UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

RESTAURAÇÃO DE IMAGENS MONOCROMÁTICAS UTILIZANDO PROCESSAMENTO PARALELO

Autor: _____
Jaime Grande Vela

Orientador: _____
D. Sc. Eugenius Kaszkurewicz

Co-orientador: _____
Ph. D. Antônio Petraglia

Examinador: _____
Ph. D. Amit Bhaya

DEL
Setembro de 2007

AGRADECIMENTO

Gostaria de agradecer a colaboração das pessoas que contribuíram na conclusão da minha formação como engenheiro e sem as quais a realização deste projeto não teria sido possível.
Ao meu Orientador de Projeto de Fim de Curso Eugenius Kaszkurewicz, pela sua paciência, motivação e constante apoio, assim como meu co-orientador, Antônio Petraglia.
Aos meus pais pelo constante apoio e confiança que sempre depositaram em mim.

RESUMO

Este trabalho tem por objetivo viabilizar o processamento e restauração de imagens que sofreram efeitos de degradação, utilizando para tanto, algoritmos e redes neurais implementados em computadores paralelos.

A necessidade de se dispor de algoritmos que possam restaurar imagens que sofreram degradações utilizando a computação paralela, deve-se ao alto custo computacional dessas restaurações num ambiente seqüencial, especialmente nos casos em que se lida com imagens de alta resolução.

A quantidade de dados a ser processada nesses casos é elevada, o que torna o processamento seqüencial lento em ambiente computacional convencional. Assim, algoritmos paralelos mostram-se como soluções adequadas.

As degradações de imagens podem ser de diversos tipos: perda de foco, deslocamento, ausência ou excesso de iluminação e ruído. A tecnologia de obtenção e transmissão de imagens tem imperfeições, o que provoca distorção de algum tipo.

Neste trabalho consideramos que as distorções sofridas pelas imagens são invariantes no espaço, ou seja, todos os *pixels* (unidade mínima de discretização de uma imagem) da imagem sofreram o mesmo tipo de distorção e podem ser descritos por modelos lineares. Os modelos utilizados podem ser resolvidos pela utilização de algumas classes de métodos iterativos, que na realidade constituem-se em especializações de redes neurais. A opção pelos métodos iterativos em relação aos diretos se deve à facilidade e conveniência de paralelização.

São apresentadas as equações integrais que descrevem o modelo contínuo e linear de formação da imagem, e a discretização da equação recaindo numa equação que descreve um modelo de sistema linear. Também são introduzidas as classes de métodos iterativos que podem solucionar o problema e que atendem ao critério de convergência e as variações da matriz degradação de acordo com o tipo de distorção sofrida pela imagem.

Apresenta-se e analisa-se a proposta de uma solução paralela para a recuperação de imagens, utilizando uma rede neural baseada no método iterativo de Jacobi, assim como no método de Polyak.

São apresentados os resultados computacionais com as curvas de desempenho do algoritmo paralelo quando aplicado às imagens submetidas a diferentes tipos de degradação.

PALAVRAS-CHAVE

Restauração de imagens

Processamento de imagens

Distorção de imagens

Processamento paralelo

Métodos iterativos

SUMÁRIO

ÍNDICE DE FIGURAS

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A melhoria na qualidade de imagens ou processamento das mesmas para outros fins requer processos computacionalmente pesados e complexos. O desempenho destes processos pode ser melhorado utilizando-se implementações paralelas. Estas implementações paralelas podem dar uma maior agilidade neste âmbito.

Este trabalho consiste no desenvolvimento de processos para restauração e tratamento de imagens que sofreram efeitos de degradação, utilizando para tanto, processos de redes neurais implementados em computadores paralelos assim como a otimização de alguns algoritmos usados no processamento de sinais.

Uma melhoria destes algoritmos, assim como a paralelização dos mesmos pode reduzir significativamente o tempo de execução dos processos, assim como melhores resultados desde o ponto de vista de medida de degradação.

1.2 DESCRIÇÃO DO PROJETO

O objetivo deste projeto é poder restaurar imagens digitais monocromáticas que sofreram algum tipo de degradação ou distorção. Para realizar esta restauração usamos métodos iterativos para resolver os sistemas lineares que representam o fenômeno de degradação e recuperação da imagem. Estes métodos iterativos são implementados com processos que usam redes neurais e são executados em um ambiente de computadores paralelos.

Uma imagem monocromática pode ser representada digitalmente por uma matriz bidimensional, cada valor da matriz (*pixel*) representa o valor de brilho da imagem nesse ponto. À medida que a dimensão da matriz vai aumentando, temos uma imagem com maior resolução. Este aumento na resolução da imagem vai ser refletido em um aumento na memória necessária para poder armazenar o arquivo da imagem assim como na memória necessária no processador para poder tratar a mesma.

Para trabalhar na restauração destas imagens usamos um modelo linear (Jacobi e Polyak) [3], [4] que faz necessário o uso de matrizes de dimensões exponencialmente maiores que a dimensão da matriz que representa a imagem original. Este fato dificulta a execução destes processos em computadores comuns, já que para trabalhar com estas matrizes se requer de muita memória e de alto poder de processamento.

Os modelos lineares antes mencionados (Jacobi e Polyak) têm uma estrutura que facilita a divisão do processo em vários subprocessos menores que podem ser resolvidos ao mesmo tempo, é por isso que a computação paralela aparece como uma opção viável para trabalhar com estas grandes quantidades de dados. Alem da paralelização dos processos de restauração foram otimizados alguns dos algoritmos já conhecidos no processamento de imagens e que são utilizados neste trabalho.

As imagens tratadas foram codificadas no formato PGM (*Portable Gray Map*) [11], este formato facilita a manipulação destas imagens já que as representa como uma matriz de valores arbitrários de escala de cinzas colocados num arquivo tipo texto.

2 FUNDAMENTOS TEÓRICOS

2.1 PROCESSAMENTO DE IMAGENS

O processamento de imagens tem como objetivo melhorar a aparência das imagens e fazer mais evidente nelas determinados aspectos que se deseja ressaltar. O processamento pode ser feito por meios óticos ou por meios digitais usando métodos computacionais. Neste trabalho utilizamos o meio digital.

Da mesma maneira do método ótico, os fundamentos do processamento digital de imagens têm sido estabelecidos faz muitos anos, mas não eram realizados devido à falta de computadores eficientes. Com o surgimento de computadores de alto desempenho e memória, começou-se a desenvolver este campo.

Um dos primeiros lugares onde se realizaram projetos de processamento de imagens foi no Laboratório de Propulsão a Jato (*Jet Propulsion Laboratory*) [18], no ano de 1959 com a finalidade de melhorar as imagens enviadas pelos satélites. Os resultados obtidos por este projeto foram tão impressionantes e num período de tempo tão curto que as aplicações foram levadas rapidamente a outros campos.

2.2 PROCESSAMENTO DIGITAL DE IMAGENS

Nos últimos anos, o processamento digital de imagens tem sido usado amplamente em diversos campos da ciência humana tais como: Medicina, Biologia, Física e Engenharia [14].

Por meio do processamento digital, é possível manipular imagens digitais em um computador com o objetivo de obter informações objetivas e claras da cena capturada pela câmera. O processamento digital de imagens tem dois objetivos fundamentais:

- Melhoramento de uma imagem digital com fins de interpretação.
- Tomada de decisão de maneira automática de acordo com o conteúdo da imagem.

Como aplicações típicas, podemos mencionar: detecção de presença de objetos, inspeção visual automática, medição de características geométricas e de cor de objetos, classificação de objetos, restauração de imagens e melhoramento da qualidade das imagens.

2.2.1 Representação de Imagens Digitais

O termo imagem monocromática, ou simplesmente imagem, refere-se à função bidimensional de intensidade da luz $f(x,y)$, onde x e y denotam as coordenadas espaciais e o valor f em

qualquer ponto *(x,y)* é proporcional ao brilho (ou níveis de cinza) da imagem naquele ponto [14].

Uma imagem digital é uma imagem *f(x,y)* discretizada tanto em coordenadas espaciais quanto em brilho (intensidade da luz). Uma imagem digital pode ser considerada como uma matriz cujos índices de linhas e de colunas identificam um ponto na imagem, e o correspondente valor do elemento da matriz, identifica o nível de cinza naquele ponto. Os elementos dessa matriz digital são chamados: elementos da imagem, elementos da figura, *pixels* ou *pels* (abreviações de *picture elements*). Quanto mais *pixels* uma imagem tiver melhor é a sua resolução e qualidade [14].

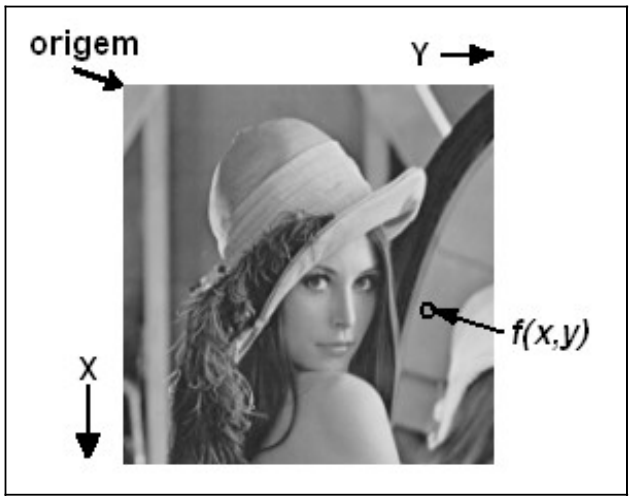

Figura 2.1: Convenção dos eixos para representação de imagens digitais.

2.3 FILTROS DIGITAIS

Em muitas aplicações no tratamento de imagens digitais é necessária a filtragem destas tanto na remoção de ruído quanto na segmentação, enfoque, suavizado, etc. Para estes propósitos são utilizados os filtros digitais 2-D lineares e não lineares.

Os processos de melhoria da imagem podem agrupar-se tendo em conta o mecanismo utilizado na modificação da imagem digital. Assim existem algoritmos que modificam o histograma ou função de distribuição de intensidade, algoritmos que modificam o conteúdo de cada *pixel* da imagem de acordo com os valores dos *pixels* vizinhos aplicando uma determinada função pré-determinada e outros que filtram determinadas freqüências.

Uma parte muito importante no processo de restauração de imagens baseia-se na aplicação de funções, cujo resultado depende unicamente dos níveis de intensidade de cada *pixel* da imagem, mas não da posição dentro da imagem (*Position-invariant*), que permitem ressaltar

determinados elementos da imagem, melhorar o enfoque ou ainda reduzir o ruído no fundo da imagem.

Entre os principais objetivos da filtragem digital podemos mencionar a melhoria na qualidade de uma imagem (*realce-sharpening-enhancement*), eliminação de ruídos (*noise*), correção de imagens, segmentação, eliminação de distorções e criação de efeitos artísticos.

As funções aplicadas às imagens que por analogia com a teoria de processamento de sinais se denominam filtros, baseiam-se no processo de convolução e nas propriedades da convolução da transformação de Fourier, o que simplifica o cálculo matemático e os tempos de computação.

Por este motivo, muitos destes filtros herdam o nome segundo sua ação no espaço das freqüências (filtros Passa-Alto, Passa-Baixo ou Direcionais). Embora cada vez mais freqüente, nos sistemas de processamento de imagens, se lhes atribui nomes de acordo com o resultado visual que produzem na imagem filtrada (filtros de enfoque ou dês-enfoque).

Um filtro de convolução, para uma imagem digital, no espaço real *(x, y)*, pode representar-se como uma matriz quadrada ou retangular (máscara), de dimensões pré-definidas ($M \times N$). A matriz de convolução se desloca sob a imagem de tal forma que o elemento central da matriz de convolução coincida com cada um dos *pixels* da imagem. Em cada posição, se multiplica o valor de cada *pixel* da imagem, que coincide em posição com um elemento da matriz de convolução, pelo valor deste. O *pixel* da imagem, que coincide com o elemento central da matriz de convolução é substituído pela suma dos produtos.

2.3.1 Tipos de filtros digitais

Em filtragem digital existem dois tipos de domínio:

- Filtragem no Domínio da Freqüência. Neste domínio utilizamos a Transformada de Fourier, este tipo de filtragem é baseado no teorema da convolução.
- Filtragem no Domínio Espacial. Utilizamos máscaras.

2.3.2 Máscaras

O efeito que os filtros ocasionam representa-se digitalmente pela aplicação de máscaras nas matrizes que representam uma imagem.

Na figura 2.2 é representada uma matriz de valores onde aplicamos uma máscara de dimensão (3×3), o resultado da aplicação dessa máscara é um novo valor *P33'* na posição do

elemento *P33* (figura 2.3). Este novo valor (*P33'*) é o resultado de uma função matemática dos elementos *f* contidos na máscara.

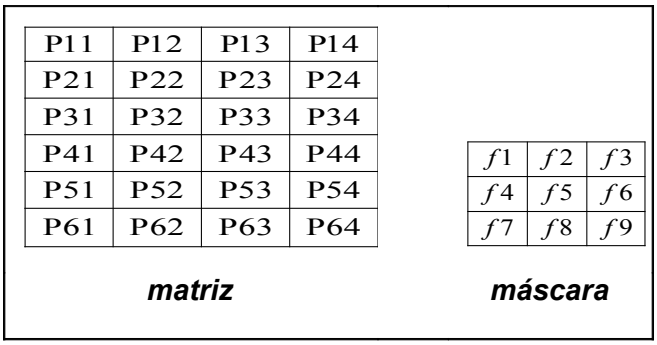

Figura 2.2. Matriz imagem e máscara.

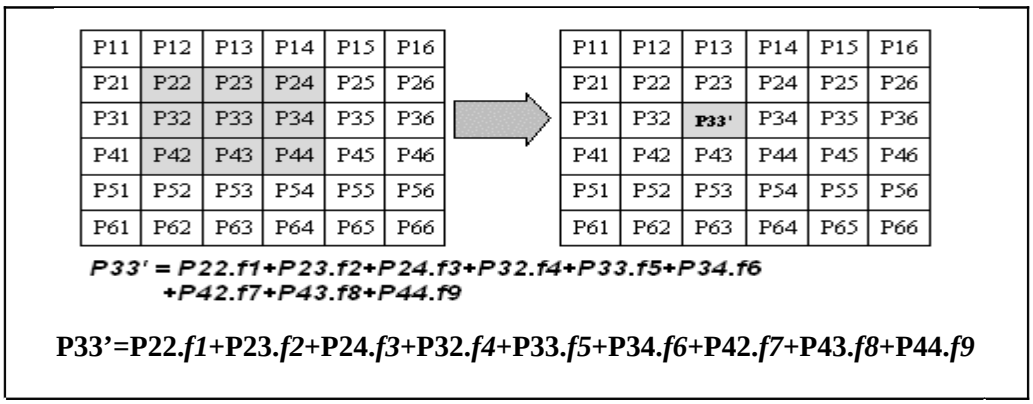

Figura 2.3. Aplicação de filtros.

2.3.3 Filtragem no domínio da freqüência (Domínio Fourier)

A base para a técnica baseada no domínio da freqüência é o teorema de Convolução. Convolução é o produto da transformada de Fourier aplicada *pixel* a *pixel* entre duas imagens envolvidas.

O efeito na aquisição de uma imagem é representado pela Função de Espalhamento de Pontos, PSF (*Point Spread Function*).

Pequenas mudanças nos níveis de cinza são representadas como sendo de baixa freqüência e mudanças bruscas nos níveis de cinza (bordas) são representados como sendo de alta freqüência.

Seja *g(x,y)* uma imagem formada pela convolução de uma imagem *f(x,y)* e um operador invariante *h(x,y)* (função de transferência do filtro), onde:

$g(x,y) = h(x,y) * f(x,y)$

Pelo teorema da convolução, tem-se a seguinte relação no domínio da freqüência.

$G(x,y) = H(x,y) \cdot F(x,y)$ onde:

G, H e *F* são as transformadas de Fourier de *g, h* e *f*.

A convolução no domínio da freqüência é mais fácil que no domínio espacial já que é simplesmente o produto da transformada de Fourier *pixel* a *pixel* entre as imagens envolvidas, neste caso se o núcleo for menor, completa-se a matriz *H* com zeros.

Os principais filtros no domínio da freqüência são o Ideal, Batherworth, Cheby, Gaussiano e Filtro Wiener.

2.3.4 Filtragem Espacial

Esta técnica é baseada na aplicação de mascaras na imagem, estas máscaras são normalmente chamadas de Filtros Espaciais e são divididos em dois tipos principais.

- Filtros lineares. Onde temos os filtros de passa - alta, passa - baixa e passa-banda.
- Filtros não lineares. Podemos mencionar a filtragem morfológica.

Filtros lineares

Os filtros lineares se baseiam na mudança do nível de cinza de um ponto (*pixel*) em função dos outros pontos da sua vizinhança mediante a aplicação de uma máscara do tipo:

ω_1	ω_2	ω_3
ω_4	ω_5	ω_6
ω_7	ω_8	ω_9

Figura 2.4: Máscara 3x3

Onde o nível de cinza do *pixel* central é dado por:

$$\omega t = \left(\omega_1 \times p_1 + \omega_2 \times p_2 + \omega_3 \times p_3 + \omega_4 \times p_4 + \omega_5 \times p_5 + \omega_6 \times p_6 + \omega_7 \times p_7 + \omega_8 \times p_8 + \omega_9 \times p_9\right) / k \text{ s}$$

endo *k* uma constante para evitar que o nível de cinza fique fora do limite máximo dos valores dos *pixels* e p_i o valor de nível de cinza do *pixel i* dentro da máscara. Os coeficientes da máscara dependem do tipo de filtro linear, no caso do filtro passa-baixos, por exemplo, são todos positivos [16].

A seguir mostramos alguns exemplos para filtros lineares.

Exemplo 2.1: Filtro de média

Neste caso a máscara é uma matriz de dimensão (5×5) com valores iguais de 1/25.

$$\frac{1}{k}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figura 2.5: Máscara 5x5 (k = 25).

Figura 2.6: Imagem da Lena sem degradação.

Figura 2.7: Imagem da Lena após a filtragem (filtro de média 5x5).

Na figura 2.7 podemos observar os efeitos do filtro aplicado à imagem original (figura 2.6), este filtro cria o efeito de uma imagem embaçada.

Exemplo 2.2: Filtro Gaussiano

Neste caso a máscara é uma matriz com os valores mostrados na figura 2.8.

$$\frac{1}{k}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 \\ 3 & 4 & 5 & 4 & 3 \\ 2 & 3 & 4 & 3 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

Figura 2.8: Máscara 5x5 (k = 65).

Aplicando este filtro também podemos notar que o efeito na imagem (figura 2.10) é de uma imagem embaçada, a diferença deste filtro é que a influencia dos *pixels* vizinhos não é igual na mascara inteira, o peso dos vizinhos decai conforme o *pixel* vai se afastando do centro da máscara.

Figura 2.9: Imagem da Lena sem degradação.

Figura 2.10: Imagem da Lena após a filtragem (filtro Gaussiano 5x5).

2.3.5 Filtros de Mediana

Este tipo de filtro é um exemplo de Filtro não Linear, o nível de cinza do *pixel* no qual se aplica a máscara também depende dos níveis de cinza dos pontos vizinhos, mas não é uma função linear destes. Neste caso é a mediana dos níveis de cinza dos pontos vizinhos [17].

Algoritmo da mediana:

Dado um conjunto de valores $v = [a, b, c, ...]$ temos que ordenar os valores de maneira crescente ou decrescente, depois obtermos a mediana selecionando o valor que fica no meio deste novo vetor crescente. No caso de um número par de valores, podemos selecionar qualquer um dos valores que ficarem no meio.

Exemplo 2.3: Cálculo de mediana.

v = [5, 6, 8, 9, 3, 1, 2, 4, 7] (Vetor original)

v' = [1, 2, 3, 4, **5**, 6, 7, 8, 9] (Vetor reordenado de forma crescente)

Mediana(v) = **5**

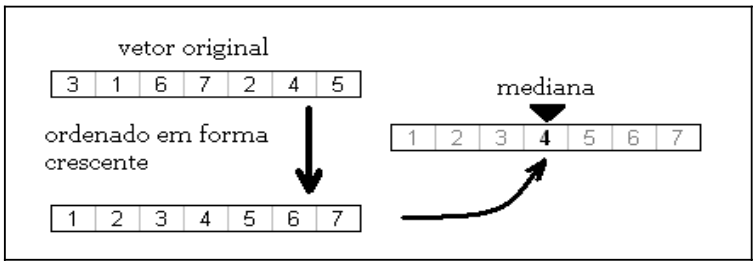

Figura 2.11: Representação gráfica do cálculo da mediana de um vetor.

Figura 2.12: Imagem da Lena com ruído (Salt & Pepper 4%).

Figura 2.13: Imagem da Lena após a filtragem (filtro de Mediana 3x3).

2.4 DISTORÇÃO DE IMAGENS

Quando capturamos uma imagem, na verdade obtemos o resultado da penetração da radiação refletida na imagem original num meio não ideal que geralmente tem propriedades físicas que ocasionam uma distorção no percurso da radiação que leva à formação da imagem.

Na figura 2.14 temos uma representação gráfica do processo de captura de uma imagem. Quando uma determinada radiação (luz visível) atinge um objeto não opaco, este reflete parte da radiação recebida.

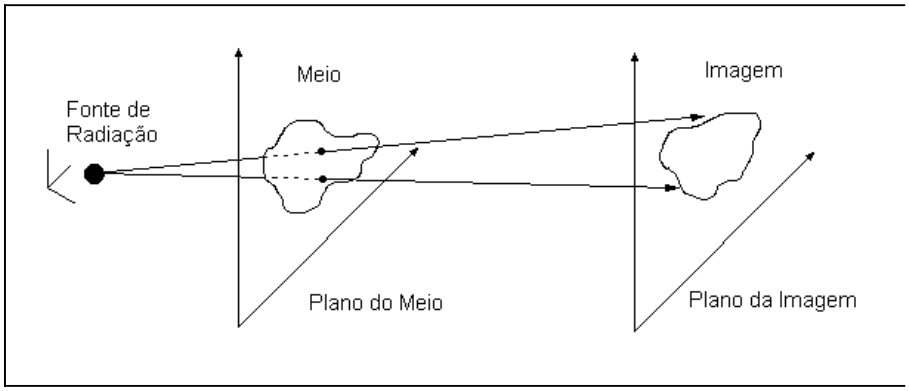

Figura 2.14: Formação de uma imagem pela passagem de radiação num meio não ideal.

As degradações que são encontradas na imagem gerada são introduzidas pelo sistema de formação da imagem e/ou por fatores externos que geram mudanças no ambiente em que é feita a aquisição da imagem, como variação de temperatura ou pressão. As formas mais comuns de degradação são: A resolução finita dos sensores utilizados na aquisição da imagem, embaçamento devido ao movimento dos sensores ou do alvo no momento da aquisição, refra-

ção e embaçamento provocado por problemas de foco [1]. As distorções em imagens podem ser classificadas em duas categorias [2]

- Variantes no espaço. A distorção sofrida depende da região da imagem que estamos analisando. Este tipo de distorção é ocasionado por problemas na aquisição ou por fatores externos.
- Invariantes no espaço. A distorção é a mesma em todas as regiões da imagem. Este tipo de distorção pode ser provocado por problemas de foco ou movimento da câmera.

