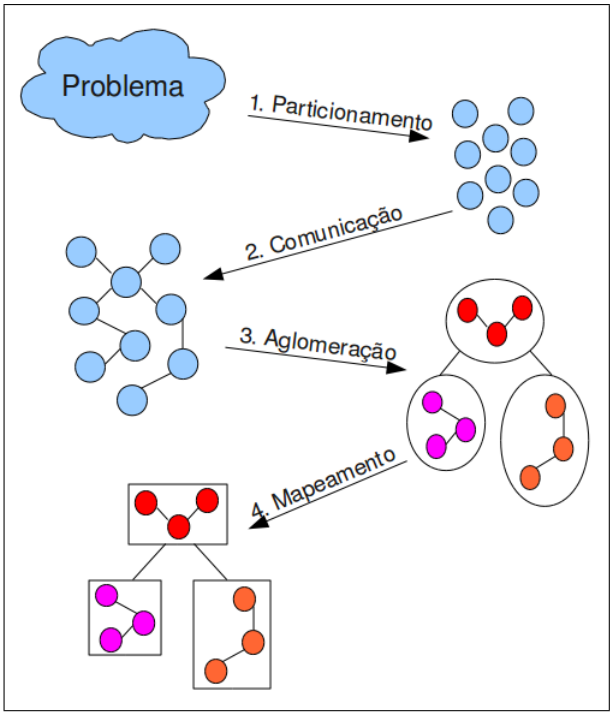

Figura 3.4: Etapas da metodologia PCAM

a granularidade apresentada, melhor será o desempenho obtido, visto que o *overhead* da comunicação será reduzido. As operações em estruturas matriciais são fáceis de paralelizar por não haver dependência sobre as linhas e colunas, o qual permite cada processador efetuar as operações de forma isolada para que, posteriormente, possam ser agrupadas pelo computador mestre.

Qualquer discussão sobre processamento paralelo é inconcebível sem a descrição do nível conceitual da granularidade em que o paralelismo ocorre. Os termos mais usados para esta classificação são o paralelismo *coarse-grain* e *fine-grain* (Tanenbaum e Woodhull, 2006), (Silberchatz et al., 2008). Esses termos estão relacionados ao nível e frequência da comunicação e sincronização que ocorre dentro de um programa paralelo.

Dongarra e Dunigan (1997) analisam o desempenho de sistemas de passagem de mensagem em diversas arquiteturas, incluindo um *cluster* de estações de trabalho. Os autores concluíram que, em *clusters*, a escolha correta da granularidade do paralelismo é um fator que afeta diretamente no desempenho de uma aplicação.

O termo *fine-grain* (granularidade fina) representa um conjunto de dados em um nível mais baixo. O termo baixo nível neste contexto quer dizer que o nível de paralelismo é baseado em instruções mais próximas das instruções de máquina (Tanenbaum e Woodhull, 2006). As instruções do código fonte podem ser executadas em paralelo sendo que cada instrução no código possui poucas instruções para serem enviadas.

O paralelismo *coarse-grain* (granularidade grossa) é baseado conceitualmente em uma abstração mais alta do que o paralelismo de granularidade fina. Esse paralelismo está acima da estrutura lógica do programa. Normalmente, é implementado em procedimentos, funções e alguns outros tipos de agrupamento de códigos.

A granularidade fina e grossa podem ser comparadas à execução de máquinas forte e fracamente acopladas, respectivamente. Tanenbaum e Woodhull (2006) observa que a granularidade fina normalmente é executada em arquiteturas fortemente-acopladas enquanto o paralelismo de granularidade grossa é usualmente encontrado em máquinas fracamente-acopladas.

Paralelismo de granularidade fina

O paralelismo de granularidade fina se refere a comandos mais específicos e é considerado como instruções de nível mais próximos da máquina. As linguagens que oferecem suporte à programação de granularidade-fina podem ser muito específicas e podem não dar suporte a diversos conjuntos de estrutura de dados usados pelos programadores em linguagens de alto nível. A linguagem Occam (Fay, 1984) é um exemplo onde o programador deve construir as suas próprias estruturas antes de usá-las.

Porém, nem todas as linguagens *fine-grain* são tão restritivas como a Occam. Linguagens como Aeolus (LeBlanc e Wilkes, 1986), ParAlfl (Hudak e Smith, 1986), parLOG (Clark e Gregory, 1983) e Concurrent Prolog (Shapiro, 1987) dispõem de um vasto conjunto de estruturas de dados sintaticamente bem definidas. Essas linguagens foram projetadas para permitir a execução de um programa sobre muitos processadores e ocultam os detalhes da comunicação entre os processos do programador, mas necessitam da especificação dos componentes paralelos.

A granularidade fina é também encontrada em computadores que possuem um rápido acesso à memória como em arquiteturas de processadores vetoriais que requerem linguagens especializadas e técnicas para o acessos desses recursos (Grama et al., 2003). Nessas máquinas há sincronizações constantes e todas as mémorias são acessíveis por todos os outros processadores que precisarem fazer o acesso. Os mecannismos de acesso são extremamente rápidos para que o acesso a outros processadores não sofra com o gargalo de comunicação.

Paralelismo de granularidade grossa

O paralelismo de granularidade fina é geralmente descrito como sendo instruções de baixo nível. Algoritmos de granularidade mais grossa conceitualmente estão em um nível de

abstração mais alto do que as *fine-grain*. São baseados na construção lógica do programa. As sincronizações ocorrem quando o programa necessita de um dado para ser trocados ou quando um programa precisa realizar uma ação para garantir a operação correta. O paralelismo de grossa granularidade é geralmente caracterizado por menores quantidades de sincronismo e comunicação de processos

Pode-se citar diversas linguagens que oferecem a concorrência em granularidade grossa: Linda (Ahuja et al., 1988), Orca (Bal et al., 1992) e Argus (Liskov, 1988). Nessas linguagens um programa pode ser executado em diversos computadores, porém o tempo de execução podem não ser os mesmos, então a sincronização e comunicação dos processos com um programa ocorre de diferentes formas do que nas aplicações de granularidade fina.

As máquinas que oferecem suporte a execução de algoritmos *coarse-grain* são normalmente arquiteturas fracamente-acopladas. Um exemplo de arquitetura fracamente-acoplada e utilizado neste trabalho é o *cluster*. Neste tipo de arquitetura, os processos não compartilham fisicamente a mesma memória, precisando assim, um ambiente para a comunicação de processos por meio de alguma forma de mensagem.

Neste sentido, o algoritmo da aplicação desta dissertação se encaixa na classificação de granularidade grossa e utiliza um *cluster* Beowulf, a linguagem C e a biblioteca Open MPI e HLRC para o desenvolvimento e execução das versões paralelas.

3.3 Trabalhos relacionados

O poder computacional oferecido pelos *clusters* e pela plataforma MPI são empregados e descritos em trabalhos de diversas áreas. Na genética e biologia molecular, o sequenciamento de filetes de DNA tem aproveitado enormemente dos recursos computacionais que um *cluster* oferece. Chen e Schmidt (2003) apresenta uma aplicação para equacionar os segmentos dito como ótimos ou próximos de tal para duas sequências em tempo linear e quadrático, mostrando que a paralelização pode ser aplicada neste tipo de algoritmo, conseguindo reduzir significativamente o tempo de execução em *clusters*. O experimento obteve um *speedup* de aproximadamente 41 ao analisar duas cadeias de 816.394 e 580.074 de comprimento, respectivamente. Técnicamente, foram utilizados 64 nós e a implementação MPICH. Apesar dos métodos heurísticos propostos por Chen mostrarem maior rapidez, uma outra abordagem definida por (Boukerche et al., 2004) apresenta maior precisão em longos segmentos de DNA requerida pelos biólogos. A avaliação do desempenho foi realizada em um *cluster* e os resultados apontaram que a solução de Boukerche é melhor em relação aos resultados conquistados na abordagem anterior.

Na área de previsão metereológica, Carpenter Jr (2001) descreve uma aplicação concebida para análise de fenômenos meteorológicos cujos resultados são publicados para várias companhias aéreas comerciais e usinas energéticas. Os dados são oriundos de uma área de aproximadamente 5.760 x 3.600 km e possui também grades horizontais de 40 km, com 32 níveis de altura. Esta aplicação foi executada em um *cluster* sobre a plataforma MPI com 32 processadores Pentium III *dual*. Fischer et al. (1999) relata a paralelização de uma aplicação para previsão metereológica utilizada por um comitê de 13 países que apresentou uma portabilidade e eficiência computacional satisfatória ao ser executado num *cluster*.

Outro segmento beneficiado com o uso da computação paralela em *clusters* é o estudo da Dinâmica dos Fluidos em (Rehm et al., 2003). Neste trabalho, os resultados adquiridos com a simulação numérica de fluxos reativos permitem analisar a combustão do hidrogênio, para propor melhorias na segurança dos sistemas de combustão e, consequentemente, reduzir os acidentes que utilizem esse gás como fonte de energia.

A partir do final da década de 90, a computação em *cluster* deixou de ser usada apenas em aplicações científicas complexas que demandavam milhões de intensivos ciclos de processamento. Diversos setores da indústria comercial que movimentam bilhões de dólares passaram a utilizar os *clusters* para otimizar tarefas que, anteriormente, gastavam muito tempo a ser processadas ou até mesmo eram inviáveis de ser realizadas (Chiola, 1998).

A equipe de automobilismo *F1 Red Bull Racing* também utiliza a computação baseada em *clusters* para realizar simulações dos componentes aerodinâmicos para melhorar o desempenho de seus carros, num esporte onde cada fração de segundo melhorada pode ser decisiva.

Os *clusters* também são usados pelo setor petrolífero para a execução de simulações sísmicas para exploração de novas reservas de petróleo submarinas. A *Leading Oilfield Services Provider* mantém em seu centro de pesquisas 57 diferentes programas para a simulação de novos pontos potenciais de petróleo que são executados em *clusters* HP, IBM, Sun e Dell sobre uma variedade de arquiteturas e sistemas operacionais.

Na indústria cinematográfica, por exemplo, os *clusters* permitiram aperfeiçoar a qualidade dos efeitos especiais e das animações gráficas. O filme *"Titanic"* foi um dos primeiros casos que teve os seus efeitos visuais renderizados em um *cluster*. Outro caso, a produção de filmes animados de 90 minutos com computação gráfica de 24 quadros por segundo resultaria em aproximadamente 130.000 quadros. Cada quadro é quebrado em diversos elementos renderizados: os personagens, pedras, núvens, árvores. Um único elemento pode conter geometrias complexas e levando em conta os diversos efeitos de iluminação, a renderização de um filme completo poderia levar no mínimo 17.7 milhões de horas,

aproximadamente 2.020 anos-processador. Neste panorama, os estúdios cinematográficos passaram a utilizar *clusters* com 2000 a 3000 processadores que permitem renderizar um filme por completo em até 1 ano.

O Instituto Wellcome Trust Sanger emprega 12 *clusters* no processamento de pesquisas sobre o genoma humano em busca da cura de doenças a partir da análise das sequências de DNA. Com a compra de 30 novas máquinas de sequenciamento de genoma, cada uma delas produz dados duas vezes maior que as máquinas anteriores. Logo, uma máquina nova dessas é capaz de gerar dados que ocuparia todo o poder computacional do instituto há 3 anos atrás. A execução de uma única reação de sequenciamento é capaz de levar 3 dias e consumir 2 terabytes de resultado. Para dar suporte a esta necessidade computacional, o Instituto Sanger adquiriu um conjunto de computadores com 710 heterogêneos envolvendo máquinas IBM e HP *dual* e *quad core*, bem como diversas estações SGI Altix com processadores Opteron e Itanium de 64-bit, dando suporte à toda carga computacional necessária.

Capítulo

4

Aplicação Alvo: Processamento de Imagens e o Índice de Fragmentação

Os métodos de filtragem de imagem permitem melhor visualizar certas características de uma imagem, transformando-a em uma forma adequada à aplicação (Pedrini e Schwartz, 2008). Tais métodos estão estruturados em dois domínios: espacial e de frequência. O domínio espacial está relacionado ao processamento da vizinhança de um *pixel* sobre ele.

No domínio espacial, o processamento de um ponto p(x,y) depende do nível de cinza original do próprio ponto e de seus *pixels* vizinhos. A convolução é uma técnica que atua sobre o domínio espacial por meio de uma matriz, denominada de *janela* ou *máscara*, que é aplicada sobre todos os *pixels* da imagem original a fim de produzir outra imagem de saída. Esta técnica pode fazer uso de diferentes algoritmos: soma ponderada dos *pixels* vizinhos, mediana dos *pixels*, número de vizinhos distintos, entre outros. A cada elemento da janela está associado um valor numérico denominado de coeficiente. A aplicação da janela com centro na coordenada (a,b), sendo a uma linha e b uma coluna, consiste na substituição do valor do *pixel* central por um novo valor que depende de operações com os coeficientes vizinhos.

Em filtragens de imagens digitais é possível aplicar os filtros em máscaras de quaisquer dimensões. Porém, alguns programas definem um tamanho máximo das janelas devido a limitações computacionais (Crósta, 1992).

Turner (1989) apresenta o índice de fragmentação para medir o grau de variabilidade da paisagem e revelar as possíveis influências da atividade humana sobre a mesma. O

índice de fragmentação pode ser adotado como uma medida local de textura, com valores no domínio entre zero e um, sendo calculado para cada *pixel* da imagem por meio da aplicação de uma convolução específica na região de viziança do *pixel*. Se todos os *pixels* posicionados em relação à janela de convolução forem de classes diferentes, seu valor será um, indicando máxima variabilidade na região em análise. Em contrapartida, se todos os *pixels* da imagem forem iguais ($c=0$), a variabilidade será baixa e o resultado da medida será zero (Galo e Novo, 1998). Em termos de reconhecimento de padrões, utilizar o índice de fragmentação multidimensional (IFM) significa considerar, além da vizinhança do *pixel* em cada banda, seu contexto nas múltiplas bandas para classificá-lo (Miller et al., 1995).

Neste trabalho foi adotada uma versão sequencial do IFM implementada pelo Grupo de Pesquisas em Geodésia da Universidade Estadual de São Paulo (UNESP) - Campus Presidente Prudente, a qual foi utilizada como referência no presente trabalho e serviu de código base para a criação das versões paralelas. Ele considera imagens com qualquer número de bandas e após o processamento gerar uma única banda de saída. De modo conceitual, cada pixel da imagem é varrido considerando uma janela quadrada de *pixels* vizinhos para determinação da fragmentação que pode ser parametrizada pelo usuário, sendo a janela n-dimensional com linhas e colunas ímpar e profundidade determinada pelo número de bandas de entrada para o processamento.

O algoritmo de cálculo do índice de fragmentação multidimensional, aqui denominado apenas de Algoritmo IFM, foi aplicado sobre a Figura 4.1 (da Cruz e Galo, 2005) que representa o reservatório de Salto Grande situado no estado de São Paulo. A Figura 4.2 ilustra o resultado obtido após o processo de IFM sobre a combinação de duas bandas: vermelho e infravermelho próximo. O foco deste trabalho é estritamente a paralelização do Algoritmo IFM em imagens capturadas por meio de sensoriamento remoto.

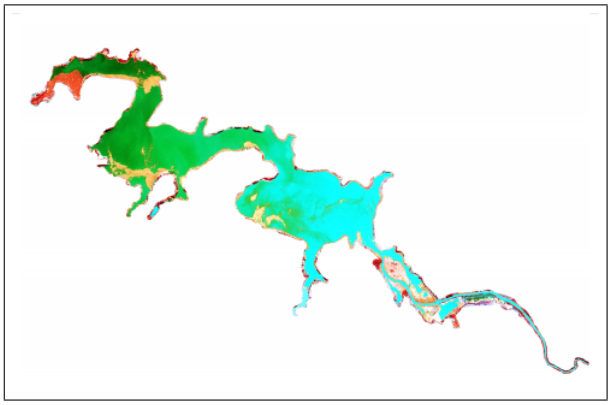

Figura 4.1: Composição da área do reservatório de Salto Grande

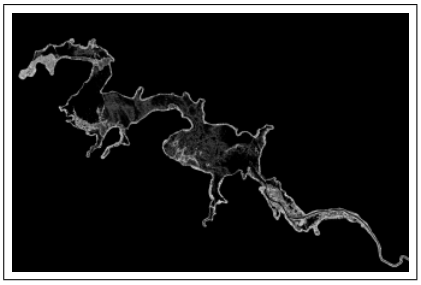

Figura 4.2: Índice de fragmentação aplicado às bandas do vermelho e do infravermelho próximo

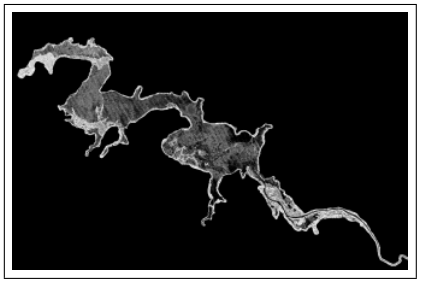

Figura 4.3: Índice de fragmentação aplicado às quatro bandas do sensor Ikonos

Para cada posição da janela, os *pixels* da imagem são mapeados para classes e indexados numericamente em intervalos a que cada *pixel* pertence. O tamanho desses intervalos é determinado a partir da divisão do número total de níveis de cinza da imagem pelo número de elementos da janela (n). Após o mapeamento dos *pixels* da janela para as classes de intervalo, realiza-se a contagem do número de *pixels* distintos (c). O IFM (F) é calculado pela Equação 4.1. A ideia do algoritmo de cálculo do índice de fragmentação é representado na equação:

$$F = \frac{c-1}{n-1} \tag{4.1}$$

Considerando-se uma imagem de 8 bits, tem-se 256 níveis de cinza. Para uma janela de tamanho 3 por 3, os intervalos gerados pela aplicação desenvolvida estão sintetizados na Tabela 4.1. O primeiro procedimento de processamento pode ser visualizado na Tabela 4.2 para uma banda de imagem de estudo, em que cada posição da janela possui apenas um valor que representam o número digital de brilho do pixel para cada banda. Na Tabela 4.3, os valores de brilho foram mapeados de acordo com as classes de intervalos definidos na Tabela 4.1. A atribuição do novo valor de brilho para o pixel central depende do número de valores presentes na janela em cada iteração. No caso da Tabela 4.3, existem

3 valores distintos. Assim, pela Equação 4.1, tem-se o valor de fragmentação 0.250 que é multiplicado, ao fim do passo de processamento de cada pixel da imagem, por 255 para gerar uma imagem final cujos valores de brilho sejam inteiros.

