2018 | Volume 15 | Nº 4 | Pág. 11234 a 1255

DOI http://dx.doi.org/10.15536/thema.15.2018.1234-1255.1027

CIÊNCIAS EXATAS E DA TERRA

# Uma análise de eficiência do uso das ferramentas de programação paralela OpenMP e TBB na remoção de ruídos em imagens

## *An efficiency analysis on the use of the parallel programming tools OpenMP and TBB on images noises removal*

Rodolfo Migon Favaretto[1]

## RESUMO

O uso de ferramentas de programação paralela é uma alternativa viável cada vez mais adotada em sistemas computacionais, uma vez que estas podem contribuir significativamente para o aumento do desempenho dos programas que as utilizam. Este artigo tem por objetivo analisar o impacto e a eficiência da utilização das ferramentas de programação paralela *Intel® Threading Building Blocks* – TBB e *Open Multi-Processing* – *OpenMP* em algoritmos de remoção de ruídos em imagens, por meio da aplicação do filtro da média. Foram desenvolvidos três algoritmos, um sequencial, ou seja, sem o uso de ferramentas paralelas e um para cada uma das ferramentas estudadas. Os resultados comprovam que o uso de ferramentas paralelas contribui para o aumento de desempenho deste tipo de aplicação. O algoritmo paralelo com *OpenMP* obteve o melhor desempenho, apresentando mais de 90% de eficiência em relação ao algoritmo sequencial.

***Palavras-chave:*** *Processamento de Imagens, Remoção de ruídos, Ferramentas de Programação Paralela.*

## ABSTRACT

*The use of parallel programming tools is a viable alternative increasingly adopted in computational systems, since these tools can contribute significantly to increase the performance of programs that use them. This article aims to analyze the impact of using the parallel programming tools Intel® Threading Building Blocks - TBB and Open Multi-Processing - OpenMP on image processing, with noise removal algorithms, by applying the average filter. Three algorithms were developed, one sequentially, i.e. without the use of parallel tools and one to each tool studied. The results show that the use of parallel tools contributes to the improved performance of this type of application. The OpenMP parallel algorithm had the best performance, with over 90% efficiency compared to the sequential algorithm.*

***Keywords:*** *Image Processing, Noise removal, Parallel Programming Tools.*

[1] IFSul - Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Charqueadas/RS - Brasil.

## 1. INTRODUÇÃO

O processamento de imagens é fundamental para que se possa manipular objetos gráficos com a finalidade de melhorá-los ou deixá-los visualmente possíveis de serem analisados e esse processo de manipulação, normalmente, é demorado e trabalhoso, pois envolve diversas questões complexas que envolvem a concepção de uma imagem (HONGJUN, L; CHING, Y. S, 2016).

Qualquer área onde se possa capturar uma imagem e dela obter resultados é uma área onde pode ser usado um sistema de processamento de imagens. Devido a esta vasta amplitude de áreas de aplicação, sistemas de processamento de imagens são, à *priori*, interdisciplinares. Alguns exemplos de área são: Sensoriamento Remoto, Microscopia, Medicina, Manutenção de Obras de Arte, Identificação de Impressões Digitais, Astronomia, Fotografia, Vídeo, Efeitos Especiais, entre outros (FREEMAN, K; REICHER, 2015).

No processo de aquisição, alguns ruídos são inseridos nas imagens por diversas causas, como iluminação inadequada, sujeira na lente ou poeira entre a câmera e o objeto a ser fotografado. Geralmente, os pixels com ruído aparecem como pontos com cores bem diferentes da sua vizinhança (escuros (pretos) ou saturados (brancos)). Estes pontos ruidosos podem aparecer distribuídos aleatoriamente ou de forma sistemática como listras verticais e horizontais (GONZALEZ; WOODS, 2012).

PRAJAPATI e VIJ (2011) mostram que uma das técnicas de processamento mais utilizadas para melhorar a qualidade de uma imagem digital é a aplicação de algoritmos que permitam eliminar regiões indesejáveis, causadas pelos métodos de aquisição, ou pelas condições em que a imagem foi capturada e que introduzem algum tipo de ruído na imagem. Este tipo de técnica é chamado de filtragem de imagem.

O processamento de imagens é fundamental para que se possa manipular objetos gráficos com a finalidade de melhorá-los ou deixá-los visualmente possíveis de serem analisados e esse processo de manipulação, normalmente, é demorado e trabalhoso, pois envolve diversas questões complexas que envolvem a concepção de uma imagem. Com a necessidade de se aproveitar da melhor maneira possível os recursos de processamento que um computador oferece, a fim de se obter ganhos em desempenho, buscou-se a utilização de ferramentas de programação paralela para que o computador consiga processar mais rápido e de maneira eficiente (GONZALEZ; WOODS, 2012).

Segundo Pilla, Santos e Cavalheiro (2009), hoje em dia, por questões de economia de energia e pela redução de danos causados ao meio ambiente, é fundamental que se tenha sistemas computacionais que consumam menos e que consigam utilizar os recursos disponíveis da melhor maneira possível. Por esses fatores, optou-se por utilizar alternativas promissoras da área de computação paralela em aplicações práticas na área de processamento de imagens que é muito utilizada nas áreas médica, meteorológica, geográfica, entre outras.

Neste contexto, o objetivo geral deste trabalho consiste em investigar o impacto da utilização das ferramentas de programação paralelas *Intel*® TBB e *OpenMP* em algoritmos de remoção de ruídos em imagens, comparando com a versão sequencial do algoritmo no que diz respeito ao desempenho e ao aproveitamento dos recursos computacionais disponíveis.

Uma das principais contribuições deste trabalho consiste na análise detalhada de duas ferramentas de programação paralela (*Intel* TBB e *OpenMP*) aplicadas em algoritmos de remoção de ruídos em imagens, mostrando com isso que existem alternativas de remoção de ruídos em imagens muito mais eficientes e com desempenho superior à atual maneira sequencial de desenvolvimento de algoritmos e ainda, por executarem em menor tempo, consomem menos energia.

## 2. CONTEXTUALIZAÇÃO TEÓRICA

### 2.1 Processamento de imagens digitais

Segundo Frery at. al. (2011) o processamento de imagens é uma forma de processamento de dados no qual a entrada e a saída é uma imagem fotográfica ou quadros de vídeos. Esse processo tem o objeto de otimizar a extração de informações e, portanto, ajudar na interpretação da imagem, que envolve a detecção e reconhecimento de elementos contidos na imagem. O processamento de imagens consiste na aplicação de vários algoritmos para chegar aos resultados pré-definidos Bastos (2010).

A primeira etapa do processamento de imagens consiste em adquirir a imagem digital com um equipamento digital. Segue-se então o pré-processamento, onde a imagem adquirida é submetida a métodos de filtragem para eliminar ruídos e/ou aumentar o contraste (OLIVEIRA et. al., 2010) preparando-a para a etapa de segmentação.

O processo de segmentação é o mais complexo e difícil de ser realizado em processamento de imagens e se não for feito com sucesso compromete todas as etapas subsequentes (GONZALEZ; WOODS, 2007).

Uma imagem é definida matematicamente como uma função f(x, y), onde x e y são coordenadas espaciais e o valor de f, para um par qualquer de coordenadas, é a intensidade do nível de cinza da imagem naquele ponto. Quando todos os valores de x, y e da intensidade de f são finitos e discretos, a imagem é chamada de imagem digital e os elementos desta imagem são chamados *pixels* (GONZALEZ; WOODS, 2007).

Para o processamento computacional, de acordo com Gonzalez e Woods (2010), uma função f(x,y) precisa ser digitalizada tanto espacialmente quanto em amplitude. A digitalização das coordenadas espaciais (x,y) é denominada amostragem da imagem e a digitalização da amplitude é chamada quantização em níveis de cinza.

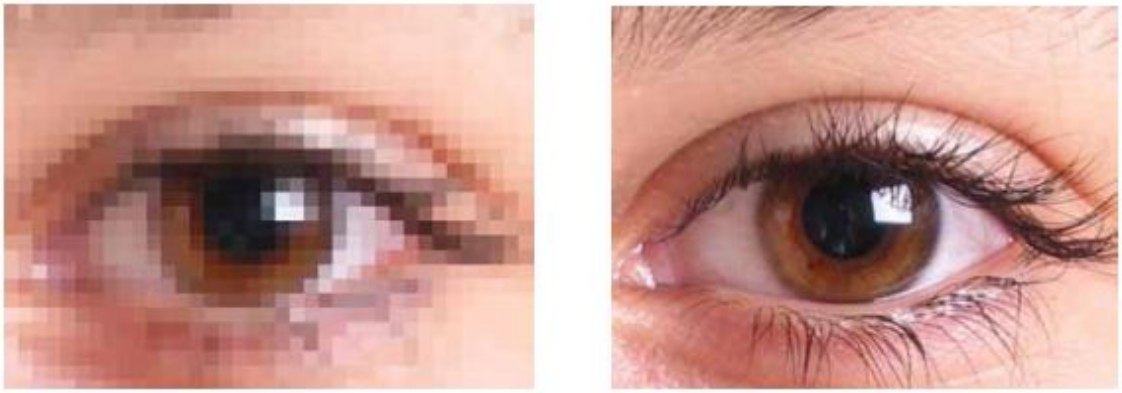
Segundo Lopes (2018) e seus colaboradores, uma imagem digital é composta por *pixels*. O termo *pixel* é uma abreviação do inglês para *picture element*, que em português significa elemento de figura. É a menor unidade de uma imagem digital, de forma quadrada, onde são descritos a cor e o brilho específico de uma célula da imagem, conforme ilustra a Figura 1. Cada *pixel* é criado quando a cor e brilho de uma dada posição na matriz são medidos e gravados como uma quantidade discreta (LEÃO, 2008).