No pressente trabalho, consideramos somente as distorções invariantes no espaço e lineares por elas terem um modelo matemático mais simples. Muitas das distorções não-lineares podem ser aproximadas por distorções lineares.

2.5 MODELAGEM MATEMÁTICA DA DISTORÇÃO DE IMAGENS

O processo de formação da imagem pode ser representado matematicamente pela equação integral de Fredholm [1] que mostramos a seguir.

$$g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(x,\xi,y,\eta)\, f(\xi,\eta)\, d\xi\, d\eta + \varepsilon(x,y) \qquad \textbf{(2.1)}$$

Sendo $f(\xi,\eta)$ e $g(x,y)$ as funções que descrevem o objeto (no plano de coordenadas (ξ,η), que é referido como plano do objeto) e a imagem gravada (no plano de coordenadas (x,y), que é referido como plano da imagem), $h(x,\xi,y,\eta)$ é a forma analítica da função degradação e $\varepsilon(x,y)$ representa o ruído presente na imagem.

Na figura 2.15 mostramos um esquema representativo do objeto e da imagem em seus respectivos sistemas de coordenadas.

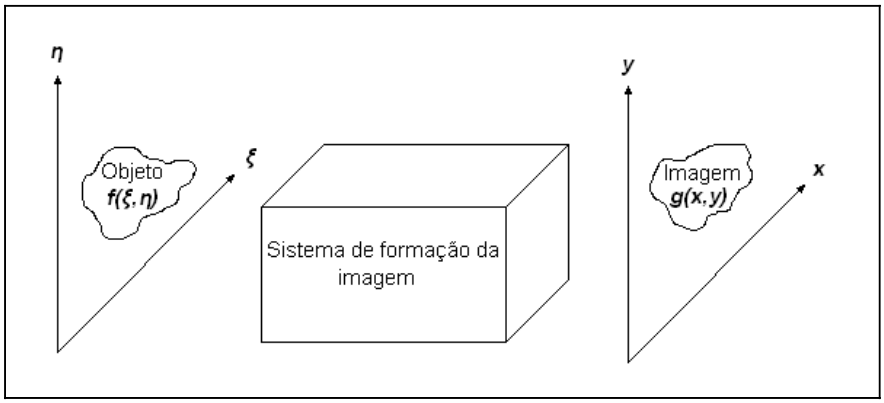

Figura 2.15: Representação esquemática do sistema de formação de uma imagem.

O conhecimento do ruído $\varepsilon(x,y)$ é limitado pelos métodos estatísticos. No processamento de imagens, a função de degradação $h(x,\xi,y,\eta)$ geralmente é conhecida a priori e representa a resposta do sistema a um impulso unitário nas coordenadas (x,y). Este impulso unitário é representado, no campo da ótica como um ponto de luz, é por isso que a função $h(x,\xi,y,\eta)$ é também conhecida como *Point Spread Function* (PSF). A função PSF assume um papel fundamental no processo de restauração de uma imagem assim como na escolha do método de restauração.

O modelo de Fredholm [1] para formação de imagens, mostrado na equação *(2.1)* pode ser reescrito no caso de sistemas discretos como se mostra na equação a seguir.

$$g(x,y) = \sum_{\alpha}^{N} \sum_{\beta}^{M} f(\xi,\eta)h(x,y;\xi,\eta) + \varepsilon(x,y) \qquad \textbf{(2.2)}$$

Se considerarmos $h(x,\xi,y,\eta)$ uma função linear então a equação *(2.1)* pode ser reescrita considerando $f(\xi,\eta)$, $g(x,y)$ e $\varepsilon(x,y)$ como vetores coluna de dimensão $(M\times N)$. Para poder deixar estas matrizes (f, g e ε) na forma de vetor temos que usar um rearranjo lexicográfico destas informações. Um rearranjo lexicográfico consiste em reagrupar o conteúdo da matriz num vetor único que tem cada linha da matriz empilhada uma após a outra.

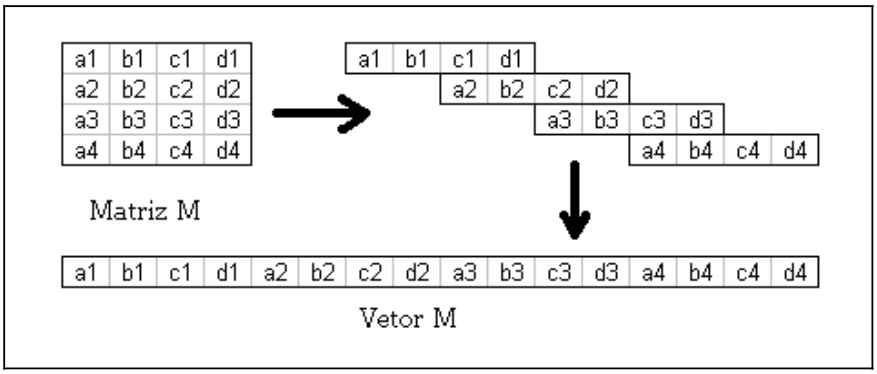

Figura 2.16: Representação gráfica do rearranjo lexicográfico de uma matriz M.

Agora podemos representar a equação *(2.1)* como uma equação linear representada na equação (*2.3*).

$$g = Hf + \varepsilon . \qquad \textbf{(2.3)}$$

Na equação linear acima *(2.3)* g e f representam os arranjos lexicográficos das matrizes de imagem degradada e imagem original respectivamente, ε representa a componente de ruído

aditivo e H é uma matriz operador que representa a função $h(x,\xi,y,\eta)$. A multiplicação da matriz H com o vetor imagem f executa a mesma operação que convoluir $f(\xi,\eta)$ com $h(x,\xi,y,\eta)$.

2.6 TIPOS DE DISTORÇÃO

Existem muitos tipos de distorção, mas nesta seção vamos mencionar somente os que foram tratados no processo de restauração.

2.6.1 Motion Blur (Distorção por movimento)

No caso da distorção *Motion Blur* (Distorção por Movimento) a função *PSF* é representa por uma matriz que descreve as contribuições de brilho de cada um dos vizinhos do *pixel* central que entram no processo de distorção. Para uma distorção deste tipo (*Motion Blur*) temos dois parâmetros a considerar, o tamanho da distorção e o ângulo da distorço.

Tamanho da distorção: É o comprimento da distorção em *pixels*, este valor representa o número de *pixels* que entram na movimentação da câmera.

Ângulo da distorção: Este valor representa o ângulo em sentido anti-horário no qual se movimenta a câmera.

Quando o ângulo é 0 (zero) é fácil notar que a distorção *Motion Blur* é basicamente uma média dos valores dos *pixels* vizinhos que entram na distorção.

Exemplo 2.4: Neste exemplo mostramos uma distorção do tipo *Motion Blur* com 10 (dez) *pixels* de comprimento e 0° (zero graus).
PSF = [0.05 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.05]

Na figura 2.18 podemos ver que o efeito de aplicar esta distorção é a aparência de ter capturado a imagem em movimento.

Figura 2.17: Lena original.

Figura 2.18: Lena após filtro Motion Blur de 10 pixels e 0°.

Exemplo 2.5: Neste caso mostramos uma distorção do tipo *Motion Blur* com 15 (quinze) *pixels* de comprimento e 0° (zero graus). Podemos notar que a distribuição dos valores dos pesos dos *pixels* vizinhos da uma média aritmética dos mesmos.

$PSF = [a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a]$, $a = 0.0667$

Na figura 2.20 podemos ver o efeito deste filtro com 15 pixels, é evidente que a distorção é mais forte que no exemplo com 10 *pixels*.

Figura 2.19: Lena original.

Figura 2.20: Lena após filtro Motion Blur de 15 pixels e 0°.

Exemplo 2.6: Neste caso mostramos uma distorção do tipo *Motion Blur* com 21 (vinte e um) *pixels* de comprimento e 0° (zero graus).

$PSF = [a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a]$, $a = 0.0476$

Neste ultimo exemplo podemos ver o aumento no numero de pixels que participam da distorção aumenta também o grau de distorção da imagem (ver figura 2.22).

Figura 2.21: Lena original.

Figura 2.22: Lena após filtro Motion Blur de 21 pixels e 0°.

Quando mudamos o ângulo da distorção a representação deste efeito inclui *pixels* que de diferentes linhas e colunas já que neste caso, devido ao formato da matriz da imagem, se faz impossível colocar o vetor de distorção ocupando somente o número de *pixels* do vetor original.

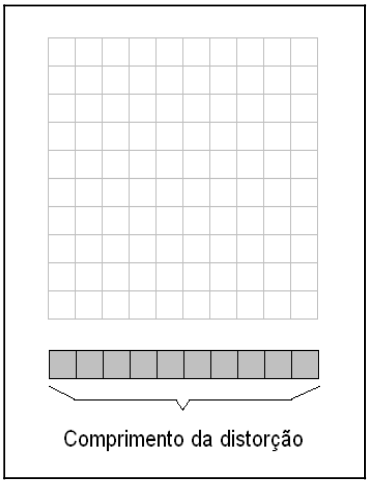

Figura 2.23: Representação gráfica do comprimento da distorção.

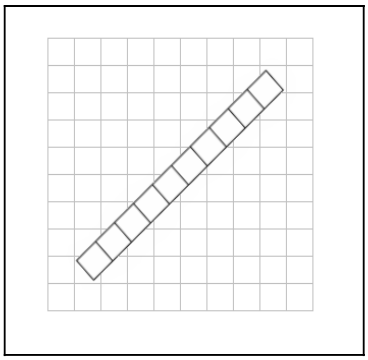

Figura 2.24: Distorção aplicada com um ângulo de 45°.

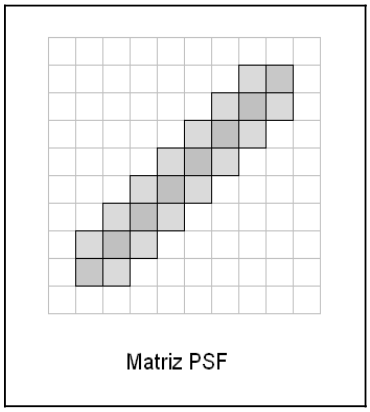

Figura 2.25: Distribuição dos pesos da distorção aplicada a 45°.

Exemplo 2.7: Neste caso mostramos uma distorção do tipo *Motion Blur* com 10 (dez) *pixels* de comprimento e 45° (quarenta e cinco graus).

$$PSF = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & b & a & c \\ 0 & 0 & 0 & 0 & 0 & b & a & b & 0 \\ 0 & 0 & 0 & 0 & b & a & b & 0 & 0 \\ 0 & 0 & 0 & b & a & b & 0 & 0 & 0 \\ 0 & 0 & b & a & b & 0 & 0 & 0 & 0 \\ 0 & b & a & b & 0 & 0 & 0 & 0 & 0 \\ c & a & b & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{aligned} a &= 0.089584 \\ b &= 0.026239 \\ c &= 0.014511 \end{aligned}$$

Figura 2.26: Lena original.

Figura 2.27: Lena após filtro Motion Blur de 10 pixels e 45°.

Exemplo 2.8: Neste caso mostramos uma distorção do tipo *Motion Blur* com 15 (quinze) *pixels* de comprimento e 45° (quarenta e cinco graus).

$$PSF = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b & a \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & b & a & b \\ 0 & 0 & 0 & 0 & 0 & 0 & b & a & b & 0 \\ 0 & 0 & 0 & 0 & 0 & b & a & b & 0 & 0 \\ 0 & 0 & 0 & 0 & b & a & b & 0 & 0 & 0 \\ 0 & 0 & 0 & b & a & b & 0 & 0 & 0 & 0 \\ 0 & 0 & b & a & b & 0 & 0 & 0 & 0 & 0 \\ 0 & b & a & b & 0 & 0 & 0 & 0 & 0 & 0 \\ b & a & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{aligned} a &= 0.059824 \\ b &= 0.017522 \\ c &= 0.055572 \end{aligned}$$

Figura 2.28: Lena original.

Figura 2.29: Lena após filtro Motion Blur de 15 pixels e 45°.

No exemplo 2.7 e 2.8 podemos notar que a distorção é numa determinada direção do plano da imagem, isso vai depender de um dos parâmetros da distorção (ângulo).

2.6.2 Gaussian Blur (Distorção Gaussiana)

Este tipo de distorção é muito usado, ocasiona redução de níveis de detalhe nas imagens e pode servir para diminuir o ruído das mesmas. O efeito visual desta distorção é semelhante ao suavizado numa imagem vista através de uma tela translúcida, distintamente do efeito *bokeh* [10] produzido por uma lente do fora de foco.

Matematicamente falando, aplicar uma distorção gaussiana numa imagem equivale a convoluir esta imagem com uma distribuição gaussiana ou normal. Esta convolução tem o efeito de filtro passa-baixas.

Para gerar este efeito aplicamos a distribuição gaussiana em duas dimensões representada na seguinte equação.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)} \qquad \textbf{(2.4)}$$

Onde σ é o desvio padrão da distribuição gaussiana e x e y são as coordenadas do *pixel* na matriz imagem. Esta formula aplicada na matriz imagem produz uma superfície cujos contornos são círculos concêntricos com uma distribuição gaussiana do centro para fora.

A seguir mostramos um exemplo onde aplicou-se uma distorção gaussiana com um raio de 4 *pixels* e um σ igual a 2.

$$PSF = \begin{bmatrix} g & j & i & d & i & j & g \\ j & f & h & c & h & f & j \\ i & h & e & b & e & h & i \\ d & c & b & a & b & c & d \\ i & h & e & b & e & h & i \\ j & f & h & c & h & f & j \\ g & j & i & d & i & j & g \end{bmatrix}$$

a= 0.0467 g= 0.0049

b= 0.0412 h= 0.0250

c= 0.0283 i= 0.0134

d= 0.0152 j= 0.0092

e= 0.0364

f= 0.0172

Figura 2.30: Lena original.

Figura 2.31: Lena após filtro gaussiano 4 pixels de raio e σ = 2.

2.6.3 Distorção Toeplitz

A matriz Toeplitz é uma matriz onde todo elemento que se situa na mesma diagonal possui o mesmo valor. Nas figuras 2.32 e 2.33 temos alguns exemplos de matriz de Toeplitz.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

Figura 2.32: Matriz Toeplitz com valores crescentes.

$$\begin{bmatrix} 1 & 1/n & 1/n^2 & 1/n^3 & 1/n^4 & 1/n^5 & \cdots & 1/n^n \\ 1/n & 1 & 1/n & 1/n^2 & 1/n^3 & 1/n^4 & \cdots & \vdots \\ 1/n^2 & 1/n & 1 & 1/n & 1/n^2 & 1/n^3 & \cdots & 1/n^5 \\ 1/n^3 & 1/n^2 & 1/n & 1 & 1/n & 1/n^2 & \cdots & 1/n^4 \\ 1/n^4 & 1/n^3 & 1/n^2 & 1/n & 1 & 1/n & \cdots & 1/n^3 \\ 1/n^5 & 1/n^4 & 1/n^3 & 1/n^2 & 1/n & 1 & \cdots & 1/n^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 1/n \\ 1/n^n & \cdots & 1/n^5 & 1/n^4 & 1/n^3 & 1/n^2 & 1/n & 1 \end{bmatrix}$$

Figura 2.33: Matriz Toeplitz com valores decrescentes.

Se a nossa matriz H da equação *(2.3)* for representada por uma matriz Toepliz podemos ter uma distorção similar a distorção por movimento (*Motion Blur*) com a diferença que a medida

que os pixels vizinhos de uma mesma linha vão se afastando o peso que eles têm na distorção do elemento central diminui. Neste tipo de distorção usou-se o modelo da figura 2.33, neste modelo existe um parâmetro que é dado para poder aplicar a distorção, é o valor de *n*. A seguir mostramos alguns exemplos deste tipo de distorção.

Na figura 2.35 podemos ver como o efeito desta distorção se assemelha à distorção *motion blur*.

Figura 2.34: Imagem da Lena sem degradação.

Figura 2.35: Imagem da Lena após a filtragem (matriz Toeplitz com n = 1,2).

2.7 MÉTODOS DIRETOS PARA RESOLUÇÃO DE SISTEMAS LINEARES

Estes métodos são os que após um número finito de etapas nos dão a solução exata do problema. Esta solução é exata se assumimos a não existência de erros de erredondamento [21].

Nesta cessão apresentaremos os principais métodos numéricos diretos para a resolução de sistemas lineares do tipo $Ax=b$ semelhante ao sistema proposto na equação (*2.3*). Neste tipo de sistemas assumimos que $A \in \mathbb{R}^{MN}$ onde $M \leq N$. Os métodos diretos nos dão uma resposta exata do sistema e com esses métodos é possível determinar, a priori, o tempo máximo gasto para resolver um sistema, uma vez que sua complexidade é conhecida. $\mathbb{R}$

2.7.1 Método de eliminação de Gauss

O método de Gauss consiste em aplicar transformações elementares sobre as equações do sistema $Ax = b$ até que, depois de $(M - 1)$ passos se obtenha um sistema triangular superior do tipo $Ux = c$, após a obtenção deste novo sistema a resposta é obtida por substituições retroativas (*back substitution*).

$Ax = b \qquad A \in \mathbb{R}^{MN}$

$A = LU$

Após (M-1) passos: $Ux = c$

A matriz L é uma matriz triangular inferior com a diagonal principal igual a um e a matriz U é uma matriz triangular superior. A matriz A também pode precisar de permutações para chegar ao modelo LU.

$PA=LU$

2.7.2 Métodos de Fatorização

No método de eliminação de Gauss vimos que quando não há permutações, podemos obter a fatorização da matriz A da forma $A=LU$, onde U seria a matriz triangular superior obtida ao final da fatorização e L a matriz triangular inferior com diagonal unitária, cujos elementos representariam os multiplicadores m_{ij}.

Método de Doolittle

Neste método pretendemos obter as matrizes L e U tal que $A=LU$

$$A = \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & 0 \\ l_{n1} & \cdots & l_{nn-1} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_{11} & \cdots & \cdots & u_{1n} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix}}_{U}$$

Depois, efetuando o produto, podemos obter as formulas correspondentes ao método de Doolittle:

Passo 1: $u_{1j} = a_{1j} \qquad (j=1,\ldots,n)$

$l_{i1} = a_{i1} / u_{11} \quad (i=2,\ldots,n)$

Passo k: $u_{kj} = a_{kj} - \sum_{r=1}^{k-1} l_{kr} u_{rj} \qquad (j=k,\ldots,n)$

$l_{ik} = (a_{ik} - \sum_{r=1}^{k-1} l_{ir} u_{rk}) / u_{kk} \quad (i=k+1,\ldots,n)$

Para fatorizar a matriz A usando este método é usado o mesmo número de operações que no método de eliminação de Gauss. Mesmo assim há uma grande vantagem neste método que é o melhor uso de memória para armazenar os elementos da matriz, já que não são sucessivamente alterados como no método de Gauss.

Método de Cholesky

Este método só pode ser aplicado quando a matriz A é simétrica ($A=A^T$) e positiva definida. Na prática para verificar se a matriz é positiva definida gastaríamos mais tempo do que resolvendo este sistema. Por isto só aplicamos este método quando sabemos a priori se a matriz A é positiva definida e simetrica.

Para estes tipos de matrizes é valida a afirmação: $A=LL^T$, e o método consiste nos seguintes passos:

Passo 1: $l_{11} = \sqrt{a_{11}}$

$l_{i1} = a_{i1} / l_{11}$ para i=2,…,n

Passo k: $l_{kk} = \sqrt{a_{kk} - \sum_{m=1}^{k-1} (l_{km})^2}$

$$l_{ik} = \frac{(a_{ik} - \sum_{m=1}^{k-1} l_{im} l_{km})}{l_{kk}} \qquad \text{para } i=k+1,\ldots,n$$

2.8 MÉTODOS ITERATIVOS PARA RESOLUÇÃO DE SISTEMAS LINEARES

Estes métodos nos dão uma resposta aproximada do sistema de equações. A precisão destas respostas depende do número de iterações que são efetuadas. Quando o número de iterações tende ao infinito a resposta do sistema será exata. Este tipo de métodos são mais simples de serem implementados computacionalmente que os métodos diretos e são facilmente paralelizaveis, é por este motivo que a implementação deste projeto foi feita usando estes métodos.

2.8.1 Método de Jacobi

Embora o método de Jacobi não seja viável para muitos problemas este método proporciona um conveniente ponto de partida para a discussão de métodos iterativos [3].

Sendo A uma matriz não singular ($N \times N$) e

$$Ax = b \tag{2.5}$$

o sistema pode ser resolvido. Assumiremos que os elementos da diagonal principal, a_{ii}, de A são diferentes de zero. O método de Jacobi pode ser representado como:

$$x_j^{k+1} = \frac{1}{a_{ii}}\left(- \sum_{j \neq i} a_{ij} x_j^k + b_i \right), \qquad i=1,...,n \quad k=0,1,... \tag{2.6}$$

onde os superíndices indicam o número da iteração que eles assumem, como se faz em todo método iterativo, a seqüência $x_1^0, x_2^0, ..., x_n^0$ representa o chute inicial.

Para poder entender melhor este algoritmo vamos considerar que a matriz A pode ser decomposta da seguinte maneira:

$$A = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix} + \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} \\ a_{21} & 0 & a_{23} & a_{24} \\ a_{31} & a_{32} & 0 & a_{34} \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix}$$

isto é, $A = D + B$, sendo D uma matriz diagonal. Podemos substituir agora a nova representação de A em *(2.5)* deixando a equação da seguinte forma:

$$\begin{aligned} &(D + B)x = b \\ &Dx + Bx = b \\ &Dx = b - Bx \\ &x' = \frac{1}{a_{ii}}(b - Bx) \end{aligned}$$

O método de Jacobi não é sempre convergente, existem dois teoremas de convergência que serão mostrados a seguir.

Teorema 2.1. Se A é estritamente diagonal dominante, então as iterações de Jacobi sempre convergirão para qualquer valor inicial de $\boldsymbol{x}^0$.

Teorema 2.2. Se $A = D + B$ é positiva definida, então as iterações de Jacobi convergirão para qualquer valor inicial de $\boldsymbol{x}^0$ se e somente se $(D - B)$ é positivo definido.