O tempo de execução do Algoritmo IFM cresce de acordo com alguns parâmetros: a dimensão da imagem, o número de bandas de cada imagem e o tamanho da janela (máscara). A dimensão da imagem e o número de bandas alteram linearmente o tempo de execução desde que os demais parâmetros sejam os mesmos. O terceiro parâmetro influencia exponencialmente o tempo de execução.

Nível de cinza inicial do intervalo	Nível de cinza final	Índice
0	28	0
29	57	1
58	86	2
87	115	3
116	144	4
145	173	5
174	202	6
203	231	7
232	255	8

Tabela 4.1: Intervalos gerados para imagem de 8 bits (tom de cinza)

25	91	182
201	14	51
205	35	243

Tabela 4.2: Números originais de uma janela de imagem

0	3	6
6	0	1
7	1	8

Tabela 4.3: Valores mapeados a partir dos números originais da imagem

4.1 Metodologia

Esta seção mostra a metodologia empregada para paralelizar o Algoritmo IFM. Ela foi adaptada em quatro fases a partir da metodologia proposta por Foster (1995). Cada fase adotada é descrita a seguir:

- **Fase 1:** análise do algoritmo sequencial: nesta fase é determinado o domínio do problema bem como o delineamento de quais partes do algoritmo serão beneficiadas com a execução da versão paralela. De acordo com Foster (1995), o particionamento de domínio compreende a tarefa da divisão dos dados que serão distribuídos no ambiente paralelo, em busca do melhor paralelismo possível, tanto em computação quanto nos dados. É importante salientar que quanto menor o tamanho do número de instruções e de dados, melhor é a eficiência do particionamento, desde que não exista um *overhead* de comunicação que possa degradar o desempenho da aplicação. Para realizar o particionamento do Algoritmo IFM descrito no capítulo anterior foi feito um estudo dos possíveis modelos de paralelização dos dados e da definição da lógica do algoritmo que é executada pelo nó mestre e pelos demais nós de processamento. Os demais estágios de Comunicação, Aglomeração e Mapeamento propostos por Foster também foram definidos nesta fase.

- **Fase 2:** nesta fase é realizada basicamente a implementação das versões paralelas. Para este trabalho foram projetos dois modelos distintos: o Modelo MPI e Modelo HLRC.

- **Fase 3:** validação das versões paralelas por meio da comparação dos resultados da implementação sequencial e das versões paralelas. Nesta fase os resultados gerados pela versão paralela devem ser idênticos ou chegar próximo aos dados esperados pelos autores do Algoritmo IFM, validando a aplicação do algoritmo na plataforma MPI.

- **Fase 4:** comparação do tempo de execução das versões paralelas, levando em consideração a acurácia de cada implementação. Esta fase analisa qual versão oferece o menor tempo de execução de acordo com a precisão do resultado esperado.

4.2 Descrição do Algoritmo Sequencial

Para facilitar o entendimento do algoritmo e dos modelos de paralelização propostos, o Quadro 4.2.1 mostra o pseudo-código geral do algoritmo sequencial focando os laços e as instruções globais mais importantes.

```
1  Especificacao dos parametros inicias (mascara, imagem de entrada, ...)
2  Alocacao das matrizes para manipulacao das imagens de entrada e saida
3  Carga dos valores dos pixels da imagem na matriz
4  Divisao dos intervalos de representacao
5  Inicio da convolucao:
6     Para cada linha L, faca:
7        Para cada coluna C, faca:
8           Determina o pixel central da regiao L,C:
9              Para cada linha L1 da Mascara, faca:
10                Para cada coluna C1 da Mascara, faca:
11                          Determina o intervalo de cada pixel(l1,c1) da Mascara
12                   Determina o numero de intervalos distintos na Mascara
13                   Calcula o valor percentual do pixel central
14          Calcula o pixel sobre o intervalo do espectro de imagem cinza
15          Converte o valor do pixel no formato da imagem de saida
16          Pinta o pixel na imagem de saida
17 Fim da convolucao
18 Salva a imagem
```

Código 4.2.1: Pseudo-código resumido da versão sequencial

Primeiramente, as variáveis são inicializados de acordo com os parâmetros passados pelo usuário. Nesta etapa, são fornecidos como parâmetros a máscara e o arquivo da imagem de entrada. Após ler os dados de inicialização, é realizada a alocação das estruturas matriciais de entrada e saída de acordo com a dimensão da imagem de entrada. Na linha 3, é carregado o valor de cada pixel da imagem na matriz de entrada. A instrução da linha 4 divide os intervalos de representação em tons de cinza conforme o tamanho da máscara.

As instruções das linhas 5, 8, 11 e 12 representam funções que contém muito laços de repetição aninhados. A sub-rotina de início da convolução (linha 5) realiza L x C chamadas ao procedimento de cálculo do índice do *pixel* central (linha 8); por sua vez, esta função percorre cada *pixel* da matriz para classificar o intervalo de cada elemento (linha 11) e, ainda, realiza a contagem de números distintos contidos na janela (linha 12). O número de iterações dos laços de repetição aninhados pelas linhas 6, 7, 9 e 10 depende diretamente dos parâmetros iniciais que definem quantas iterações serao realizadas. Para cada cálculo do *pixel* central realizado na rotina de convolução é calculado o valor do *pixel* central da janela N x N. No final do processo de convolução em toda a imagem, é gravada uma imagem de saída.

4.3 Análise para Paralelização do Algoritmo

O primeiro passo para paralelizar o Algoritmo IFM foi analisar o problema em busca de segmentos particionáveis. Após analisá-lo e compreender as suas peculiariedades, observou-se que o cálculo de cada *pixel* central da imagem de saída pode ser realizado de maneira independente do cálculo de *pixels* vizinhos. O nível de granularidade de paralelismo encontrado nesse algoritmo pode ser classificado como *coarse-grain* (granularidade grossa).

O Algoritmo IFM trabalha conforme segue. Para cada vizinhança NxN de *pixels* da imagem de entrada é gerada uma máscara NxN contendo os intervalos de cada *pixel* desta vizinhança. A partir desta máscara gera-se o *pixel* central, o qual representa o percentual de diversidade entre todos os intervalos da máscara. Este procedimento independe de qualquer outro resultado previamente calculado. Então, é possível efetuar em paralelo os cálculos de vários *pixels* centrais sem a necessidade de espera por outras instruções serem executadas.

Deste modo, distribuir os segmentos da imagem de entrada entre os nós do *cluster* é uma maneira coerente para se particionar o domínio dos dados. Assim, cada nó será responsável por calcular cada *pixel* central do segmento de imagem recebida e, posteriormente, devolver o segmento de saída calculado à máquina mestre. Este modelo de particionamento independe do número de nós existentes, pois o cálculo da divisão da imagem é dado pela razão entre a *altura da imagem* e *número de nós* do *cluster*. Assim, se temos uma imagem de entrada de dimensão 8460 x 9530 pixels e um *cluster* formado por 12 nós de processamento, podemos processar segmento de imagem de 705 x 9530 pixels em cada nó como pode ser visualizado na Figura 4.4. Entretanto, observa-se na Figura 4.4 que cada segmento pode fazer uso de linhas que ultrapassam os seus limites. Assim, algumas linhas além da divisão exata do segmento são passadas para que se possa obter o processo de convolução completo.

É importante observar que este trabalho trata o particionamento e distribuição dos dados entre os nós não levando em consideração as características individuais de cada um. Neste trabalho não tratamos o balanceamento de carga para o caso de haver variação do poder computacional de cada nó. Existem diversas técnicas para executar o balanceamento de carga, como as elencadas por (Gorino, 2006), que geralmente adotam soluções baseadas em análise da capacidade de processamento do hardware e o status dos recursos computacionais disponíveis.

O segundo estágio estabelecido pela metologia de Foster, o de comunicação, envolveu a definição de como as trocas de mensagens serão realizadas. A determinação da granu-

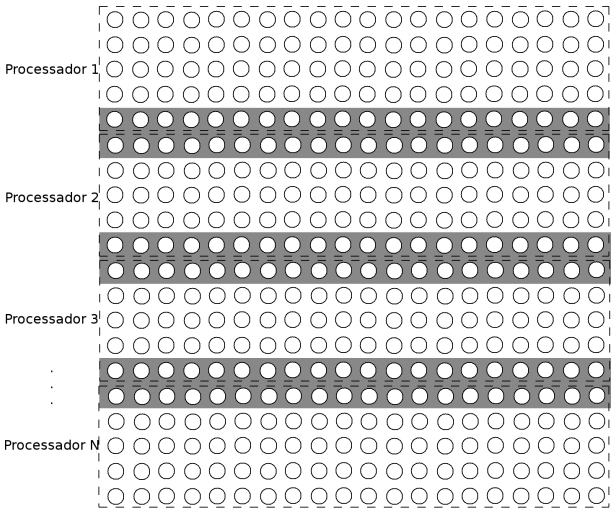

Figura 4.4: Modelo de distribuição dos segmentos da imagem de entrada em paralelo

laridade procurou minimizá-las entre os processos, conforme (Evans e Goscinski, 1995). No Modelo MPI foi estabelecido que o nó mestre é o responsável pela leitura da imagem de entrada e parâmetros iniciais, os quais são transmitidos a todos os outros nós. Após o término do processamento em cada nó, estes retornam o resultado para a máquina mestre. O Modelo HLRC carrega a imagem de entrada na memória compartilhada distribuída. Além disso, aloca uma matriz para armazenar os pontos de saída do processo de índice de fragmentação.

Capítulo

5

Modelos e Implementações Paralelas

Neste trabalho foram desenvolvidos dois modelos distintos de paralelização do Algoritmo IFM descrito no Capítulo 4. Ambos modelos foram projetados para duas plataformas de programação paralela: MPI e HLRC, originando 4 versões paralelas que são discutidas neste capítulo. O primeiro modelo, denominado Modelo Básico, paraleliza a versão sequencial original do algoritmo usando a técnica de paralelização de dados de forma que a convolução é aplicada a todos os pontos da imagem. O segundo modelo, denominado Modelo Seletivo, é uma otimização sobre o primeiro modelo, aplicando seletivamente a convolução de forma a poupar processamento ainda que tenha uma perda aceitável de qualidade.

5.1 Modelo Básico

A idéia do Modelo Básico é a de aplicar a convolução sobre todos os *pixels* da imagem original da mesma maneira que a versão sequencial, porém de forma paralela. Este modelo foi subdividido em 2 versões conforme mostra as próximas seções.

5.1.1 Versão MPI Básico

A Versão MPI Básico usa uma abordagem de computação distribuída *mestre-escravo*. Neste modelo, representado pela Figura 5.1, inicialmente o nó mestre da aplicação faz a leitura da imagem de entrada e de todos os parâmetros necessários para o processamento.

Em seguida, são particionados os dados de modo proporcional ao número de nós escravos disponíveis. Este particionamento é feito nas linhas da imagem. Por exemplo, supondo que uma imagem de entrada possua a dimensão igual a 12 linhas e 80 colunas e um *cluster* composto por três nós escravos, o nó mestre divide as 12 linhas por 3, resultando em segmentos de 4 linhas e 80 colunas; após, cada segmento é enviado para um nó escravo por meio de passagem de mensagem. Após receber o seu respectivo segmento da imagem de entrada, cada nó escravo executa o Algoritmo IFM sobre ele e envia o segmento resultante da aplicação da convolução ao nó mestre novamente por meio de passagem de mensagem. Este por sua vez, concatena adequadamente os segmentos gerados individualmente e grava em disco a imagem de saída completa. Convém ressaltar que o Algoritmo IFM é o mesmo executado em cada nó escravo, variando apenas os dados manipulados. O Quadro 5.1.1 apresenta o pseudo-código da Versão MPI Básico.

```
1  Especificacao dos parametros iniciais
2  Divisao dos intervalos de representacao
3
4  === NO MESTRE ===
5  Alocacao das matrizes de manipulacao das imagens de entrada e saida
6  Leitura da imagem de entrada
7  Carga dos valores dos pixels da imagem de entrada na matriz
8  Segmenta a imagem de entrada de acordo com o numero de escravos
9  Envia cada segmento da imagem a um NO ESCRAVO
10 Aguarda os resultados
11
12 === NO ESCRAVO ===
13 Recebe os parametros da dimensao da matriz a ser recebida
14 Aloca a matriz do segmento
15 Recebe o segmento da imagem
16 Inicia a convolucao:
17    Para cada linha L, faca:
18       Para cada coluna C, faca:
19          Determina o pixel central da regiao L,C:
20             Para cada linha L' da Mascara, faca:
21                Para cada coluna C' da Mascara, faca:
22                                      Determina o intervalo de cada pixel(L',C') da Mascara
23                                      Determina o numero de intervalos distintos na Mascara
24                                      Calcula o valor percentual do pixel central
25          Calcula o pixel sobre o intervalo do espectro de imagem cinza
26          Converte o valor do pixel no formato da imagem de saida
27          Pinta o pixel na matriz de saida
28 Fim da convolucao
29 Envia o resultado do segmento ao NO MESTRE
30
31 === NO MESTRE ===
32 Recebe o resultado de cada NO ESCRAVO
33 Agrupa todos os resultados
34 Salva a matriz de resultado final em arquivo
```

Código 5.1.1: Pseudo-código resumido da versão paralela MPI Básico

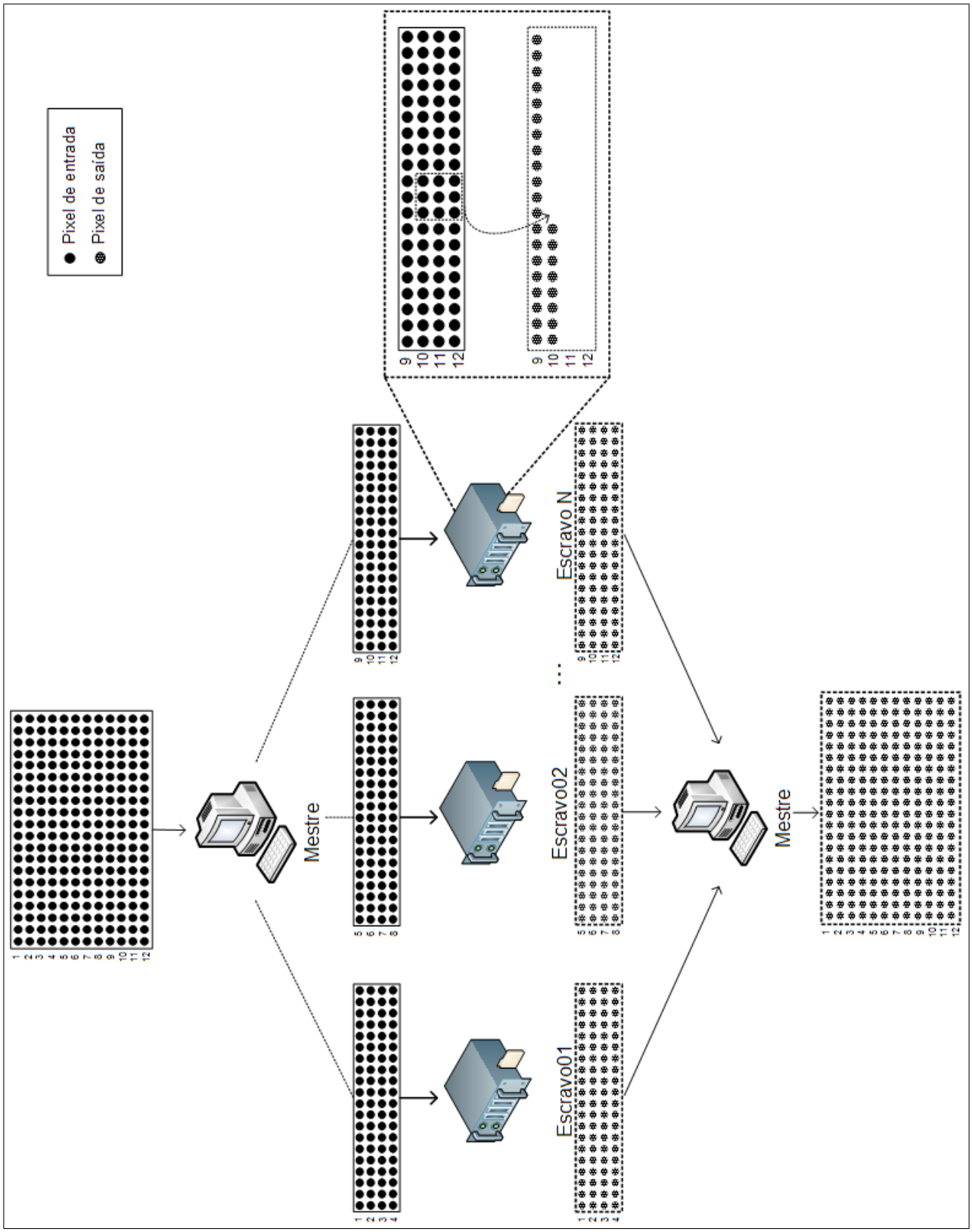

Figura 5.1: Modelo MPI Básico

5.1.2 Versão HLRC Básico

O protocolo HLRC cria um ambiente de execução com uma visão da memória única para todos os computadores, ainda que a memória seja fisicamente distribuída. Dessa forma, o mesmo endereço de memória acessado por um nó A, pode ser também acessado por um nó B, tanto para leitura quanto para a escrita. De fato, sabe-se que para prover este mecanismo de memória compartilhada distribuída, a plataforma deve internamente trocar informações entre os processadores, mas isto é transparente para o programador.