**Figura 1:** Detalhe de pixels visíveis em uma imagem digital.

Fonte: Leão (2008).

Leão (2008) afirma que uma imagem é formada pela quantidade de *pixels* que a compõe. O tamanho da imagem digital não se refere apenas à quantidade de detalhe visível – resolução espacial, mas também ao número de cores presentes – profundidade de cor. Quanto maior a quantidade de *pixels*, maior será o tamanho da imagem e melhor a qualidade de detalhe visível da mesma, conforme é possível ver na Figura 2. O tamanho de uma imagem digital é descrito, por exemplo: 640 x 480 *pixels*, 1024 x 768 *pixels* ou quaisquer outros valores.

**Figura 2:** Diferentes tamanhos para a mesma imagem: 40x30 pixels (lado esquerdo) e 640x480 pixels (lado direito).



Fonte: Leão (2008).

Gomes e Velho (2008) mostram que a representação de uma cor em uma imagem digital pode ser feita de acordo com diversos sistemas, que são escolhidos de maneira a atender às peculiaridades de cada aplicação. Representar cores em um determinado sistema significa reduzir o espaço espectral de cor para um sistema de coordenadas de dimensão finita.

Em 1931, a CIE - *Commission Internationale de L'eclairage* - organização internacional responsável pela padronização nas áreas relacionadas à iluminação - criou o sistema padrão RGB (GOMES; VELHO, 2008). Baseado nas propriedades dos fotorreceptores existentes no olho humano, sensíveis a radiações eletromagnéticas nas frequências baixa, média e alta, o RGB define uma cor em função de três componentes primárias: vermelho (*Red*), verde (*Green*) e azul (*Blue*).

Cada cor é definida pela quantidade de vermelho, verde e azul que a compõem. Por conveniência, os arquivos digitais atuais usam números inteiros entre 0 e 255 para especificar estas quantidades. O número 0 indica ausência de intensidade e o número 255 indica intensidade máxima.

Neste contexto, cada cor no sistema RGB é identificado por uma tripla ordenada (R, G, B) de números inteiros, com $0 \leq R \leq 255$, $0 \leq G \leq 255$ e $0 \leq B \leq 255$. Sendo assim, pode-se associar cada cor do sistema RGB com pontos com coordenadas inteiras de um cubo com aresta de tamanho 255.

## 2.2. Ruídos em imagens

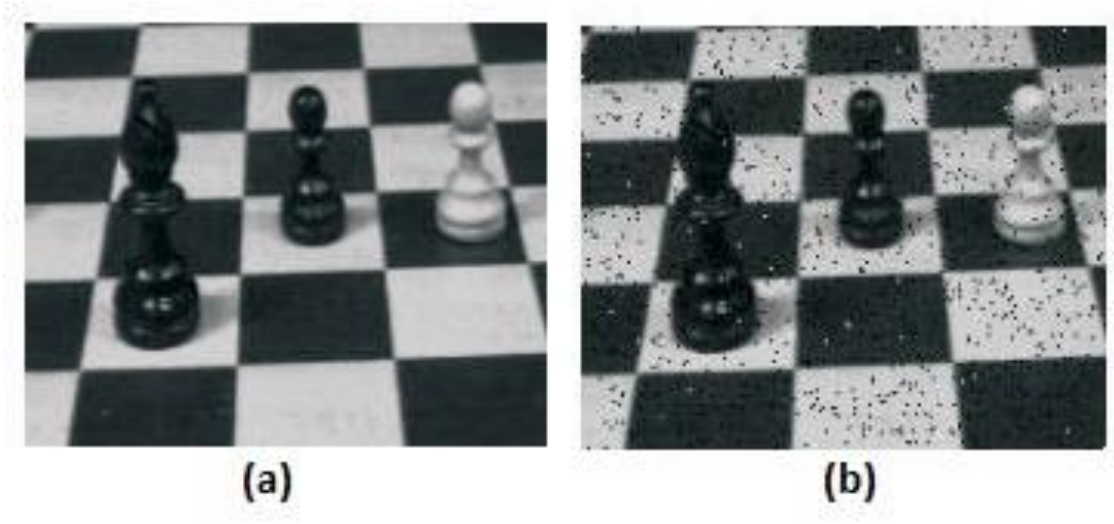
No processo de geração de imagens, alguns ruídos são inseridos. Geralmente, os pixels com ruído aparecem como pontos com níveis de cinza bem diferentes da sua vizinhança (escuros (pretos) ou saturados (brancos)). Estes pontos ruidosos podem aparecer distribuídos aleatoriamente ou de forma sistemática como listras verticais e horizontais (GONZALEZ; WOODS, 2010). Os tipos mais comuns são Sal e Pimenta e Gaussiano. Ambos são descritos a seguir.

### 2.2.1. Ruídos Sal e Pimenta

Ruído impulsivo, ou ruído sal e pimenta (do inglês *salt and pepper noise*), em imagens digitais é geralmente proveniente do processo de transmissão de dados (GEORGE, S; JOSEPH, S; 2017).

Segundo as pesquisas de Plataniotis e Venetsanopoulos (2008), as imagens com ruído, em geral, fornecem informações errôneas durante o processo de aquisição de informações, prejudicando, dessa forma, as demais etapas de processamento. Os *pixels* corrompidos ou são alterados para o valor máximo, ou tem alguns bits alterados, causando uma diferença brusca de tons entre este *pixel* e seus vizinhos. A Figura 3 ilustra o ruído sal e pimenta (SONG; SUDIRMAN; MERABTI, 2012).

**Figura 3:** Imagem original (a) e com Sal e Pimenta (b).

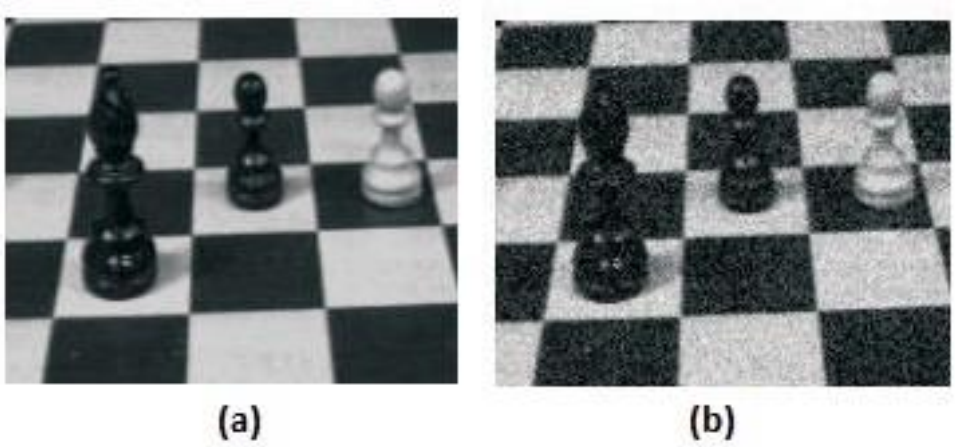



Fonte: Dorini e Rocha (2007).

### 2.2.2. Ruído Gaussiano

Esse tipo de ruído é muito comum em imagens, recebe este nome porque segue a distribuição de frequência de Gauss. Há maior probabilidade de ocorrência do ruído gaussiano nos tons de cinza próximos ao valor médio e menor probabilidade nos tons de cinza máximos e mínimos (ARÊDES, 2009). A Figura 4 ilustra esse tipo de ruído.

**Figura 4:** Imagem original (a) e com ruído Gaussiano (b).




Fonte: Dorini e Rocha (2007).

## 2.3. Filtros em imagens

Prajapati e Vij (2011) e Thakur e Maurya (2017) mostram que uma das técnicas de processamento mais utilizadas para melhorar a qualidade de uma imagem digital é a aplicação de algoritmos que permitam eliminar regiões indesejáveis, por causa de métodos de aquisição, ou das condições em que a imagem foi capturada, que introduzem algum tipo de ruído na imagem. Este tipo de técnica é chamado de filtragem de imagem.

De acordo com Santos (2018), as técnicas de filtragem de imagens são transformações *pixel* a *pixel*, que não dependem apenas do nível de cinza de um determinado *pixel*, mas também do valor dos níveis de cinza dos *pixels* vizinhos, na imagem. O processo de filtragem é feito utilizando matrizes denominadas máscaras, as quais são aplicadas sobre a imagem, esse processo também é conhecido como convolução (PRAJAPATI; VIJ, 2011).

A ideia de máscara com os *pixels* da vizinhança considera os pixels ao redor da posição x, y que representa um *pixel* qualquer em uma imagem. Esta vizinhança é definida por uma região quadrada (ou retangular) e de tamanho (lado) impar.