Teste de convergência

Em cada iteração há um teste necessário que se faz para verificar a convergência do método. Se $\boldsymbol{x}^k$ é a seqüência de uma iteração, dois testes comuns são:

$$\|x^{k+1} - x^k\| \leq \varepsilon \quad \text{ou} \quad \|x^{k+1} - x^k\| \leq \varepsilon \|x^k\| \tag{2.7}$$

2.8.2 Iterações de Gauss-Seider e SOR

Consideremos outro método iterativo para resolver o sistema $Ax = b$, onde assumiremos novamente que os elementos da diagonal principal de *A* são diferentes que zero. Após calcular a nova iteração de Jacobi para x_1,

$$x_1^{k+1} = \frac{1}{a_{11}}\left(- \sum_{j=2}^{n} a_{1j} x_j^k + b_1 \right)$$

é razoável usar este valor atualizado em lugar de x_1^k no cálculo dos subseqüentes x_i^{k+1}. O principio de Gauss-Seider [3] é usar nova informação tão rápido como esta esteja disponível e a iteração de Gauss-Seider é dada por

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(- \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k + b_i \right), \qquad \textbf{(2.8)}$$

$$i = 1,...,n, \qquad k = 0,1,...$$

Façamos

$$A = D - L - U \qquad \textbf{(2.9)}$$

onde *D* é a diagonal de *A* e -*L* e -*U* são as partes estritamente diagonal superior e inferior de *A*. Então podemos escrever *(2.9)* como

$$Dx^{k+1} = Lx^{k+1} + Ux^k + b$$

ou

$$x^{k+1} = Hx^{k+1} + d, \quad k = 0,1,..., \quad H = (D - L)^{-1}U, \quad d = (D - L)^{-1}b \qquad \textbf{(2.10)}$$

Desde que *D-L* é a parte triangular inferior de *A*, sua inversa existe pela assunção de que os elementos da diagonal de *A* são diferentes que zero.

Uma importante modificação na iteração de Gauss-Seider é dada a seguir. Deixemos $\hat{x}_i^{k+1}$ representar o lado direito de *(2.8)*, a iteração Gauss-Seider é definida como:

$$x_i^{k+1} = x_i^k + \omega(\hat{x}_i^{k+1} - x_i^k) \qquad \textbf{(2.11)}$$

Substituindo a representação de $\hat{x}_i^{k+1}$ em *(2.11)* e rearranjando os termos mostrados, x_i^{k+1} satisfaz

$$a_{ii}x_i^{k+1} + \omega \sum_{j<i} a_{ij}x_j^{k+1} = (1-\omega)a_{ii}x_i^k - \omega \sum_{j>i} a_{ij}x_j^k + \omega b_i \qquad \textbf{(2.12)}$$

ou, em termos de matrizes de *(2.9)*,

$$x^{k+1} = (D - \omega L)^{-1}[(1-\omega)D + \omega U]x^k + \omega(D - \omega L)^{-1}b, \qquad k = 0,1,... \qquad \textbf{(2.13)}$$

Isto define a iteração de sucessiva de sobre relaxação (SOR). Note que se $\omega = 1$, *(2.13)* se reduziria à iteração de Gauss-Seider. Note também que $D - \omega L$ é não singular pela assunção dos elementos da diagonal de *A*.

A seguir mostraremos um teorema que garante a convergência das iterações de Gauss-Seider.

Teorema 2.3. Se *A* é estrita ou irredutivelmente diagonal dominante, ou se *A* é uma matriz M, então as iterações de Gauss-Seider convergem para qualquer valor inicial $\boldsymbol{x^0}$**.**

2.8.3 Método de Polyak

Este método também serve para resolver um sistema linear na forma $Ax = b$, onde *b* e *x* são vetores *n* x *1* e *A* é uma matriz *n* x *n*, assumimos que num ponto x^k, medimos a variável $y^k = Ax^k - b + \xi^k$, onde $Ax^k - b$ é conhecido , e ξ^k é o ruído randômico. Assumindo que as seguintes afirmações são satisfeitas (os superíndices indicam o número da iteração que eles assumem) [4].

1. A matriz *A* é uma matriz de Hurwitz, i.e., Re(λj) > 0 para todos os autovalores de *A*.

2. O Ruído ξ^k não é correlativo, com $M(\xi^k) = 0$, $M(\xi^k(\xi^k)^T) = S > 0$ ($S > 0$ significa que *S* é positiva definida, e *M* é a média).

Podemos considerar o seguinte método iterativo para achar o ponto $x' = A^{-1}b$:

$$\begin{aligned} x^{k+1} &= x^k - \gamma^k \cdot y^k \qquad & y^k &= Ax^k - b + \xi^k \\ \hat{x}^{k+1} &= \hat{x}^k + \frac{x^k - \hat{x}^k}{k+1} \qquad & k &= 0,1,\ldots \end{aligned} \tag{2.14}$$

Neste ponto o processo para achar x^k é um processo de aproximação estocástico ordinário, e

$\hat{x}^k = \sum_{i=0}^{k-1} \frac{x^k}{k}$, i.e., os pontos $\hat{x}^k$ são valores médios das iterações x_j, $j = 0,1,\ldots,k-1$.

Considerando os valores iniciais $\boldsymbol{x^0}$ e o tamanho de passo γ^k podemos assumir o seguinte.

3. O ponto $\boldsymbol{x^0}$ é também determinístico e arbitrário, ou randômico com $M|x^0 - x'|_2 < \infty$

4. Também

$$\gamma^k \equiv \gamma, \quad 0 < \gamma < \bar{\gamma} = \min_j 2\,\mathrm{Re}(\lambda_j |\lambda_j|^{-2}), \tag{2.15}$$

ou

$$\frac{\gamma^{k}}{\gamma^{k+1}} = 1 + \delta(\gamma^{k}), \quad \gamma^{k} \to \infty \qquad \sum_{k=0}^{\infty} \gamma^{k} = \infty \tag{2.16}$$

A condição *(2.16)* é satisfeita, por exemplo, pela seqüência $\gamma^{k} = \gamma k^{-\alpha}$, $0 < \alpha < 1$, mas não por $\gamma^{k} = \gamma k^{-1}$. Em outras palavras $(\gamma^{k})^{'s}$ decresce mais lentamente que $1/k$.

2.9 ARQUIVOS PGM

O formato PGM é uma denominação pouco comum para *Grayscale File Format*. Este formato foi desenhado para ser extremamente fácil de aprender e aplicar em programas de processamento de imagens. Um arquivo PGM é texto integral e pode ser processado por ferramentas de processamento de texto [11].

Uma imagem PGM representa uma imagem em escala de cinzas. Há muitos pseudoformatos PGM em uso onde tudo é especificado como neste formato com exceção do valor individual de cada *pixel*. Para a maioria dos propósitos, uma imagem PGM pode ser entendida simplesmente como uma matriz de valores inteiros arbitrários, nesse sentido todos os programas que trabalham com imagens em escala de cinza podem ser facilmente usados para processar qualquer outra coisa. O nome PGM é o acrônimo derivado de *Portable Gray Map*.

Uma variante oficial de PGM é a máscara de transparências. Uma máscara de transparência é representada por uma imagem de PGM, com exceção que em lugar de intensidades de *pixel*, há representação de valores de opaco (vide abaixo).

Um arquivo PGM consiste numa seqüência de um ou mais imagens PGM. Não há nenhum dado, delimitadores ou recheio antes, depois ou no meio da imagem.

Cada imagem PGM está formada pelas seguintes partes:

1. Um identificador para reconhecer o tipo de arquivo. O identificador de uma imagem PGM é P5 (P2 no caso de imagens monocromáticas).
2. Os comentários. Estes comentários têm que começar pelo símbolo sustenido (#) e vão até o final de cada linha. Estes comentários serão ignorados.
3. Largura e altura da matriz que representa a imagem (*Width* × *Height*). É usada a formatação ASCII com caracteres decimais.
4. Máximo valor de brilho (Maxval). Também em ASCII decimal e menor que 65536.

5. Tom de cinza para cada *pixel*. Os elementos de cada linha são separados por um espaço em branco. Cada valor de cinza é um número proporcional à intensidade de cada *pixel* e têm que estar no intervalo de zero a Maxval (zero representa o valor de preto total).

F ***i-***

```
P2
# imagem.pgm
# autor: Jaime G. Vela
19 7
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0  0  0  0  0  0
0 5 5 5 5 0 0 9 9 9 9 0 0 15 0  0  0  15 0
0 5 0 0 5 0 0 9 0 0 0 0 0 15 15 0  15 15 0
0 5 5 5 5 0 0 9 0 9 9 0 0 15 0  15 0  15 0
0 5 0 0 0 0 0 9 0 0 9 0 0 15 0  0  0  15 0
0 5 0 0 0 0 0 9 9 9 9 0 0 15 0  0  0  15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0  0  0  0  0  0
```

IDENTIFICADOR

COMENTARIOS

LARGURA E ALTURA

MAXIMO VALOR DE BRILHO

TOM DE CINZA PARA CADA PIXEL

gura 2.36: Exemplo de arquivo PGM.

Figura 2.37: Visualização do exemplo (Fig. 2.36).

2.10 PROCESSAMENTO PARALELO

2.10.1 Motivação e aplicações

O principal motivo da aparição da computação paralela foi a demanda por poder computacional, tanto nas aplicações comercias quanto nas aplicações científicas.

Pelo lado das aplicações comerciais podemos mencionar os bancos, seguradoras e sistemas industriais cuja demanda por acesso intensivo a dados, processamento de transações e alta disponibilidade foram aumentando criticamente.

Assim como as aplicações comerciais, as aplicações científicas foram exigindo um maior uso de sistemas eficientes e de alto desempenho para poder simular ou modelar processos cada vez mais complexos e pesados desde o ponto de vista computacional.

Computação paralela tem sido tradicionalmente aplicada com muito sucesso no desenho de superfícies aerodinâmicas, motores de combustão interna, circuitos de alta velocidade e desenho de estruturas, entre outras. Mais recentemente, desenho de sistemas microeletromecânicos e nanoeletromecânicos (MEMS e NEMS) tem atraído significativamente a atenção. Enquanto muitas aplicações em engenharia e desenho propõem problemas de múltipla escala temporal e espacial e junto com fenômenos físicos. Podem-se tratar estes problemas como uma mistura de fenômenos quânticos, dinâmica molecular, modelos contínuos e estocásticos de processos físicos como condução, convecção, radiação, e estruturas mecânicas, tudo em um único sistema. Isto representa um formidável desafio para a modelagem geométrica, modelagem matemática e desenvolvimento de algoritmos, tudo no contexto de processamento paralelo [6].

Entre as mais relevantes aplicações podemos mencionar aplicações científicas, aplicações comerciais e aplicações em sistemas computacionais.

2.10.2 Organização física de plataformas paralelas

A visão da lógica tradicional sobre computadores seqüenciais consiste em uma memória conectada a um processador via um caminho ou barramento de dados, estes três componentes, processador, memória e barramento, ocasionam “gargalos” ao longo do processo do sistema computacional. Um número de inovações arquiteturais através dos anos tem se endereçado nestes gargalos. Uma das mais importantes inovações é a multiplicidade em unidades de processamento, caminhos de dados e unidades de memória. Esta multiplicidade é completamente oculta para o programador, como no caso de paralelismo implícito, ou exibida só parcialmente [6].

2.10.3 Arquitetura de um computador paralelo ideal

Uma extensão natural do modelo serial de computação (*Random Access Machine* ou RAM) consiste em *p* processadores e uma memória central de tamanho ilimitado, esta sería uniformemente acessível a todos os processadores. Todos os processadores acessam os mesmos endereços de espaço. Os processadores compartilham o mesmo *clock*, mas conseguem executar diferentes instruções em cada ciclo. Este modelo ideal também é conhecido como maquina de acesso randômico paralelo (PRAM). Desde que as PRAMs permitam acesso concorrente a vários locais de memória, dependendo de como será tratado o acesso simultâneo de memória, as PRAMs podem ser dividas em quatro subclasses.

1. Leitura exclusiva, escrita exclusiva (**EREW**). Nesta classe o acesso à memória é exclusivo. Nenhuma operação concorrente de leitura nem escrita são permitidas.
2. Leitura concorrente, escrita exclusiva (**CREW**). Nessa classe, múltiplos acessos a leitura de memória são permitidos, mas não na escrita.
3. Leitura exclusiva, escrita concorrente (**ERCW**). Múltiplos acessos de escrita de memória são permitidos, mas a leitura é serializada.
4. Leitura concorrente, escrita concorrente (**CRCW**). Nesta classe múltiplos acessos de leitura e escrita na memória são permitidos. Este é o mais poderoso modelo de PRAM.

2.10.4 Redes de interconexão para computadores paralelos

Redes de interconexão provem mecanismos para transmissão de dados entre os módulos de processamento ou entre os processadores e os módulos de memória. Vistos como caixas pretas, a redes podem ser consideradas como *n* entradas e *m* saídas. Redes de interconexão são construídas basicamente usando *links* e *switches*. Um *link* corresponde à mídia física usada para transmitir os dados.

Redes de interconexão podem ser classificadas como estáticos ou dinâmicos. Redes estáticas consistem em comunicação ponto a ponto entre módulos de processamento e são chamadas também redes diretas. Redes dinâmicas por outro lado são construídas usando *switches* e *links* de comunicação. Os *links* de comunicação são conectados uns a outros dinamicamente mediante os *switches* para estabelecer caminhos entre módulos de processamento e bancos de memória. Redes dinâmicas são chamadas também redes indiretas [6].

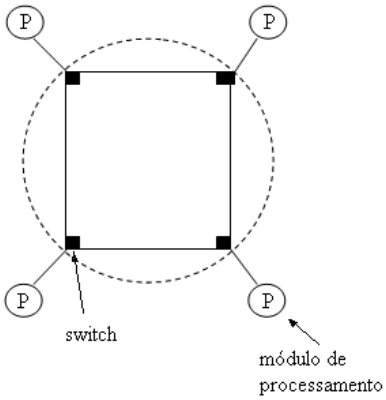

Figura 2.38: Rede direta.

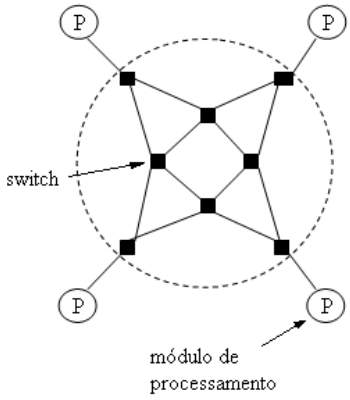

Figura 2.39: Rede indireta.

2.10.5 Topologias de redes

Uma ampla variedade de topologias de rede tem sido usada em redes de interconexão. Estas topologias tentam conciliar custos e escalabilidade com desempenho. Enquanto topologias puras têm propriedades matemáticas atraentes, na prática redes de interconexão tendem a ser combinações ou modificações destas topologias puras que mostraremos a seguir.

Rede de bus compartilhado

A rede de bus de dados é provavelmente o tipo de rede mais simples, consiste em um meio compartilhado comum para todos os nodos.

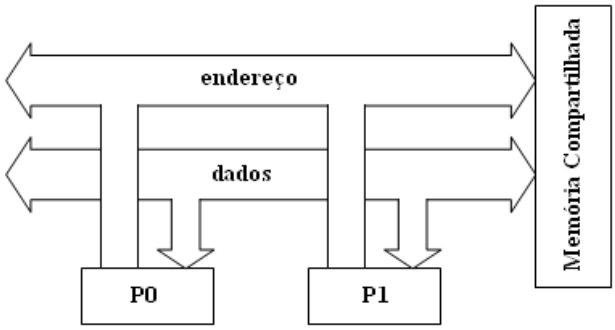

Figura 2.40: Rede de bus compartilhado.

Rede Crossbar

Uma maneira simples de conectar *p* processadores com *b* bancos de memória é usando uma rede crossbar. Esta rede utiliza um grupo de chaves ou nodos de chaveamento como mostrado na figura 2.41.

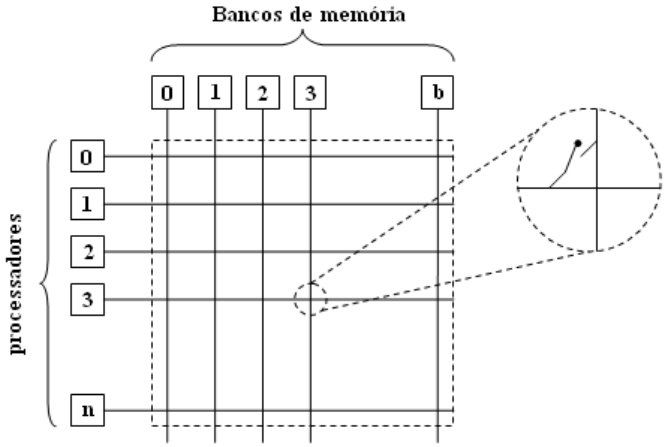

Figura 2.41: Rede crossbar não bloqueante

Rede multiestágio

A rede de interconexão *crossbar* é escalavel em termos de performance, mas não em termos de custo. Contrariamente a rede de bus compartilhado é escalavel em termos de custo, mas não em termos de performance. Uma classe intermediaria de rede chamada rede multiestágio de interconexão consegui-se encaixar entre estes dois extremos.

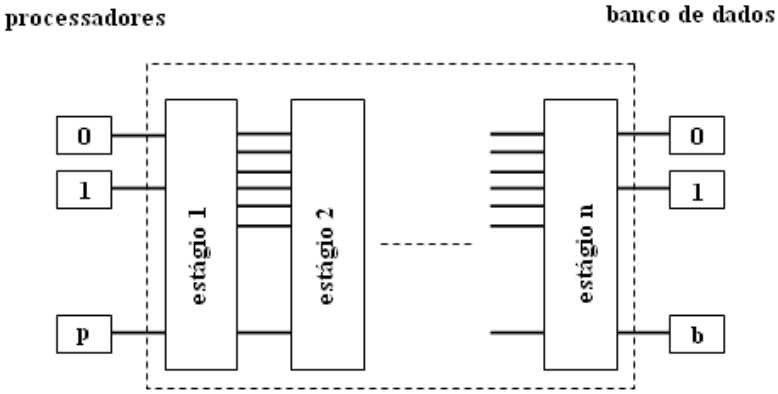

Figura 2.42: Rede multiestágio.

2.10.6 Custos de comunicação em maquinas paralelas.

Um dos maiores atrasos na execução de programas paralelos vem da comunicação entre os elementos de processamento [6].

O tempo que toma a comunicação de uma mensagem entre dois nodos de processamento numa rede é a soma do tempo para preparar a mensagem para a transmissão e o tempo que

leva a mensagem em atravessar a rede até o destino. Os principais parâmetros que determinam a latência de comunicação são:

Startup time (t_s): Este *startup* time é o tempo requerido para tratar a informação tanto no envio quanto na recepção.

Per-hop time (t_h): Este é o tempo que leva o cabeçalho da mensagem para percorrer o caminho entre os dois nodos.

Per-word transfer time (t_w): Se a banda do canal é de r palavras por segundo, então cada palavra levará $t_w=1/r$ segundos para atravessar o canal.

2.10.7 Medidas de desempenho e análise

Estas medidas são importantes para o estudo da performance dos programas paralelos tentando determinar os melhores algoritmos, avaliando performance de plataformas de *hardware* e analisando os benefícios do paralelismo [6].

Tempo de execução

O tempo serial de execução de um programa é o tempo transcorrido entre o começo e o final da sua execução em um computador seqüencial. O tempo de execução paralelo é o tempo transcorrido entre o momento que o programa paralelo começou até o momento que o último dos elementos termina sua execução. O tempo seqüencial é denotado por Ts e o tempo paralelo é denotado por Tp.

Speedup

Quando avaliamos um sistema paralelo, o interesse mais freqüente é conhecer o ganho de performance ocasionado pela paralelização de um determinado processo paralelo. *Speedup* é uma medida que mostra o benefício relativo de resolver um problema em paralelo. O *speedup* é definido como a razão do tempo que leva resolver um problema numa simples unidade de processamento e o tempo que leva resolver este mesmo problema num computador paralelo com p elementos de processamento idênticos.

$$S = \frac{T_p}{T_s}$$

Eficiência

Somente um sistema paralelo ideal que contem p unidades de processamento pode obter um *speedup* igual a p. Na prática, um comportamento ideal não é alcançado porque enquanto o algoritmo paralelo é executado, as unidades de processamento não conseguem operar no 100% da capacidade o tempo todo. Eficiência é uma medida da fração de tempo que cada unidade de processamento é totalmente utilizada; este é definido como a razão do *speedup* e o número de elementos de processamento.

$$E = \frac{S}{p}$$

3 ANÁLISE DA SOLUÇÃO

3.1 REPRESENTAÇÃO DIGITAL DA IMAGEM

Para poder trabalhar com as imagens monocromáticas que vamos restaurar, usaremos o formato PGM que nos permite o tratamento da imagem como uma matriz de valores. A vantagem deste formato é que não existe diferença entre a representação matemática da imagem (matriz de valores) e o formato do arquivo da imagem (PGM).

O valor de cada *pixel* da imagem será representado por dois bytes (16 bits), isto garante uma gama de valores de cinza que vai de 0 ate 65536.

Figura 3.1: Representação discreta da cada pixel da imagem

3.2 PROCESSO DE DEGRADAÇÃO E RESTAURAÇÃO

A imagem que vamos tratar passa por vários processos. Inicialmente temos uma captura da imagem por meio de um equipamento digital ou analógico (câmera). No momento da captura existe um meio entre a câmera e o objeto, este meio é representado por *H*.

O meio (*H*) é representado matematicamente por uma matriz bidimensional cuja dimensão vai depender da dimensão da matriz que representa a imagem (*f*).

Na figura 3.2 podemos acompanhar o processo completo, desde a degradação da imagem até a restauração.

Para começar temos a representação matemática da imagem original (*f*). Esta representação é feita mediante um rearranjo lexicográfico (ver capitulo 2.5) da matriz que representa tal imagem.

Continuando com o processo, se a dimensão do nosso vetor imagem *f* fosse ($N^2 \times 1$) temos a matriz de degradação *H* quadrada de dimensão($N^2 \times N^2$), que pode representar qualquer tipo de degradação na imagem.

O vetor da imagem (*f*) passa por esta degradação e se transforma no vetor *g* com a mesma dimensão do vetor *f*. Matematicamente esta passagem pode ser representada como uma multiplicação entre a matriz *H* e o vetor *f*.

Neste ponto temos uma nova imagem representada por *g* e vai sofrer o efeito da adição de um ruído externo que é representado na figura 3.2 como *r*.

Este ruído (*r*) vai ser inserido na nova imagem *g* mediante a soma de valores aleatórios, geralmente muito menores do que os valores da matriz original da imagem.

Depois da adição do ruído, temos uma imagem com valores diferentes aos da imagem original, e é neste ponto que começa o processo de restauração.

O nosso processo de restauração pode ser dividido em duas partes. A primeira parte é encarregada de eliminar ou diminuir o ruído aditivo da imagem e é chamada de pré-filtragem, neste caso é usado um filtro de mediana. Devido às propriedades do ruído aditivo, o uso deste filtro é mais vantajoso.

Uma vez que a imagem passou pela primeira fase (pré-filtragem), podemos passar à fase mais importante da restauração que é a aplicação de algoritmos iterativos.

Nesta fase temos duas variantes, já que podemos usar dos algoritmos diferentes para tentar recuperar a imagem. Um dos algoritmos usados é o algoritmo de Jacobi [3] e o outro é o algoritmo de Polyak [4].

Depois de atingir certo grau de melhoria da imagem (pré-definido) podemos ter uma imagem representada pelo vetor *f'*. Este novo vetor (*f'*) vai ser o produto final deste processo de restauração.