Assim, na Versão HLRC Básico o fluxo de dados real que cada nó faz nas memórias distribuídas não é visível ao programador, somente ao desenvolvedor do protocolo. A Versão HLRC Básico, mostrada na Figura 5.2, começa com a leitura da imagem de entrada e dos parâmetros necessários para o início do processamento. A imagem é carregada pelo processo identificado pelo *identificador 0* que aloca os espaços na memória compartilhada. A partir disso, os demais nós executam o algoritmo que lê diretamente os dados na memória compartilhada, sem a necessidade de passagem de mensagem explícita. Isso significa que o protocolo HLRC isenta o programador de se preocupar com a coordenação de mensagens trocadas entre os nós, porém ainda é de sua responsabilidade o particionamento dos dados. Da mesma forma, o resultado de cada iteração do algoritmo é atribuído diretamente no espaço de memória compartilhada, sem a necessidade de utilizar a passagem de mensagem explícita para o nó responsável pela gravação da imagem de saída em disco. O Quadro 5.1.2 apresenta o pseudo-código da Versão HLRC Básico.

```
1  Especificacao dos parametros iniciais
2  Divisao dos intervalos de representacao
3  Leitura da imagem de entrada
4  Carga dos valores dos pixels da imagem de entrada na DSM
5  Inicia a convolucao:
6     Determina o segmento a ser calculado de cada processo
7     Para cada linha L referente ao processo, faca:
8        Para cada coluna C, faca:
9           Determina o pixel central da regiao L,C:
10             Para cada linha L' da Mascara, faca:
11                Para cada coluna C' da Mascara, faca:
12                                      Determina o intervalo de cada pixel(L',C') da Mascara
13                                      Determina o numero de intervalos distintos na Mascara
14                                      Calcula o valor percentual do pixel central
15          Calcula o pixel sobre o intervalo do espectro de imagem cinza
16          Converte o valor do pixel no formato da imagem de saida
17          Pinta o pixel na matriz de saida
18 Fim da convolucao
19 Salva a matriz de resultado final em arquivo
```

Código 5.1.2: Pseudo-código resumido da versão paralela HLRC Básico

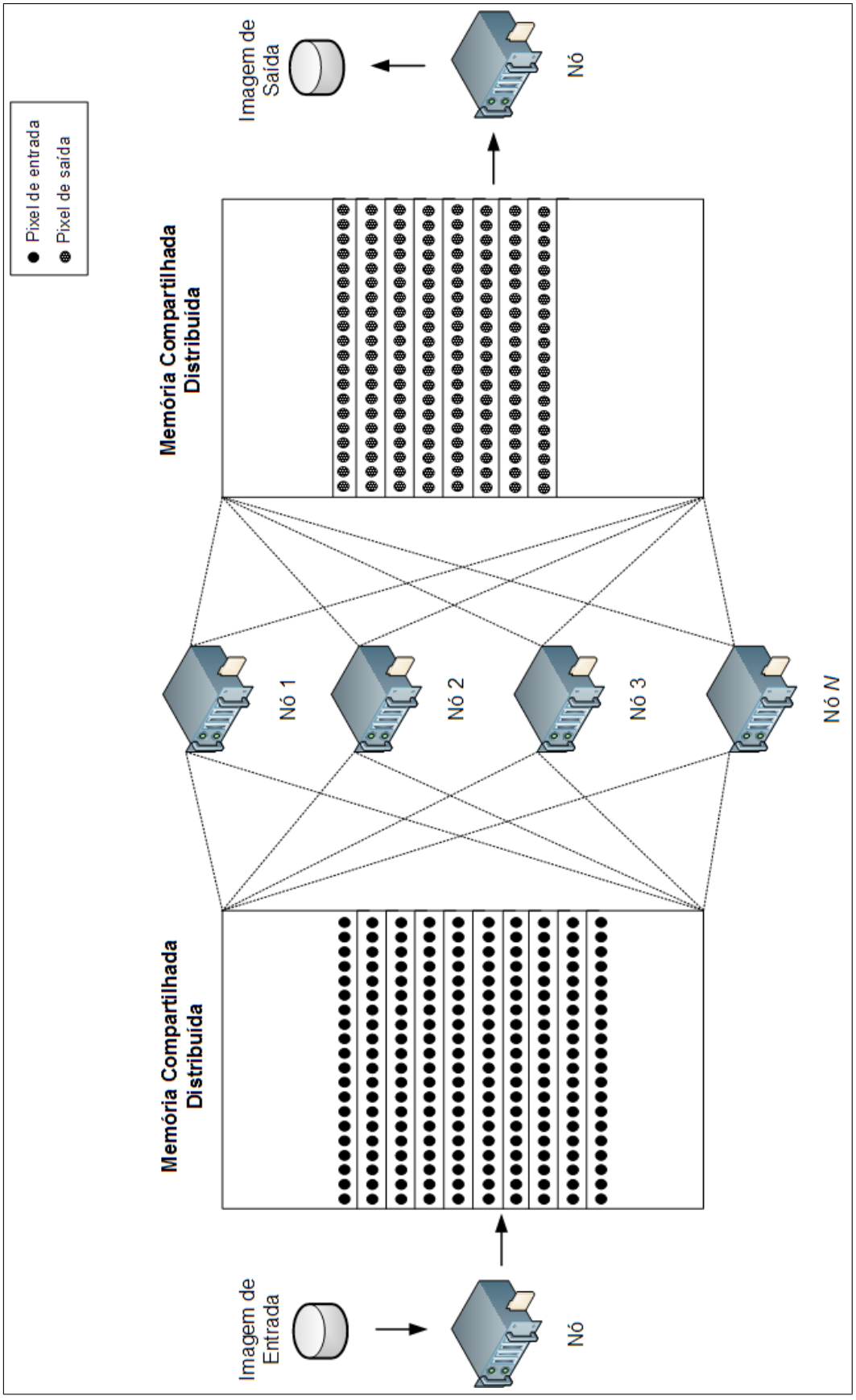

Figura 5.2: Modelo HLRC Básico

5.2 Modelo Seletivo

O Modelo Seletivo, também conhecido como Modelo de Saltos, é uma evolução do Modelo Básico. A idéia geral deste modelo é a de não aplicar a convolução sobre todos os pixels, mas aplicar parcialmente, de acordo com uma Política de Seleção, estimando os valores para os demais pixels, baseado em uma Política de Estimativa, poupando o tempo das convoluções não usadas. Obviamente, o tempo do cálculo da estimativa deve ser menor do que o tempo da convolução para que este modelo seja considerado.

A Política de Seleção de pixel utilizada neste trabalho é a ŤPíxel EquidistanteŤ, a qual é uma política estática. Esta política seleciona pixels equidistantes a um certo intervalo para sofrerem a convolução, entretanto, outras políticas poderão ser experimentadas futuramente, inclusive políticas dinâmicas, as quais podem selecionar *pixels* em função da dinâmica de execução.

5.2.1 Versão MPI Seletivo

Esta versão particiona e distribui os segmentos da imagem igualmente a Versão MPI Básico, conforme mostra a Figura 5.3. Assim, o nó mestre lê a imagem de entrada e os parâmetros necessários para inicializar o processamento. Logo em seguida, a imagem é particionada e os segmentos são distribuídos aos demais nós do cluster por meio de passagem de mensagem. Cada nó ao receber o seu respectivo segmento de imagem, inicia a geração de cada *pixel* resultante de seu respectivo segmento. De acordo com um parâmetro de inicialização do algoritmo, denominado Intervalo de Salto, o algoritmo deixa de aplicar a convolução a cada intervalo de N *pixels* durante a verredura da imagem. Os *pixels* que foram "saltados"têm os seus valores estimados de acordo com uma Política de Estimativa de Pixel. O caracter "X"na Figura 5.3 exemplifica os pixels saltados. As 3 políticas de estimativa utilizadas neste trabalho são discutidas na seção 5.2.3. Após o término do processamento dos nós escravos, estes enviam os segmentos gerados ao nó mestre por meio de passagem de mensagem, que os concatena em uma única imagem de saída. O Quadro 5.2.1 apresenta o pseudo-código da Versão MPI Seletivo.

```
1  Especificacao dos parametros iniciais
2  Divisao dos intervalos de representacao
3
4  === NO MESTRE ===
5  Alocacao das matrizes de manipulacao das imagens de entrada e saida
6  Leitura da imagem de entrada
7  Carga dos valores dos pixels da imagem de entrada na matriz
8  Segmenta a imagem de entrada de acordo com o numero de escravos
9  Envia cada segmento da imagem a um NO ESCRAVO
10 Aguarda os resultados
11
12 === NO ESCRAVO ===
13 Recebe os parametros da dimensao da matriz a ser recebida
14 Aloca a matriz do segmento
15 Recebe o segmento da imagem
16 Inicia a convolucao:
17    Para cada linha L, faca:
18       Para cada coluna C do intervalo a ser realmente processado, faca:
19          Determina o pixel central da regiao L,C:
20             Para cada linha L' da Mascara, faca:
21                Para cada coluna C' da Mascara, faca:
22                                      Determina o intervalo de cada pixel(L',C') da Mascara
23                                      Determina o numero de intervalos distintos na Mascara
24                                      Calcula o valor percentual do pixel central
25          Calcula o pixel sobre o intervalo do espectro de imagem cinza
26          Converte o valor do pixel no formato da imagem de saida
27          Pinta o pixel na matriz de saida
28                  Para cada Coluna C'' pertencente ao intervalo saltado, faca:
29                         Pinta a metade-esquerda saltada com o pixel anterior
30             Pinta a metade-direita saltada com o pixel posterior
31 Fim da convolucao
32 Envia o resultado do segmento ao NO MESTRE
33
34 === NO MESTRE ===
35 Recebe o resultado de cada NO ESCRAVO
36 Agrupa todos os resultados
37 Salva a matriz de resultado final em arquivo
```

Código 5.2.1: Pseudo-código resumido da versão paralela MPI Seletivo

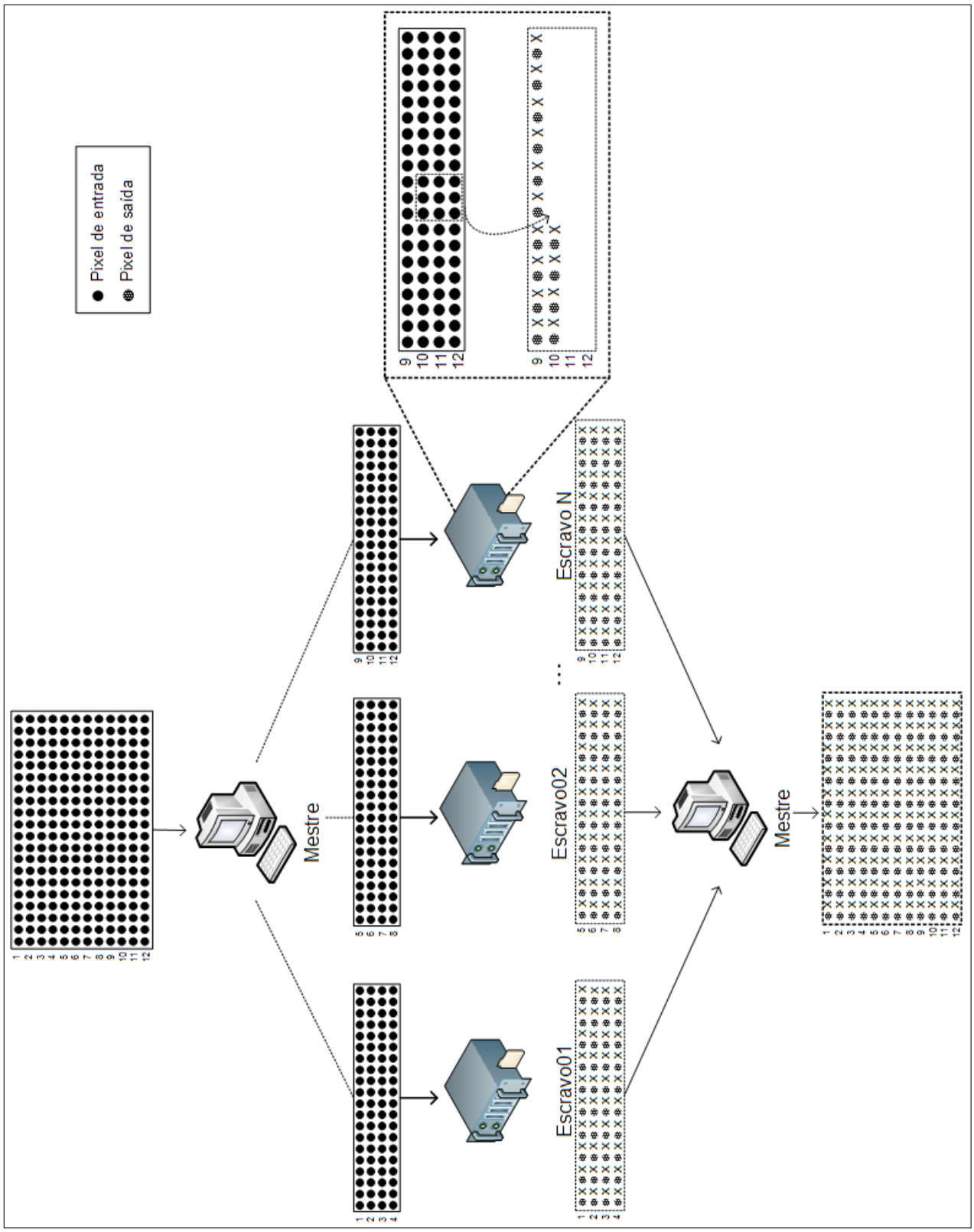

Figura 5.3: Modelo MPI Seletivo

5.2.2 Modelo HLRC Seletivo

Por fim, o Modelo HLRC Seletivo conta com um nó responsável pela leitura da imagem de entrada e dos demais parâmetros. A imagem de entrada é alocada no espaço de memória compartilhada, visível por todos os nós do *cluster*. A leitura ou escrita das imagens são realizadas diretamente na memória compartilhada. Cada nó processa o seu segmento de imagem utilizando o algoritmo seletivo mencionado na seção anterior conforme a Figura 5.4. Após o término da execução de todos os nós envolvidos no processamento, a imagem de saída gerada a partir do endereço de memória compartilhada é gravada em disco pelo nó responsável. O Quadro 5.2.2 apresenta o pseudo-código da Versão HLRC Seletivo.

```
1  Especificacao dos parametros iniciais
2  Divisao dos intervalos de representacao
3  Leitura da imagem de entrada
4  Carga dos valores dos pixels da imagem de entrada na DSM
5  Inicia a convolucao:
6     Determina o segmento a ser calculado de cada processo
7     Para cada linha L referente ao processo, faca:
8        Para cada coluna C, faca:
9           Determina o pixel central da regiao L,C:
10             Para cada linha L' da Mascara, faca:
11                Para cada coluna C' da Mascara, faca:
12                                      Determina o intervalo de cada pixel(L',C') da Mascara
13                                      Determina o numero de intervalos distintos na Mascara
14                                      Calcula o valor percentual do pixel central
15          Calcula o pixel sobre o intervalo do espectro de imagem cinza
16          Converte o valor do pixel no formato da imagem de saida
17          Pinta o pixel na matriz de saida
18                  Para cada Coluna C'' pertencente ao intervalo saltado, faca:
19                         Pinta a metade-esquerda saltada com o pixel anterior
20             Pinta a metade-direita saltada com o pixel posterior
21 Fim da convolucao
22 Salva a matriz de resultado final em arquivo
```

Código 5.2.2: Pseudo-código resumido da versão paralela HLRC Seletivo

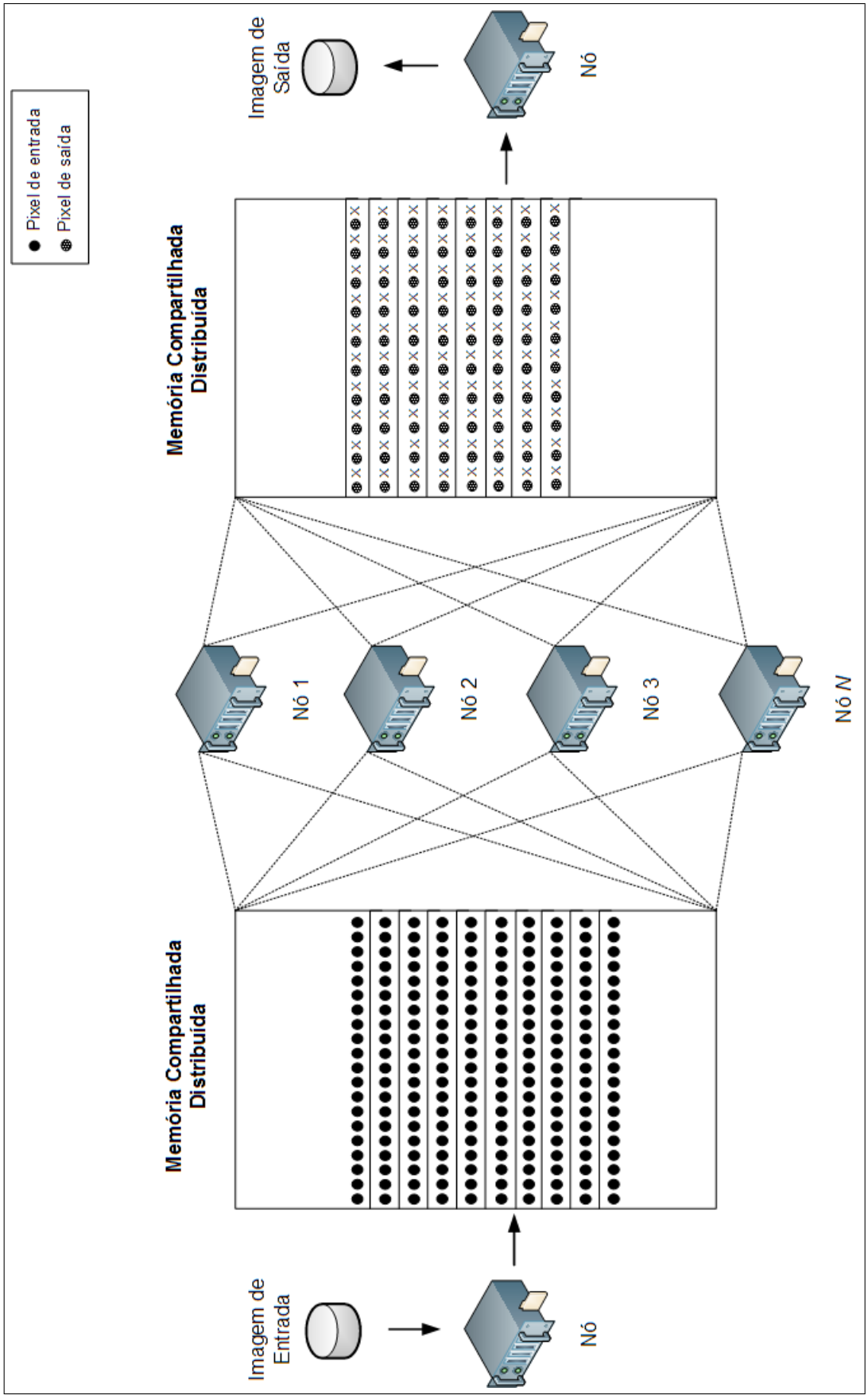

Figura 5.4: Modelo HRLC Seletivo

5.2.3 Variações da Abordagem Seletiva: Politicas de Estimativa

O Modelo Seletivo foi proposto neste trabalho como alternativa para diminuir o tempo de execução da aplicação, objetivando uma menor taxa de erro possível dos *pixels* estimados. Estimamos os valores dos pixels de saída para aqueles pertencentes a um determinado intervalo de N pixels, substituindo os *pixels* T̆saltadosT̆ por um valor estimado por uma função de complexidade inferior ao processamento gasto na aplicação da convolução. Certamente, os *pixels* estimados poderão ser ou não diferentes daqueles que seriam gerados pela aplicação da convolução, dependendo da precisão da Política de Estimativa utilizada. Este erro poderá ser aceitável se estiver dentro de limites aceitáveis pela aplicacão.