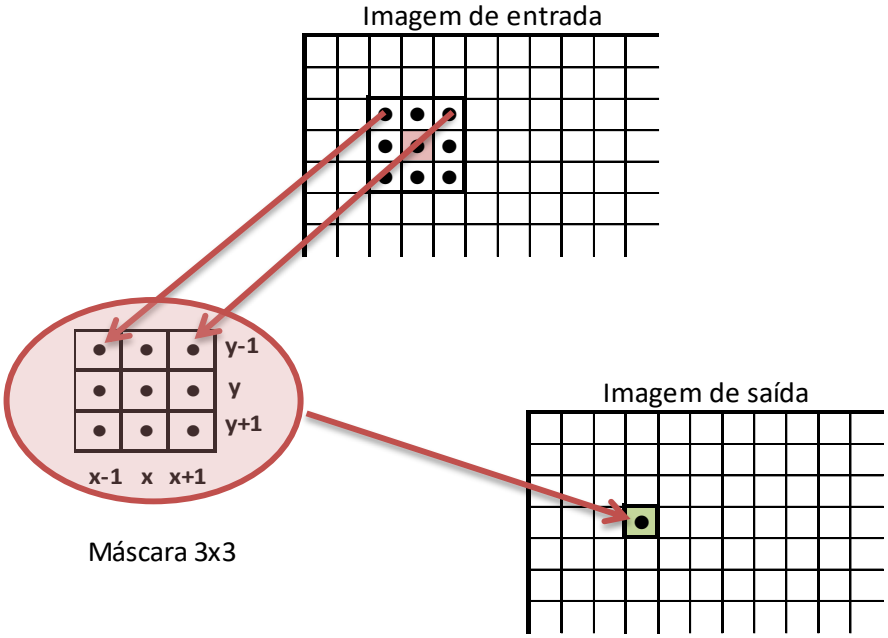
De acordo com Gomide e Araújo (2009), os filtros em imagens podem ser: (i) lineares, que suavizam, realçam detalhes da imagem e minimizam efeitos de ruído, sem alterar o nível médio de cinza da imagem, e (ii) não-lineares, que aplicam transformações sem o compromisso de manterem o nível médio de cinza da imagem original. O filtro da média é um filtro linear e foi utilizado no desenvolvimento deste trabalho.

Esse filtro refere-se a uma redução das frequências que é responsável pelos detalhes finos na imagem. Uma maneira mais fácil de se criar um filtro da média, é realizada quando se utiliza uma máscara 3x3, com pesos numéricos iguais a 1, com isto, o retorno trazido pela máscara torna-se a soma dos níveis de cinza dos pixels. Porém, existe a possibilidade do valor deste resultado ser grande o suficiente para ultrapassar o limite máximo da escala, e para que isto não ocorra, este valor é normalizado por 9 (isto é, a soma resultante é dividida por 9) o que é equivalente ao tamanho da máscara (3x3). Com isso, o resultado torna-se a média dos tons de cinza contidos na vizinhança de cada *pixel* (ROMAN, 2010).

Ainda segundo Roman (2010), há a possibilidade de se utilizar outras máscaras, com maiores dimensões (5x5, 7x7), mas a resposta sempre será normalizada através de seus tamanhos (uma máscara 7x7 será normalizada por 49, por exemplo).

Em outras palavras, o filtro da média utiliza uma máscara que percorre a imagem e substitui cada *pixel* da imagem pela média de seus vizinhos. O objetivo desse filtro não é de eliminar o ruído e sim suavizá-lo, a fim de que se torne imperceptível. (ROMAN, 2010). A Figura 5 ilustra a aplicação de uma máscara de tamanho 3x3 em uma imagem.

**Figura 5:** Ilustração da aplicação de uma máscara.




O exemplo apresentado na Figura 5, com base na ideia de Roman (2010), mostra uma máscara de dimensão 3x3, O elemento que está na posição (x, y) da imagem receberá um novo valor que será calculado a partir dos demais elementos da máscara (que neste caso variam de x-1 a x+1 e de y-1 a y+1). O cálculo do filtro da média pode ser representado através da equação:

$$\boldsymbol{Média} = \frac{\mathbf{1}}{\boldsymbol{e}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{e}} \boldsymbol{p(i)} \qquad (1)$$

Onde **e** é o número de elementos na máscara, **p** é o valor numérico do pixel da imagem e **i** é o pixel a ser calculado.

Para manipular os pixels de uma imagem, são necessárias bibliotecas de manipulação de imagens, como é o caso da biblioteca OpenCV (*Open Source Computer Vision*).

Segundo Marengoni e Stringhini (2011), o OpenCV implementa uma variedade de ferramentas de interpretação de imagens, indo desde operações simples como um filtro de ruído, até operações complexas, tais como a análise de movimentos, reconhecimento de padrões e reconstrução em 3D. Neste trabalho, a biblioteca OpenCV foi utilizada para fazer a leitura dos valores dos níveis de cinza de cada pixel das imagens com ruídos, gerando uma matriz numérica que serviu de entrada para os algoritmos de remoção de ruídos desenvolvidos.

## 2.4. Sistemas paralelos

De acordo com Carissimi, Oliveira e Toscani (2008), um processo é definido como um programa em execução. Desta forma, um algoritmo de remoção de ruídos em imagens, quando executado, passa a ser um processo. Todo processo possui um fluxo de execução. Por sua vez, uma *thread* nada mais é do que um fluxo de execução. Em sistemas paralelos, os programas são executados em processadores paralelos para alcançar altas taxas de desempenho e, geralmente, existem tantos processos quanto processadores, tentando resolver um problema de forma mais rápida ou um problema maior na mesma quantidade de tempo (ANDREWS, 2009).

Segundo Pilla, Santos e Cavalheiro (2009), no processamento dos computadores de primeira geração, chamado de processamento sequencial, a execução de um programa se dá em um único processador ou Unidade Central de Processamento - CPU, com as instruções sendo processadas uma de cada vez.

O processamento paralelo, ao contrário do sequencial, é realizado pela execução simultânea de instruções de programas que foram alocados em múltiplos processadores com o simples objetivo de rodar um programa em menos tempo ou conseguir resolver problemas mais complexos e de maior dimensão. De uma forma simples, o processamento paralelo pode ser definido como o uso simultâneo de vários recursos computacionais (arquiteturas paralelas com diversos processadores ou núcleos de processamento) de forma a reduzir o tempo necessário para resolver um determinado problema (BOTOR, 2017).

Os principais desafios da programação paralela, decorrentes da utilização de arquiteturas paralelas, são: (i) desenvolvimento de algoritmos paralelos eficientes, (ii) gerenciamento de recursos e de dados para as tarefas paralelas, (iii) utilização de mecanismos de comunicação e balanceamento de carga e o (iv) gerenciamento da complexidade de um programa paralelo (JAQUIE, 2009). Ainda segundo Jaquie (2009), como tentativa de solucionar esses problemas surgem as ferramentas de auxílio à construção de programas paralelos, permitindo uma fácil manutenção e apoiando a documentação desses programas. Neste trabalho, foram investigas duas ferramentas de programação paralela: (i) *Threading Building Blocks* – TBB e *OpenMP*, as quais foram utilizadas na paralelização do algoritmo de remoção de ruídos em imagens e são descritas a seguir.

### 2.4.1. Threading Building Blocks

A ferramenta *Threading Building Blocks* (TBB) foi desenvolvida pela empresa Intel, cujo objetivo é fornecer uma interface para o programador para explorar maior desempenho em ambientes multicores, ou seja, com mais de um núcleo de processamento de dados. Representando assim, alto nível de abstração baseado no paralelismo de tarefas, abstraindo detalhes e mecanismos de threads para fornecer escalabilidade e desempenho (REINDERS, 2007). TBB é uma biblioteca de programação escrita em C++ que fornece *templates* que possibilitam o desenvolvimento de aplicações com diversos padrões de projeto paralelos em construções de alto nível, essencialmente baseados em paralelismo de dados. Além disso, fornece uma interface para programação de tarefas e um escalonador de tarefas que dá suporte a execução de todas as construções de TBB que geram unidades de trabalho concorrentes (STPICZYNSKI, P; 2018).

### 2.4.2. OpenMP

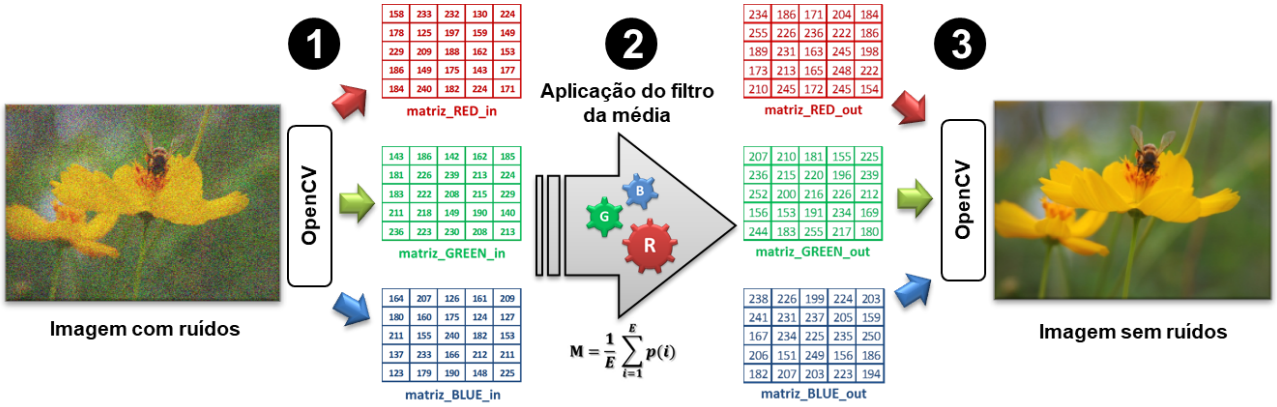
A ferramenta *Open Multi-processing – OpenMP* (CHANDRA et al., 2011) é uma interface de programação que oferece diversos recursos para a criação, desenvolvimento e execução de programas *multithread*. O suporte a esta API encontra-se implementado em C/C++ e Fortran sobre diferentes sistemas computacionais, incluindo plataformas Unix e Microsoft Windows. É composta por uma série de recursos, tais como: diretivas de compilação, chamadas a funções de biblioteca e variáveis de ambiente (SCHWARZROCK, J; 2018).