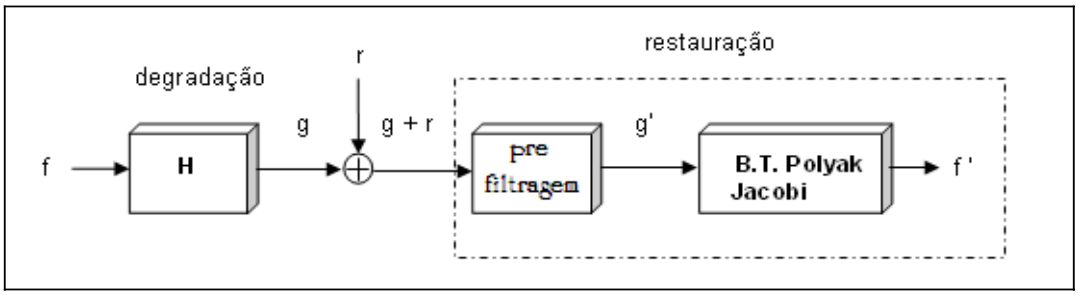

Figura 3.2: Diagrama do processo de distorção e recuperação da imagem.

3.3 PROCESSO DE DEGRADAÇÃO DA IMAGEM

As imagens tratadas neste trabalho foram previamente modificadas simulando algum tipo de distorção já seja por embaçamento ou por movimento. Alem de distorcer estas imagens aplicou-se diversos níveis de ruído aditivo.

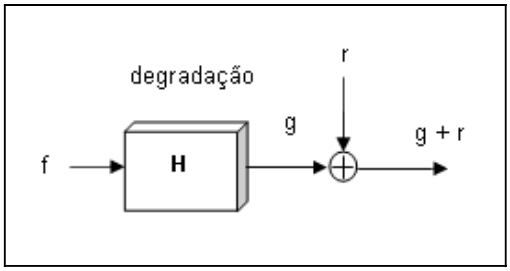

Figura 3.3: Diagrama do processo de distorção da imagem.

3.4 PROCESSO DE RESTAURAÇÃO DA IMAGEM

O nosso processo de restauração está dividido em duas etapas. A primeira etapa elimina o ruído aditivo da imagem e a segunda etapa aplica um algoritmo iterativo (Jacobi ou Polyak) para poder restaurar a imagem.

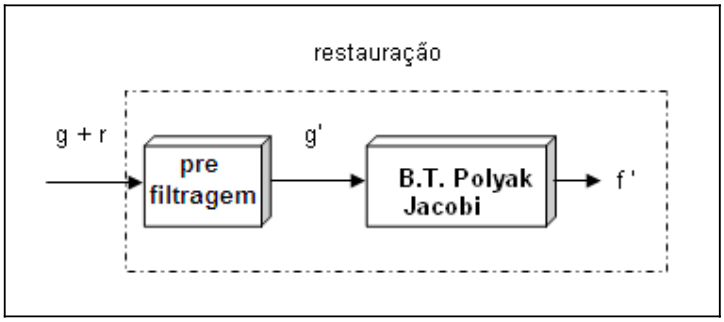

Figura 3.4: Diagrama do processo de restauração.

3.5 PRÉ-FILTRAGEM

No presente trabalho optamos por usar um filtro de mediana modificado de dimensões (3×3) e (5×5) para poder eliminar total ou parcialmente o ruído aditivo. Este filtro é aplicado na imagem inteira *pixel* por *pixel* e quando os pixels vizinhos necessários para filtrar um elemento estiverem fora da imagem estes são substituídos pelo elemento a ser filtrado. Por este motivo a filtro usado mostra algumas imperfeições nas bordas da imagem.

A seguir mostramos um exemplo de imagem filtrada com o filtro de mediana (3×3).

Figura 3.5: Imagem com distorção Motion Blur de 21 pixels e 45° e 6% de ruído.

Figura 3.6: Imagem da Lena após a filtragem (filtro de Mediana 3x3).

Nas figuras acima (figura 3.5 e 3.6) podemos ver a remoção de ruído na imagem da Lena, neste caso o filtro teve um resultado satisfatório, deixando só alguns vestígios do ruído aditivo nas bordas da imagem, isto se deve a que o nosso filtro de mediana não tem boa resposta nas bordas da imagem.

3.6 MÉTODO ITERATIVO DE RESTAURAÇÃO

Nesta fase do processo temos a imagem total ou parcialmente livre de ruído aditivo e já podemos aplicar os algoritmos iterativos para livrar a imagem dos efeitos da distorção no momento da captura da mesma. No presente trabalho foram aplicadas distorções que inviabilizam o uso do algoritmo de Jacobi (ver capitulo 2.8.1) por isso optamos por usar o algoritmo de Polyak (ver capitulo 2.8.3).

A seguir mostramos o diagrama de atividades do algoritmo seqüencial usado assim como uma breve descrição dos principais pontos, para um entendimento mais detalhado destes algo-

ritmos pode-se ver o apêndice B onde se faz uma descrição detalhada sobre cada ponto do algoritmo.

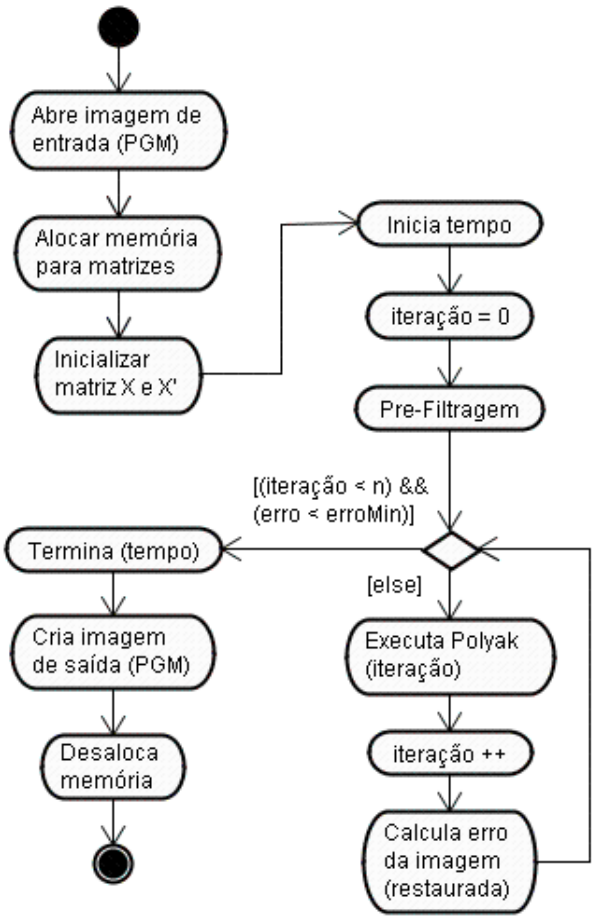

Figura 3.7: Diagrama de atividades do algoritmo seqüencial.

Na figura 3.7 podemos ver o diagrama de atividades do algoritmo usado, este algoritmo tem como objetivo restaurar uma imagem que sofreu algum tipo de distorção por movimentação ou por outro efeito similar e contém ruído aditivo. Para este propósito usamos um algoritmo iterativo (Polyak) até atingir um erro menor que o erro especificado previamente ou alcançar um número de iterações máximo. Neste trabalho o erro usado foi o *Root Mean Square* (RMS).

$$erro = RMS(Hx'(k) - g)$$

3.7 APLICAÇÃO DO ALGORITMO ITERATIVO SEQÜENCIAL

Nesta cessão mostraremos alguns exemplos de restauração usando nosso algoritmo seqüencial. Para obter maiores detalhes sobre o código fonte do programa seqüencial, ver o apêndice C.

Os códigos foram compilados usando o compilador da Intel icc e com nível de otimização 3. O critério usado para escolher este compilador dentre os compiladores disponíveis para Linux foi o menor consumo de tempo do programa compilado. A segui mostramos um gráfico contendo as curvas de tempo *vs* número de iterações do programa para códigos compilados com dois tipos de compiladores e opções de otimização nível 3 (O3).

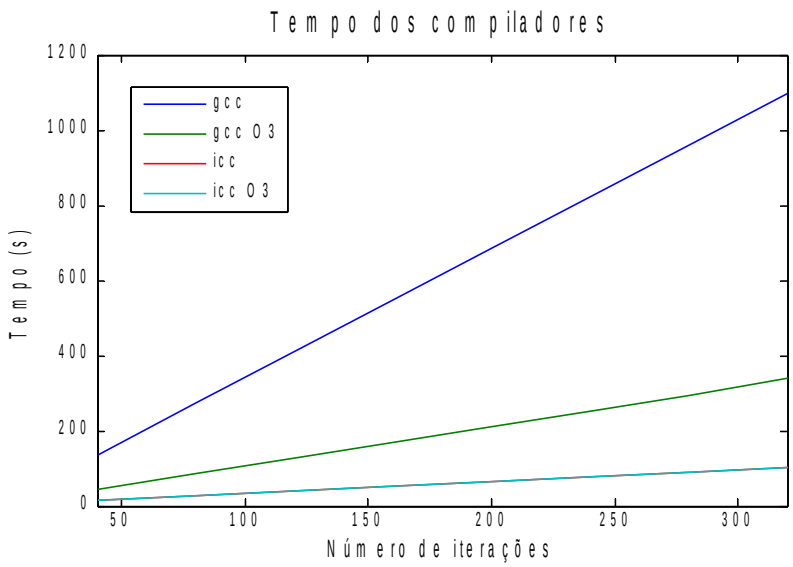

Figura 3.8: Tempo do processo vs número de iterações.

Na figura 3.8 podemos observar a diferença entre as velocidades do programa compilado com gcc (GNU) e icc (Intel), no caso do código compilada com icc não teve diferença entre as opções de compilação com otimização e sem otimização.

Os dados plotados na figura 3.8 foram extraídos das tabelas 1, 2, 3 e 4 no apêndice A.

Teste1. Neste teste usamos uma imagem de resolução 1024×744 e 2 Bytes por *pixel*. Esta imagem sofreu uma distorção tipo Toeplitz com n=1.3 e um ruído aditivo tipo *salt & pepper* em 7 % dos *pixels* (ver capitulo 2.6.3).

O programa foi compilado e rodado num ambiente Linux com processador Intel Itanium2 de 1.5 GHz e 2 GBytes de memória. O compilador usado foi o icc da Intel com nível de otimização 3 (-O3).

Figura 3.8:Imagem original sem degradação.

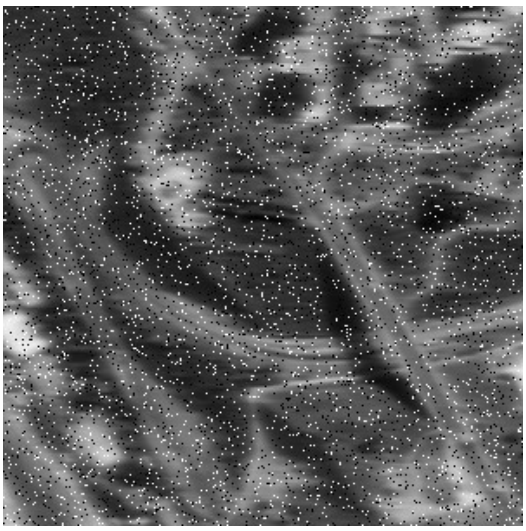

Figura 3.9: Imagem degradada com efeito Toeplitz 1.3 e 7 % de ruído.

Figura 3.10: Imagem recuperada após 40 iterações.

Figura 3.11: Imagem recuperada após 320 iterações.

Nas figuras 3.8, 3.9, 3.10 e 3.11 podemos apreciar só parte da imagem tratada por ser esta de alta resolução. Na imagem da figura 3.11 podemos apreciar uma melhoria significativa comparada co a imagem degradada (figura 3.9).

A seguir mostraremos os resultados numéricos deste teste. Na figura 3.12 temos uma curva que representa o tempo do processo *vs* o número de iterações este aumento no tempo é praticamente linear. Já na figura 3.13 podemos ver o erro (RMS) *vs* a número de iterações, neste caso vemos um decaimento exponencial no erro da imagem.

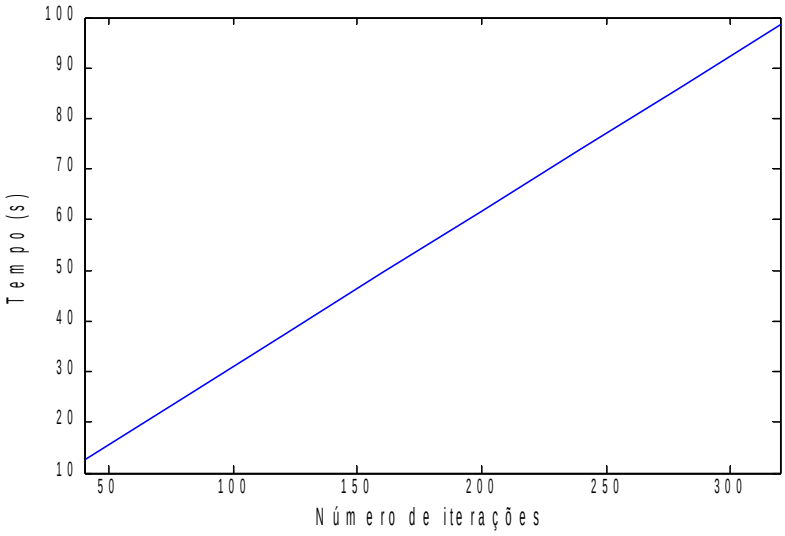

Figura 3.12: Tempo do processo vs número de iterações.

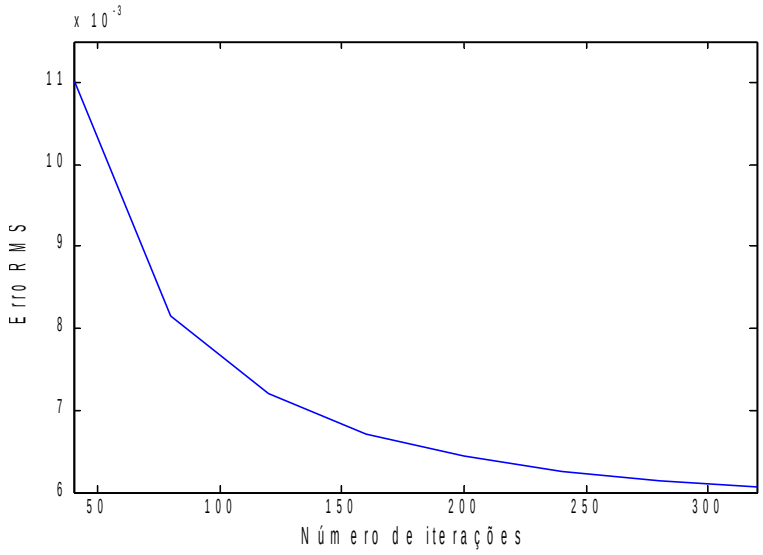

Figura 3.13: Erro do processo vs número de iterações.

Os dados plotados nas figuras 3.12 e 3.13 foram extraídos da tabela 6 no apêndice A.

Teste2. Neste este aplicamos o método de Polyak para restaurar imagens que sofreram degradação tipo *Motion Blur* (ver capitulo 2.6.1) pra diferentes valores de número de *pixels* e ângulo.

Figura 3.14: Lena após distorção Motion Blur de 21 pixels e 45° e ruído aditivo 4%.

Figura 3.15: Imagem degradada após a pré-filtragem (Filtro Mediana Seletivo).

Figura 3.16: Imagem restaurada (Método de Polyak) após 4 iterações.

Figura 3.17: Imagem restaurada (Método de Polyak) após 20 iterações.

Neste teste podemos ver que o método de Polyak não é muito eficiente para esse tipo de distorção, já na figura 3.21 podemos ver alguma melhoria na imagem restaurada após 30 iterações.

Figura 3.18: Lena após distorção Motion Blur de 15 pixels e 45° e ruído aditivo 4%.

Figura 3.19: Imagem degradada após a pré-filtragem (Filtro Mediana Seletivo).

Figura 3.20: Imagem restaurada (Método de Polyak) após 4 iterações.

Figura 3.21: Imagem restaurada (Método de Polyak) após 30 iterações.

4 PROPOSTA DE SOLUÇÃO PARALELA

Nosso algoritmo seqüencial pode ser paralelizado em diversos pontos, esta paralelização permite uma independência de trabalho dos nós nos pontos em que a paralelização é aplicada. A seguir mostramos uma explicação da implementação deste algoritmo paralelo assim como das principais sub-rotinas que compõem este processo.

4.1 ALGORITMO PARALELO

O objetivo deste algoritmo é dividir as tarefas do algoritmo seqüencial em tarefas menores que possam ser feitas de maneira independente e ao mesmo tempo. Estas tarefas não funcionam de maneira independente o tempo todo, tendo que parar para se sincronizar com os outros processos em determinados momentos, mesmo com estas "paradas" o algoritmo consegue ser mais eficiente que o seqüencial.

A seguir são listados os passos do algoritmo paralelo com uma breve descrição de cada um.

1. Determina número de processos (*n*): Neste passo é determinado e registrado o número de nós de processamento com os quais contamos.
2. Determina processo local (*eu*): É determinado e registrado o número do nó que será usado como processo principal.
3. Abre imagem de entrada: Neste passo abrimos o arquivo que contem a imagem a tratar.
4. Alocar memória para matrizes: Alocamos dinamicamente a memória para as matrizes que usamos no programa.
5. Distribui número de linhas da imagem nos *n* processos: Todas as linhas da matriz da imagem são distribuídas nos nós, tentando colocar o mesmo número de linhas em cada nó.
6. Inicia tempo: A contagem do tempo de processamento começa neste ponto.
7. Pré-Filtragem: Aplicamos o filtro de Mediana para poder livrar a imagem a tratar do ruído aditivo.
8. Executa Polyak: Neste ponto começamos a restauração da imagem usando o algoritmo de Polyak.
9. Calcula erro da imagem: O cálculo do erro se faz aplicando o erro meio quadrático (RMS) à diferença entre a imagem restaurada multiplicada pela matriz de distorção e a imagem distorcida que entrou no processo.

   $$erro = RMS(Hx'(k) - g)$$

10. Envia erro (*eu*): Todos os subprocessos enviam os erros calculados.
11. Recebe erros: O subprocesso principal (*eu*) recebe os erros de todos os outros subprocessos.
12. Comparação de erros nos *n* processos: Os erros de todos os subprocessos são comparados e se faz uma estimativa do erro total da imagem.
13. Termina tempo: A contagem do tempo de processamento acaba neste ponto.
14. Envia subMatriz(*eu*): Todos os subprocesso enviam o resultado da restauração das parcelas da matriz imagem que eles restauraram.
15. Recebe todas subMatrizes(*n*): O nó principal (*eu=0*) recebe todos as parcelas dos subprocessos.
16. Monta matriz final: O nó principal (*eu*) monta a imagem completa a partir da informação que cada subprocesso enviou.
17. Cria imagem de saída: Salva os dados da nova imagem restaurada no formato PGM.
18. Desaloca memória: A memória que foi alocada para trabalhar no processo de restauração é desalocada.

A seguir é apresentado o fluxo de atividades que representa o comportamento deste algoritmo.

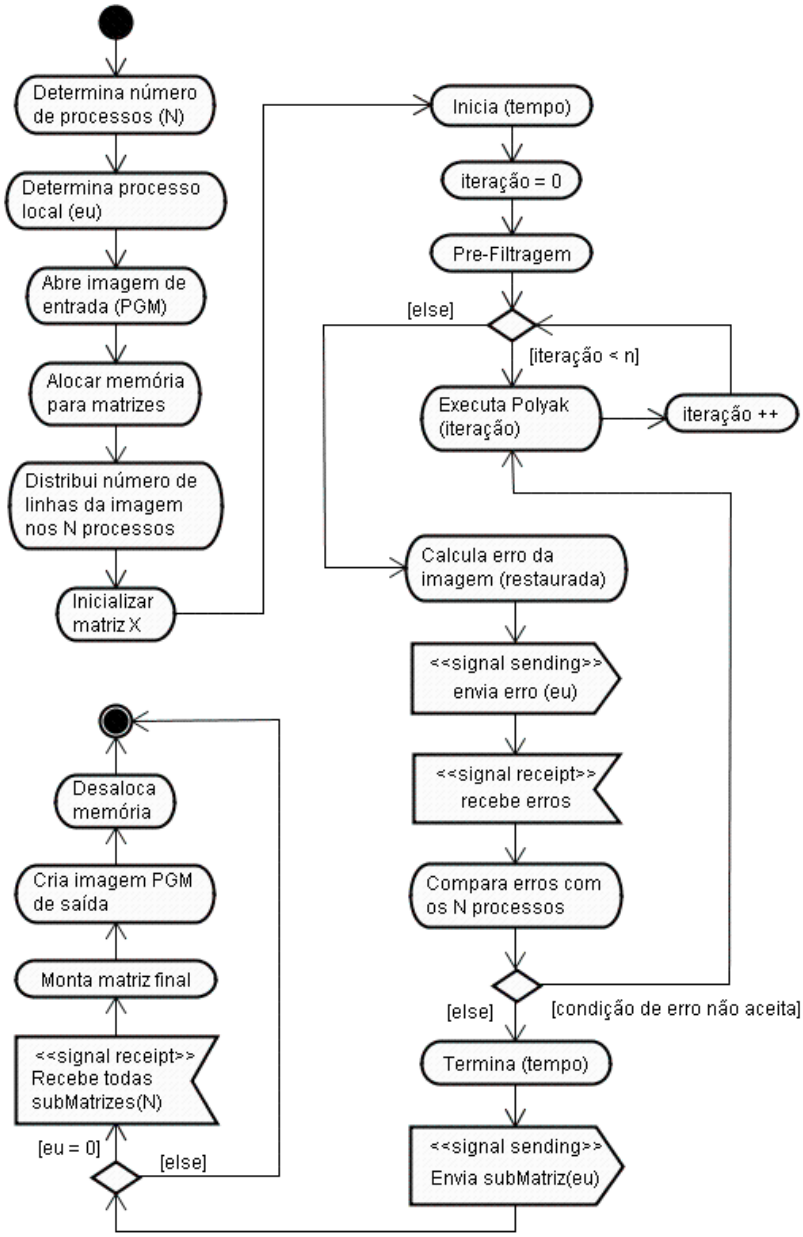

Figura 4.1: Diagrama de atividades do algoritmo paralelo.

4.2 APLICAÇÃO DO ALGORITMO PARALELO

Especificações técnicas da máquina paralela usada.

Este projeto foi implementado e testado no computador paralelo SGI Altix 350 do laboratório NACAD da COPPE-UFRJ.

Esta máquina usa um sistema de memória compartilhada (tecnologia NUMA) com 16 CPUs Intel Itanium 2 (64-bits) de 1.5 GHz de *clock* e 32GB de memória. O sistema operacional é

nativo de 64 bits (Red Hat Enterprice Linux + SGI ProPack) e possui compiladores Intel e GNU (Fortran 90 e C/C++) com suporte OpenMP e MPI.

Para maiores informações ao respeito do ambiente paralelo usado, acessar a página web do NACAD [19] e a pagina oficial da Intel [20].

Testes realizados com o modelo paralelo.

Para testar o modelo paralelo usamos uma imagem monocromática em três resoluções, 700x700, 900x900, 1100x1100 e 1300x1300 com 2 Bytes por *pixel*. Estas imagens foram armazenadas no formato PGM.