O exemplo de salto de 1 (uma) coluna é ilustrado pela Figura 5.5. O círculo representa os *pixels* realmente calculados por meio do algoritmo originalmente proposto e os elementos estimados são definidos pelo elemento "X". Para cada *pixel* saltado usa-se uma política para estimar qual o valor que será atribuído. Nesse trabalho experimentamos três políticas para estimar o valor das colunas saltadas, conforme segue:

- Política do Pixel Médio: consiste em calcular o valor dos *pixels* saltados a partir da média entre os *pixels* imediatamente anterior e posteior ao intervalo saltado, os quais foram realmente calculados pela convolução;
- Política Repete Anterior: faz a estimativa sempre utilizando o valor do último *pixel* realmente calculado pela convolução;
- Política Repete Anterior-posterior: Usa o *pixel* anterior realmente calculado para estimar a primeira metade (da esquerda) dos *pixels* saltados. A segunda metade (da direita) é estimada com o valor do *pixel* posterior realmente calculado.

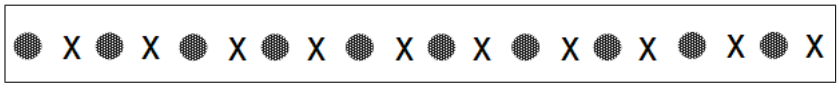

Figura 5.5: Exemplo da abordagem seletiva

Capítulo

6

Avaliação de Desempenho

6.1 Ambiente de Execução

O Laboratório Experimental de Computação de Alto Desempenho (LECAD) do Departamento de Informática da Universidade Estadual de Maringá possui um *cluster* com nós escravos homogêneos onde todos os experimentos desta dissertação foram executados. Este *cluster* é formado por 7 máquinas: 1 nó servidor de acesso e 6 nós da Sun Microsystems. A Tabela 6.1 apresenta a configuração detalhada de cada um dos nós do *cluster*.

Nó	CPU	Clock (Ghz)	Núcleos	RAM (MB)	Rede (Mbits)
Nó 1	Intel Core 2 Quad	2.4	4	2048	1000
Nó 2	AMD Opteron 1218	2.6	2	4096	1000
Nó 3	AMD Opteron 1218	2.6	2	4096	1000
Nó 4	AMD Opteron 1218	2.6	2	4096	1000
Nó 5	AMD Opteron 1218	2.6	2	4096	1000
Nó 6	AMD Opteron 1218	2.6	2	4096	1000
Nó 7	AMD Opteron 1218	2.6	2	4096	1000

Tabela 6.1: Configuração dos nós do *cluster*

Além de possuir um processador quad-core, o servidor de acesso é composto ainda por 8MB de memória cache L2 e capacidade de 400GB para armazenamento. Cada um dos 6 nós da Sun Microsystems dispõe de 1MB de cache L2 por núcleo e 1TB de disco

rígido. Assim, esse *cluster* é conectado por um *switch gigabit*, totalizando 16 núcleos de processamento, 26GB de memória e 6,4TB de espaço de armazenamento.

A plataforma operacional de execução é provida pelo sistema operacional Linux CentOS com *kernel* 2.6.18. A compilação é feita pelo *GCC* 4.1.2. A implementação da plataforma MPI é a Open MPI 1.3.2 e o HLRC na versão com suporte ao protocolo TCP/IP. Para realizar o monitoramento do *cluster* durante a execução dos experimentos foi utilizada a ferramenta Ganglia (Sacerdoti et al., 2003). Este ambiente de execução e gerenciado pela ferramenta ROCKS (Papadopoulos et al., 2003). Além disso, foi adotada a biblioteca OpenCV 2.1 para a leitura e gravação das imagens (Bradski e Kaehler, 2008).

Nos experimentos com MPI é considerado o modelo distribuído *mestre-escravo*. Neste caso, o servidor de acesso é usado como nó mestre responsável pela distribuição e agrupamento das mensagens. Os demais 6 nós da Sun Microsystems são os responsáveis pela execução do núcleo do algoritmo de convolução. Os modelos em HLRC descartam o servidor de acesso por não utilizar o modelo *mestre-escravo*, totalizando em 6 nós da Sun. Deste modo, o modelo HLRC não sofre as interferências de uma máquina heterogênea.

Os resultados das avaliações se deram pela média da execução de 3 baterias de cada experimento utilizando os seguintes parâmetros de entrada:

- imagem com dimensão de 8460x9530 *pixels*;
- tamanhos de máscara de 3, 5, 7, 9, 11, 13 e 15 *pixels*;
- número de processos: 1 a 12 processos no MPI; e 1 a 6 processos no HLRC;

A partir dos experimentos realizados, os dados relativos à execução foram coletados e os gráficos que permitam avaliar o desempenho de cada modelo proposto foram gerados. As métricas ilustradas nas próximas seções incluem o *speedup*, a eficiência e o percentual entre comunicação e computação. Os modelos HLRC e MPI são comparados a fim de se obter uma análise dos eventuais aspectos que influenciaram no desempenho de cada um.

6.2 Avaliação do Modelo Básico

6.2.1 Versão MPI Básico

A Versão MPI Básico é centrada basicamente na ideia da divisão da imagem de entrada, distribuição e processamento de cada segmento e agrupamento das partes enviadas por cada nó. A Figura 6.1 ilustra o *speedup* obtido. O eixo X define o número de processos;

o eixo Y representa o tamanho da máscara; e o eixo Z mostra o *speedup* alcançado na intersecção entre os eixos X e Y.

O MPI permite a execução de N processos por nó. No caso deste *cluster* que conta com máquinas de 2 núcleos, foram executadas baterias de testes de 1 a 12 processos. A sequência de escalonamento de processos inicia-se com 1 processo por máquina até se completar 6 processos e os próximos são alocados novamente a partir da primeira máquina.

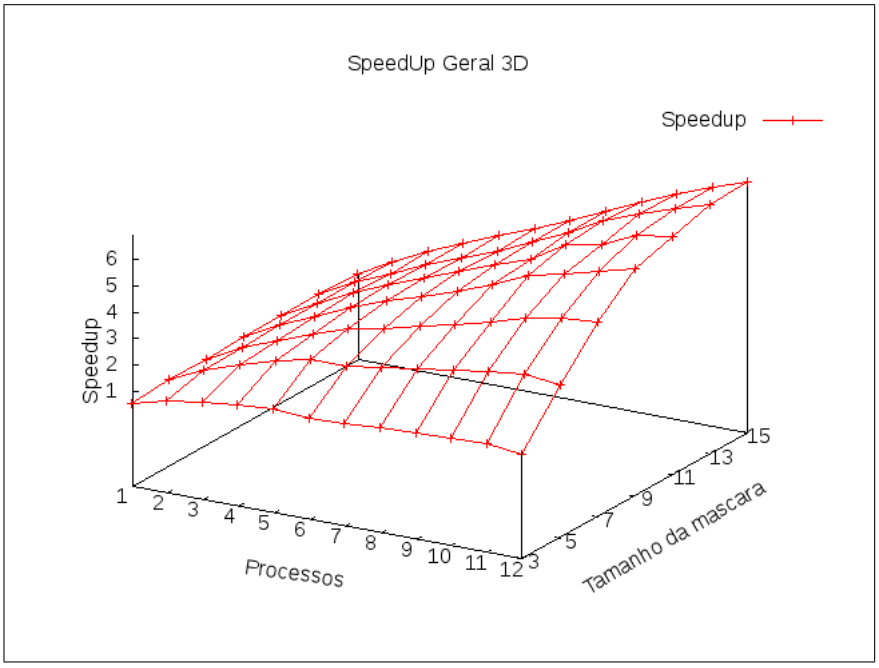

Figura 6.1: Speedup Modelo MPI Básico

Pode se observar que a versão paralela com 1 processo apresenta desempenho inferior à sequencial, independentemente do tamanho da máscara. O tamanho de máscara de 3 pixels mostra-se pouco viável mesmo com o aumento do número de processos. Com o uso de 5 pixels de máscara, o tempo de execução é reduzido de acordo com o número de processos. Assim, observa-se que na situação de 15 pixels de máscara e 12 processos é obtido *speedup* 6,91.

A Figura 6.2 descreve a eficiência de acordo com o crescimento do número de processos e do tamanho da máscara. O eixo X define o número de processos; o eixo Y caracteriza o tamanho da máscara; e o Z esboça a eficiência alcançada pelo cruzamento entre os dois eixos. Pode-se observar que o índice de eficiência com 6 e 12 processos tem maior queda. Essa redução de desempenho ocorre em função do tempo de comunicação gasto pela sexta máquina escrava cujo dever é executar o sexto e décimo segundo processo. Contudo, essa deficiência torna-se praticamente invisível conforme o percentual computacional cresce de

acordo com o tamanho da máscara. Ao analisar a curva do sexto processo, por exemplo, é possível notar que o índice computacional aumenta de acordo com o tamanho da máscara, ocultando assim a parcela de tempo gasta com comunicação.

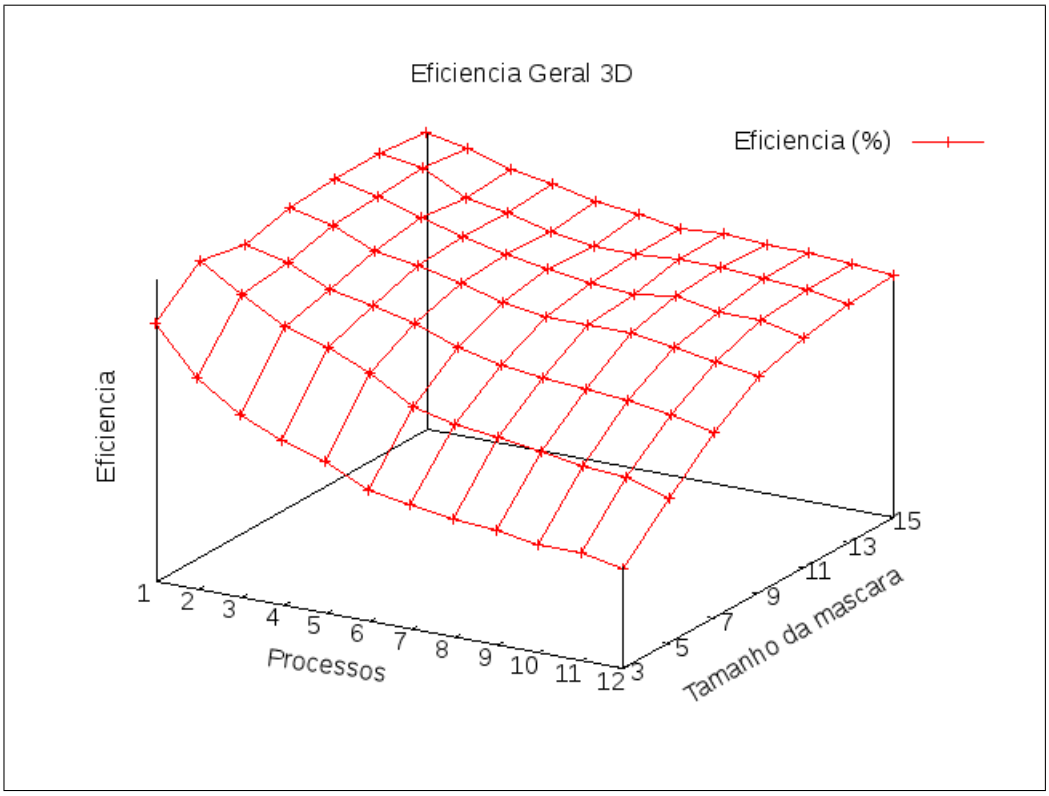

Figura 6.2: Eficiência Modelo MPI Básico

Podemos consolidar que o ganho de desempenho com 3 pixels de máscara é cada vez menor com o aumentar do número de processos. Apesar de mais amena, esta situação é válida para os demais tamanhos de entrada de máscara. Isso se deve diretamente ao tempo de comunicação que se torna significante conforme o crescimento do número de processos. A Figura 6.3 ilustra o percentual gasto com comunicação enquanto a Figura 6.4 mostra o tempo gasto com computação.

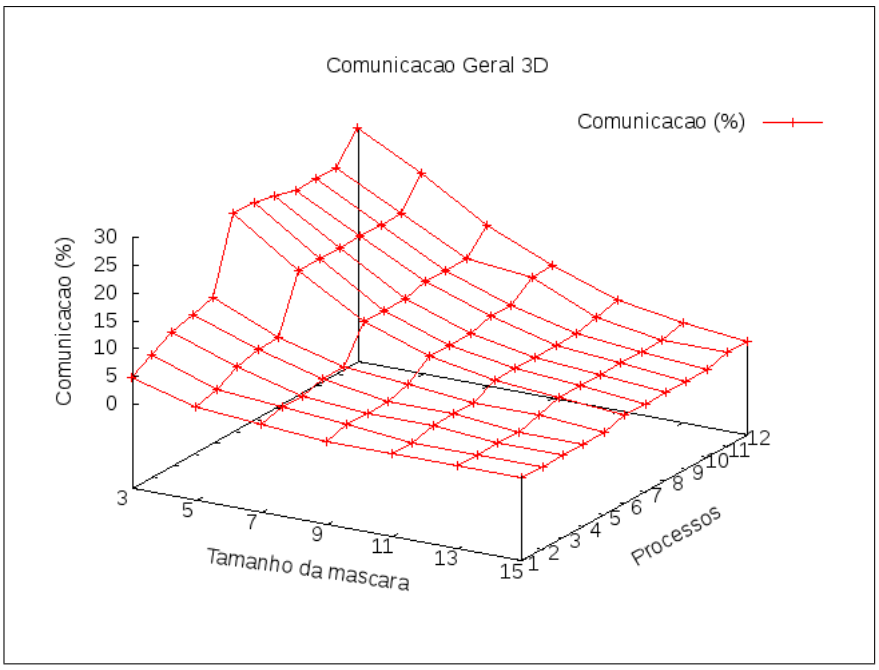

Figura 6.3: Percentual de Comunicação do Modelo MPI Básico

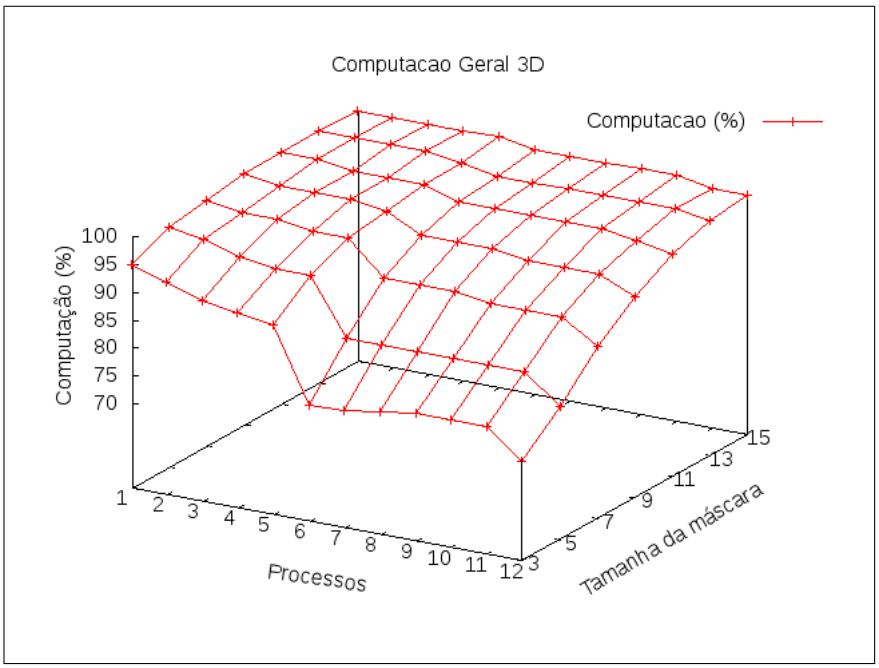

Figura 6.4: Percentual de Computação do Modelo MPI Básico

Se observarmos o impacto da comunicação nas execuções com 3 pixels de máscara, a parcela de comunicação do ambiente com 12 processos atinge aproximadamente 27% do tempo total do experimento. E, com 15 pixels de máscara nota-se que a parcela de

computação torna-se mais significativa diante do número de instruções geradas gastando aproximadamente 98% com computação.