Uma aplicação com *OpenMP* utiliza o modelo de programação *fork-and-join*, com uma *thread* principal sendo criada no início do programa e a mesma executando sozinha até localizar uma diretiva que permita uma execução paralela, voltando a thread principal a executar sozinha após o termino da região paralela (MARONGIU; BURGIO; BENINI, 2011).

## 3. SOLUÇÃO PROPOSTA

Para efetuar a remoção dos ruídos de uma imagem, o algoritmo desenvolvido passa por três passos distintos, são eles: (i) aquisição dos valores de entrada, (ii) aplicação do filtro da média e (iii) geração da imagem final. Cada um desses passos está descrito na Figura 6.

**Figura 6:** Visão geral da solução proposta.




O primeiro passo do algoritmo é a captação dos valores dos níveis de cinza da imagem com ruídos. Para essa captura, foi utilizada a ferramenta OpenCV, através dessa ferramenta é possível transformar uma imagem em uma matriz numérica com os valores dos níveis de cinza de cada *pixel*. Por se tratar de uma imagem colorida, optou-se por utilizar imagens que fazem uso do padrão de cores RGB, por ser um dos padrões mais comuns para serem utilizados em imagens. Neste caso, por tratar-se do padrão RGB, faz-se necessário o uso de três matrizes diferentes, para armazenar os níveis de cinza de cada componente (*Red*, *Green* e *Blue*).

Após a execução do primeiro passo do algoritmo, três matrizes são criadas. A matriz *matriz_RED_in* contém os valores dos níveis de cinza referentes a cor vermelha, a matriz *matriz_GREEN_in* contém os valores dos níveis de cinza referentes a cor verde e a matriz *matriz_BLUE_in* contém os valores dos níveis de cinza referentes a cor azul da imagem. Estas três matrizes com os valores dos níveis de cinza da imagem com ruídos servem de entrada para o segundo passo do algoritmo, aplicação do filtro da média.

O segundo passo do algoritmo de remoção de ruídos consiste em aplicar o filtro da média sobre as três matrizes de níveis de cinza capturadas no passo anterior. Após a aplicação desse filtro, três novas matrizes são criadas. A matriz *matriz_RED_out* contém os novos valores dos níveis de cinza referentes a cor vermelha, a matriz *matriz_GREEN_out* contém os novos valores dos níveis de cinza referentes a cor verde e a matriz *matriz_BLUE_out* contém os novos valores dos níveis de cinza referentes a cor azul. Estas três novas matrizes contém os valores dos níveis de cinza já com a remoção dos ruídos, pois são resultantes da aplicação do filtro da média e servem de entrada para o terceiro e último passo do algoritmo, a geração da imagem final.

O último passo é responsável por compor a imagem final, com a remoção dos ruídos e salvá-la em um arquivo no computador. Para fazer essa conversão das matrizes com os níveis de cinza para a imagem final, novamente é utilizada a ferramenta OpenCV, que através dos valores de cada componente RGB consegue gerar uma nova imagem e salvá-la no disco. Este filtro não realiza a remoção total dos ruídos na imagem, mas os ameniza significativamente.

Estes três passos foram implementados em um algoritmo sequencial e, posteriormente, foi paralelizado em duas versões, uma com a ferramenta TBB e outra com OpenMP, os quais são descritos na seção a seguir.

### 3.1. Algoritmos desenvolvidos

Para o desenvolvimento do código, optou-se pela utilização da linguagem de programação C++, que é uma linguagem de programação multi-paradigma e de uso geral. A linguagem combina características de linguagens de alto e baixo níveis. Desde os anos 1990 é uma das linguagens comerciais mais populares, sendo bastante usada também na academia por seu grande desempenho. (ROMAN, 2010). A versão sequencial do algoritmo foi desenvolvida em C++ juntamente com OpenCV e descreve exatamente os passos ilustrados na Figura 6.

A versão paralela com TBB, apresentada na Figura 7, também utiliza C++ como base, onde são inseridos trechos de códigos da biblioteca TBB para prover o paralelismo ao código. No início do código (linhas 1-2 da Figura 7), são incluídas as bibliotecas necessárias para a utilização das funcionalidades da ferramenta Intel TBB. A biblioteca "*parallel_for.h*" é um *template* que contém métodos ideais para serem utilizados quando a aplicação tiver partes específicas de código sobre as quais se deseja interagir ou se deseja executar um número conhecido de vezes o mesmo código, como é o caso do processamento das imagens, pois se conhece a largura e a altura de cada uma.

A classe *removeNoise{ }*, apresentada entre as linhas 4-33 da Figura 7, recebe como parâmetros a posição do *pixel* inicial da imagem e também a altura, em pixels, da imagem *(blocked_range<int>(0,img->height))*, para que cada linha seja transformada em uma tarefa. O outro parâmetro (*img->width*) diz respeito à largura, também em pixels, da imagem, enviado para o método junto com a chamada da classe.

A classe *removeNoise*{ }, transforma cada uma das linhas da imagem em uma tarefa, e aplica o filtro da média para cada uma dos pixels que compõem essa determinada tarefa. Dessa maneira, os núcleos de processamento consomem essas tarefas uma a uma, quanto mais núcleos de processamento, mais tarefas serão executas e paralelo (ao mesmo tempo).

**Figura 7:** Classe principal do algoritmo paralelo com TBB.

```
1  #include <tbb/parallel_for.h>
2  #include <tbb/blocked_range.h>
3  [...]
4  class removeNoise{
5      int largura;
6  public:
7      void operator()(blocked_range<int> line) const {
8          for (int i = line.begin(); i != line.end(); ++i) {
9              for (int j = 0; j < largura; ++j) {
10                 int somaR, somaG, somaB, cont;
11                 somaR = somaG = somaB = cont = 0;
12                 if (i >= int(MASC/2) && i != line.end()){
13                     if (j >= int(MASC/2) && j <= largura -(int(MASC/2))){
14                         for(int l=i-(MASC/2); l<=i+(MASC/2); l++){
15                             for(int c=j-(MASC/2); c<=j+(MASC/2); c++){
16                                 somaR += matriz_RED_in[l][c];
17                                 somaG += matriz_GREEN_in[l][c];
18                                 somaB += matriz_BLUE_in[l][c];
19                                 cont++;
20                             }
21                         }
22                         matriz_RED_out[i][j] = somaR/cont;
23                         matriz_GREEN_out[i][j] = somaG/cont;
24                         matriz_BLUE_out[i][j] = somaB/cont;
25                     }
26                 }
27             }
28         }
29     }
30     removeNoise(int w){
31         largura = w;
32     }
33 };
```

A versão paralela com *OpenMP* foi desenvolvida de maneira semelhante, também em C++, adicionando-se apenas as diretivas da biblioteca para tornar o código paralelo. Trechos desta versão do algoritmo são apresentadas na Figura 8. Para que se possa utilizar as funções da ferramenta *OpenMP*, deve-se fazer a inclusão da biblioteca "#include <omp.h>" (linha 1). É muito simples de se trabalhar com essa ferramenta de programação paralela, neste caso, para que se possa paralelizar um laço de repetição, basta a diretiva "#pragma omp parallel for" descrito na linha 6. O construtor paralelo "#pragma omp parallel" é a diretiva mais importante do OpenMP, uma vez que é o responsável pela indicação da região do código que será executada em paralelo. Se esse construtor não for especificado o programa será executado de forma sequencial.

Quando essa linha de código (linha 6) é executada, a própria ferramenta se encarrega de criar o tamanho das tarefas a serem executas, realiza a distribuição da carga de trabalho entre os núcleos de processamento disponíveis no computador e realiza a gestão desse ambiente onde as tarefas estão sendo executas. Se um processador acabar de executar sua carga de trabalho e ficar ocioso, a própria ferramenta se encarregar de "roubar" parte das tarefas de um processador que estiver sobrecarregado e distribuir entre os processadores ociosos ou pouco sobrecarregados.