O código fonte foi compilado com o compilador icc versão 8.1 e usou-se a opção de otimização de nível três (–O3). O critério na escolha do compilador foi a significativa diferença no tempo de execução do código compilado com o icc comparado com o código compilado com o compilador GNU gcc (vide figura 4.2).

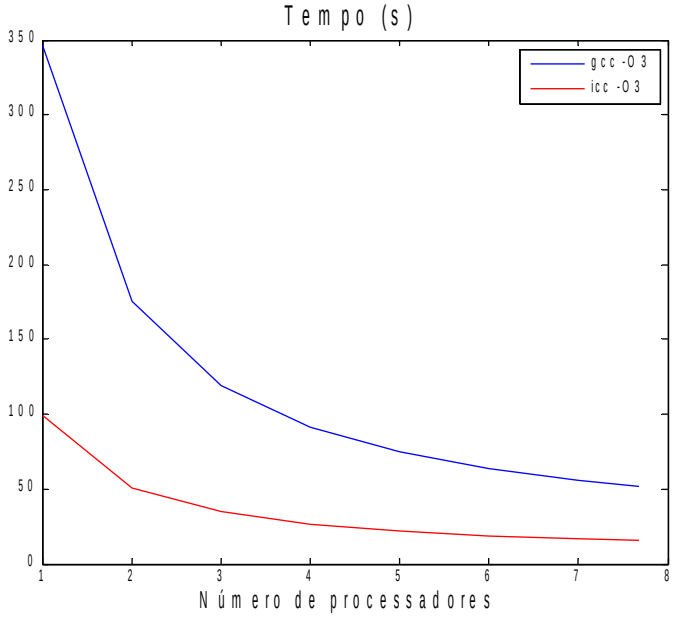

Figura 4.2 Tempos de execução do programa compilado com gcc (GNU) e icc (Intel) para 320 iterações com uma imagem de 744x1024 pixels.

Na figura 4.2 podemos ver o tempo de execução do programa (para 320 iterações) para diferente número de processadores. Pode-se notar uma diferença significativa no tempo do código compilado com icc (Intel) e gcc (GNU), enquanto para um processador o programa com compilação gcc demorou aproximadamente 350 segundos, o programa compilado com icc demorou aproximadamente 100 segundos, isto quer dizer que o programa compilado com icc só

consumiu o 28,5% do tempo do programa equivalente com gcc. Já no caso de 8 processadores a diferença entre o programa compilado com gcc e icc foi de 34,93 segundos, isto quer dizer que o programa compilado com icc consumiu só o 29,7 % do tempo consumido pelo compilado com gcc. Por este motivo os testes de desempenho foram feitos usando o compilador da Intel e a opção de otimização de terceiro nível (O3).
Os dados usados para plotar a figura 4.2 encontram-se na tabela 7 e 8 do apêndice A.

Teste 1

Para este teste usamos uma imagem com distorção tipo Toeplitz com n=1.3 (ver capitulo 2.6.3) e ruído aditivo (*salt & pepper*) em 3% dos *pixels*.
Estas curvas de *speed-up* foram montadas tomando-se como referência o tempo de execução do mesmo modelo só que substituindo-se a parte iterativa por um método direto. O método direto foi implementado usando-se a biblioteca mkl - BLAS da Intel [22], esta biblioteca permite a resolução de sistemas lineares calculando a inversa da matriz distorção (ver capítulo 2.5) aproveitando suas propriedades de esparcidade e simetria. Cada valor dentro da curva de *speed-up* é o resultado da divisão do tempo de execução do modelo serial direto pelo tempo de execução no nosso modelo paralelo para diferente número de processadores. No caso da curva de eficiência paralela *Ep* cada valor é o resultado da divisão do valor de *speed-up* pelo número de processadores usados, a eficiência ideal é representada por uma linha reta paralela ao eixo x na posição 1 no eixo y, o que significa que a eficiência deveria se manter em 100% independente do número de processadores.

A figura 4.3 mostra as curvas de *speed-up* do programa paralelo rodado com imagens de 4 dimensões diferentes (1300x1300, 1100x1000, 900x900 e 700x700) para diferente número de processadores (de 1 a 8). Neste gráfico podemos ver que a maior dimensão da imagem a curva de *speed-up* se aproxima mais à curva ideal. Isto se deve a que para dimensões pequenas a maior parte do tempo é consumida em carregar o arquivo da imagem na memória do processador e na comunicação dos processos, já para dimensões maiores o tempo consumido para carregar o arquivo da imagem e de comunicação se faz menos significativo.

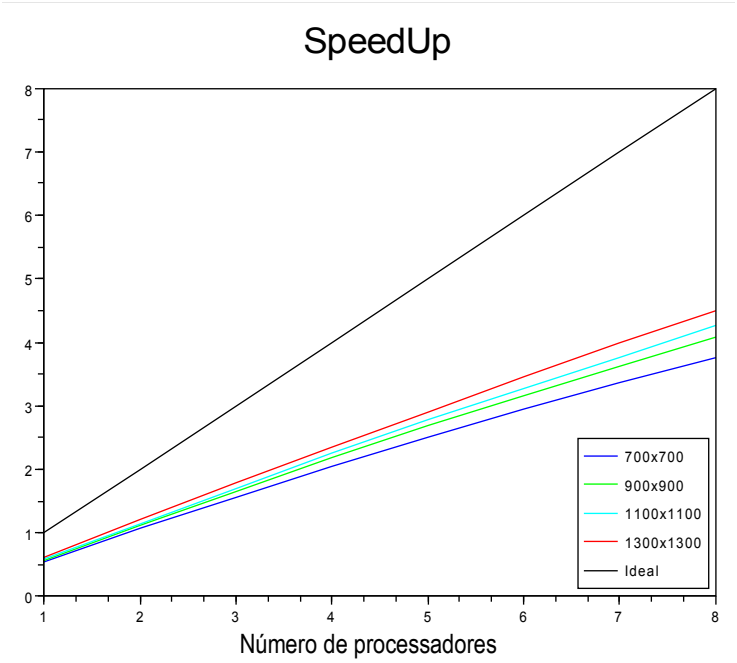

Figura 4.3: Speed-up paralelo.

Na figura 4.4 podemos ver a eficiência paralela (*Ep*) do programa rodado com varias dimensões de imagem, neste caso também se faz evidente o aumento da eficiência com o aumento da dimensão.

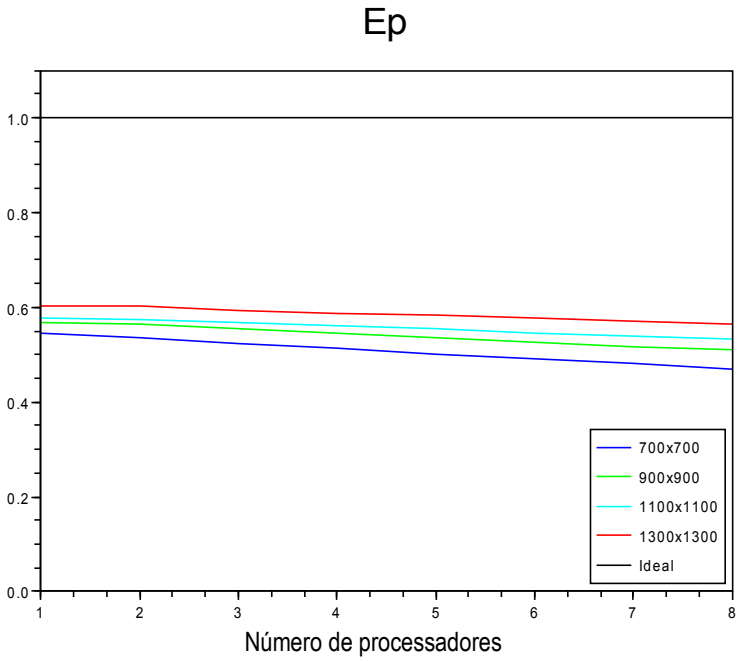

Figura 4.4: Eficiência paralela.

Os dados usados nas figuras 4.3 e 4.4 encontram-se na tabelas 13 e 14 do apêndice A.

Nas figuras 4.5 a 4.7 são mostrados segmentos das imagens usadas nos testes tanto para o método iterativo (Polyak) quanto para o método direto (BLAS - mkl). Em estas imagens podemos ver que o método iterativo comportou-se melhor que o método direto, isso se deve a que o método direto é aplicado após submeter a imagem a uma pré-filtragem (mediana), este filtro altera os valores dos *pixels*, por isso que aplicando a inversa da matriz que gerou a distorção não retorna necessariamente a matriz original. No caso do método iterativo, já que o modelo de Polyak tolera pequenos ruídos na imagem podemos obter uma boa restauração mesmo tendo variação nos valores dos *pixels* originais.

Figura 4.5: Imagem original degradada com efeito Toeplitz 1.3 e 3% de ruído.

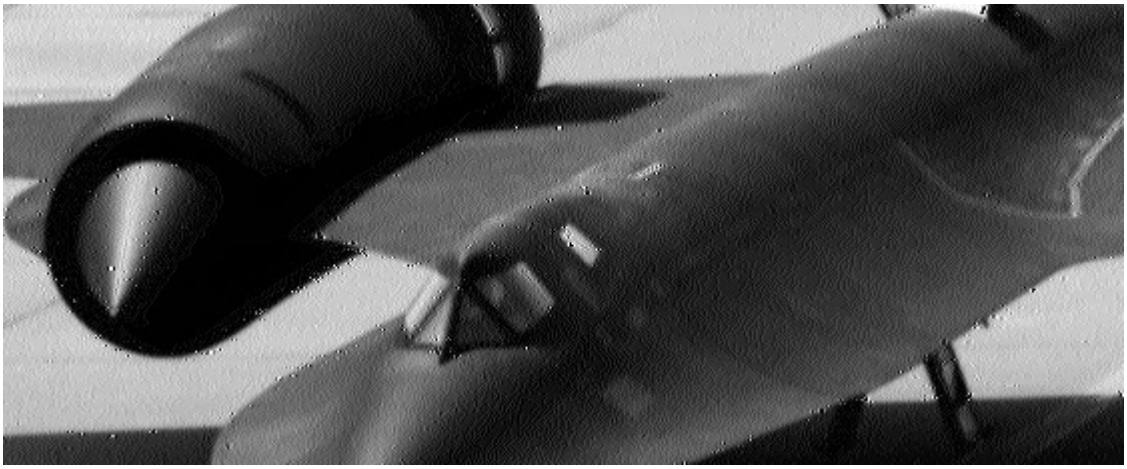

Figura 4.6: Imagem recuperada com método direto (BLAS mkl).

Figura 4.7: Imagem recuperada com Polyak após 350 iterações em 8 processadores .

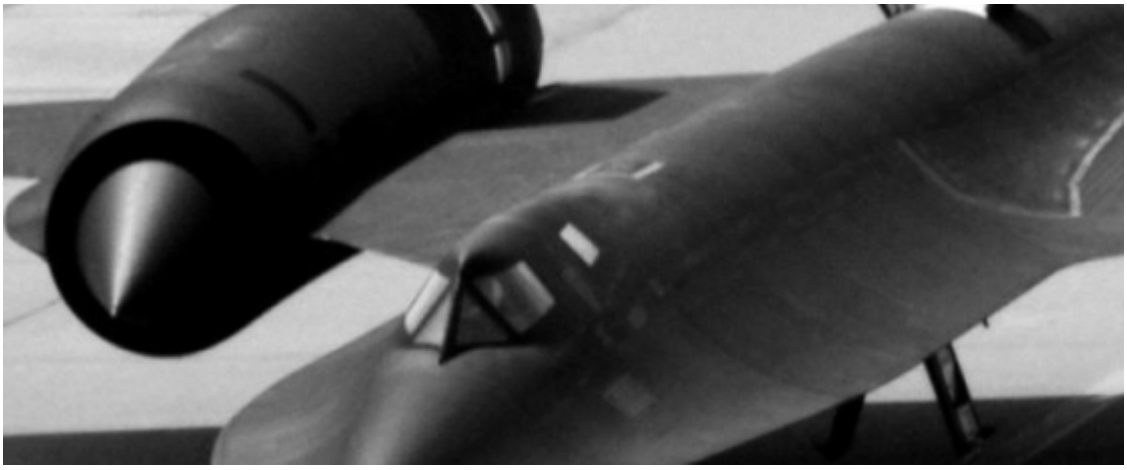

Figura 4.8: Imagem original.

Figura 4.9: Imagem recuperada com Polyak após 350 iterações em 1 processador.

CONCLUSÕES

O objetivo principal deste trabalho é demonstrar que é possível fazer uso de ambientes paralelos para implementar métodos de restauração de imagens que sofreram degradação de algum tipo. Este objetivo foi alcançado já que pudemos demonstrar com as curvas de *speed-up* e *Ep* que o nosso algoritmo consegue fazer o trabalho de um algoritmo seqüencial em menos tempo e que o aumento do número de processadores diminui este tempo de processamento.

O método de restauração usado nos testes foi o método de iterações de Polyak, este método não é muito usado no processamento de imagens, mas comportou-se de maneira satisfatória na maioria dos testes feitos. Outros métodos iterativos como o método de Jacobi apresentaram problemas para convergir num resultado ótimo nas distorções usadas nos estes. O método de Polyak é um método simples de fácil implementação já que trabalha aplicando a mesma transformação que ocasionou a distorção no processo de restauração e não tenta descompor a matriz *H* em submatrizes como o faz o método de Jacobi e LU. Este fato facilitou o desenvolvimento deste projeto.

Na figura 4.3 podemos observar as curvas de *speed-up* nas quais podemos ver que o nosso algoritmo comportou-se de uma maneira aceitável no sentido que a curva de *speed-up* do nosso algoritmo se manteve sempre com uma inclinação positiva o que indica que o aumento de processadores no processo não prejudicou o nosso desempenho. O tempo gasto em rodar o programa em um processador ficou quase duas vezes mais lento que no modelo aplicando o método direto, isto se deve a que o método direto geralmente é mais rápido que método iterativo, além do fato que a biblioteca usada (mkl) já é otimizada para trabalhar com processadores da Intel. Cabe ressaltar que por limitações das contas de treinamento no *cluster* usado (SGI Altix 350) só pudemos tesar o processo com até oito processadores.

Pudemos constatar que existe uma considerável diferença de tempo entre os programas compilados com o compilador da Intel (icc) e os programas compilados com o compilador GNU gcc, isto teve um profundo impacto nos tempos de execução dos nossos testes iniciais que foram feitos usando o compilador padrão dos sistemas Linux (gcc).
Alem da escolha do compilador da Intel usamos as ferramentas de otimização do compilador, o que diminuiu ainda mais os tempos de execução dos programas. Estas otimizações podem

ser aplicadas em três níveis, neste trabalho optou-se pelo nível máximo de otimização (-O3) o que gera um maior tempo de compilação, mas traz benefícios no programa final.

Para melhorar ainda mais o desempenho do programa podem ser usadas técnicas de perfilagem para as quais existem programas específicos nos quais podemos identificas que partes do programa consomem mais tempo e se faz correto uso da memória alocada.

No presente trabalho trabalhou-se com imagens armazenadas como arquivos PGM de modo ASCCI, isto faz com que no momento de carregar a imagem se faça uma varredura do arquivo inteiro para poder coletar a informação dos *pixels* da imagem, este processo poderia ser mais eficiente se trabalhássemos com arquivos binários. O uso de arquivos binários reduz o espaço necessário para armazenar a imagem e conseqüentemente reduziria o tempo que o programa consome para carregar os dados da imagem na memória do computador.

As imagens restauradas nos testes deram erros menores que 0.008 (RMS) para 350 iterações. Este erro reduzido pode ser percebido visualmente nas imagens restauradas ficando estas muito próximas das originais. O erro calculado nas imagens não foi diminuído acima de 355 iterações, o que quer dizer que o método tem um ponto ótimo de iterações, este ponto ótimo pode variar dependendo do estado de degradação da imagem.

REFERÊNCIAS

[1] WANG, WAHL. Yuan Wang, Friedrich M. Wahl. *Vector-entropy optimization-based neural-network approach to image reconstruction from projections*. IEEE Transaction on Neural Networks, v. 8, n. 5, pp. 1008-1014, Sep 1997.

[2] PERRY, WONG, GUAN. Stuart W. Perry, Hau San Wong, Ling Guan, *Adaptive Image Processing: A Computational Intelligence Perspective*. Bellingham, Spice Press, 2002.

[3] ORTEGA, James M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York and Londom, USA, 1989.

[4] POLYAK, B. T. Polyak. *New Method of Stochastic Approximation Type*. Vol. 51, no. 7, pt. 2, July 1990.

[5] STRANG. Gilbert Strang. *Linear Algebra and Its Applications*. Academic Press, United Kingdom (UK), 1976.

[6] GRAMA, GUPTA, KARYPIS, KUMAR. A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, Second Edition, USA, January 2003.

[7] CICHOCKI, UNBEHAUEN. A. Cichocki, R. Unbehauen. *Neural Networks for Optimization and Signal Processing*. John Wiley & Sons Ltd., 1992.

[8] FIGUEIREDO, LEITÃO. MAT Figueiredo, JMN Leitão. *Sequential and Parallel Image Restoration: neural network implementations*. Vol. 3, no. 6, November 1994.

[9] BANHAM, KATSAGGELOS. M. R. Banham, A. K. Katsaggelos. *Digital image restoration*. IEEE Signal Processing Magazine 14(2): 24–41. 1997.

[10] Efeito Bokeh. Disponível em: <http://www.vanwalree.com/optics/bokeh.html>. Acesso em 14 junho 2007.

[11] Arquivos PGM. Disponível em: <http://netpbm.sourceforge.net/doc/pgm.html#lbAC>. Acesso em 14 junho 2007

[12] MPI Routines. Disponível em: < http://www-unix.mcs.anl.gov/mpi/www/www3/> Acesso em 14 junho 2007.

[13] Common Research Images. Disponível em: <http://iul.eng.fiu.edu/Resources/Images/ImagesPage.htm>. Acesso em: 14 junho 2007.

[14] Processamento de Imagens UCPEL. Disponível em: <http://atlas.ucpel.tche.br/~vbastos/pi.htm>. Acesso em: 14 junho 2007.

[15] Processamento Paralelo, Mendes C. L. Disponível em: < http://www.lac.inpe.br/~celso/cap334/>. Acesso em: 14 junho 2007.

[16] Filtros Digitales de Imágenes IIE. Disponível em: <http://iie.fing.edu.uy/ense/asign/dlp/proyectos/2004/imag2_filtro/index.htm>. Acesso em: 14 junho 2007.

[17] Operações Espaciais GBDI USP. Disponível em: <http://gbdi.icmc.usp.br/disciplinas/sce-230/aulas/aula04/>. Acesso em: 14 junho 2007.

[18] Jet Propulsion Laboratory JPL NASA. Disponível em: <http://www.jpl.nasa.gov/index.cfm>. Acesso em: 14 junho 2007.

[19] Núcleo de Atendimento em Computação de Alto Desempenho NACAD. Disponível em: <http://www.nacad.ufrj.br/ >. Acesso em: 23 julho 2007.

[20] Intel C++ compiler 10.0. Disponível em: < http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm >. Acesso em: 23 julho 2007.

[21] Métodos numéricos, Freitas S. R. Disponível em: <http://www.dct.ufms.br/~evely/metodos.pdf>. Acesso em: 30 julho 2007.

[22] Intel Math Kernel Library 9.1 – BLAS/LAPACK. Disponível em: <http://www.intel.com/cd/software/products/asmo-na/eng/266858.htm>. Acesso em: 27 agosto 2007

APÊNDICE A – Tabelas de Dados

icc

np	iterações	Erro RMS	Tempo (s)
1	40	0,0110	13,1545
1	80	0,0082	25,8894
1	120	0,0072	38,6067
1	160	0,0067	51,3620
1	200	0,0064	64,2716
1	240	0,0063	76,8795
1	280	0,0061	89,9160
1	320	0,0061	102,6079

Tabela 1: Dados do programa seqüencial compilado com icc.

icc O3

np	iterações	Erro RMS	Tempo (s)
1	40	0,0110	12,8139
1	80	0,0082	25,3174
1	120	0,0072	37,8005
1	160	0,0067	50,2933
1	200	0,0064	62,7900
1	240	0,0063	75,2633
1	280	0,0061	87,7795
1	320	0,0061	100,3094

Tabela 2: Dados do programa seqüencial compilado com icc –O3.

gcc

np	iterações	Erro RMS	Tempo (s)
1	40	0,0110	135,6894
1	80	0,0082	273,3005
1	120	0,0072	411,0600
1	160	0,0067	548,2553
1	200	0,0064	685,9904
1	240	0,0063	823,3438
1	280	0,0061	960,2444
1	320	0,0061	1.097,5003

Tabela 3: Dados do programa seqüencial compilado com gcc.

gcc O3

np	iterações	Erro RMS	Tempo (s)
1	40	0,0110	41,8382
1	80	0,0082	84,0180
1	120	0,0072	126,4808
1	160	0,0067	168,7211
1	200	0,0064	211,0746
1	240	0,0063	253,1080
1	280	0,0061	295,4576
1	320	0,0061	337,7507

Tabela 4: Dados do programa seqüencial compilado com gcc –O3.

np	iterações	Erro RMS	Tempo (s)
1	320	0.053843	69.45086
1	280	0.051684	60.85913
1	240	0.049219	52.22706
1	200	0.046364	43.58068
1	160	0.042995	34.95837
1	120	0.038924	26.3289
1	80	0.033858	17.67861
1	40	0.027562	9.042315

Tabela 5: Testes do algoritmo seqüencial (MB 21p 0g).

np	iterações	Erro RMS	Tempo (s)
1	320	0.006074	98.54249
1	280	0.006147	86.26766
1	240	0.006261	74.01463
1	200	0.006437	61.69296
1	160	0.006717	49.42204
1	120	0.007196	37.14819
1	80	0.008157	24.86815
1	40	0.011017	12.59691

Tabela 6: Testes do algoritmo seqüencial (T 1.3 N 7%).

np	Iter-ações	Erro RMS	Tempo (s)
1	320	0,00666	346,48195
2	320	0.007045	175,33645
3	320	0.008123	119,35797
4	320	0.007949	91,52342
5	320	0,00826	74,84261
6	320	0.008801	63,74158
7	320	0.009772	55,99410
8	320	0.009108	49,72232

Tabela 7: Testes do algoritmo paralelo (gcc –O3).

np	Iter-ações	Erro RMS	Tempo (s)
1	320	0,00666	99,20194
2	320	0,00705	50,66221
3	320	0,00812	34,66589
4	320	0,00795	26,70369
5	320	0,00826	21,95382
6	320	0,00880	18,74961
7	320	0,00977	16,58126
8	320	0,00911	14,78738

Tabela 8: Testes do algoritmo paralelo (icc –O3).

np	Dimensão	Iter-ações	Erro RMS	Tempo (s)
1	700x700	350	0.002619	69.120320
2	700x700	350	0.005152	35.217984
3	700x700	350	0.006167	24.048640
4	700x700	350	0.006298	18.377104
5	700x700	350	0.006037	15.006976
6	700x700	350	0.007021	12.789504
7	700x700	350	0.007947	11.175200
8	700x700	350	0.007234	10.015712

Tabela 9: Programa paralelo rodado com imagem de 700x700 pixels.

np	Dimensão	Iterações	Erro RMS	Tempo (s)
1	900x900	350	0.002606	114.497488
2	900x900	350	0.005066	57.512752
3	900x900	350	0.005623	39.101488
4	900x900	350	0.006112	29.775808
5	900x900	350	0.006411	24.206752
6	900x900	350	0.006771	20.516496
7	900x900	350	0.006742	17.940832
8	900x900	350	0.006986	15.949792

Tabela 10: Programa paralelo rodado com imagem de 900x900 pixels.

np	Dimensão	Itera-ções	Erro RMS	Tempo (s)
1	1100x1100	350	0.003120	171.161120
2	1100x1100	350	0.004681	86.073440
3	1100x1100	350	0.005311	58.005632
4	1100x1100	350	0.005091	44.079088
5	1100x1100	350	0.005708	35.672800
6	1100x1100	350	0.005787	30.201344
7	1100x1100	350	0.006227	26.307104
8	1100x1100	350	0.006797	23.259056

Tabela 11: Programa paralelo rodado com imagem de 1100x1100 pixels.

np	Dimensão	Iter-ações	Erro RMS	Tempo (s)
1	1300x1300	350	0.002699	239.176608
2	1300x1300	350	0.004556	119.793264
3	1300x1300	350	0.005151	80.755216
4	1300x1300	350	0.005481	61.294752
5	1300x1300	350	0.005629	49.453920
6	1300x1300	350	0.006147	41.608832
7	1300x1300	350	0.006238	36.076864
8	1300x1300	350	0.006167	31.946432

Tabela 12: Programa paralelo rodado com imagem de 1300x1300 pixels.

np	Sp	Ep
1	0,5447	0,5447
2	1,0690	0,5345
3	1,5655	0,5218
4	2,0486	0,5122
5	2,5087	0,5017
6	2,9437	0,4906
7	3,3689	0,4813
8	3,7589	0,4699

np	Sp	Ep
1	0,5673	0,5673
2	1,1293	0,5647
3	1,6611	0,5537
4	2,1813	0,5453
5	2,6831	0,5366
6	3,1657	0,5276
7	3,6202	0,5172
8	4,0721	0,5090

Tabela 13: Speed-Up do programa paralelo com imagem de 700x700 pixels.