6.2.2 Versão HLRC Básico

A Versão HLRC Básico é centrada na ideia de carga direta da imagem na memória compartilhada distribuída. Todos os nós tem a visão de uma única memória onde fazem diretamente a leitura e escrita dos dados. A Figura 6.5 ilustra o *speedup* obtido. O eixo X define o número de processos; o eixo Y representa o tamanho da máscara; e o eixo Z mostra o *speedup* alcançado na intersecção entre os eixos X e Y. A execução dos experimentos HLRC é realizada apenas com um único processo por máquina devido às restrições da biblioteca. Diante deste fato, os gráficos contemplam o desempenho de 1 a 6 processos.

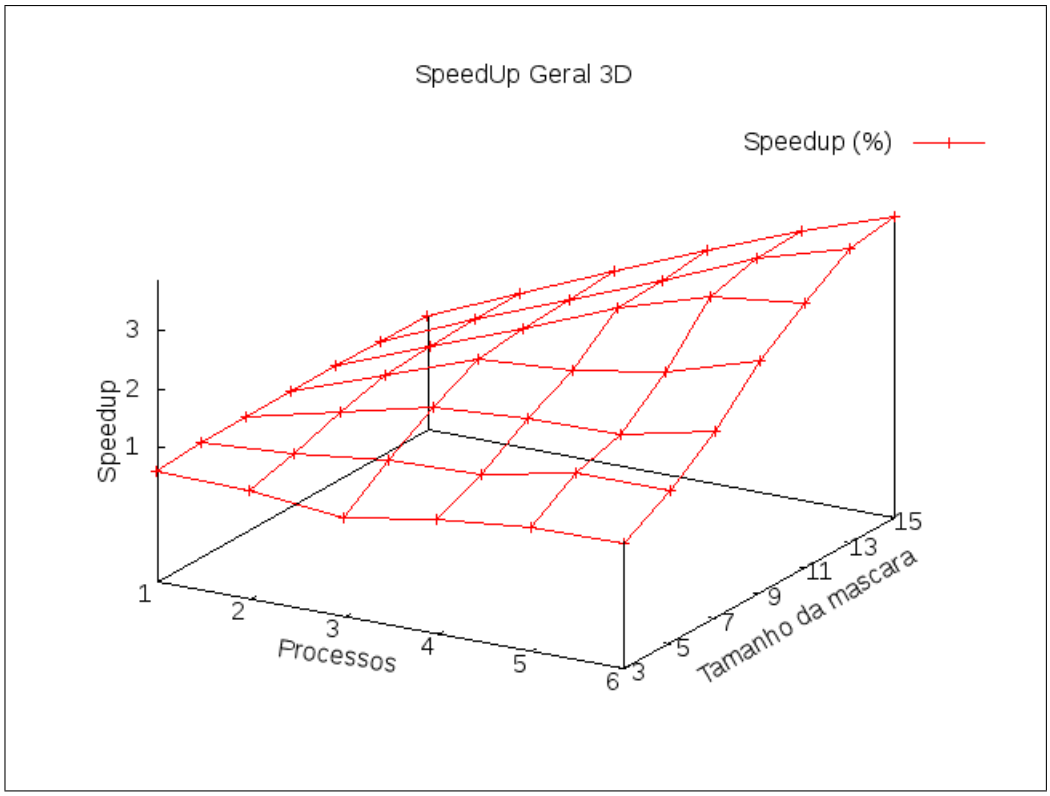

Figura 6.5: Speedup Modelo HLRC Básico

Nesta versão todas as execuções com 1 processo também apresentam performance inferior à sequencial devido ao *overhead* imposto pelo gerenciamento da plataforma HLRC. Com 5 pixels de máscara, o ganho de desempenho começa a ser significativo a partir de 5

processos em diante. O pico máximo desempenho encontra-se quando é adotado 15 pixels de máscara de entrada executando-se com 6 processos ao aproximar de 3,84 de *speedup*.

A Figura 6.6 ilustra a eficiência conforme o crescimento do número de processos e do tamanho da máscara. O eixo X define o número de processos; o eixo Y caracteriza o tamanho da máscara; e o Z quantifica a eficiência alcançada pelo cruzamento entre os dois eixos.

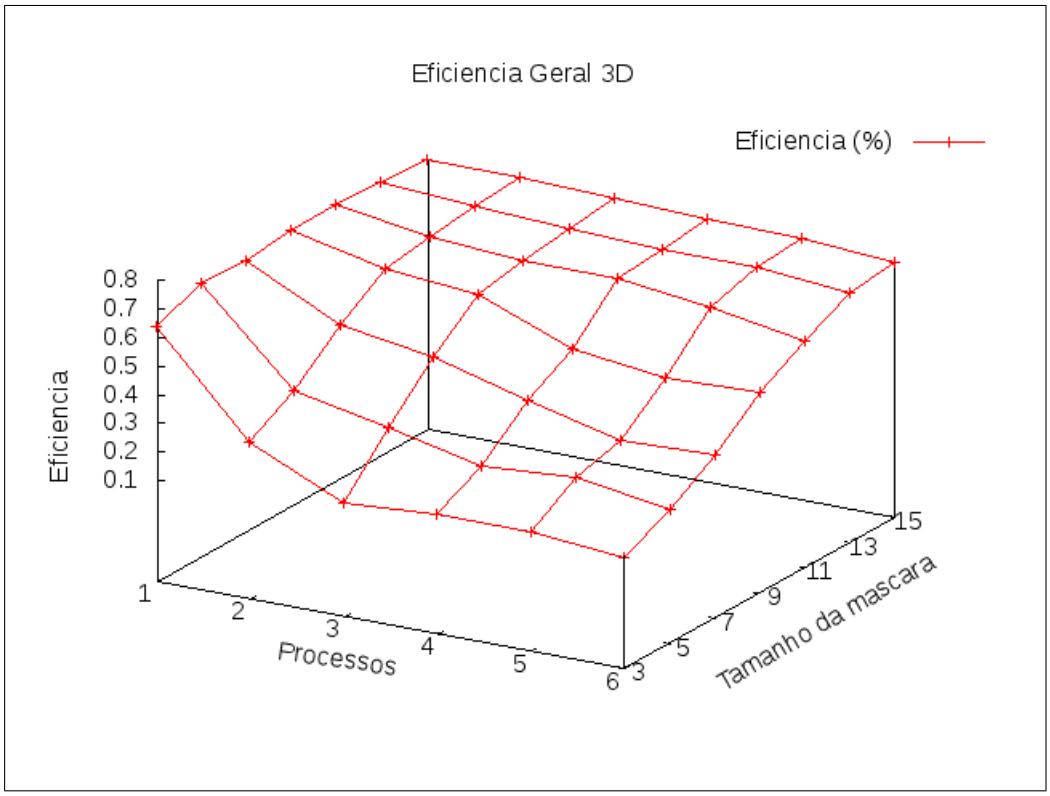

Figura 6.6: Eficiência Modelo HLRC Básico

O ganho de desempenho com 3 pixels de máscara também é cada vez menor com o crescimento do número de processos. As Figuras 6.7 e 6.8 apresentam o percentual gasto com comunicação e computação. Assim, é possível observar que com aumento do tamanho da máscara, o percentual de comunicação torna-se menos significativo, melhorando a eficiência da aplicação.

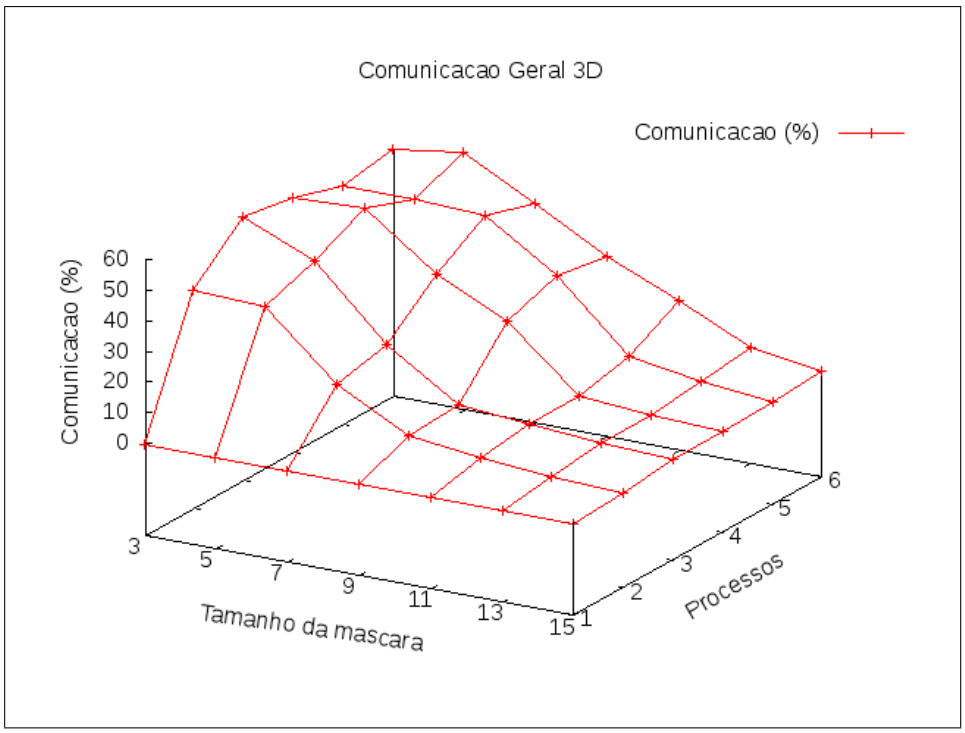

Figura 6.7: Percentual de Comunicação do Modelo HLRC Básico

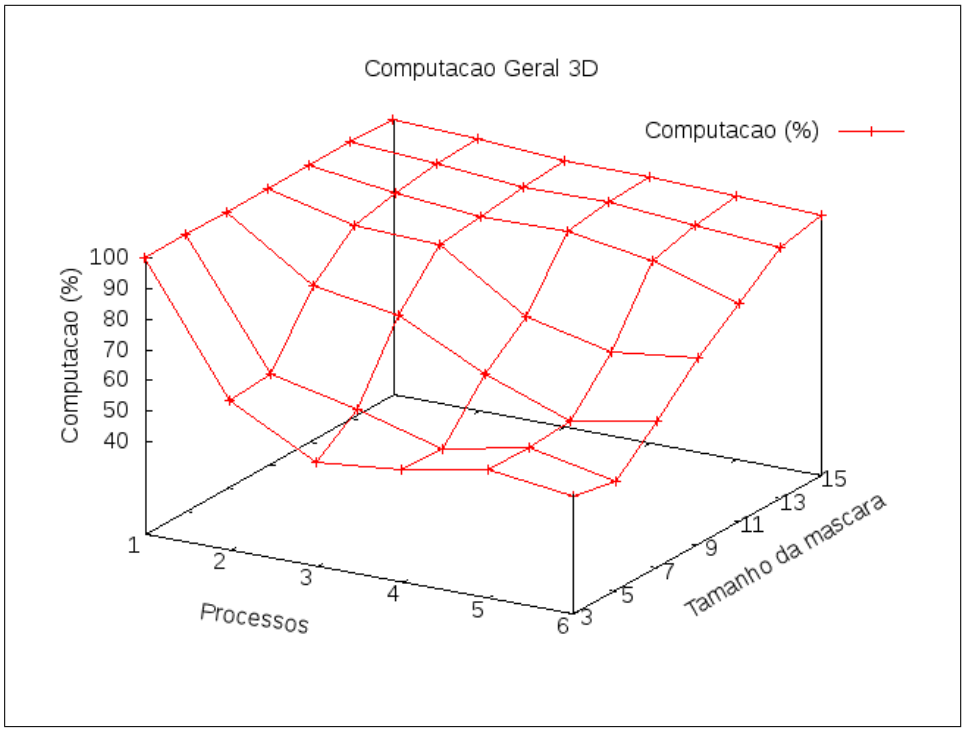

Figura 6.8: Percentual de Computação do Modelo HLRC Básico

Os *speedup* com tamanho de máscara entre 9 e 15 *pixels* apresenta um comportamento regular, porém os de máscaras menores indicam algumas variações. A Figura 6.7 permite observar que a origem dessa irregularidade está relacionada com o gerenciamento da comunicação na plataforma HLRC.

Se observarmos o impacto da comunicação nas execuções HLRC com 3 pixels de máscara, a parcela de comunicação do ambiente com 3 processos atinge aproximadamente 56% do tempo total do experimento. E, com 15 pixels de máscara nota-se que a parcela de computação torna-se mais significativa diante do número de instruções geradas gastando aproximadamente 95% com computação.

6.3 Avaliação do Modelo Seletivo

A Abordagem Seletiva descrita na Seção 5.2 estima os índices de fragmentação dos pixels pertencentes aos pixels saltados em busca da redução do tempo de execução em detrimento a uma taxa de erro aceitável. Para avaliar o modelo seletivo, criamos 2 métricas de avaliação.

A primeira, denominada Taxa de Erro, mede quantitativamente o percentual de *pixels* diferentes entre os valores correspondentes entre a imagem de saída totalmente processada e a imagem gerada pelo modelo seletivo. A Figura 6.9 apresenta o comparativo percentual entre as políticas utilizadas no que se refere a Taxa de Erro. Observa-se que o método "anterior-posterior"produz a menor taxa de erro entre os demais, independente do tamanho da máscara de entrada. Assim, na avaliação de desempenho constante do Capítulo 6 consideramos apenas o uso desta política.

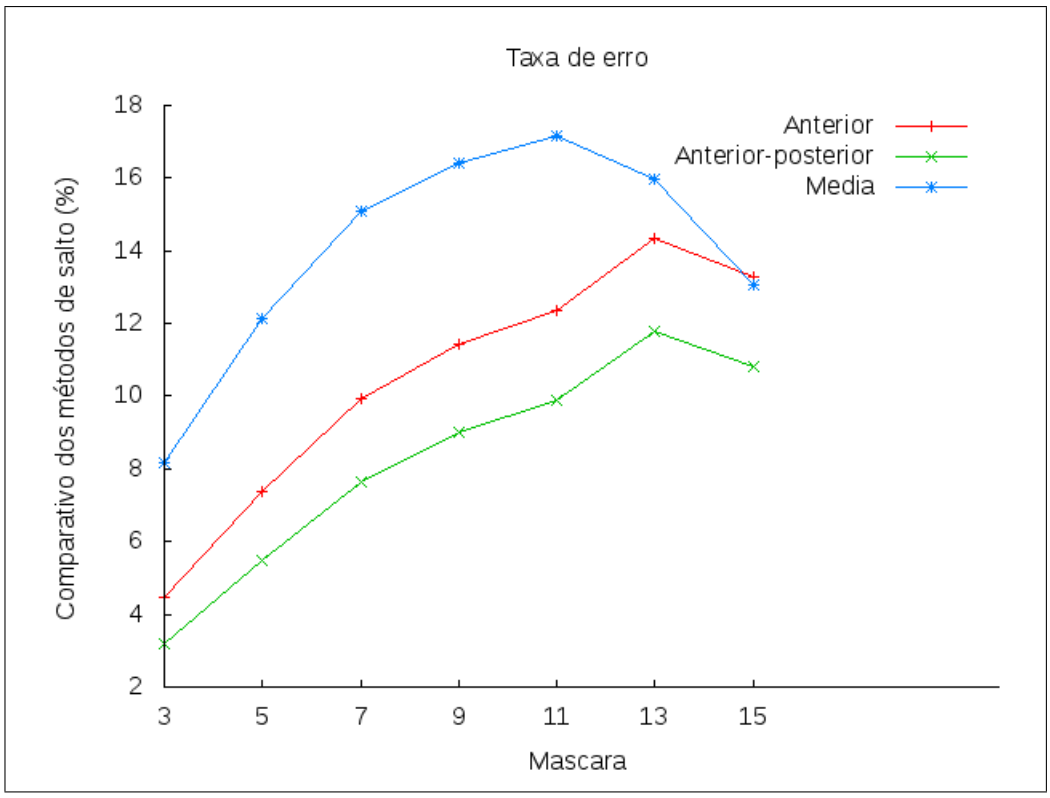

Figura 6.9: Comparativo das variações da abordagem seletiva

A segunda métrica, denominada Grau de Variação Média do erro, mede qualitativamente a variação média do erro para aqueles *pixels* estimados incorretamente. Em todos os experimentos realizados neste trabalho, medimos a Taxa de Erro e o Grau de Variação Média do erro. Verificamos que o Grau de Variação Média aumenta na medida em que a Taxa de Erro aumenta. Analisamos estas duas métricas em função da variação do tamanho da máscara e do intervalo de saltos.

Notamos que usando Intervalos de Saltos de 2 pixels, a Taxa de Erro gira em torno de 10tamanho 13x13 *pixels* como pode ser visto nas Figuras 6.10 6.11. Com isso, nas seções seguintes, nossa avaliação de desempenho mostra apenas os resultados obtidos com o uso da política "anterior-posterior"e Intervalos de Saltos de tamanho 2, pelo fato de enterdermos que estes limites são bem aceitáveis.