**Figura 8:** Trechos do código do algoritmo paralelo com OpenMP.

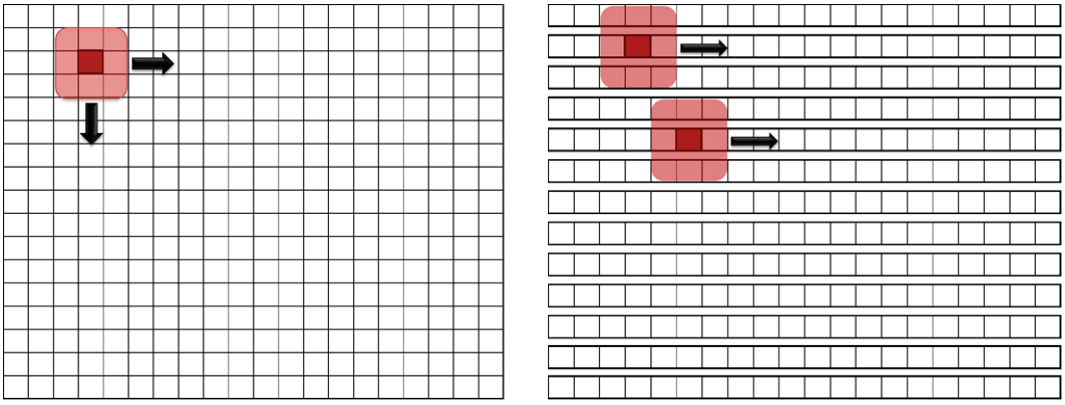
```
1  #include <omp.h>
2  [...]
3  int main(void){
4      [...]
5      time_log_start();
6      #pragma omp parallel for
7      for(i=0; i<ALTURA; i++){
8          for(j=0; j<LARGURA; j++){
9              somaR = somaG = somaB = cont = 0;
10             if (i >= int(MASC/2) && i <= ALTURA -(int(MASC/2))){
11                 if (j >= int(MASC/2) && j <= LARGURA -(int(MASC/2))){
12                     for(int l=i-(MASC/2); l<=i+(MASC/2); l++){
13                         for(int c=j-(MASC/2); c<=j+(MASC/2); c++){
14                             somaR += matriz_RED_in[l][c];
15                             somaG += matriz_GREEN_in[l][c];
16                             somaB += matriz_BLUE_in[l][c];
17                             cont++;
18                         }
19                     }
20                     matriz_RED_out[i][j] = somaR/cont;
21                     matriz_GREEN_out[i][j] = somaG/cont;
22                     matriz_BLUE_out[i][j] = somaB/cont;
23                 }
24             }
25         }
26     }
27     time_log_stop();
28     [...]
29     return 0;
30 }
```

### 3.2 Funcionamento dos algoritmos

Na Figura 9 está ilustrada a política de escalonamento da execução dos algoritmos, onde, no algoritmo sequencial Figura 9 (a), a imagem toda é tratada com uma única tarefa, sendo assim, o processador vai aplicando o filtro da média *pixel* a *pixel*, da esquerda para a direita e de cima para baixo até chegar ao último *pixel* da imagem.

Já na execução paralela Figura 9 (b), cada uma das linhas da imagem é transformada em uma tarefa que, posteriormente, é consumida pelos núcleos de processamento. Neste trabalho foi utilizada uma arquitetura "*Dual Core*", trata-se de um processador com 2 núcleos de processamento, onde 2 tarefas podem ser executadas ao mesmo tempo. Assim que uma das tarefas acaba de ser executada, uma nova tarefa é adquirida pelo núcleo de processamento. Nesse sentido, as linhas de *pixels* que compõem a imagem, podem não ser executadas de maneira ordenada, ou seja, podem não seguir a ordem em que aparecem na imagem.

**Figura 9:** Política de execução dos algoritmos: (a) sequencial, (b) paralelo com TBB e *OpenMP*.




## 4. Resultados e Discussões

### 4.1 Metodologia utilizada

Cada versão do algoritmo (Sequencial, Paralelo com TBB e Paralelo com *OpenMP*) foi executada 30 vezes sobre a mesma imagem a fim de se obter uma média de tempo de execução confiável para ser analisada.

Foi utilizada uma imagem (representada na Figura 10) de altíssima resolução com tamanho de 8.260 x 6.000 *pixels* e, com a finalidade de investigar qual o tamanho de máscara que melhor remove o ruído das imagens preservando a sua qualidade, cada um dos algoritmos foi testado com 3 tamanhos de máscara diferentes (3x3, 5x5 e 7x7). Para cada tamanho de máscara, cada um dos algoritmos foi executado 30 vezes.

**Figura 10:** Imagem utilizada nos testes dos algoritmos.

Após as execuções dos algoritmos, os tempos coletados foram inseridos no *software* de planilha eletrônica Microsoft Office Excel, onde foram manipulados e analisados. Essas análises consistem no cálculo das métricas de desempenho adotadas neste trabalho: o cálculo do *Speed-UP* e da Eficiência.

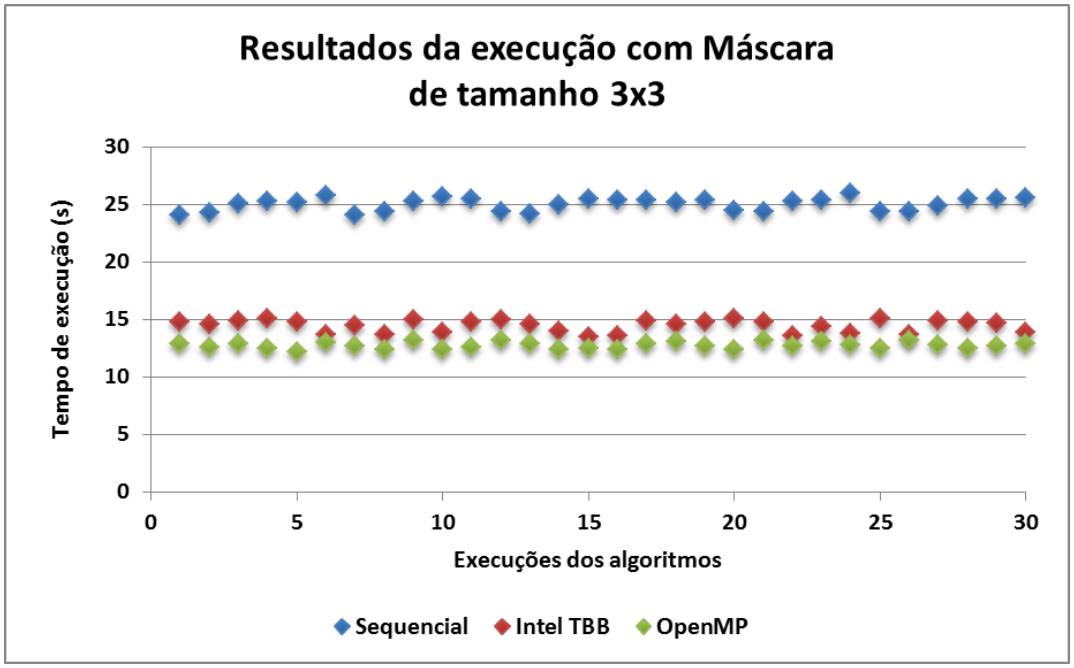
Para uma análise mais criteriosa e de fácil percepção e entendimento, foram gerados vários gráficos contendo os dados coletados e as análises realizadas, como a média dos tempos de execução de cada um dos algoritmos desenvolvidos, para cada tamanho de máscara adotado. Estas análises são apresentadas na seção a seguir.

### 4.1 Testes e resultados

Após a realização dos testes e análise dos dados coletados, os resultados para cada uma das versões dos algoritmos são apresentados seguindo o critério de tamanho de máscara utilizado, para se ter uma ideia de qual algoritmo obteve o melhor desempenho ou foi mais eficiente.

- **Mascara de tamanho 3x3:** o gráfico apresentado na Figura 11, a seguir, ilustra os 30 tempos de execução de cada uma das versões do algoritmo (Sequencial em azul, Intel TBB em vermelho e *OpenMP* em verde), fazendo uso da máscara de tamanho 3x3.

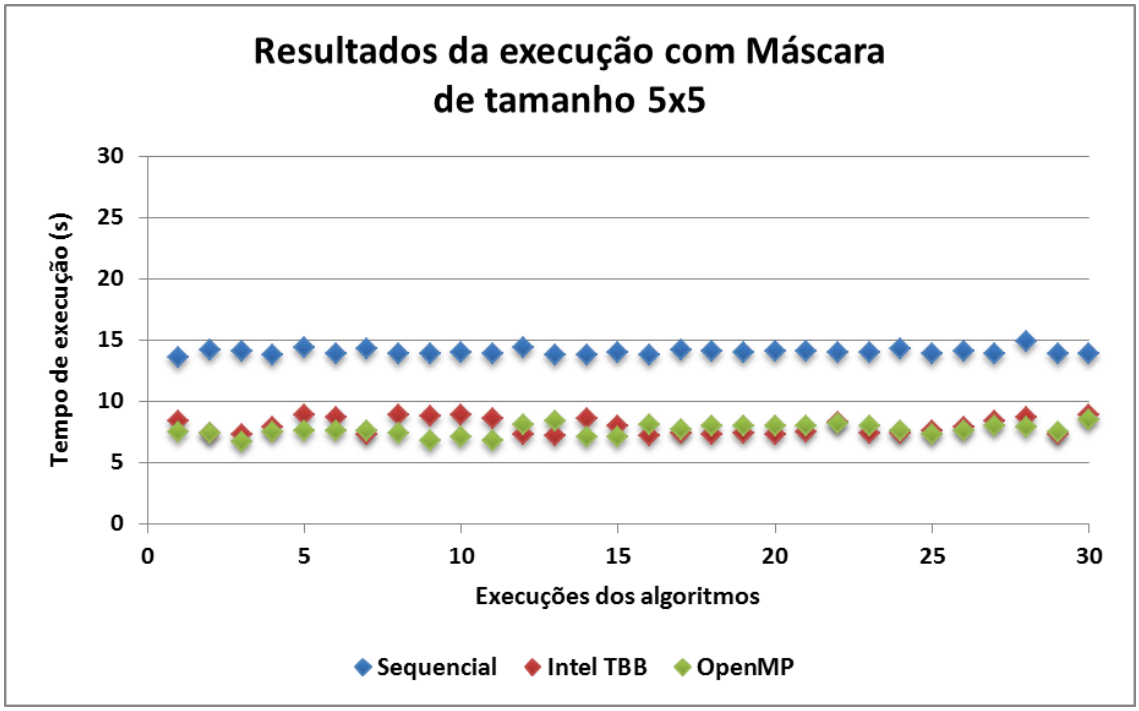
**Figura 11:** Tempos de execução com a máscara 3x3.