Tabela 14: Speed-Up do programa paralelo com imagem de 900x900 pixels.

APÊNDICE B – Diagramas de Atividades

Algoritmo Seqüencial

Descrição

Este algoritmo tem como objetivo restaurar uma imagem que sofreu algum tipo de distorção por movimentação e contém ruído aditivo. Para este propósito usamos um algoritmo iterativo (Polyak) até atingir um erro menor que o erro especificado previamente ou alcançar um número de iterações máximo.

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento deste algoritmo.

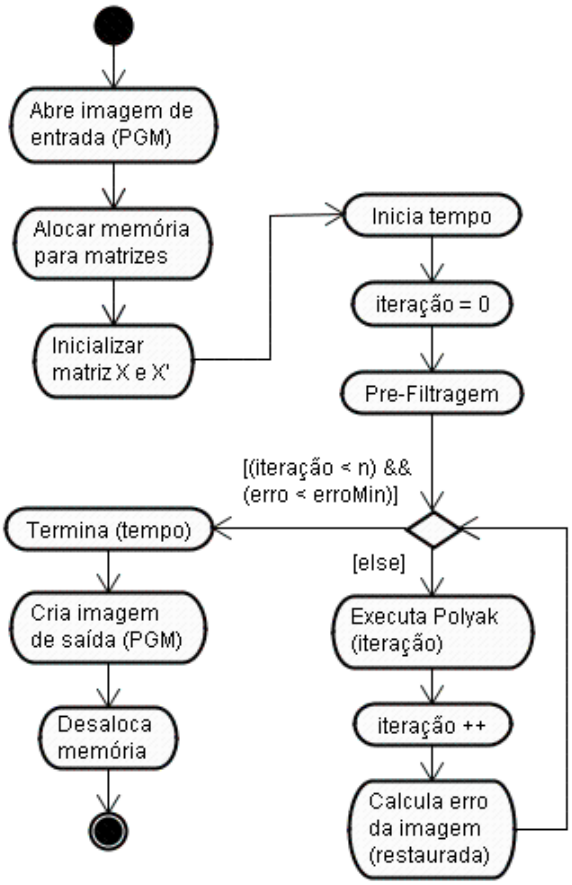

Figura B.1: Diagrama de atividades do algoritmo seqüencial.

Abre imagem de entrada (PGM): Neste passo abrimos o arquivo que contem a imagem a tratar (PGM).

Alocar memória para matrizes: Alocamos dinamicamente a memória para as matrizes que usamos no programa.

Inicializar matriz X e X': Damos os valores iniciais para as matrizes que intervém no algoritmo de Polyak.

Inicia tempo: A contagem do tempo de processamento começa neste ponto.

Pré-Filtragem: Aplicamos o filtro de Mediana Seletivo para poder livrar a imagem a tratar do ruído aditivo.

Executa Polyak: Neste ponto começamos a restauração da imagem usando o algoritmo de Polyak.

Calcula erro da imagem: O cálculo do erro se faz aplicando a norma infinita à diferença entre a imagem restaurada multiplicada pela matriz de distorção e a imagem distorcida que entrou no processo.

$$erro = RMS(Hx'(k) - g)\_$$

Termina (tempo): A contagem do tempo de processamento acaba neste ponto.

Cria imagem de saída (PGM): A imagem restaurada é passada para o formato PNG.

Desaloca memória: A memória que foi alocada para trabalhar no processo de restauração é desalocada.

Entradas e saídas

Entrada: Nome do arquivo (PGM) que contem a imagem que irá ser restaurada.

Saída: Imagem restaurada no formato PGM.

Função mediana( )

Descrição

Esta função tem como objetivo calcular a mediana de um conjunto de valores dentro de um vetor.

Comportamento e Arquitetura Interna

A seguir é apresentado o fluxo de atividades que representa o comportamento desta função.

Figura B.2: Fluxo de atividades da função mediana().

Receber o vetor: o vetor contendo todos os valores é recebido.

Ordena o vetor em forma crescente: reordena os valores dentro do vetor de forma crescente.

Seleciona o elemento central no vetor: seleciona o elemento que ficar no meio do vetor depois de ter sido ordenando de forma crescente.

Retorna o elemento central: retorna o elemento selecionado no passo anterior.

Entradas e saídas

Entradas: ponteiro do vetor (*vetor), dimensão do vetor (n).

Saídas: mediana do vetor.

Função filtroMediana3x3( )

Descrição

Esta função tem como objetivo aplicar um filtro de mediana (3x3) em uma determinada posição dentro de uma matriz.

Comportamento e Arquitetura Interna

A seguir é apresentado o fluxo de atividades que representa o comportamento desta função.

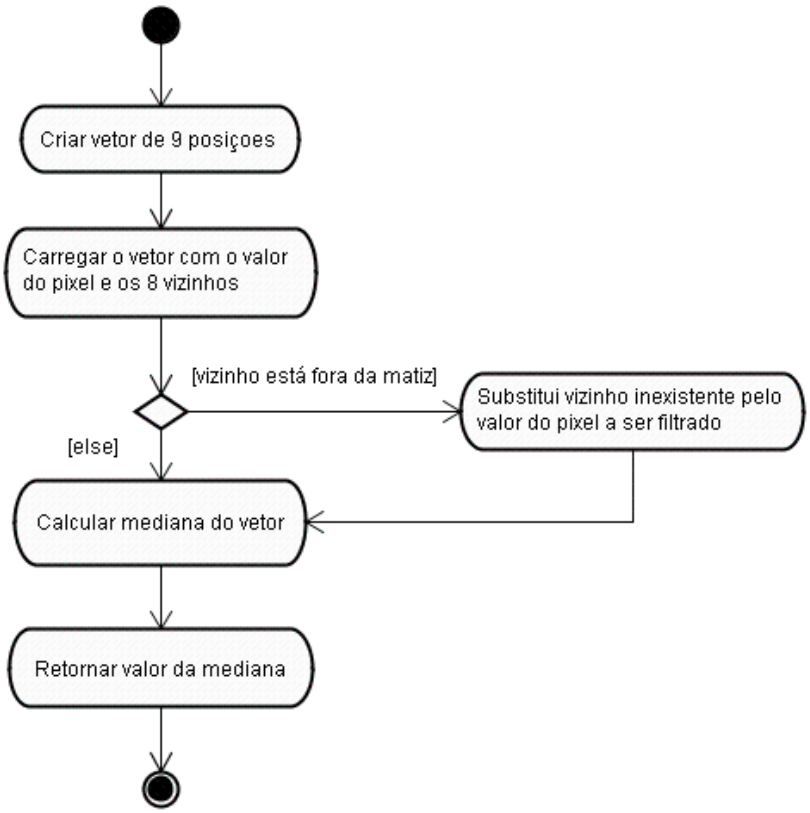

Figura B.3: Fluxo de atividades da função filtroMediana3x3().

Criar vetor de 9 posições: alocar memória para um vetor de 9 posições e valores de 2 bytes (16 bits).

Carregar o vetor com o valor do *pixel* e os 8 vizinhos: Carregar o vetor com os valores dos 8 vizinhos mais próximos do *pixel* a ser filtrado incluindo ele mesmo.

Calcular mediana do vetor: usar a função mediana() para calcular este valor.

Substituir vizinho inexistente pelo valor do *pixel* a ser filtrado: no momento de carregar o vetor de 9 posições se não existir um ou mais dos vizinhos este valor deve ser substituído pelo valor do *pixel* a ser filtrado.

Retornar valor da mediana: retornar o valor do resultado da filtragem do *pixel* de coordenadas x, y.

Entradas e saídas

Entradas: ponteiro da matriz que contem o elemento (**matriz), dimensão da matriz (lin, col), coordenadas do elemento a ser filtrado (x,y).

Saídas: valor do *pixel* filtrado.

Função filtroMedianaSeletivo3x3( )

Descrição

Esta função tem como objetivo aplicar um filtro de mediana "seletivo" (3x3) em uma determinada posição de uma matriz.

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento desta função.

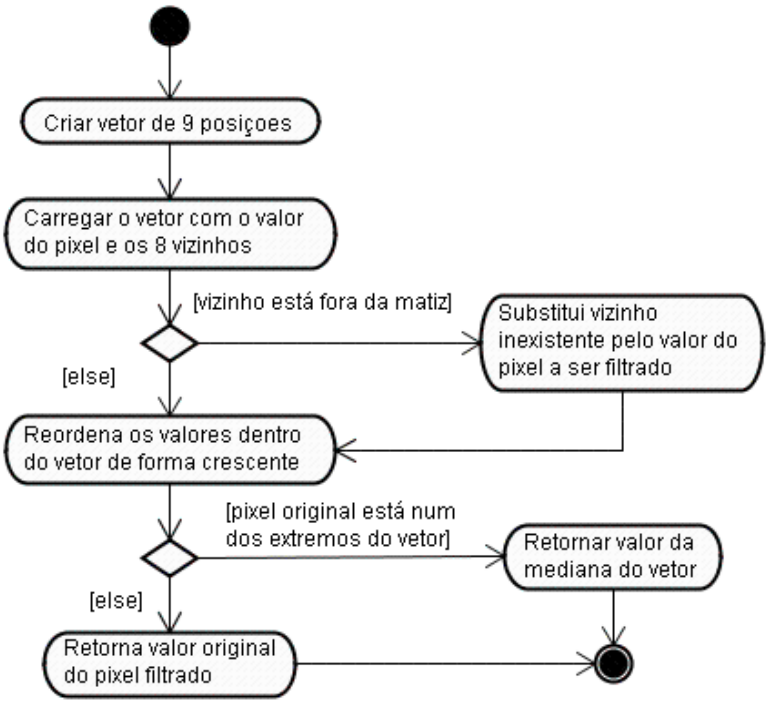

Figura B.4: Fluxo de atividades do algoritmo filtroMedianaSeletivo3x3().

Criar vetor de 9 posições: alocar memória para um vetor de 9 posições e valores de 2 bytes (16 bits).

Carregar o vetor com o valor do *pixel* e os 8 vizinhos: Carregar o vetor com os valores dos 8 vizinhos mais próximos do *pixel* a ser filtrado incluindo ele mesmo.

Reordenar os valores dentro do vetor de forma crescente: Os valores dentro do vetor são ordenados de forma crescente.

Substituir vizinho inexistente pelo valor do *pixel* a ser filtrado: No momento de carregar o vetor de 9 posições se não existir um ou mais dos vizinhos este valor deve ser substituído pelo valor do *pixel* a ser filtrado.

Retornar valor da mediana do vetor: retornar o valor do resultado do cálculo da mediana do vetor alocado.

Retornar valor original do *pixel* filtrado: O valor original do *pixel* filtrado é retornado.

Entradas e saídas

Entradas: ponteiro da matriz que contem o elemento (**matriz), dimensão da matriz (lin, col), coordenadas do elemento a ser filtrado (x,y).

Saídas: valor do *pixel* filtrado.

Observações

O cálculo da mediana "seletiva" de um conjunto de valores é representado nas figuras B.5 e B.6. Esta mediana serve para poder identificar quando é necessário fazer o cálculo da mediana e quando podemos prescindir deste resultado.

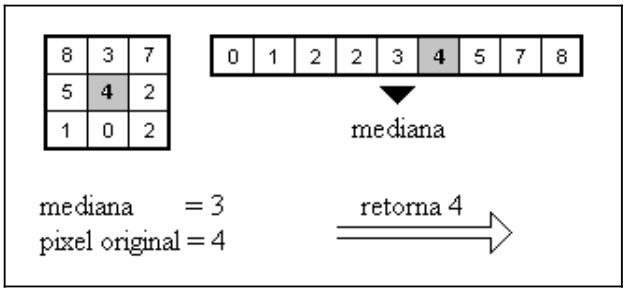

Figura B.5: Representação gráfica do cálculo da mediana "seletiva" (caso 1).

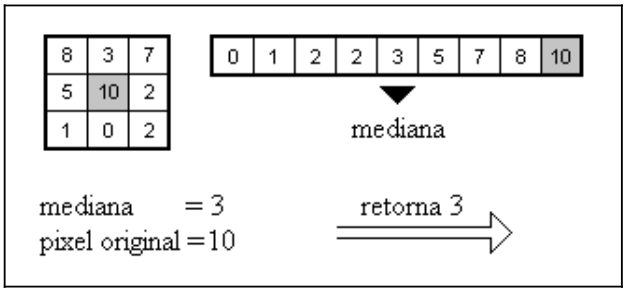

Figura B.6: Representação gráfica do cálculo da mediana "seletiva" (caso 2).

Função polyak()

Descrição

Esta função tem como objetivo realizar as iterações do algoritmo de Polyak até atingir um erro menor que um erro especificado previamente (erro mínimo).

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento desta função.

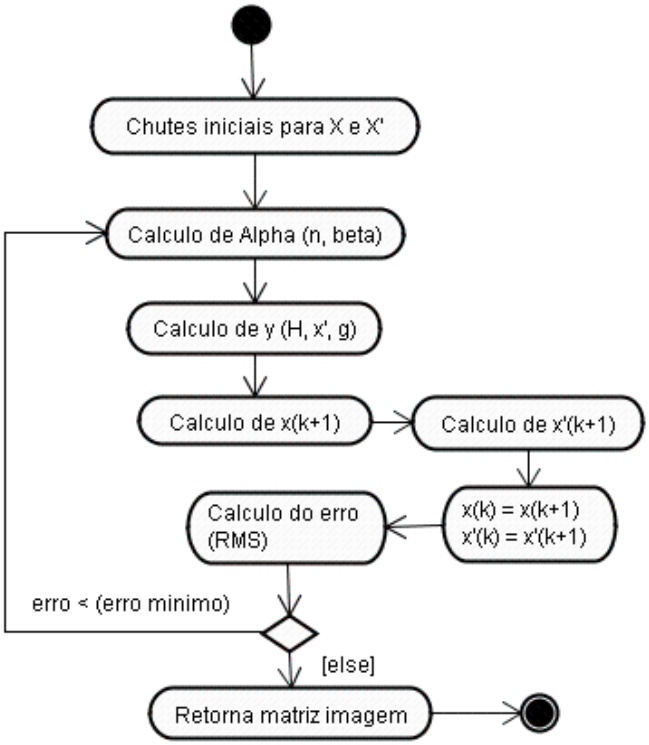

Figura B.7: Fluxo de atividades da função polyak().

Chutes iniciais para *X* e *X'*: As matrizes *X* e *X'* são carregadas com valores iniciais para começar o algoritmo de Polyak.

Calculo de Alpha (n, beta): Alpha é calculado como a inversa do número da iteração elevado a β.

$$\alpha = \frac{1}{n^{\beta}}$$

Calculo de *y (H, x', g)*: Este valor é obtido como resultado da seguinte operação:

$$y = Hx'(k) - g$$

Calculo de *x(k+1)*: Este valor é obtido como resultado da seguinte operação:

$$x_{k+1} = x_k - \alpha y$$

Calculo de *x'(k+1)*: Este valor é obtido como resultado da seguinte operação:

$$x'_{k+1} = x'_k + \frac{x_k - x'_k}{n+1}$$

Calculo do erro (Norma infinita): O cálculo do erro se faz aplicando a Norma Infinita à diferença entre a imagem restaurada multiplicada pela matriz de distorção e a imagem distorcida que entrou no processo.

$$erro = \|Hx'(k) - g\|$$

Retorna matriz imagem: Se o erro ficar menor que o erro esperado, esta função retorna a imagem restaurada.

Entradas e saídas

Entradas: Imagem a restaurar (*g*), matriz de distorção (*H*), valor de beta ($0.5<\beta<1$) e erro mínimo.

Saídas: matriz restaurada.

Observações

As iterações de Polyak são mostradas a seguir:

$$x^{k+1} = x^k - \gamma^k \cdot y^k \qquad y^k = Ax^k - b + \xi^k$$

$$\hat{x}^{k+1} = \hat{x}^k + \frac{x^k - \hat{x}^k}{k+1} \qquad k = 0,1,...$$

Algoritmo Paralelo

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento deste algoritmo.

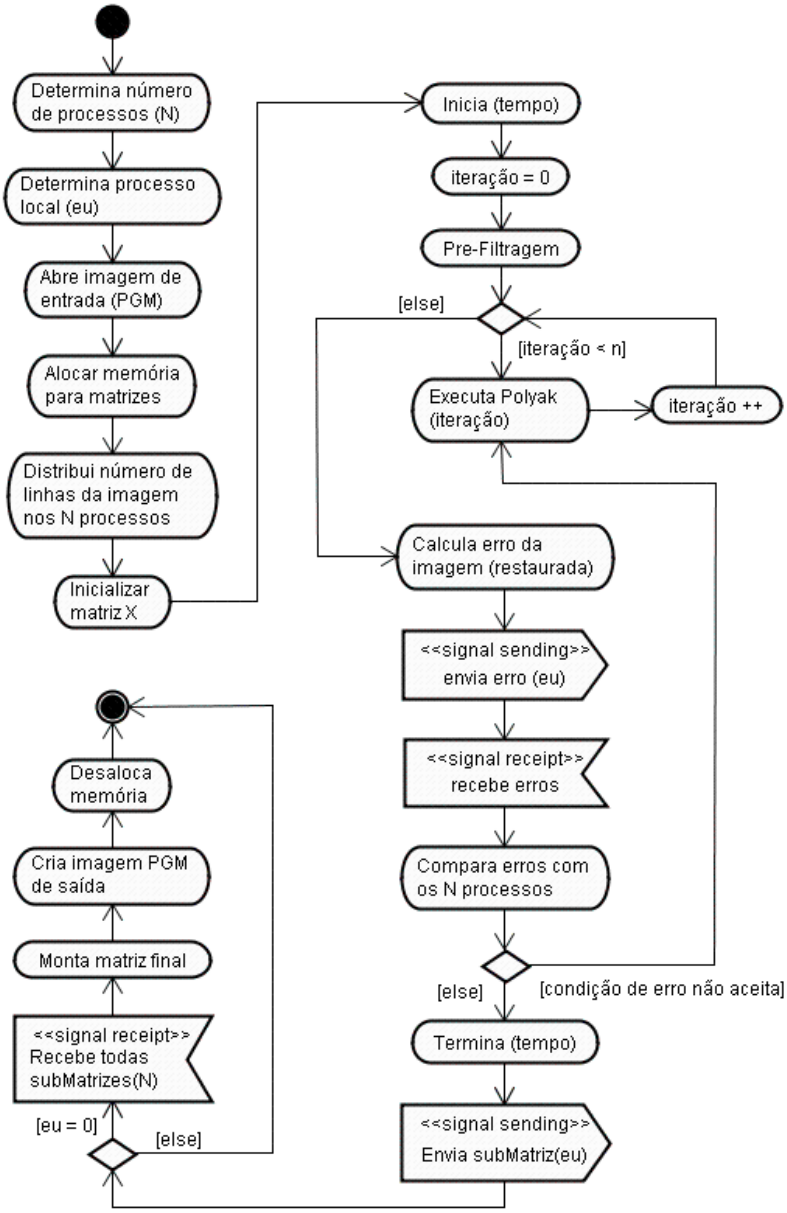

Figura B.8: Fluxo de atividades do algoritmo paralelo.

Determina número de processos (*n*): Neste passo é determinado e registrado o número de nós de processamento com os quais contamos.

Determina processo local (*eu*): É determinado e registrado o número do nó que será usado como processo principal.

Abre imagem de entrada (PGM): Neste passo abrimos o arquivo que contem a imagem a tratar (PGM).

Alocar memória para matrizes: Alocamos dinamicamente a memória para as matrizes que usamos no programa.

Distribui número de linhas da imagem nos *n* processos: Todas as linhas da matriz da imagem são distribuídas nos nós, tentando colocar o mesmo número de linhas em cada nó.

Inicia tempo: A contagem do tempo de processamento começa neste ponto.

Pré-Filtragem: Aplicamos o filtro de Mediana Seletivo para poder livrar a imagem a tratar do ruído aditivo.

Executa Polyak: Neste ponto começamos a restauração da imagem usando o algoritmo de Polyak.

Calcula erro da imagem: O cálculo do erro se faz aplicando a norma infinita à diferença entre a imagem restaurada multiplicada pela matriz de distorção e a imagem distorcida que entrou no processo.

$$erro = RMS(Hx'(k) - g)$$

Envia erro (*eu*): Todos os subprocessos enviam os erros calculados.

Recebe erros: O subprocesso principal (*eu*) recebe os erros de todos os outros subprocessos.

Compara erros com os *n* processos: Os erros de todos os subprocessos são comparados e se faz uma estimativa do erro total da imagem.

Termina (tempo): A contagem do tempo de processamento acaba neste ponto.

Envia subMatriz(*eu*): Todos os subprocesso enviam o resultado da restauração das parcelas da matriz imagem que eles restauraram.