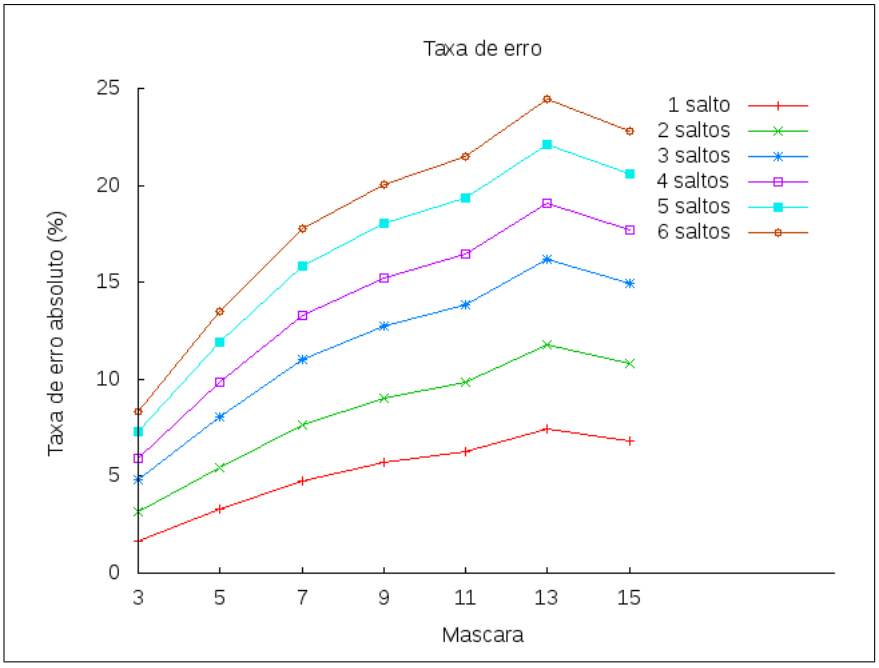

Figura 6.10: Taxa de erro entre variações de salto

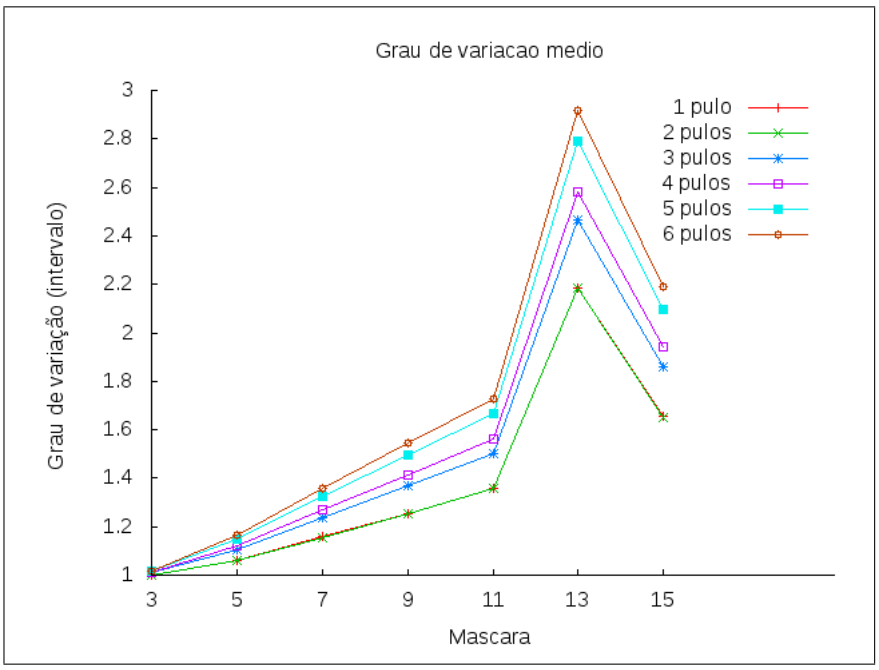

Figura 6.11: Grau de variação média de intervalos dos saltos

6.3.1 Versão MPI Seletivo

A Versão MPI Seletivo é centrada na ideia da divisao da imagem de entrada, distribuição da imagem e agrupamento dos segmentos de saída. A diferença entre este modelo com o

Modelo MPI Básico é o núcleo do processamento que salta e estima o valor de n colunas da imagem de saída. A Figura 6.1 ilustra o *speedup* obtido. O eixo X define o número de processos; o eixo Y representa o tamanho da máscara; e o eixo Z mostra o *speedup* alcançado na intersecção entre os eixos X e Y.

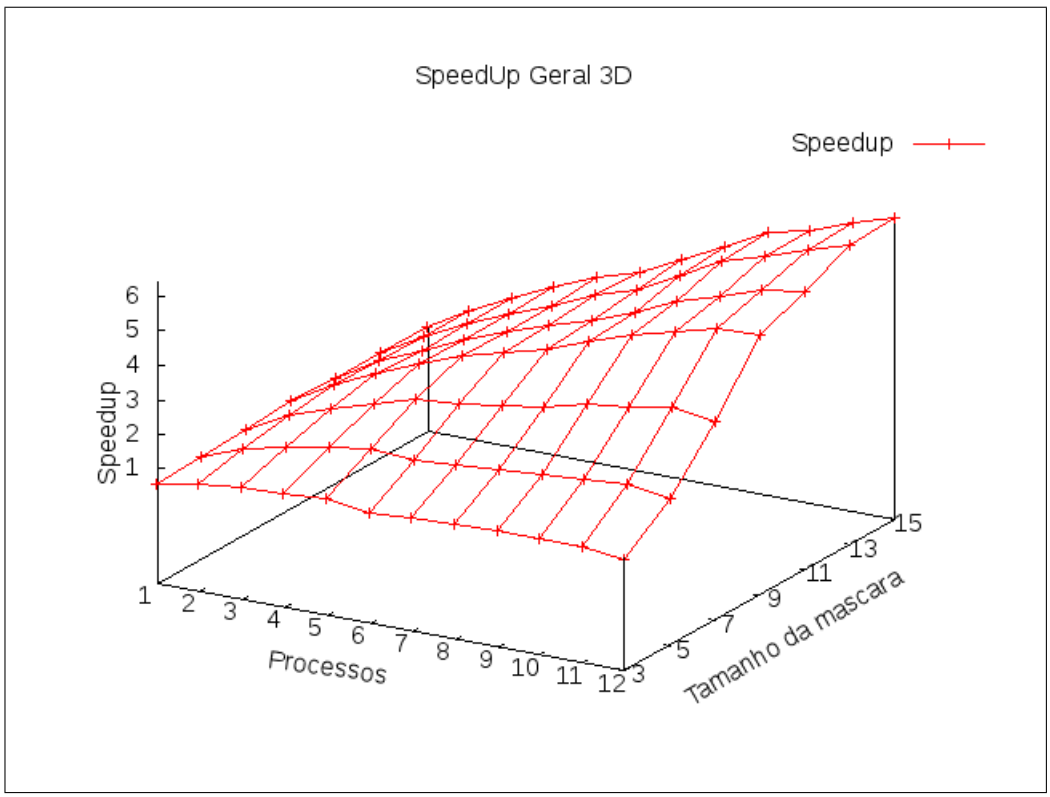

Figura 6.12: Speedup Modelo MPI Seletivo

A máscara de 3 pixels explora levemente a capacidade computacional do *cluster* mesmo com 12 processos em execução. Como esperado, a versão paralela deste modelo apresenta desempenho inferior à sequencial quando executada com 1 processo. Em contra-partida, o desempenho alcançado com uma máscara de 15 *pixels* é de aproximadamente 6,39. Todavia, a partir de 5 *pixels* de máscara já é o suficiente para se obter o dobro de desempenho em relação à versão sequencial.

A Figura 6.13 esboça a eficiência de acordo com o crescimento do número de processos e do tamanho da máscara. O eixo X define o número de processos; o eixo Y caracteriza o tamanho da máscara; e o Z esboça a eficiência alcançada pelo cruzamento entre os dois eixos.

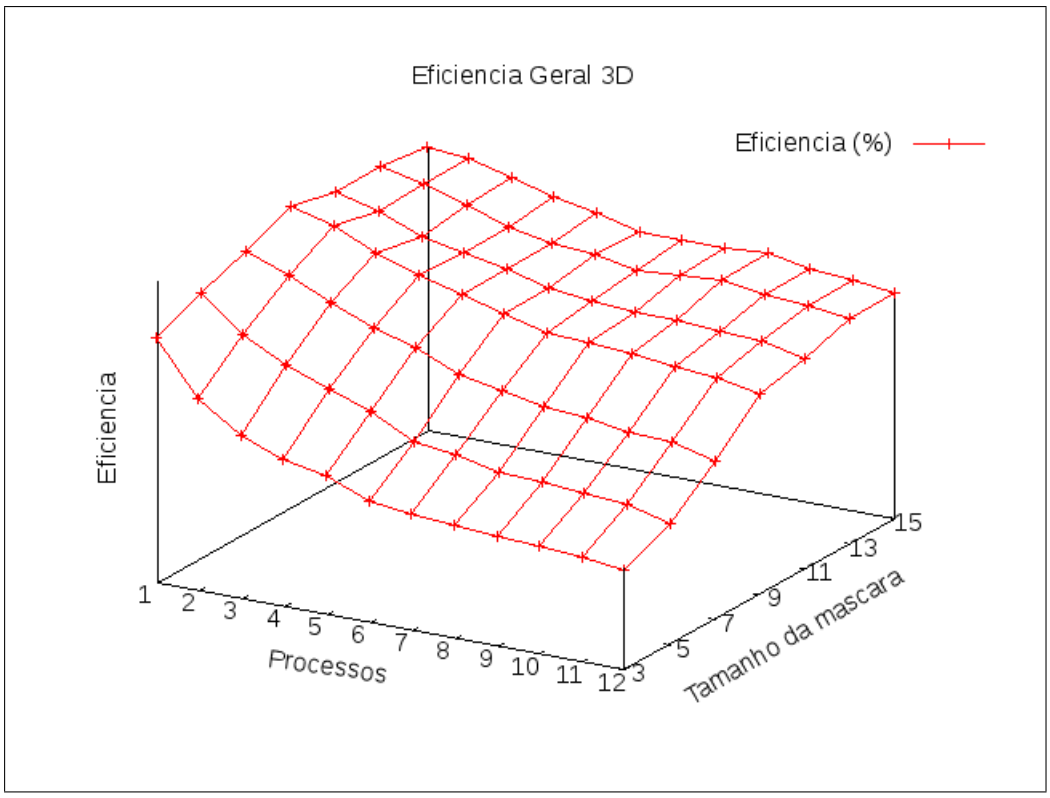

Figura 6.13: Eficiência Modelo MPI Seletivo

A eficiência da paralelização é afetada por dois fatores. O primeiro é o crescimento do número de processos que gera um *overhead* de comunicação. E o pequeno tamanho da máscara que proporciona um baixo conjunto de instruções a serem processadas. Portanto, o pior índice de eficiência é apontado no experimento com 12 processos e máscara de 3 *pixels*; e o melhor aproveitamento é encontrado na execução de apenas 1 processo e 15 *pixels* de máscara.

Deste modo, podemos notar que o ganho de desempenho com 3 pixels de máscara é cada vez menor com o aumentar do número de processos. Apesar de mais amena, esta situação é válida para os demais tamanhos de entrada de máscara. Isso se deve diretamente ao tempo de comunicação que se torna significante conforme o crescimento do número de processos. A Figura 6.14 ilustra o percentual gasto com comunicação, enquanto a Figura 6.15 mostra o tempo gasto com computação.

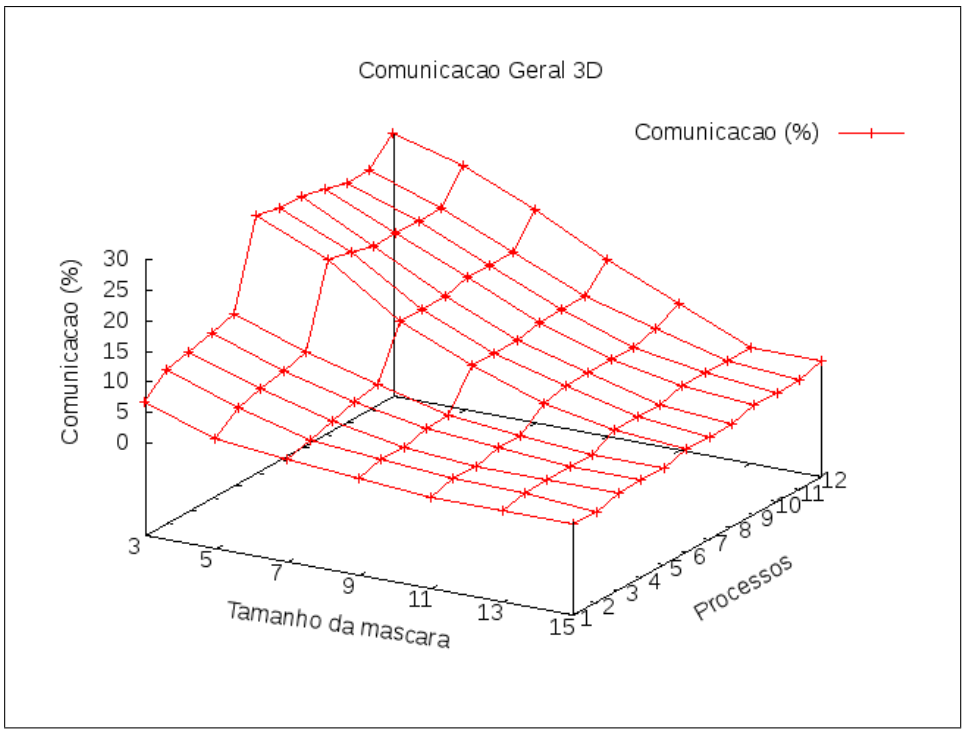

Figura 6.14: Percentual de Comunicação do Modelo MPI Seletivo

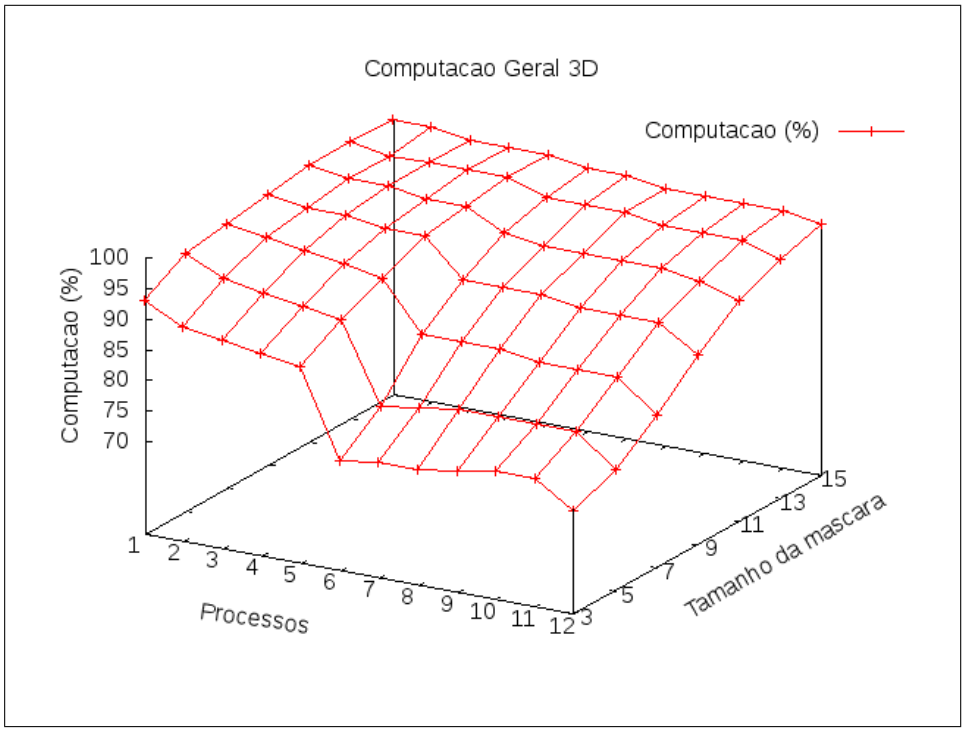

Figura 6.15: Percentual de Computação do Modelo MPI Seletivo

Se observarmos o impacto da comunicação nas execuções com 3 pixels de máscara, a parcela de comunicação do ambiente com 12 processos atinge aproximadamente 28% do tempo total do experimento. E, com 15 pixels de máscara nota-se que a fatia de comunicação torna-se menos efetiva diante do número de instruções geradas gastando aproximadamente 2% com comunicação.

6.3.2 Versão HLRC Seletivo

Derivado da ideia de carga direta da imagem na memória compartilhada distribuída, todos os nós na Versão HLRC Seletivo possuem a visão de uma única memória onde fazem diretamente a leitura e escrita dos dados. A Figura 6.16 ilustra o *speedup* obtido. O eixo X define o número de processos; o eixo Y representa o tamanho da máscara; e o eixo Z mostra o *speedup* alcançado na intersecção entre os eixos X e Y.

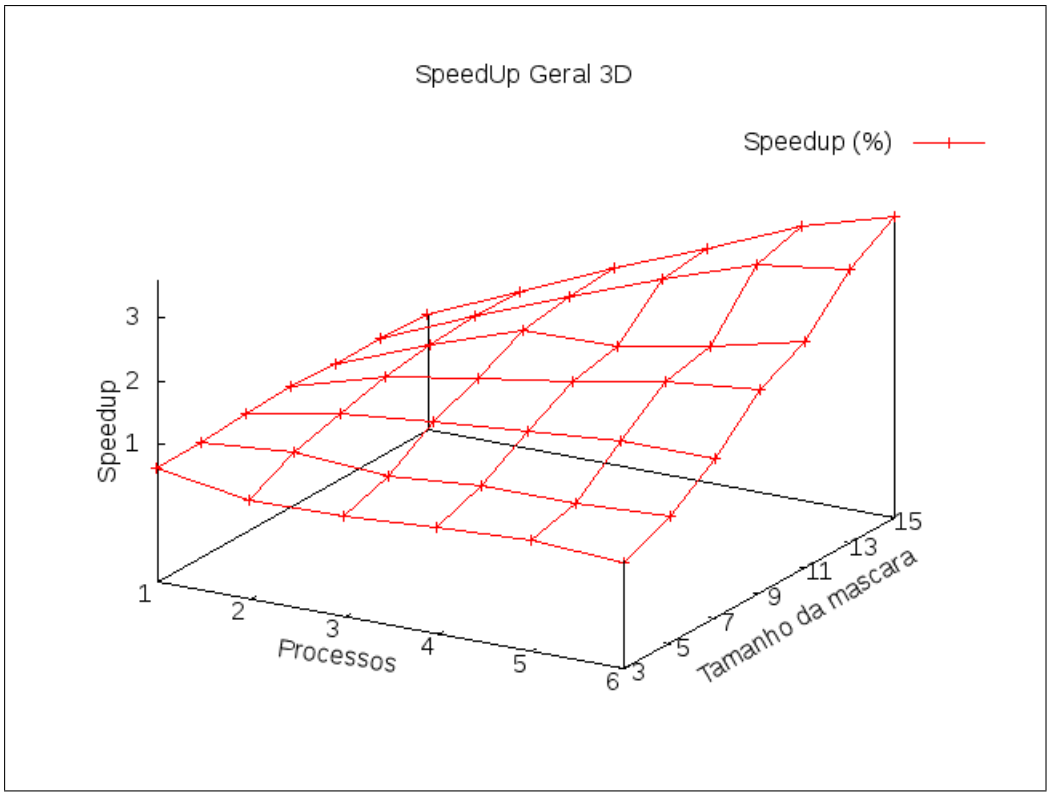

Figura 6.16: Speedup Modelo HLRC Seletivo

Todas as execuções com 1 processo neste modelo indicam desempenho inferior à versão sequencial devido ao *overhead* imposto pelo gerenciamento da plataforma HLRC. Devido ao *overhead* de comunicação, o *speedup* obtido com 1 processo apresenta fator menor que 1. Com 7 pixels de máscara, o *speedup* é superior a 1 a partir de 3 processos. O

pico máximo desempenho encontra-se quando é adotado 15 pixels de máscara de entrada executando-se com 6 processos ao aproximar de 3,57 de *speedup*.