Pode-se observar que ambas as versões paralelas demoraram menos tempo para executar, ou seja, obtiveram um melhor desempenho. Em todas as execuções, a ferramenta OpenMP (em vermelho) se saiu melhor, pois demorou menos tempo para executar. Pode-se concluir que, dentre as ferramentas paralelas utilizadas, a ferramenta *OpenMP* é a mais indicada para trabalhar com o tamanho de máscara 3x3, sendo cerca de 13,4% melhor que a Intel TBB.

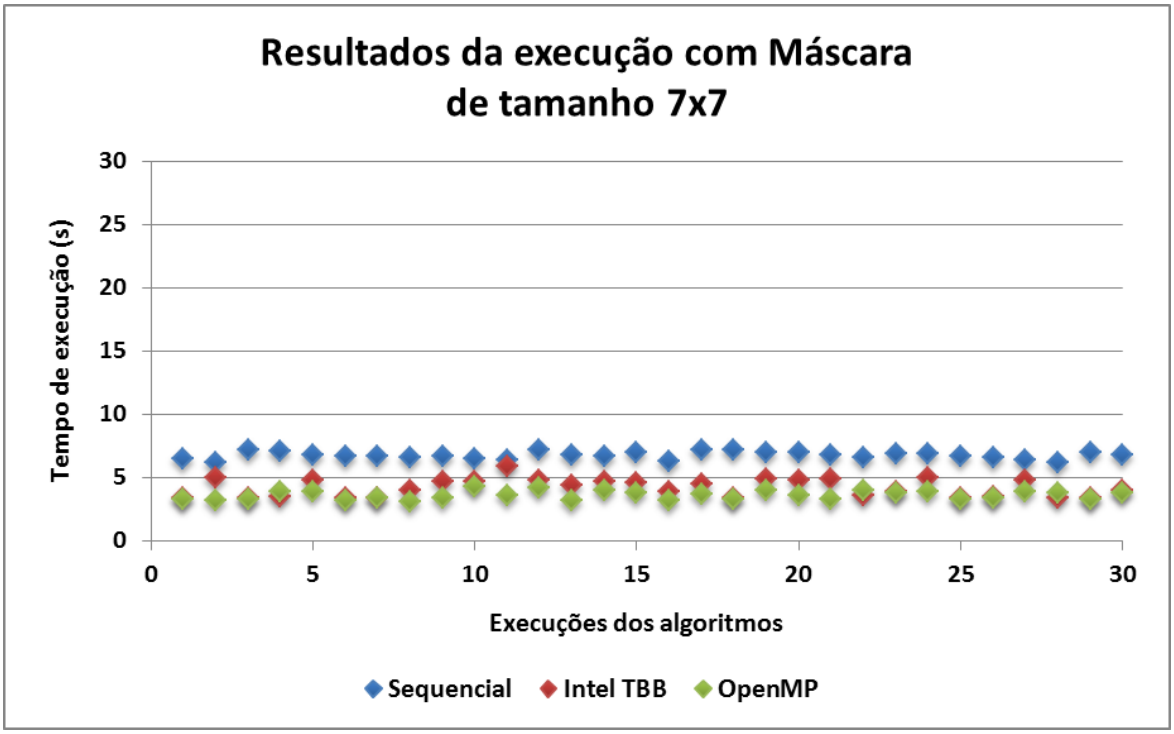
- **Mascara de tamanho 5x5:** o gráfico exibido na Figura 12 mostra os 30 tempos de execução de cada uma das versões dos algoritmos fazendo uso da máscara com tamanho 5x5. Pode-se notar que as duas ferramentas paralelas obtiveram um desempenho bastante similar, mas no geral a ferramenta *OpenMP* também obteve um melhor desempenho.

**Figura 12:** Tempos de execução com a máscara 5x5.




- **Mascara de tamanho 7x7:** o gráfico ilustrado na Figura 13 mostra os 30 tempos de execução de cada uma das versões dos algoritmos fazendo com a máscara de tamanho 7x7.

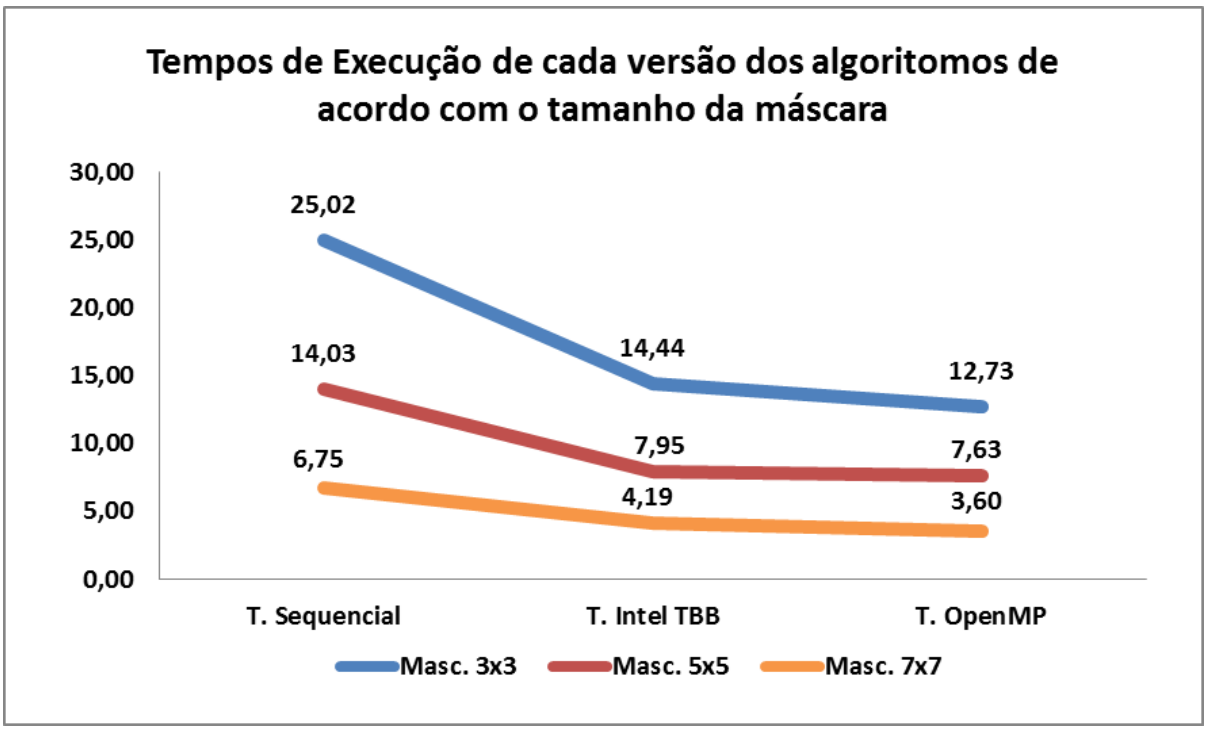
**Figura 13:** Tempos de execução com a máscara 7x7.




Pode se notar que, com a máscara de tamanho 7x7, os tempos de execução dos algoritmos desenvolvidos fazendo-se o uso das ferramentas paralelas, não estão tão distantes da versão sequencial, conforme apresentado nos gráficos anteriores (Figuras 11 e 12). Acredita-se que isso se deve ao fato de uma maior carga de trabalho em cada tarefa, uma vez que a quantidade de *pixels* a serem processados aumenta consideravelmente em relação ao tamanho de máscara 5x5. De qualquer maneira, o algoritmo que faz uso da ferramenta paralela *OpenMP* obteve o melhor desempenho. Para

uma melhor visualização dos desempenhos obtidos por cada algoritmo, o gráfico apresentado na Figura 14 traz a média dos tempos de execução de cada versão dos algoritmos, para cada um dos tamanhos de máscaras testados.

**Figura 14:** Tempos de execução dos algoritmos em média.




Com todos os tamanhos de máscaras testados, o algoritmo paralelo com a ferramenta *OpenMP* apresentou, em média, os melhores tempos de execução com os tempos 12,73 segundos para a máscara de tamanho 3x3, 7,63 segundos para a máscara de tamanho 5x5 e 3,60 para a máscara de tamanho 7x7.