Recebe todas subMatrizes(*n*): O nó principal (*eu*) recebe todos as parcelas dos subprocessos.

Monta matriz final: O nó principal (*eu*) monta a imagem completa a partir da informação que cada subprocesso enviou.

Cria imagem PGM de saída: Salva os dados da nova imagem restaurada no formato PGM.

Desaloca memória: A memória que foi alocada para trabalhar no processo de restauração é desalocada.

Entradas e saídas

Entrada: Nome do arquivo (PGM) que contem a imagem que irá ser restaurada.

Saída: Imagem restaurada no formato PGM.

Função distribuirLinhas()

Descrição

Esta função se encontra dentro do algoritmo paralelo e tem como objetivo criar e preencher com os correspondentes valores da matriz principal um número *n* de submatrizes que irá depender do número de processos.

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento deste algoritmo.

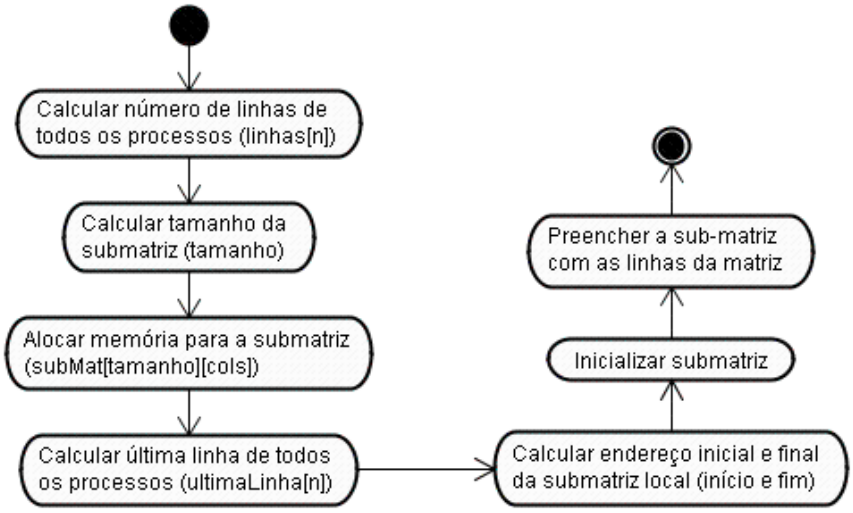

Figura B.9: Fluxo de atividades da função distribuirLinhas().

Calcular número de linhas de todos os processos (linhas[n]): Neste passo é calculado o número de linhas que corresponde a cada um dos nós de tal maneira que todos os nós fiquem com igual ou parecido número de linhas da matriz original.

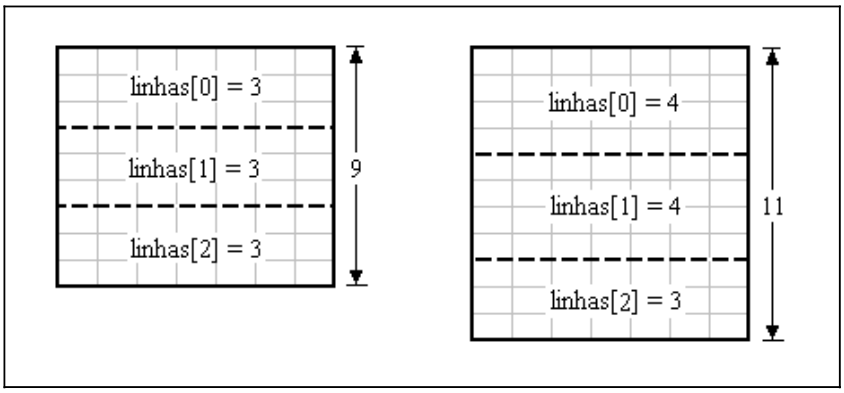

Figura B.10: Distribuição de linhas da matriz original em cada nó.

Calcular tamanho da submatriz (tamanho): O tamanho de cada submatriz é definido como o número de linhas de cada subprocesso mais o número de linhas de *overlapping*.

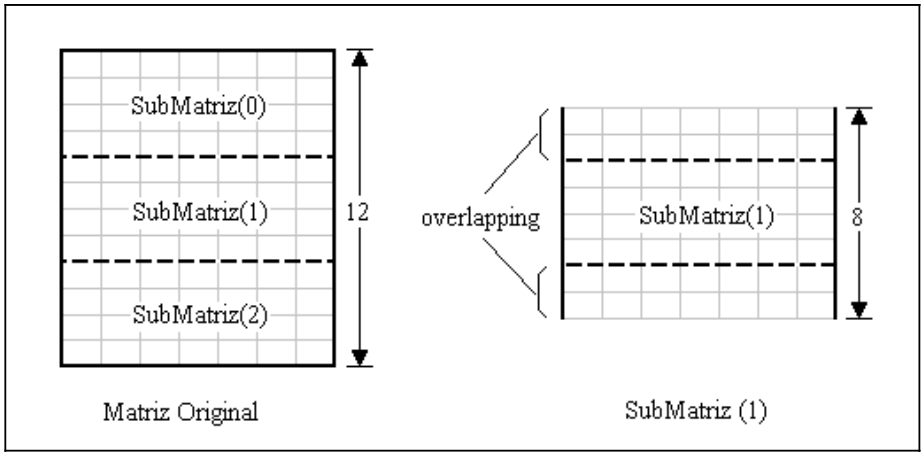

Figura B.11: Tamanho da submatriz do processo local.

Alocar memória para a submatriz (subMat[tamanho][cols]): Uma vez definida a dimensão de cada submatriz é alocada memória em cada nó.

Calcular última linha de todos os processos (ultimaLinha[n]): Como parte deste algoritmo, é calculado o número da linha da matriz original que será a última linha da correspondente submatriz de cada processo.

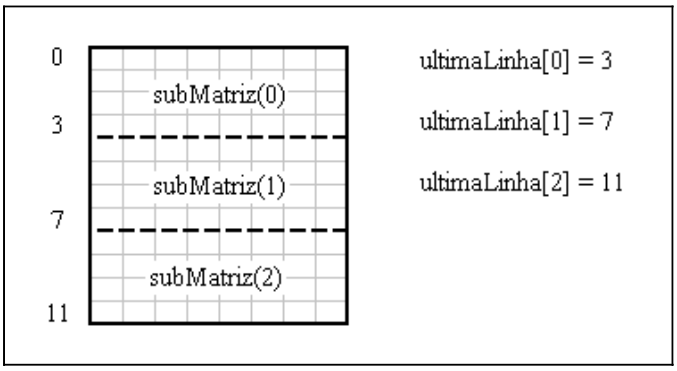

Figura B.12: Cálculo do número de cada última linha das submatrizes.

Calcular endereço inicial e final da submatriz local (início e fim): Os números da primeira e última linha do segmento da matriz original que será colocado na submatriz são armazenados em duas variáveis.

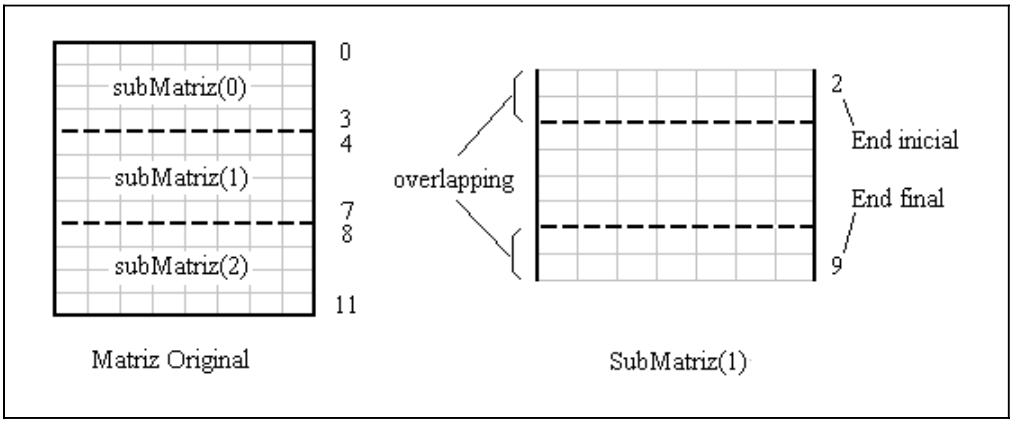

Figura B.13: Calculando endereço inicial e final da submatriz.

Inicializar submatriz: As matrizes alocadas em cada nó são inicializadas com um valor inicial (geralmente zero).

Preencher a submatriz com as linhas da matriz: Como último passo os valores da matriz original são carregados nas submatrizes de cada nó.

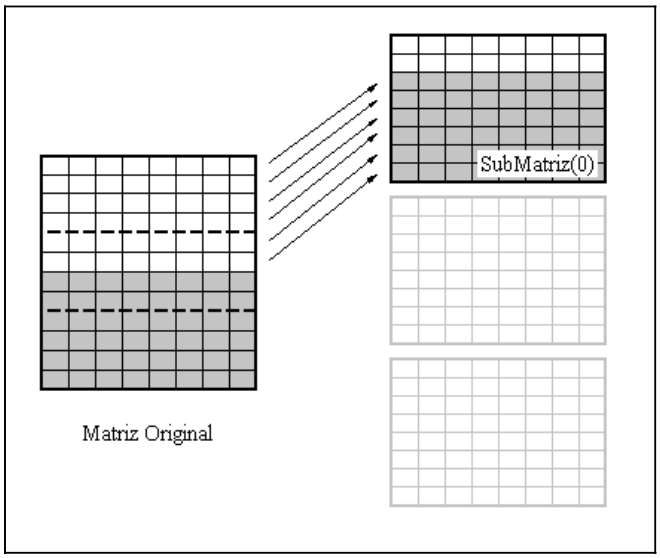

Figura B.14: Preenchendo submatriz(0).

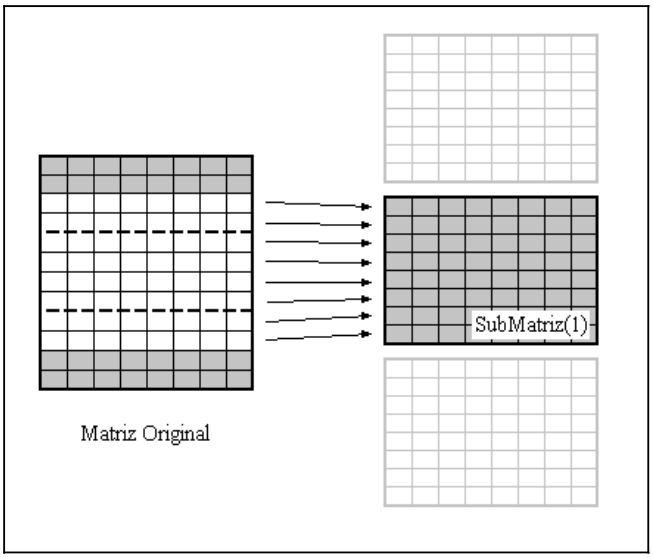

Figura B.15: Preenchendo submatriz(1).

Entradas e saídas

Entrada: Matriz de valores da imagem a ser restaurada, número de processos, dimensão da matriz e número do processo local.

Saída: submatriz local, número de linhas e tamanho da submatriz local.

Montagem da matriz final

Descrição

Esta rotina encontra-se dentro do algoritmo paralelo e é executado unicamente no nó principal (nó zero). Esta rotina monta uma matriz de saída com os valores que recebe de todos os subprocessos no momento em que estes acabam o processamento.

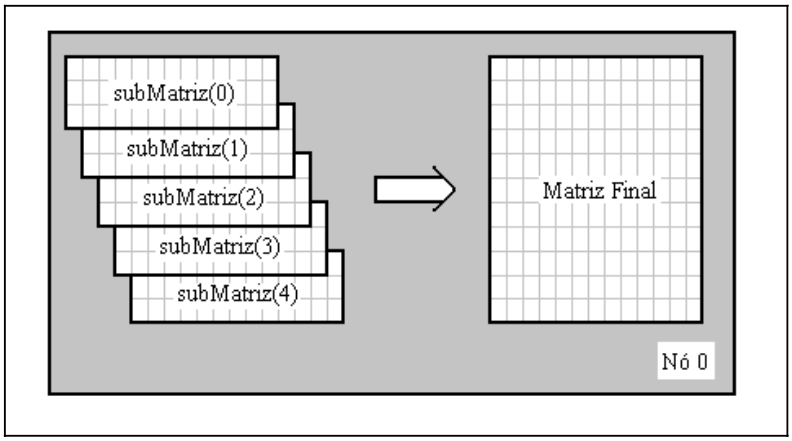

Figura B.16: Montagem da matriz final.

Comportamento/Arquitetura interna

A seguir é apresentado o fluxo de atividades que representa o comportamento desta rotina.

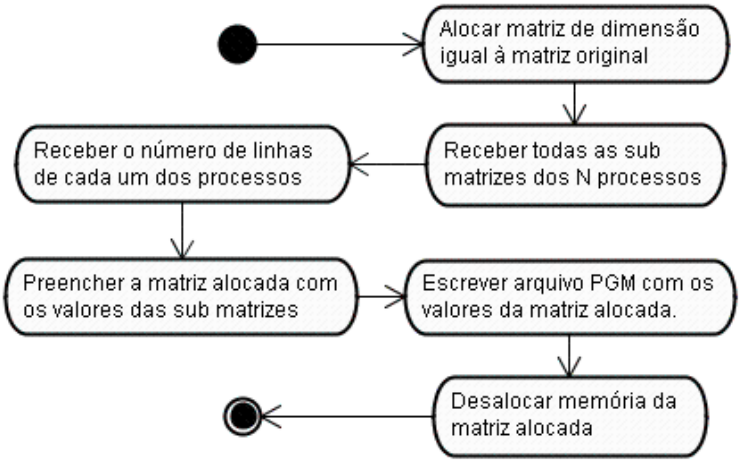

Figura B.17: Fluxo de atividades do algoritmo de montagem da matriz.

<u>Alocar matriz de dimensão igual á matriz original</u>: nesta atividade é alocada memória dinamicamente para uma matriz com a mesma dimensão da matriz original, é usada a função alocarMatriz().

Receber todas as submatrizes dos *n* processos: nesta atividade são recebidas todas as sub matizes dos *n* processos.

Receber o número de linhas de cada um dos processos: esta função recebe um vetor com a informação do número de linhas que cada processo pegou da matriz original (linhas[]).

Preencher a matriz alocada com os valores das submatrizes: nesta etapa se realiza a reconstrução da matriz restaurada, colocando as correspondentes linhas das submatrizes dentro da matriz de saída, para realizar esta tarefa é necessário saber o número de linhas de cada processo e o número das linhas de *overlapping*.

Escrever arquivo PGM com os valores da matriz alocada: uma vez reconstruída a matriz um arquivo PNG é criado contendo os valores da matriz reconstruída, este passo é realizado usando a função escritaArquivoPGM().

Desalocar memória da matriz alocada: uma vez terminado o processo de juntar as submatrizes e antes de sair da função a memória alocada para a matriz reconstruída é liberada, este passo é realizado usando a função liberarMatriz().

Entradas e saídas

Entradas: número de linhas de cada processo (linhas[n]), ponteiros das submatrizes dos processos (**subMatriz), tamanho do *overlapping* (tamOv).

Saídas: ponteiro da matriz reconstruída (**matSaida).

Observações

A seguir mostramos uma representação gráfica da montagem da matriz.

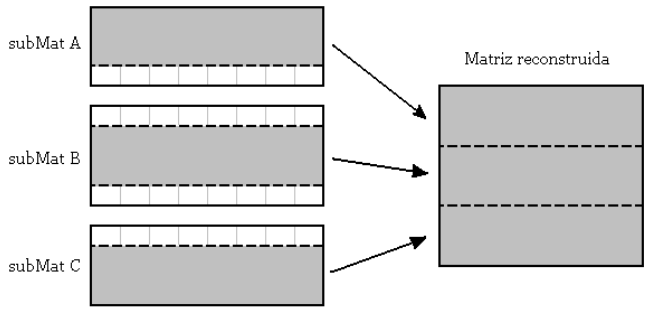

Figura B.18: Representação gráfica da reconstrução da matriz de saída.

APÊNDICE C – Código Seqüencial

```
/*
 * Universidade Federal do Rio de Janeiro
 * Escola Politecnica
 * Departamento de Eletronica e de Computacao
```

```
 *
 * Autor       : Jaime Grande Vela
 * Orientador  : Eugenius Kaszkurewicz
 * Co-orintador: Antonio Petraglia
 * 2007-2
 *
 * Descricao: Projeto de fim de curso
 * Nome do projeto: Restauracao de Imagens Monocromaticas
 *                  Utilizando Processamento Paralelo
 *
 * arquivo: retauradorSeq.c
 */

#include "funRest.h"

int main(int argc, char *argv[])
{
   unsigned short int max;
   char fname[20];

   unsigned short int **yp = NULL;
   unsigned short int **y  = NULL;
   unsigned short int **x  = NULL;
   unsigned short int **xp = NULL;
   unsigned short int **x2 = NULL;
   int ret, i, j;
   int lin = 0;
   int col = 0;
   int n , numIt;
   float beta;
   float alpha;
   int g = 0;
   float erro;
   clock_t startclock, stopclock;
   float cond;
   char *nomeArqPGM;
   char *nomeOUT;

   numIt = atoi(argv[1]); // numero de iteracoes limite
   nomeArqPGM = argv[2];  // nome do arquivo
   nomeOUT    = argv[3];  // nome do arquivo de saida

startclock = clock();

   // Abre arquivo PGM de entrada
   yp = leituraArquivoPGM(nomeArqPGM, &lin, &col, &max);
   y  = alocarMatriz(lin, col);
   x  = alocarMatriz(lin, col);
   xp = alocarMatriz(lin, col);
   x2 = alocarMatriz(lin, col);

   for(i = 0; i < lin; i++)
      for(j = 0; j < col; j++)
         y[i][j] = filtroSeletivo3(yp,i,j,lin,col);

   for(i = 0; i < lin; i++)
      for(j = 0; j < col; j++)
         x2[i][j]=32000;

   for(i = 0; i < lin; i++)
      for(j = 0; j < col; j++)
         x[i][j]=32000;

   beta = 0.78;
   n = 1; erro = 100.00; //numIt = 30;
   while( n<numIt && erro>0.001)  {
      for(i = 0; i < lin; i++)  {
         for(j = 0; j < col; j++)  {
            alpha = (float)(pow((float)n,-beta));
            g = (int)toeplitz1_3(x,col,i,j) - (int)y[i][j];
            cond = (float)x[i][j] - alpha*g;
            if(cond > 0 && cond < max)
               xp[i][j] = (unsigned short int)cond;
            else {
               if(cond > max )
                  xp[i][j] = max;
               else
```

```
                xp[i][j] = 0;
            }
            x2[i][j] = (n*x2[i][j]+xp[i][j])/(n+1);  //x2=x2+(xp-x2)/(n+1);
         }
      }
      copiaMatriz(xp, x, lin, col);
      n++;

      // calcula RMS
      for(i = 0; i < lin; i++)
         for(j = 0; j < col; j++)
            yp[i][j]=(unsigned short int)abs(y[i][j]-toeplitz1_3(x2,col,i,j));

      erro = rms(yp,lin,col)/max;
   }

stopclock = clock();

   printf("Numero de iteracoes: %d\n",n);
   printf("Erro(rms):           %f\n", erro);
   printf("Tempo de exec:       %f seg.\n",(stopclock-startclock)/(double)CLOCKS_PER_SEC);

   escritaArquivoPGM(x2,nomeOUT,lin,col,max);

   liberarMatriz(lin ,yp);
   liberarMatriz(lin ,y);
   liberarMatriz(lin ,x);
   liberarMatriz(lin ,xp);
   liberarMatriz(lin ,x2);

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

APÊNDICE D – Código Paralelo

```
/*
 * Universidade Federal do Rio de Janeiro
 * Escola Politecnica
 * Departamento de Eletronica e de Computacao
 *
 * Autor       : Jaime Grande Vela
 * Orientador  : Eugenius Kaszkurewicz
 * Co-orintador: Antonio Petraglia
 * 2007-2
 *
 * Descricao      : Projeto de fim de curso
 * Nome do projeto: Restauracao de Imagens Monocromaticas
 *                  Utilizando Processamento Paralelo
 *
 * arquivo: restImg.c
 */

#include "funRest.h"
#include <mpi.h>

int main(int argc, char *argv[])
{
   unsigned short int max;
   unsigned short int **y = NULL;
   unsigned short int **yp = NULL;
   unsigned short int **x  = NULL;
   unsigned short int **xp = NULL;
   unsigned short int **x2 = NULL;

   unsigned short int **subMatLocal = NULL;
   unsigned short int *rbuf      = NULL;
   unsigned short int *sendarray = NULL;

   int *lins = NULL;
   signed long int g = 0;
   int i, j, numIt, lin, col, n, tamSubMat, eu, np, z, k;
   float alpha, beta;
   clock_t startclock, stopclock;
   char *nomeArqPGM;
   int *recvcnts;
   int *displs;
   float cond;
   float erro;

   int *ultimaLinha = NULL;
   int *ini = NULL;
   int resto;
   int overlap = 8;
   unsigned short int *subMatEnvio  = NULL;
   MPI_Status status;

   // Inicio do processo paralelo
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&np);
   MPI_Comm_rank(MPI_COMM_WORLD,&eu);

   numIt = atoi(argv[1]); // numero de iteracoes limite
   nomeArqPGM = argv[2];  // nome do arquivo

   if (eu == 0)  startclock = clock();

   // Distribuir numero de linas nos N processos
   lins = alocarVetorInt(np);

//##################################################
   // Carrega imagem
   if (eu == 0)  y = leituraArquivoPGM(nomeArqPGM,&lin,&col,&max);

   // Envia informações do arquivo
   MPI_Bcast(&lin,1,MPI_INT,0,MPI_COMM_WORLD);
   MPI_Bcast(&col,1,MPI_INT,0,MPI_COMM_WORLD);
   MPI_Bcast(&max,1,MPI_UNSIGNED_SHORT,0,MPI_COMM_WORLD);
```

```
if (eu == 0)  {
   ultimaLinha = alocarVetorInt(np);
   ini         = alocarVetorInt(np);

   // Calculando número de linhas.
   resto = (unsigned char)(lin%np);

   for (i = 0; i < np; i++)  {
      if (resto == 0)
         lins[i] = lin/np;
      else  {
         lins[i] = lin/np + 1;
         resto -= 1;
      }
   }

   //if (np==1) overlap = 0;
   tamSubMat = lins[0] + overlap*2; //overlapping
}

MPI_Bcast(&tamSubMat,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(lins,np,MPI_INT,0,MPI_COMM_WORLD);

if (eu == 0)  {
   // Calculando as ultimas linhas de cada sub-matriz.
   ultimaLinha[0] = lins[0]-1;
   for (n = 1; n < np; n++)
      ultimaLinha[n] = ultimaLinha[n-1] + lins[n];

   // Calculando os endereços inicial de cada sub-matriz.
   ini[0] = 0;
   for (n = 1; n < np; n++)
      ini[n] = ultimaLinha[n-1] - (overlap-1);  // overlapping
}