A Figura 6.17 ilustra a eficiência conforme o crescimento do número de processos e do tamanho da máscara. O eixo X define o número de processos; o eixo Y caracteriza o tamanho da máscara; e o Z quantifica a eficiência alcançada pelo cruzamento entre os dois eixos.

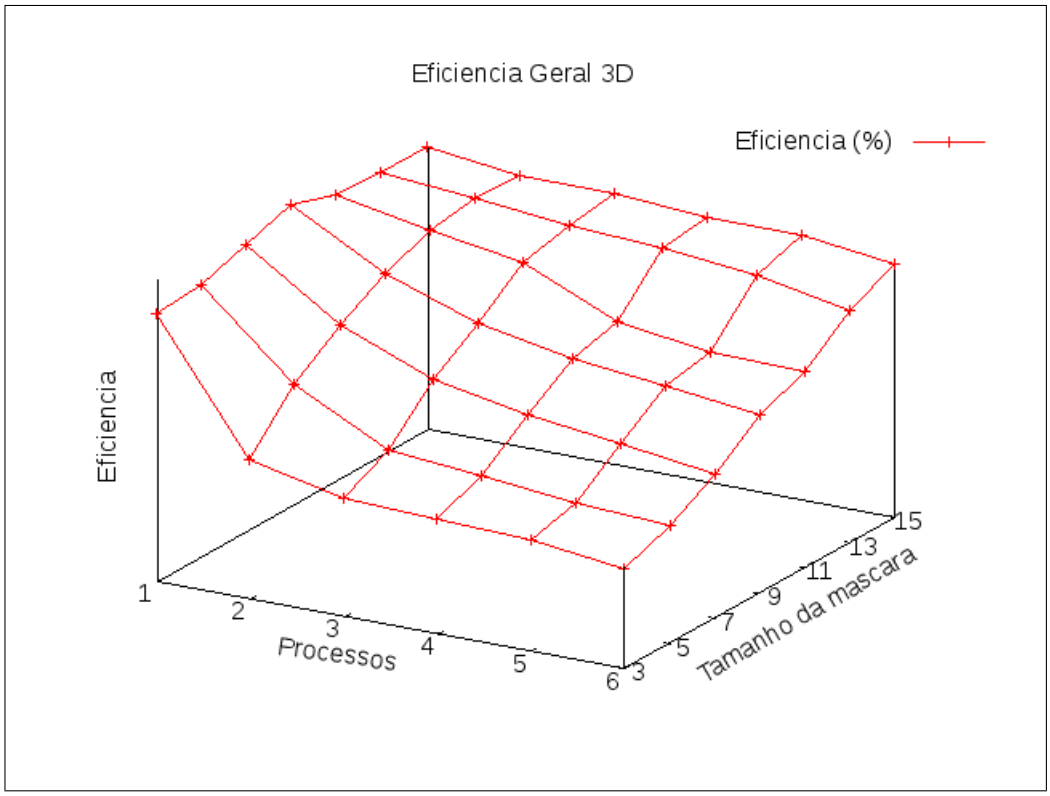

Figura 6.17: Eficiência Modelo HLRC Seletivo

A eficiência com 3 pixels de máscara também é cada vez menor com o crescimento do número de processos. As Figuras 6.18 e 6.19 apresentam o percentual gasto com comunicação e computação. É possível observar que com aumento do tamanho da máscara, o percentual de comunicação torna-se menos significativo, melhorando a eficiência da aplicação.

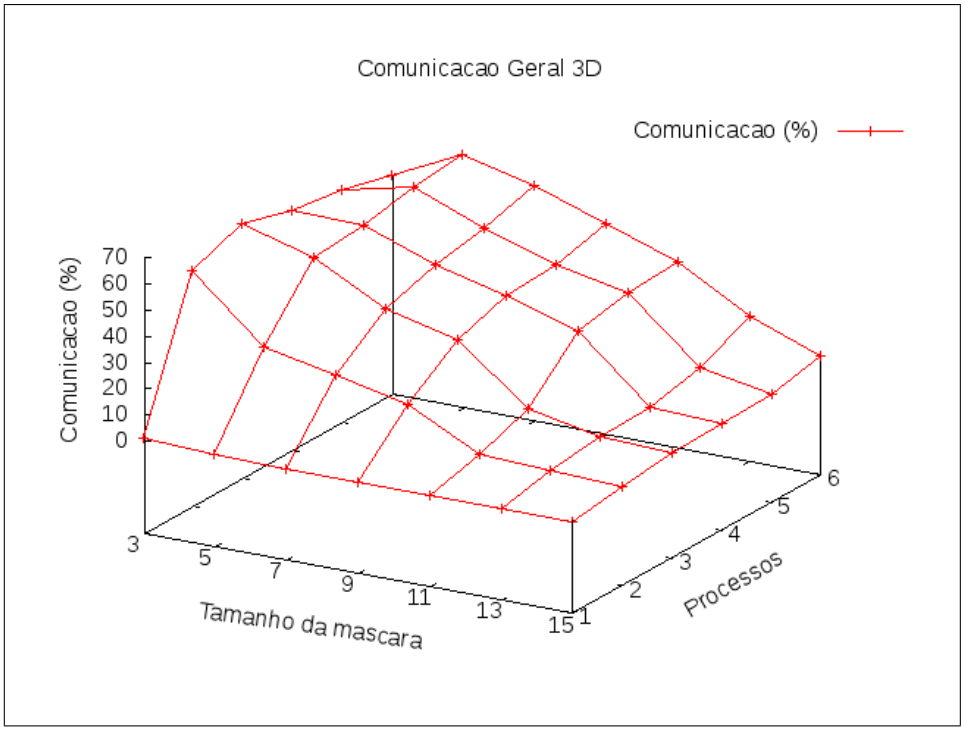

Figura 6.18: Percentual de Comunicação do Modelo HLRC Seletivo

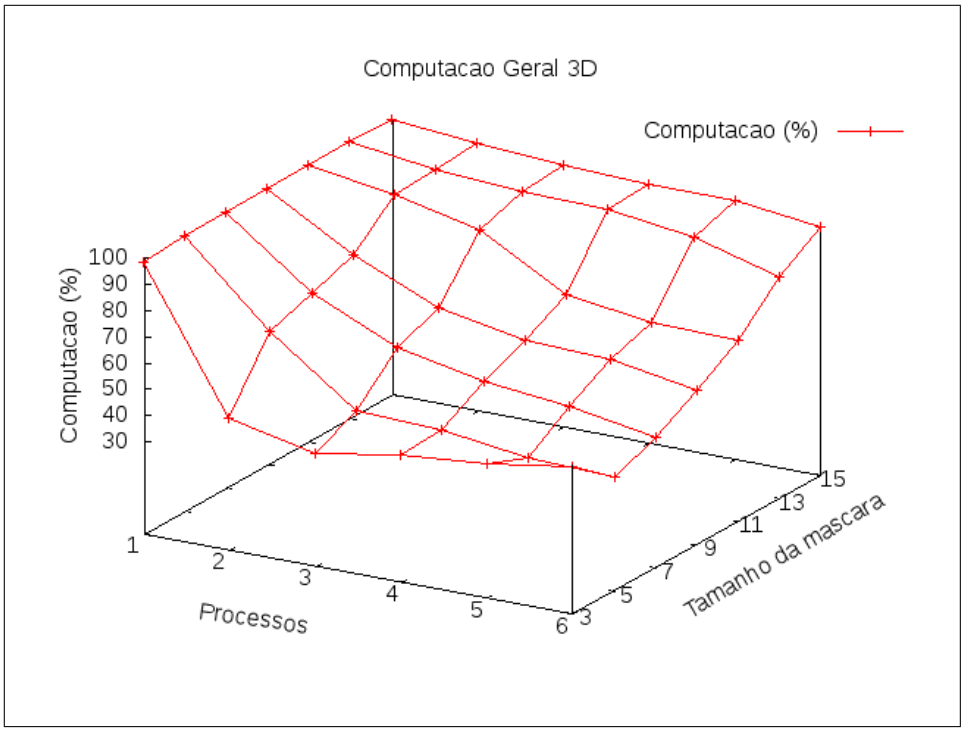

Figura 6.19: Percentual de Computação do Modelo HLRC Seletivo

O impacto da comunicação nas execuções HLRC com 3 pixels de máscara, a parcela de comunicação do ambiente com 3 processos atinge aproximadamente 61,8% do tempo total do experimento. E, com 15 pixels de máscara nota-se que a parcela de computação torna-se mais significativa diante do número de instruções geradas gastando aproximadamente 95% com computação. É válido observar que a plataforma HLRC não adota o modelo *mestre-escravo*. Quando executada com apenas um processo, o *overhead* de comunicação não é significante, mas ainda existe o custo do gerenciamento da plataforma.

6.4 Comparativo dos Modelos

6.4.1 Modelo Básico

Para avaliar a diferença entre as execuções entre o MPI e HLRC são comparados os *speedup* entre cada configuração. A Figura 6.20 apresenta o comparativo da abordagem completa entre as duas plataformas. O eixo X representa as máscaras de entrada, o Y indica o número de processos para a execução e o eixo Z ilustra o índice de desvantagem da implementação HLRC em relação à plataforma MPI.

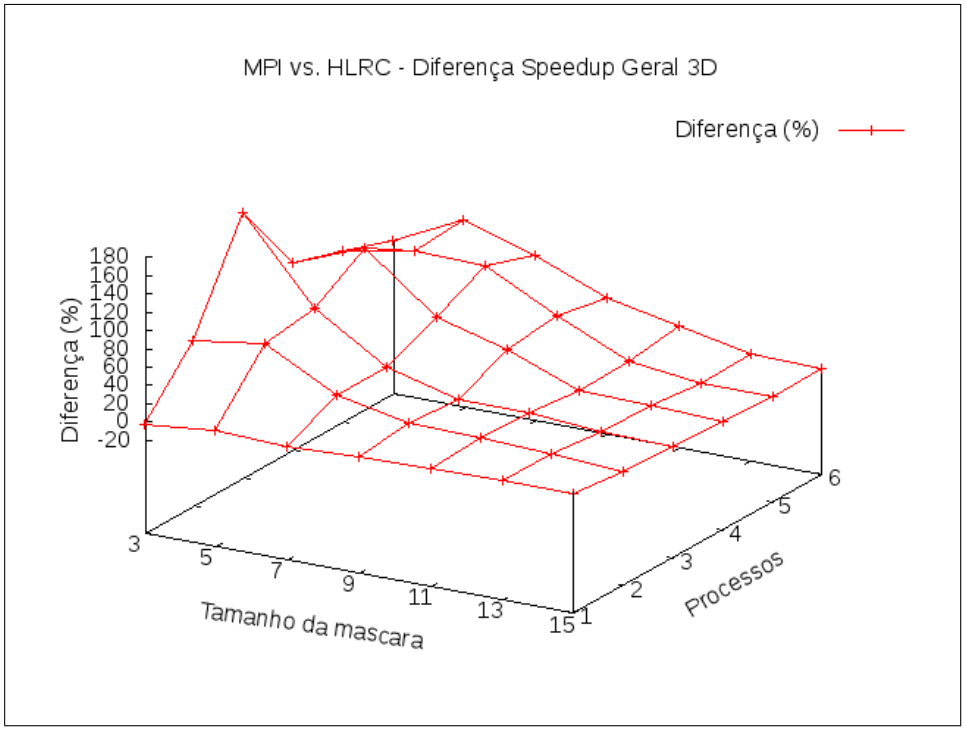

Figura 6.20: Percentual de Computação do Modelo HLRC Seletivo

Podemos observar a irregularidade no comportamento dos experimentos realizados com máscara 3 a 7 indicando picos de 168%. Esta variação acontece por causa do pequeno tempo de computação de máscaras pequenas. A partir da execução com 9 *pixels* de máscara, o HLRC começa a se equiparar com o desempenho do MPI. Os experimentos com 13 e 15 *pixels* de máscara com 5 e 6 processos apontam que o HLRC obteve um aproveitamento superior em relação à plataforma MPI, chegando a aproximadamente 2,8% mais eficiente.

6.4.2 Modelo Seletivo

O comparativo da Abordagem Seletiva ilustrado pela Figura 6.21 esboça as configurações em que cada plataforma de execução obteve vantagem e desvantagem. Nesta abordagem, não ocorre a discrepância tão grande conforme a Abordagem Completa. O comportamento das curvas dos experimentos com 3 e 5 *pixels* de máscara sofre alguns picos de até 118%. O compartamento das execuções entre as duas plataformas se estabiliza com o aumento do tamanho das máscaras. As máscaras de 13 e 15 *pixels* proporcionam ao HLRC uma eficiência superior ao MPI.

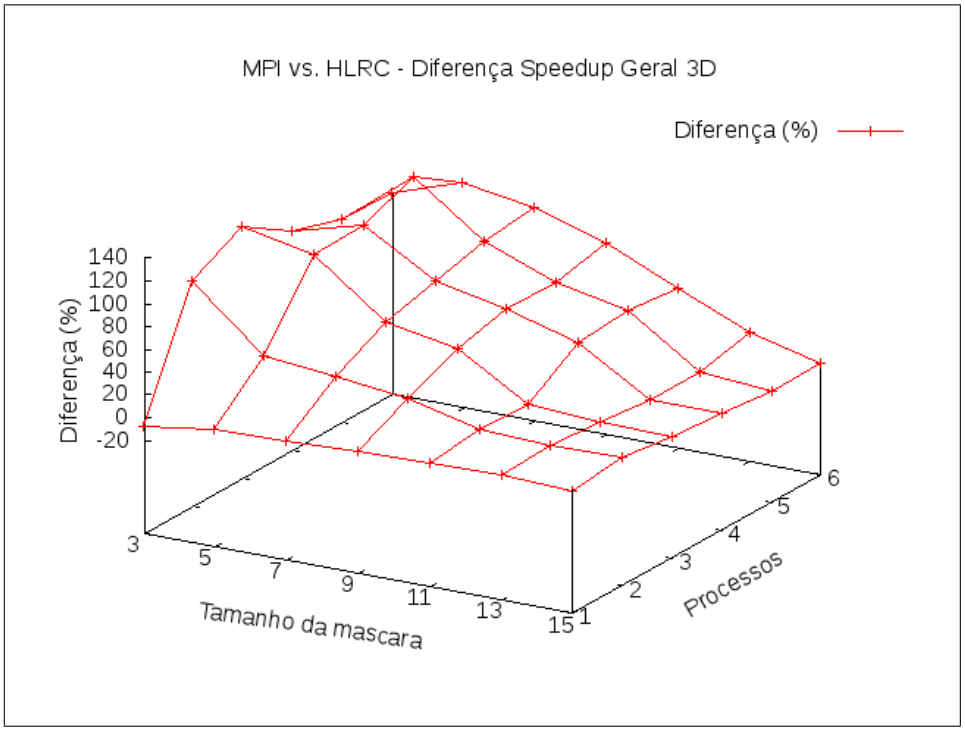

Figura 6.21: Percentual de Computação do Modelo HLRC Seletivo

Apesar desta aplicação ser altamente paralelizável, observou-se que o *speedup* é relativamente baixo nas execuções com máscaras de 3, 5 e 7 *pixels*. Nos experimentos realizados com máscaras de 9, 11, 13 e 15 *pixels* obteve-se melhor desempenho, porém não próximo da curva ideal. Esse fato é decorrente pelos seguintes motivos:

- *overhead* do protocolo de comunicação;
- tamanho dos dados de entrada e saída de aproximadamente 108MB;
- recebimento síncrono dos resultados

Capítulo

7

Conclusões

As simulações computacionais podem ser uma ferramenta de fundamental importância para que pesquisas obtenham a acurácia dos resultados próxima aos reais. Além disso, com um baixo custo, permitem a execução de experimentos que, na prática, envolvem riscos de magnitude inesperadas sem o risco de quaisquer danos materiais.

A computação paralela é indiscutivelmente uma solução atrativa para aumentar o desempenho de aplicações com investimento relativamente pequeno aos supercomputadores. Neste panorama, os *clusters* podem apresentar um desempenho igual ou até mesmo superior aos supercomputadores, sendo uma alternativa para instituições que necessitam de poder computacional, mas não dispõem de recursos financeiros para adquirir um supercomputador. Entretanto, ainda existe uma carência quanto a proposição e utilização de metodologias de paralelização em aplicações complexas do mundo real. O presente trabalho objetiva diminuir esta lacuna, propondo e avaliando um modelo arrojado de paralelização de uma aplicação importante para a ciência geográfica.

Apesar dos 20 anos de pesquisas nesta plataforma, o desenvolvimento de programas paralelos ainda é uma tarefa complexa, sendo necessário realizar mais estudos em busca de técnicas que auxiliem na escrita de programas paralelos. A aplicação alvo apresentada neste trabalho, necessita de muito tempo computacional dependendo dos parâmetros de entrada utilizados, impedindo, muitas vezes, que seja implementada como um módulo de sistemas que fazem o uso de processamento de imagens digitais. A paralelização desta aplicação permite que parâmetros que, anteriormente, eram inviáveis de serem

experimentados, possam ser executados de maneira a contribuir com as pesquisas de áreas relacionadas a este tipo de processamento de imagens digitais.