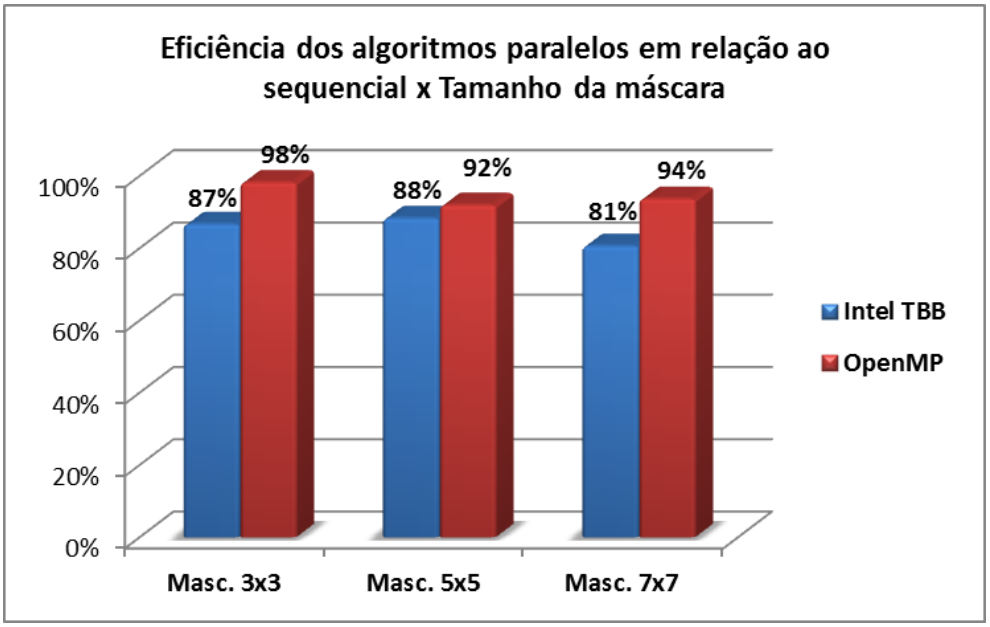
Em segundo lugar ficou o algoritmo paralelo com uso da ferramenta *Intel* TBB com os tempos de, em média, 14,44 segundos para o tamanho de máscara 3x3, 7,95 segundos para o tamanho de máscara 5x5 e 4,19 segundo para o tamanho de máscara 7x7.

Com os piores tempos de execução ficou o algoritmo sequencial, sendo em média, 25,02 segundos para o tamanho de máscara 3x3, 14,03 segundos para o tamanho de máscara 5x5 e 6,75 segundos para o tamanho de máscara 7x7.

Era esperado que o algoritmo sequencial obtivesse um desempenho aquém dos demais, pois o objetivo das ferramentas de programação paralela é fazer com que os programas utilizem os recursos disponíveis no computador da melhor maneira possível, fazendo com que o tempo de execução dos algoritmos que as utilizem demore menos tempo para executar.

Com a finalidade de saber quanto os algoritmos que fazem uso das ferramentas paralelas foram mais eficientes que o algoritmo sequencial, o gráfico de barras (Figura 15) demonstra a eficiência dos algoritmos paralelos em relação ao sequencial, de acordo com cada um dos tamanhos de máscara utilizados. A eficiência indica a taxa de utilização média dos processadores, permitindo verificar se os recursos de hardware foram bem aproveitados.

**Figura 15:** Eficiência dos algoritmos paralelos em relação ao sequencial.




Com o tamanho de máscara 3x3, o algoritmo com *OpenMP* (barra vermelha) foi 98% mais eficiente que o algoritmo sequencial, enquanto o algoritmo com Intel TBB (barra azul) foi 87% mais eficiente que a versão sequencial. Para a máscara de tamanho 5x5, o algoritmo com *OpenMP* foi 92% mais eficiente que o sequencial enquanto o algoritmo com Intel TBB foi 88% mais eficiente. Já para a máscara de tamanho 7x7, o algoritmo com *OpenMP* foi 94% mais eficiente que o sequencial e o algoritmo com Intel TBB obteve uma eficiência de 81%.

Pode-se perceber que o algoritmo que faz uso da ferramenta *OpenMP* obteve uma melhor eficiência para todos os tamanhos de máscara utilizados como teste. O que antecipava o gráfico da Figura 14 que apresentava os tempos médios de execução de cada algoritmo, onde o algoritmo com *OpenMP* executou em menos tempo.

Uma outra análise realizada foi o cálculo do *Speed-UP*. Trata-se de uma métrica de desempenho que mostra o quanto mais rápido um programa paralelo é em relação ao seu correspondente sequencial, ou seja, o quanto ele é escalável. Nesse sentido, a Tabela 2 apresenta todos os valores de *Speed-UP* calculados para cada um dos tamanhos de máscara.

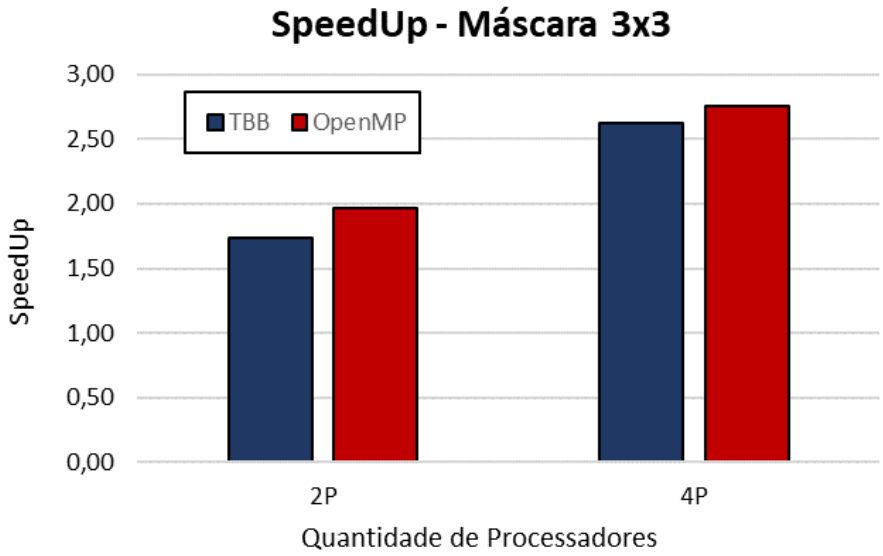
**Tabela 1:** *Speed-UP* das ferramentas paralelas x algoritmo sequencial.

| ***Speed-UP* das ferramentas** | | | |
|---|---|---|---|
| **Ferramentas** | **Masc. 3x3** | **Masc. 5x5** | **Masc. 7x7** |
| **Intel TBB** | 1,73 | 1,77 | 1,61 |
| **OpenMP** | 1,96 | 1,84 | 1,87 |

Pode-se observar que o algoritmo paralelo *OpenMP* obteve um melhor *Speed-UP* quando comparado com o algoritmo paralelo Intel TBB. Como os testes foram realizados em um computador com 2 núcleos de processamento, o algoritmo que obtivesse o *Speed-UP* mais próximo de 2, seria o melhor. Pode-se afirmar com isso, que o algoritmo baseado na ferramenta paralela *OpenMP* é mais escalável que o algoritmo com Intel TBB.

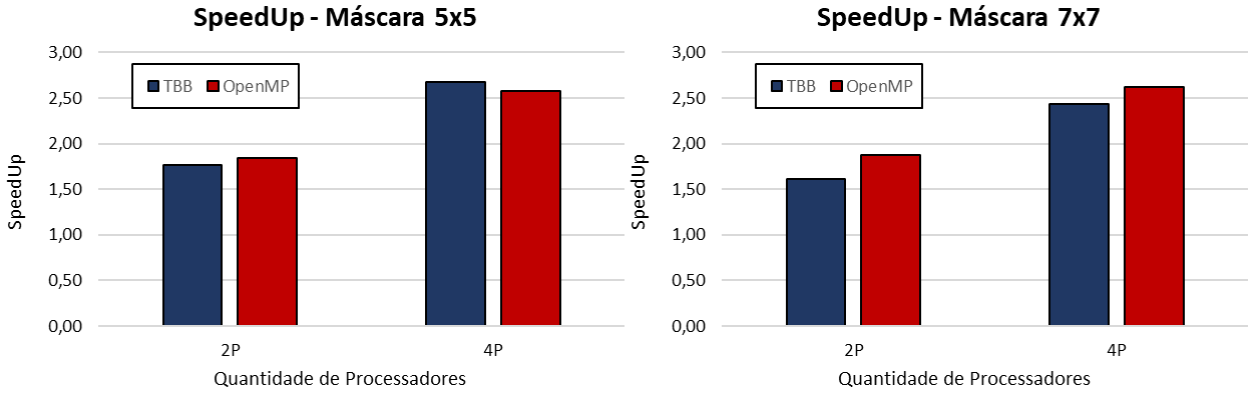
Adicionalmente, também foi realizado um experimento em uma máquina com 4 processadores, comparando os *speedUps* obtidos pelas duas ferramentas. A Figura 16 mostra os *speedUps* das ferramentas com a máscara de tamanho 3x3.

**Figura 16 -** *SpeedUp* das ferramentas com máscara de tamanho 3x3.




Já era esperado que o *speedUp* aumentasse em uma máquina com 4 processadores, pois a máquina possui mais recursos para processar as tarefas produzidas na versão paralela do algoritmo. A Figura 17 apresenta os *speedUps* obtidos com as máscaras de tamanho 5x5 e 7x7.