// Aloca memoria para matrizes
subMatEnvio = alocarVetor2B(tamSubMat*col);
subMatLocal = alocarMatriz(tamSubMat,col);
yp          = alocarMatriz(tamSubMat,col);

if (eu == 0)  {
   k=0;
   for (i=overlap; i<tamSubMat; i++)
      for (j=0; j<col; j++)
         yp[i][j] = y[i-overlap][j];

   for (n = 1; n < (np-1); n++)  { // overlapping
      k=0;
      for (i=0; i<tamSubMat; i++)
         for (j=0; j<col; j++)
            subMatEnvio[k++] = y[ini[n]+i][j];

      MPI_Send(subMatEnvio,(lins[n]+overlap*2)*col, //
               MPI_UNSIGNED_SHORT,n,10,MPI_COMM_WORLD);
   }

   k=0; // n = np
   for (i=0; i<lins[np-1]+overlap; i++)
      for (j=0; j<col; j++)
         subMatEnvio[k++] = y[ini[np-1]+i][j];

   MPI_Send(subMatEnvio,(lins[np-1]+overlap)*col,  //
            MPI_UNSIGNED_SHORT,np-1,20,MPI_COMM_WORLD);
}


// Recebe matriz enviada por 0
if (eu == np-1)
   MPI_Recv(subMatEnvio,(lins[eu]+overlap)*col,  //
            MPI_UNSIGNED_SHORT,0,20,MPI_COMM_WORLD,&status);
else
if (eu != 0) {
   MPI_Recv(subMatEnvio,(lins[eu]+overlap*2)*col,  //
            MPI_UNSIGNED_SHORT,0,10,MPI_COMM_WORLD,&status);
}

// Monta matriz recebida
```

```
   if(eu != 0)  {
      k=0;
      for(i=0; i<tamSubMat; i++)
         for(j=0; j<col; j++)
            yp[i][j] = subMatEnvio[k++];
   }
//##################################################

   // Prepara dados para 'MPI_Gatherv'
   recvcnts = alocarVetorInt(np);
   sendarray = alocarVetor2B(lins[0]*col);

   if (eu == 0) rbuf = alocarVetor2B(lin*col); // vetor de saida

   // Aloca memoria para matrizes
   x  = alocarMatriz(tamSubMat, col);
   xp = alocarMatriz(tamSubMat, col);
   x2 = alocarMatriz(tamSubMat, col);

   for(i=0; i<tamSubMat; i++)
      for(j=0; j<col; j++)
         subMatLocal[i][j] = filtroSeletivo3(yp,i,j,tamSubMat,col);

   beta = 0.78;
   n = 1; erro = 100.00;
   while( n<numIt && erro>0.001)  {  // Executar algoritmo de Polyak
      for(i = 0; i < tamSubMat; i++)  {
         for(j = 0; j < col; j++)  {
            alpha = (float)pow((float)n,-beta);
            g = (int)toeplitz1_3(x,col,i,j) - subMatLocal[i][j];
            cond = (float)x[i][j] - alpha*g;
            if(cond > 0 && cond < max)
               xp[i][j] = (unsigned short int)cond;
            else {
               if(cond > max)
                  xp[i][j] = max;
               else
                  xp[i][j] = 0;
            }
            x2[i][j] = (n*x2[i][j]+xp[i][j])/(n+1);
         }
      }
      copiaMatriz(xp, x, tamSubMat, col);
      n++;

      // calcula RMS
      for(i = 0; i < tamSubMat; i++)
         for(j = 0; j < col; j++)
            yp[i][j]=(unsigned short int)abs(subMatLocal[i][j]-toeplitz1_3(x2,col,i,j));

      erro = rms(yp,tamSubMat,col)/max;
   }

   z=0;
   for(i=0; i<lins[eu]; i++)
      for(j=0; j<col; j++)
         sendarray[z++] = x2[8+i][j]; // começa no overlapping

   for (i=0; i<np; i++)
      recvcnts[i] = lins[i]*col;

   if (eu == 0)  {
      displs = alocarVetorInt(np);

      for (i=0; i<np; i++)  {
         if (i==0)
            displs[0] = 0;
         else
            displs[i] = displs[i-1] + lins[i-1]*col;
      }
   }

   // Processo recebe todas as sub matrizes
   MPI_Gatherv(sendarray, lins[eu]*col, MPI_UNSIGNED_SHORT, rbuf,//
               recvcnts, displs, MPI_UNSIGNED_SHORT, 0, MPI_COMM_WORLD);

   if (eu == 0)  {
```

```
      escritaArquivoPGMv(rbuf,"OUT.pgm",lin,col,max); // Cria arquivo PGM de saida
      stopclock = clock();  // Termina tempo
      printf("Numero de processos: %d\n",np);
      printf("Numero de iteracoes: %d\n",n);
      printf("Erro(rms): %f\n", erro);
      printf("Tempo de exec: %f seg.\n",(stopclock-startclock)/(double)CLOCKS_PER_SEC);

      liberarVetor2B(rbuf);
      liberarVetorInt(displs);
   }

   // desaloca memoria
   liberarMatriz(tamSubMat, yp);
   liberarMatriz(tamSubMat, x);
   liberarMatriz(tamSubMat, xp);
   liberarMatriz(tamSubMat, x2);
   liberarMatriz(tamSubMat, subMatLocal);
   liberarVetor2B(sendarray);
   liberarVetor2B(subMatEnvio);
   liberarVetorInt(recvcnts);
   liberarVetorInt(lins);

   // Finaliza processo paralelo
   MPI_Finalize();

   return SUCESSO;
}
```

APÊNDICE E – MPI

MPI (*Message - Passing - Interface*)

Neste apêndice serão apresentados os conceitos, definições e características do Mpi (Message Passing Interface), além dos diversos tipos de comunicação “Point-to-Point” e coletiva, assim como os comandos específicos de cada tipo de comunicação.

Conceitos

MPI é uma especificação de biblioteca de *Message-Passing*, proposta como um padrão por um amplo comitê baseado em vendedores, implementadores e usuários. Desenvolvida para ser padrão em ambientes de computação paralela.

Características

- Especificada por um fórum internacional aberto, constituído por representantes da indústria, universidades e laboratórios do governo.
- Especificação de uma biblioteca de troca de mensagens
  - Modelo de troca de mensagens
  - Não é especificação de compilador
  - Não é um produto específico
- Apropriada para computadores paralelos, clusters e redes heterogêneas
- Projetada para permitir o desenvolvimento de bibliotecas de programas paralelos que sejam portáteis e eficientes.
- Projetada para fornecer o acesso a computadores paralelos avançados para:
  - Usuários finais
  - Projetistas de bibliotecas
  - Desenvolvedores de ferramentas
- Especificada em C, C++ , Fortran 77 e 90
- Razões para o uso do MPI
  - Padronização
  - Portabilidade
  - Desempenho
  - Funcionalidade

- Disponibilidade

- MPI é grande ou pequeno?
- MPI é grande

  Possui em torno de 150 funções:

  Funcionalidade sem complexidade
- MPI é pequeno

  Muitos programas paralelos podem ser escritos com somente 6 funções
- Tem o tamanho que for necessário:

  Usuário pode aumentar o conjunto de funções que vai usar
- Todo o paralelismo é explícito: o programador é responsável pela correta identificação do paralelismo e sua implementação.
- O programador tem que:
  - Inicializar os processos
  - Sincronizar os processos
  - Alocar e liberar buffer para as mensagens
  - Etc.

Utilização do MPI

- Diretiva para o MPI: contém definições e declarações necessárias para a compilação do programa

  #include <mpi.h>

  #include “mpif.h”
- MPI_ : definição para constantes e funções

  – C : 1ª letra maiúscula e o resto minúscula, palavras reservadas em maiúsculas.

- Códigos de erros

  – Fortran: a maioria das funções tem como argumento a variável de retorno de erro

  – C: a maioria das funções retorna um código de erro
- Durante a fase de execução todas as rotinas que não são do MPI executam localmente.

Inicialização e finalização

- Inicialização do MPI

```
int MPI_Init (int *argc, char ***argv)
MPI_INIT (ierror) integer ierror
```

– Nenhuma função MPI pode ser chamada antes do MPI_Init

- Finalizando o MPI

```
int MPI_Finalize (void)
MPI_FINALIZE (ierror) integer ierror
```

– Nenhuma função MPI pode ser chamada após o MPI_Finalize

Comunicadores

- Noção de grupo e contexto combinados em um objeto chamado comunicador ou *comm*
- O comunicador aparece como argumento na maioria das funções de comunicação coletiva e ponto-a-ponto
- Cada processo MPI pertence a um ou mais grupos / comunicador
- Inicialmente todos os processos pertencem ao grupo MPI_COMM_WORLD
- Cada processo possui um identificador chamado *rank* em relação a um grupo
- *rank* varia de 0 a n - 1, sendo n = número de processos do grupo
- Grupos adicionais podem ser criados pelo programador

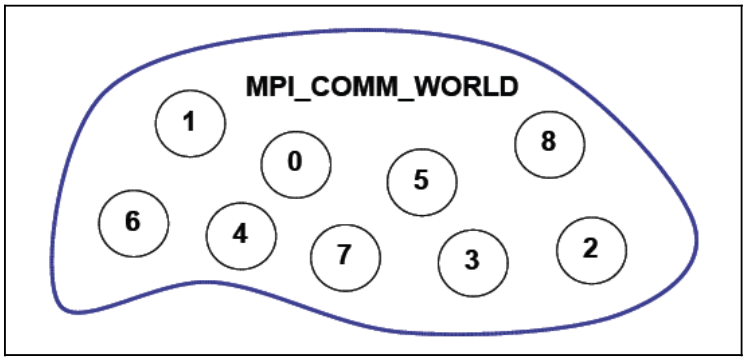

Figura E.1: Grupo principal.

- Determinando número de processos:

```
int MPI_Comm_size (MPI_Comm comm, int *size)
MPI_COMM_SIZE (comm, size, ierror) integer comm, size, ierror
```

– *size* : número de processos no grupo comm

- Determinando o identificador do processo:

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
MPI_COMM_RANK (comm, rank, ierror) integer comm, rank, ierror
```

– 0 ≤ *rank* ≤ *size* – 1

Comunicação ponto-a-ponto

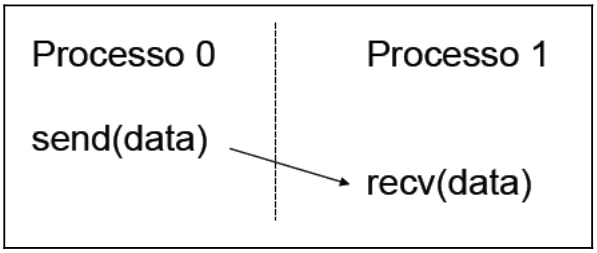

Figura E.2: Comunicação ponto-a-ponto

- Para onde a mensagem está sendo enviada
- Onde está o dado
- Qual o tamanho da mensagem
- Qual o tipo da mensagem
- Como o receptor identifica a mensagem
- Mensagem = dado + envelope
- Especificação da mensagem:

  endereço – localização da memória onde a mensagem começa

  contador – número de itens contidos na mensagem

  datatype – tipos do MPI

  fonte – *rank* do processo que envia

  destino – *rank* do processo que recebe

  tag – identificador da mensagem

  comm – comunicador utilizado
- Envia

```
int MPI_Send (void *end, int count, MPI_Datatype datatype,int dest, int tag,
MPI_Comm comm)
MPI_SEND (end, count, datatype, dest, tag, comm, ierror) <type> end(*) integer
count, datatype, dest, tag, comm, ierror
```

- Recebe

```
int MPI_Recv (void *end, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
MPI_RECV (end, count, datatype, source, tag, comm, status, ierror) <type> end(*)
integer count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror
```

- Tipos de dados usualmente utilizados no C/C++, Fortran
- Facilidade no uso de ambientes heterogêneos onde tipos variam entre máquinas
- Dados dispersos devem ser agrupados

  – armazenagem e acesso a matrizes pelos programas

  – estruturas gerais dos dados
- Outros tipos de dados podem ser criados pelo programador

Tipos de dados MPI (Datatypes)

- Principais Datatypes em C

  MPI_CHAR

  MPI_SHORT

  MPI_INT

  MPI_LONG

  MPI_FLOAT

  MPI_DOUBLE

  MPI_PACKED
- Principais Datatypes em Fortran

  MPI_CHARACTER

  MPI_INTEGER

  MPI_REAL

  MPI_DOUBLE_PRECISION

  MPI_COMPLEX

  MPI_LOGICAL

  MPI_PACKED

Enviando e recebendo mensagens

- Tags : permitem ao programador a manipulação de ordem nas mensagens

- Na recepção de mensagens:

  – MPI_ANY_SOURCE : permite a recepção de mensagem originada em qualquer fonte

  – MPI_ANY_TAG : permite a recepção de qualquer tipo
- Os parâmetros MPI_Status (C) e status (Fortran) são compostos de:

  MPI_SOURCE

  MPI_TAG

  MPI_ERROR
- Determinando a quantidade de dados de um determinado tipo que foi recebida

```
int MPI_Get_count (MPI_Status* status,  MPI_Datatype datatype, int* count_ptr)
MPI_Get_count (status, datatype, count, ierror) integer status(*), datatype, count,
ierror
```

Comunicação coletiva

- Comunicação é realizada entre um grupo de processos em um comunicador
- Mensagens não possuem *tag*
- As comunicações são bloqueantes
- Todos os processos chamam a função
- Três tipos:

  – Movimentação de dados

  – Computação coletiva

  – Sincronização
- Principais comandos:

  Broadcast

  Reduce

  Allreduce

  Scan

  Gather

  Scatter

- Broadcast : envio / recebimento de mensagem originada no processo “root” para todos os outros processos agrupados no comunicador

int MPI_Bcast (void *endereço, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

MPI_Bcast (endereço, count, datatype, root, comm, ierror) <type> endereço(*) integer count, datatype, root, comm, ierror

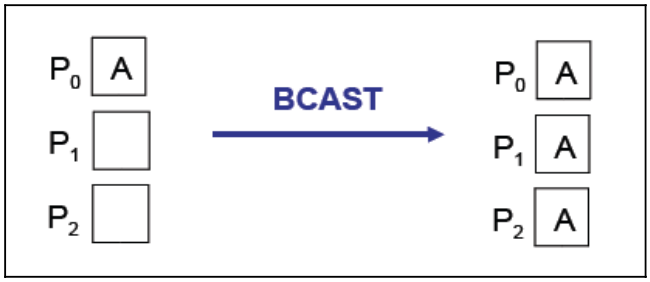

Figura E.3: Comando Broadcast.

– *root* é o *rank* da fonte da mensagem

– MPI_Bcast no envio tem que corresponder a MPI_Bcast na recepção

- Reduce : Todos os processos enviam dados, que são operados no destino e colocados no endereço de resultado

int MPI_Reduce (void *sendend, void *recvend, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) MPI_Reduce (sendend, recvend, count, datatype, op, root, comm, ierror) <type> sendend(*), recvend(*) integer count, datatype, op, root, comm, ierror

– *sendend* e *recvend* não podem ser o mesmo.

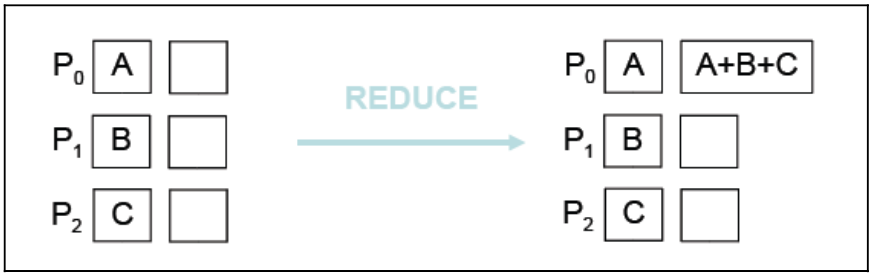

Figura E.4: Exemplo de Reduce.

- Principais operadores de redução

MPI_MAX

MPI_MIN

MPI_SUM

MPI_PROD

MPI_LAND

MPI_LOR

MPI_LXOR

- Allreduce : Combina valores de todos os processos do comunicador e retorna o resultado no endereço de saída de todos os processos.

```
int MPI_Allreduce (void *sendend, void *recvend, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_Allreduce (sendend, recvend, count, datatype, op, comm, ierror) <type> sendend(*), recvend(*) integer count, datatype, op, comm, ierror
```

Figura E.5: Exemplo de Allreduce.

- Scan : calcula resultados parciais de operações em dados armazenados no conjunto dos processos.

```
int MPI_Scan (void *sendend, void *recvend, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
MPI_Scan (sendend, recvend, count, datatype, op, comm, ierror) <type> sendend(*), recvend(*) integer count, datatype, op, comm, ierror
```

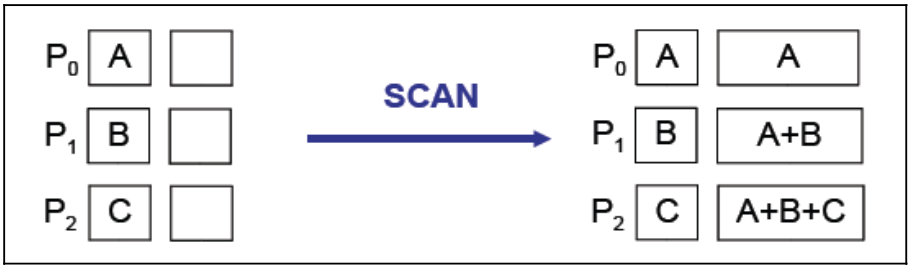

Figura E.6: Exemplo de operação de soma com Scan.

- Gather : Reúne o conteúdo dos *sendend* dos processos no *recvend* do processo "root"

int MPI_Gather (void *sendend, int sendcount, MPI_Datatype sendtype, void *recvend, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) MPI_Gather (sendend, sendcount, sendtype, recvend, recvcount, recvtype, root, comm, ierror) <type> sendend(*), recvend(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

– Os parâmetros do *recvend* só tem significado no "root"

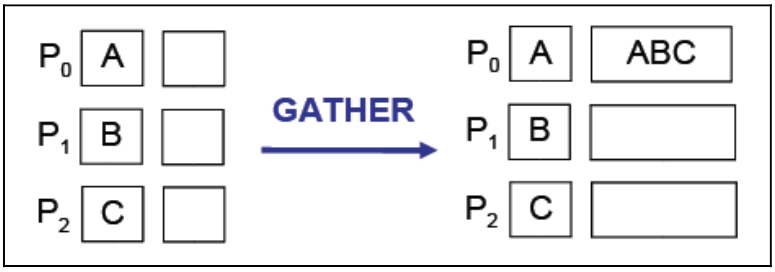

Figura E.7: Exemplo de Gather.

- Scatter : Distribui o conteúdo do *sendend* do processo "root" pelos *recvend* dos processos.

int MPI_Scatter (void *sendend, int sendcount, MPI_Datatype sendtype, void *recvend, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) MPI_Scatter (sendend, sendcount, sendtype, recvend, recvcount, recvtype, root, comm, ierror) <type> sendend(*), recvend(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

– Os parâmetros do *sendend* só tem significado no "root"

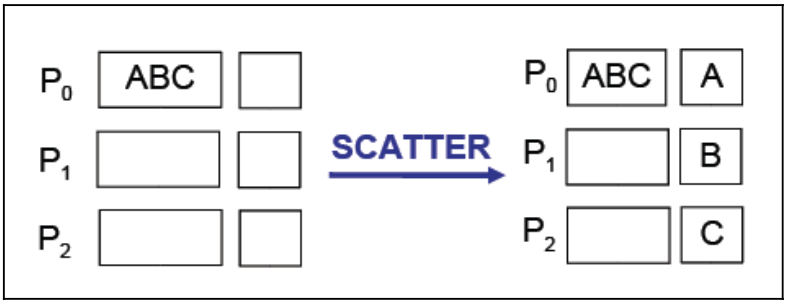

Figura E.8: Exemplo de Scatter.

- MPI_Allgather : reúne o conteúdo dos *sendend* nos *recvend* de cada processo.

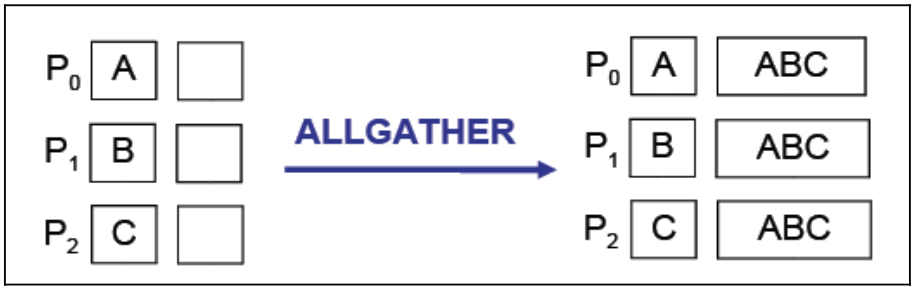

Figura E.9: Exemplo de Allgather.

- Alltoall scatter/gather (total exchange)

  MPI_Alltoall : envio dos *sendend* para cada processo, incluindo ele mesmo e recebimento do *recvend* de cada processo.

  MPI_Alltoallv : cada processo pode enviar / receber quantidades diferentes de dados.

- MPI_Barrier : sincronização dos processos Cada processo fica bloqueado até que todos os processos chamem a mesma função.

```
int MPI_Barrier ( MPI_Comm comm)
MPI_Barrier ( comm, ierror) integer comm, ierror
```

Controle de Tempo

- Obtendo o tempo de execução de programas MPI:

```
double MPI_Wtime (void)
DOUBLE PRECISION MPI_WTIME ()
```

- MPI_Wtime retorna o tempo decorrido desde a primeira chamada no programa

- Resolução do clock:

```
double MPI_Wtick (void)
DOUBLE PRECISION MPI_WTICK ()
```

- Exemplo de procedimento de tomada de tempo:

  1. Sincroniza os processos
  2. Inicia o contador

     /* código do programa */

  3. Sincroniza os processos
  4. Para o contador
  5. Calcula o tempo total gasto (elapsed time)

- Aplicar conceitos de desempenho:

  Speed-up

  Eficiência

Resumo dos principais comandos

MPI_Init() : Inicializa a comunicação.

MPI_Comm_size() : Determina o número total de processos na comunicação.

MPI_Comm_rank() : Determina o número do processo local na comunicação.

MPI_Send() : Envia dados a outro processo.

MPI_Recv() : Recebe dados de outro processo.

MPI_Get_count() : Retorna o número de elementos recebidos.

MPI_Bcast() : Retransmite uma mensagem desde o processo “root”.

MPI_Reduce() : Reduz valores de todos os processos num único valor.

MPI_Allreduce() : Reduz valores de todos os processos e distribui o resultado em todos eles.

MPI_Scan() : Calcula resultados parciais de operações em dados armazenados no conjunto dos processos.

MPI_Gather() : Reúne o conteúdo dos *sendend* dos processos no *recvend* do processo “root”.

MPI_Scatter() : Distribui o conteúdo do *sendend* do processo raiz pelos *recvend* dos processos.

MPI_Barrier() : Para temporariamente os processos para sincronização.

MPI_Finalize() : Finaliza a comunicação.