As execuções dos modelos paralelos apresentaram uma queda significativa, considerando os parâmetros que, anteriormente, eram extremamente custosos na versão sequencial. Os resultados apresentados nesta seção são definidos pela relação entre o tempo da versão sequencial pelo tempo da versão paralela. Os experimentos MPI foram executados com 12 processos, e com 6 processos no HLRC com uma imagem de entrada de 8460x9530 *pixels*.

A redução do tempo de execução foi alcançada em todos os modelos propostos para este trabalho (Capítulo 5). O Modelo MPI Básico atingiu uma redução de tempo de 30,1% em relação à versão sequencial ao utilizar uma máscara de 3 *pixels* e 85,5% com 15 *pixels* de máscara. Com uma máscara de *3 pixels*, a comunicação neste modelo é responsável por 27% do tempo total de execução e é reduzida para 2% quando a complexidade computacional é igual a 15 *pixels* de entrada. A Abordagem Completa implementada sob a plataforma HLRC começa ser vantajosa a partir de 5 *pixels* de máscara obtendo um tempo 25,3% menor a versão sequencial e 73,9% com 15 *pixels* de máscara. A inviabilidade da execução deste modelo com 3 *pixels* de máscara se dá devido aos 51% de custo de comunicação entre os nós. E, a partir dos 13 *pixels* de máscara, o HLRC demonstra ser até 4,6% mais eficaz que a plataforma MPI.

O Modelo Seletivo proposto neste trabalho permite executar de forma inteligente uma aplicação paralela economizando esforço computacional. Na Versão MPI Seletiva adotando 2 saltos, o tempo foi reduzido em 41,5% utilizando 3 *pixels* de máscara e 92,2% com 15 *pixels* de máscara. O uso de 3 *pixels* de máscara, torna a parcela de comunicação responsável por 28% do tempo total de execução. Este percentual é reduzido para 4% quando utiliza-se 15 *pixels* de máscara. A Versão HLRC Seletivo apresenta um ganho de desempenho em relação a versão sequencial a partir de 7 *pixels* de máscara com um ganho de 38,6% e 86% na configuração de 15 *pixels* de máscara. Neste modelo, a inviabilidade da execução com 3 e 5 *pixels* de máscara se dá pela diminuição do número de instruções causada pela otimização do algoritmo. Assim, a fatia de comunicação deste modelo torna-se mais significativa que no Modelo HLRC Básico atingindo 61% de comunicação. Apesar de se perder aproximadamente 10% de precisão, esta abordagem apresentou um ganho de desempenho de até 11,4% sobre a Abordagem Completa.

Assim, com os dados obtidos podemos comprovar que o Modelo Seletivo apresenta desempenho e eficiência satisfatórios para quaisquer tamanhos de máscaras. Mesmo possuindo algumas restrições de tamanho da máscara, os dois modelos HLRC apresentaram eficiência superior ao MPI ao se adotar máscaras acima de 11 *pixels*.

Certamente, os benefícios do modelo aqui apresentado poderão ser maiores se considerarmos uma convolução mais demorada. Por outro lado, algumas políticas de estimativa mais eficientes em termos de acerto poderão consumir tempo considerável a ponto de reduzir os ganhos da paralelização. Estas questões serão analisadas em trabalhos futuros.

Para trabalhos futuros sugerimos a realização de simulações em cima de imagens de multibandas para se analisar a relação computação/comunicação entre os dois tipos de imagens. Além disso, a proposição de políticas de seleção mais elaboradas, bem como o uso deste modelo em imagens de multibandas são desafios ainda a serem vencidos. Outro desafio é a implementação dessas metodologias em arquiteturas baseadas em GPU[1].

[1]Graphics Processing Unit

Referências Bibliográficas

Ahuja, S.; Carriero, N. J.; Gelernter, D. H.; Krishnaswamy, V. Matching language and hardware for parallel computation in the linda machine. *IEEE Trans. Comput.*, v. 37, n. 8, p. 921–929, 1988.

Al Geist, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V. PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing. 1994.

Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, New York, NY, USA: ACM, 1967, p. 483–485.

Baker, M. Cluster computing white. 2000.

Bal, H. E.; Kaashoek, M. F.; Tanenbaum, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, v. 18, n. 3, p. 190–205, 1992.

Bartoli, A.; Corsini, P.; Dini, G.; Prete, C. Graphical design of distributed applications through reusable components. *IEEE Parallel and Distributed Technology*, v. 3, n. 1, p. 37–50, 1995.

Boukerche, A.; Melo, A.; Walter, M.; Melo, R.; Santana, M.; Batista, R. Performance evaluation of a local dna sequence alignment algorithm on a cluster of workstations. In: *Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS2004). IEEE Society, Los Alamitos*, 2004.

Bradski, G.; Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library.* O'Reilly Media, 2008.

BRIGHTWELL, R.; UNDERWOOD, K. D. An analysis of the impact of mpi overlap and independent progress. In: *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, New York, NY, USA: ACM, 2004, p. 298–305.

BRODKIN, J. Ibm drops price on supercomputer. PCWorld - Section Computers, 2007. Disponível em `http://www.pcworld.com/article/135334/ibm_drops_price_on_supercomputer.html`

BURNS, G.; DAOUD, R.; VAIGL, J. LAM: An open cluster environment for MPI. In: *Proceedings of Supercomputing Symposium*, 1994, p. 379–386.

CARPENTER JR, R. Commercial application of the Advanced Regional Prediction System (ARPS). In: *18th Conference on Weather Analysis and Forecasting and the 14th Conference on Numerical Weather Prediction*, 2001.

CHANDRA, S.; LARUS, J.; ROGERS, A. Where is time spent in message-passing and shared-memory programs? *ACM SIGPLAN Notices*, v. 29, n. 11, p. 61–73, 1994.

CHEN, C.; SCHMIDT, B. Computing large-scale alignments on a multi-cluster. In: *IEEE International Conference on Cluster Computing*, 2003.

CHIOLA, G. Some Research Projects on Clusters of Personal Computers. 24th. In: *Proceedings of Euromicro Conference*, 1998.

CLARK, K.; GREGORY, S. PARLOG A parallel logic programming language. 1983.

CRÓSTA, A. P. Processamento digital de imagens de sensoriamento remoto. *Campinas: IG/Unicamp*, v. 170, 1992.

DA CRUZ, N.; GALO, M. Mapeamento das Infestações por Plantas Aquáticas em Reservatórios Utilizando Imagens Multiescala e Redes Neurais Aritificiais. *Universidade Estadual de São Paulo*, 2005.

DANALIS, A.; POLLOCK, L.; SWANY, M.; CAVAZOS, J. Mpi-aware compiler optimizations for improving communication-computation overlap. In: *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, New York, NY, USA: ACM, 2009, p. 316–325.

DONGARRA, J.; DUNIGAN, T. Message-passing performance of various computers. *Concurrency: Practice and Experience*, v. 9, n. 10, p. 915–926, 1997.

DUNCAN, R. A survey of parallel computer architectures. *Computer*, v. 23, n. 2, p. 5–16, 1990.

EVANS, D.; GOSCINSKI, A. *A Survey of Basic Issues of Parallel Execution on a Distributed System.* Relatório Técnico, Citeseer, 1995.

FAUSTINO, A. D. S. *Clusters java: Implementação e avaliação.* Dissertação de Mestrado, Universidade Federal do Rio de Janeiro, 2003.

FAY, D. Q. M. Experiences using inmos proto-occam (tm). *SIGPLAN Not.*, v. 19, n. 9, p. 5–11, 1984.

FISCHER, C.; ESTRADE, J.; JERMAN, J. Implementation of the Limited-Area Numerical Weather Prediction Model Aladin in Distributed Memory. *Lecture notes in computer science*, p. 1411–1416, 1999.

FLEISCH, B. D. *Distributed shared memory in a loosely-coupled environment.* Tese de Doutoramento, chair-Popek, Gerald J., 1989.

FLYNN, M. J.; RUDD, K. W. Parallel architectures. *ACM Comput. Surv.*, v. 28, n. 1, p. 67–70, 1996.

FOSTER, I. *Designing and building parallel programs: concepts and tools for parallel software engineering.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

FREISLEBEN, B.; KIELMANN, T. Approaches to support parallel programming on workstation clusters: A survey. *A Survey, Informatik Berichte, Fachgruppe Informatik, Universitat-GH Siegen*, v. 95, 1995.

GALO, M.; NOVO, E. Indices de paisagem aplicados a analise do Parque Estadual Morro do Diabo e entorno. *Anais do IX Simpósio Brasileiro de Sensoriamento Remoto. INPE/SELPER–Santos*, 1998.

GEIST, A.; GROPP, W.; LUSK, E.; HUSS-LEDERMAN, S.; LUMSDAINE, A.; SAPHIR, W.; SKJELLUM, T.; SNIR, M. MPI-2: Extending the message-passing interface. *Europar96, Lyon (France), 26-29 Aug 1996*, 1996.

GETOV, V.; GRAY, P.; SUNDERAM, V. Mpi and java-mpi: contrasts and comparisons of low-level communication performance. In: *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA: ACM, 1999, p. 21.

GONZALEZ, R.; WOODS, R. Processamento de imagens digitais. *São Paulo: Edgard Blucher*, p. 357–399, 2000.

GORINO, F. V. R. *Balanceamento de carga em clusters de alto desempenho: uma extensão para a lam/mpi.* Dissertação de Mestrado, Universidade Estadual de Maringá, 2006.

GRAHAM, R.; WOODALL, T.; SQUYRES, J. Open MPI: A flexible high performance MPI. *Lecture Notes in Computer Science*, v. 3911, p. 228, 2006.

GRAMA, A.; GUPTA, A.; KARYPIS, G.; KUMAR, V. *Introduction to parallel computing, second edition.* Second ed. Addison Wesley, 856 p., 2003.

GREGOR, D.; LUMSDAINE, A. Design and implementation of a high-performance mpi for c# and the common language infrastructure. In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA: ACM, 2008, p. 133–142.

GROPP, W.; LUSK, E. User Guide for MPICH, a Portable Implementation of MPI. 1996.

GROPP, W.; LUSK, E.; THAKUR, R. *Using mpi-2: Advanced features of the message-passing interface.* Cambridge, MA, USA: MIT Press, 1999.

GROVE, D.; CODDINGTON, P. Communication benchmarking and performance modelling of MPI programs on cluster computers. *The Journal of Supercomputing*, v. 34, n. 2, p. 201–217, 2005.

HARIRI, S.; PARASHAR, M. Tools and environments for parallel and distributing computing. *Wiley Interscience*, 2004.

HAYES, A.; BROOKS, III, E. D.; NASH, T.; WINKLER, K.-H. The role of computational clusters. In: *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, p. 162–164.

HUDAK, P.; SMITH, L. Para-functional programming: a paradigm for programming multiprocessor systems. In: *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, 1986, p. 254.

HWANG, K. Advanced Computer Architecture: Parallelism. *Scalability, Programmability (McGraw-Hill, New York, 1993)*, 1993.

IFTODE, L. *Home-based shared virtual memory.* Tese de Doutoramento, University of Princeton, 1998.

IVANOV, L. A modern course on parallel and distributed processing. *Journal of Computing Sciences in Colleges*, v. 21, n. 6, p. 29–38, 2006.

KARWANDE, A.; YUAN, X.; LOWENTHAL, D. K. Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In: *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA: ACM, 2003, p. 95–106.

KARYPIS, G.; KUMAR, V. Parallel multilevel k-way partitioning scheme for irregular graphs. *Siam Review*, v. 41, n. 2, p. 278–300, 1999.

KELEHER, P.; COX, A.; ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, v. 20, n. 2, p. 13–21, 1992.

KÜHNEMANN, M.; RAUBER, T.; RÜNGER, G. Improving the execution time of global communication operations. In: *CF '04: Proceedings of the 1st conference on Computing frontiers*, New York, NY, USA: ACM, 2004, p. 276–287.

LARSON, S.; SNOW, C.; SHIRTS, M.; PANDE, V. Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology. 2009.

LEBLANC, R.; WILKES, C. Systems Programing with Objects and Actions. In: *Proc. 5th Con Distributed Computing Systems*, 1986, p. 132–139.

LISKOV, B. Distributed programming in argus. *Commun. ACM*, v. 31, n. 3, p. 300–312, 1988.

LU, H.; DWARKADAS, S.; COX, A.; ZWAENEPOEL, W. Message passing versus distributed shared memory on networks of workstations. In: *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, 1995, p. 37–37.

LUSK, E. Programming with MPI on Clusters. In: *2001 IEEE International Conference on Cluster Computing, 2001. Proceedings*, 2001, p. 360–362.

MAJUMDER, S.; RIXNER, S. Comparing ethernet and myrinet for mpi communication. In: *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, New York, NY, USA: ACM, 2004, p. 1–7.

MARJANOVIĆ, V.; LABARTA, J.; AYGUADÉ, E.; VALERO, M. Overlapping communication and computation by using a hybrid mpi/smpss approach. In: *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, New York, NY, USA: ACM, 2010, p. 5–16.

MCCANDLESS, B.; SQUYRES, J.; LUMSDAINE, A. Object oriented MPI (oompi): a class library for the message passing interface. In: *Proceedings of the MPI Developer's Conference*, Citeseer, 1996, p. 87–94.

MEREDITH, M.; CARRIGAN, T.; BROCKMAN, J.; CLONINGER, T.; PRIVOZNIK, J.; WILLIAMS, J. Exploring beowulf clusters. *J. Comput. Small Coll.*, v. 18, n. 4, p. 268–284, 2003.

MILLER, D.; KAMINSKY, E.; RANA, S. Neural network classification of remote-sensing data. *Computers and Geosciences*, v. 21, n. 3, p. 377–386, 1995.

NASCIMENTO, W.; COUTINHO, A.; ALVES, J. Algoritmos de Decomposição de Domínios para Malhas Não Estruturadas. *1o Encontro de Usuários do NACAD-COPPE/UFRJ*, 1996.

NUPAIROJ, N.; NI, L. Performance evaluation of some MPI implementations on workstation clusters. In: *Proceedings of the SPLC Conference*, Citeseer, 1994, p. 98–105.

OJIMA, Y.; SATO, M.; BOKU, T.; TAKAHASHI, D. Design of a Software Distributed Shared Memory System using an MPI communication layer. In: *Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, 2005, p. 8.

PAPADOPOULOS, P.; KATZ, M.; BRUNO, G. NPACI Rocks: Tools and techniques for easily deploying manageable linux clusters. *Concurrency and Computation: Practice & Experience*, v. 15, n. 7, p. 707–725, 2003.

PATRICK, C. M.; SON, S. W.; KANDEMIR, M. T. Enhancing the performance of mpi-io applications by overlapping i/o, computation and communication. In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA: ACM, 2008, p. 277–278.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e projeto de computadores: A interface hardware/software.* 3 ed. Rio de Janeiro: Editora Campus, 2005.

PEDRINI, H.; SCHWARTZ, W. Análise de imagens digitais: princípios, algoritmos e aplicações. *São Paulo: Thomson Learning*, 2008.

RADOVIĆ, Z.; HAGERSTEN, E. Removing the overhead from software-based shared memory. In: *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA: ACM, 2001, p. 56–56.

REHM, W.; WANG, B.; FZJ-IWV, F.; REHM, W.; JULICH, F. Modern Field Code Cluster of Fluid Dynamics and Structure Mechanics with Meta-Computing Grids for Safety and Environment Research Studies. 2003.

ROY, A. J.; FOSTER, I.; GROPP, W.; TOONEN, B.; KARONIS, N.; SANDER, V. Mpich-gq: quality-of-service for message passing programs. In: *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA: IEEE Computer Society, 2000, p. 19.

SAAD, Y. *Iterative methods for sparse linear systems.* Society for Industrial Mathematics, 2003.

SACERDOTI, F.; KATZ, M.; MASSIE, M.; CULLER, D. Wide area cluster monitoring with ganglia. In: *Proceedings of the IEEE Cluster 2003 Conference*, 2003.

SAHNI, S.; THANVANTRI, V. Performance metrics: Keeping the focus on runtime. *IEEE Parallel and Distributed Technology*, v. 4, n. 1, p. 43–56, 1996.

SHAPIRO, E. *Concurrent prolog: collected papers.* MIT Press Cambridge, MA, USA, 1987.

SHI, W.; WEIWU, H.; TANG, Z. *Shared virtual memory: A survey.* Relatório Técnico, Institute of Computing Technology, 1998.

SILBERCHATZ, A.; GALVIN, P.; GAGNE, G. *Operating system concepts.* Willey, 2008.

SMALL, M.; YUAN, X. Maximizing mpi point-to-point communication performance on rdma-enabled clusters with customized protocols. In: *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, New York, NY, USA: ACM, 2009, p. 306–315.

SMITH, B.; BJORSTAD, P.; GROPP, W. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge Univ Pr, 2004.

STENSTROM, P. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 1990.

SUN, X.-H. *Parallel computation models: representation, analysis and applications.* Tese de Doutoramento, Ann Arbor, MI, USA, 1991.

SUN, X.-H.; NI, L. M. Another view on parallel speedup. In: *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, p. 324–333.

TANENBAUM, A. Organização estruturada de computadores. *Rio de janeiro: LTC*, v. 9, 2001.

TANENBAUM, A. S. *Organização estruturada de computadores.* 5 ed. Rio de Janeiro: LTC, 2007.

TANENBAUM, A. S.; WOODHULL, A. *Operating systems design and implementation.* Prentice Hall, 2006.

TANG, H.; YANG, T. Optimizing threaded mpi execution on smp clusters. In: *ICS '01: Proceedings of the 15th international conference on Supercomputing*, New York, NY, USA: ACM, 2001, p. 381–392.

TURNER, M. Landscape ecology: the effect of pattern on process. *Annual review of ecology and systematics*, v. 20, n. 1, p. 171–197, 1989.

WILKINSON, B.; ALLEN, M. *Parallel programming: Techniques and applications using networked workstations and parallel computers.* Pearson Prentice Hall, 2005.

YU, B.-H.; WERSTEIN, P.; PURVIS, M.; CRANEFIELD, S. Performance improvement techniques for software distributed shared memory. In: *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems*, Washington, DC, USA: IEEE Computer Society, 2005, p. 119–125.