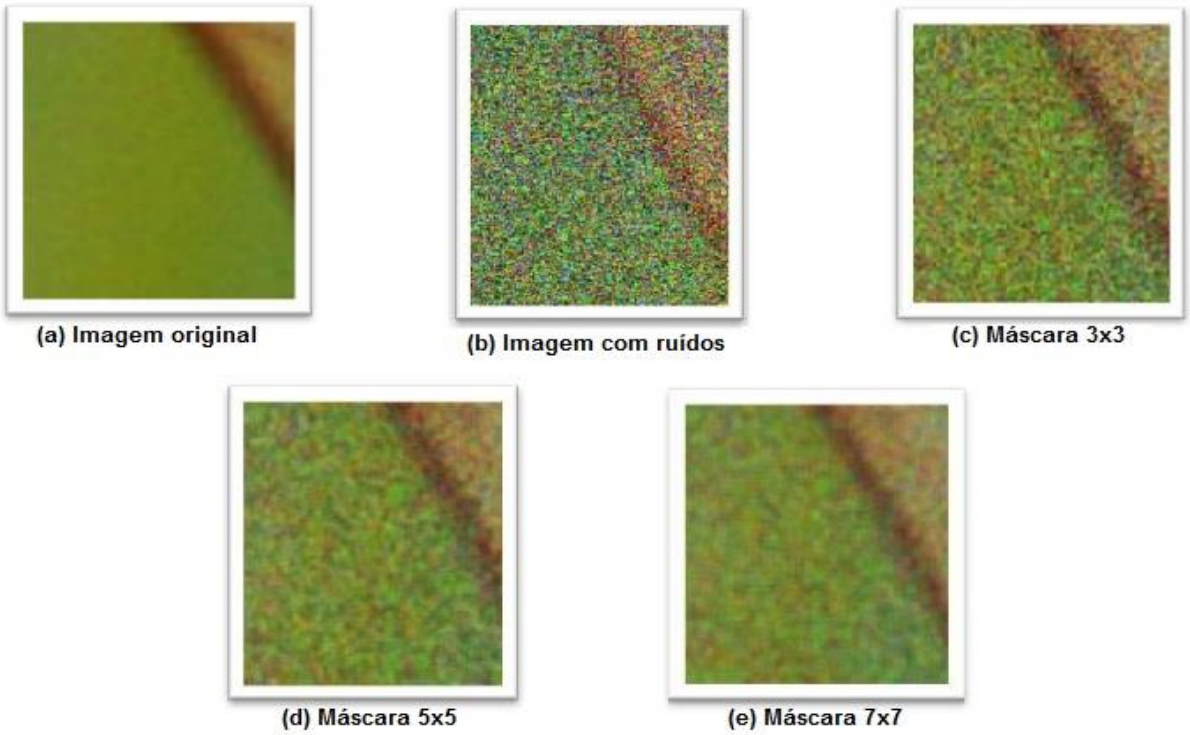
**Figura 17 -** *SpeedUp* com máscaras de tamanho 5x5 (esquerda) e 7x7 (direita).




Olhando as Figuras 16 e 17 em conjunto, para todos os tamanhos de máscaras foi possível verificar um aumento de desempenho (*speedUp*) de ambas as ferramentas na medida em que aumenta a quantidade de processadores. O *speedUp* ideal em uma máquina com 4 processadores deveria ficar próximo de 4. Isso não acontece pois quanto maior é o número de processadores, maior é o numero de dependências entre as tarefas e maior é o custo com comunicação entre eles. Nesse sentido é normal que o *speedUp* não cresça de maneira exponencial. Analisando os três gráficos, o *speedUp* está mais próximo do ideal na execução da máquina com 2 processadores (que deveria ser 2) do que com 4 processadores (onde deveria ser 4). Mesmo assim, de modo geral, a ferramenta *OpenMP* obteve os melhores resultados, sendo mais indicada para este tipo de aplicação.

Para fins de visualização da remoção dos ruídos da imagem utilizada nos experimentos, a figura 18, a seguir, ilustra a imagem final gerada por cada um dos tamanhos de máscaras adotados, comparando com a imagem original.

**Figura 18:** Análise visual das imagens após a remoção de ruídos.




Com a intenção de melhor visualizar os detalhes da imagem, a figura 18 ilustra apenas uma parte da imagem final gerada, para que se tenha uma ideia do quanto os *pixels* foram afetados. Em (a), está representada a imagem original, livre de ruídos. Já em (b), está representada a imagem carregada de ruídos do tipo sal e pimenta, esta imagem foi utilizada como entrada para todos os algoritmos desenvolvidos e com cada um dos tamanhos de máscara utilizados, a fim de se fazer uma comparação em termos de qualidade da imagem final gerada.

As imagens (c), (d) e (e) são, respectivamente, as imagens após a aplicação do algoritmo de remoção de ruídos com as máscaras de tamanho 3x3, 5x5 e 7x7. Pode-se perceber uma melhora significativa na remoção de ruídos da imagem com a utilização de todos os tamanhos de máscaras abordados. Pode-se notar também, que, quanto maior o tamanho da máscara, maior foi a suavização dos ruídos obtida. Dessa maneira, pode-se ver claramente que a imagem gerada com a máscara de tamanho 7x7 deixou a imagem mais próxima da imagem original, sem ruídos.

## 5. CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma alternativa de paralelização para algoritmos de remoção de ruídos em imagens, fazendo o uso das ferramentas de programação paralelas *Intel Threading Building Blocks* – TBB e *Open Multi-processing* – OpenMP, com o objetivo de aumentar o desempenho através da redução dos tempos de processamento.

Para realizar a remoção dos ruídos nas imagens, o método adotado foi o filtro da média, onde o valor de cada *pixel* da imagem é calculado através da média dos valores dos *pixels* vizinhos. Para tal, foi

desenvolvido um algoritmo sequencial. Após a validação desse algoritmo, dois novos algoritmos foram desenvolvidos, um para cada ferramenta de programação paralela estudada.

Com relação aos experimentos realizados, todos os algoritmos foram executados com a mesma imagem de entrada. Optou-se por uma imagem de altíssima resolução com tamanho de 8.260 x 6.000 *pixels*. Cada um dos algoritmos foi executado 30 vezes para cada um dos tamanhos de máscaras adotados (3x3, 5x5 e 7x7) para aplicar o filtro da média pelo algoritmo de remoção de ruídos, a fim de se obter uma média confiável de tempo de execução.

Os resultados experimentais obtidos na execução dos algoritmos da versão sequencial, da versão paralela com *Intel* TBB e da versão paralela com *OpenMP*, indicam que as ferramentas paralelas em geral são extremamente eficientes se comparadas a versão tradicional de desenvolvimento, a sequencial, quando utilizadas em aplicações de processamento de imagens, pois obteve-se um alto ganho de desempenho com a redução do tempo de execução, tendo como base a criação e distribuição das tarefas entre os processadores existentes.

Dentre as duas ferramentas paralelas utilizadas, a que obteve maior desempenho, em média, foi a ferramenta *OpenMP*, sendo esta a que foi mais eficiente e teve um maior *Speed-UP* em relação ao algoritmo sequencial. Para o tamanho de máscara 3x3, a eficiência foi de 98%, com um *Speed-UP* de 1,96. Para o tamanho de máscara 5x5, a eficiência foi de 92% e o *Speed-UP* ficou em 1,84 e para a máscara de tamanho 7x7, a eficiência foi de 94% e o *Speed-UP* foi 1,87.

Não obstante, existem outras políticas de escalonamento das tarefas que podem aumentar ainda mais o ganho em desempenho. Neste sentido, como trabalhos futuros, novas experimentações serão feitas para comparar com a política de escalonamento apresentada nesse trabalho, onde a cada uma das linhas da imagem foi transformada em uma tarefa e, posteriormente, consumida pelos núcleos de processamento. Outras linguagens de programação também serão incluídas nas análises, como as ferramentas *Anahy*, *Threads POSIX* e *Cilk*.

## 6. REFERÊNCIAS

ANDREWS, G. R.. **Foundations of Multithreaded, Parallel, and Distributed Programming.** Boston: Addison Wesley, 2009.

ARÊDES, B. A. R.. **Técnicas de Wavelet Thresholding Aplicadas no Processo de Denoising de Imagens Digitais.** 2009. 106 f. Dissertação (Mestrado em Engenharia Elétrica) - Departamento de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais - PUCMG, Belo Horizonte, 2009.

BASTOS, V. P.. **Técnicas de Segmentação de Imagens para Recuperação de Informações Visuais.** Pelotas: Atlas Ucpel, 2010.

BOTOR, Tomas; HABIBALLA, Hashim. **Comparison of parallel data processing and its performance**. In: AIP Conference Proceedings. AIP Publishing, 2017. p. 80010.

CARISSIMI, A. da S.; OLIVEIRA, R. S. de; TOSCANI, S. S.. **Sistemas Operacionais.** 2.ed. Porto Alegre: Bookman, 2008.

CHANDRA, R. et al. **Parallel Programming in OpenMP.** São Francisco: Morgan Kaufmann, 2011.

DORINI, L. E. B.; ROCHA, A. de R.. **Análise dos filtros não-lineares.** Campinas: Unicamp, 2007.

FILHO, J. N.. **Espacialização da temperatura para o polo de desenvolvimento Petrolina/Juazeiro utilizando computação de alto desempenho.** 2012. 54 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Departamento de Ciência da Computação da Universidade Federal do Vale do São Francisco, Juazeiro, 2012.

FREEMAN, K. REICHER, M. **A Novel Approach for Image Denoising Using Digital Filters**. International Global Journal for Engineering Research (IGJER), p. 21-24, Vol. 13, 2015.

FRERY, C. O. et al. Visão Computacional usando OpenCV. **Revista de Informática Teórica e Aplicada - Rita**, v. 15, n. 1, p.125-160, 2011.

GEORGE, Sinisha; JOSEPH, Silpa. **Survey on Various Image Denoising Techniques**. 2017.

GOMES, J. de M.; VELHO, L. C.. **Computação Gráfica:** Imagem. Rio de Janeiro: Impa/Sbm, 2008.

GOMIDE, J. V. B.; ARAÚJO, A. de A.. Efeitos Visuais, uma Abordagem a Partir do Processamento Digital de Imagens. **Revista de Informática Teórica e Aplicada - Rita**, v. 16, n. 1, p.97-124, 2009.

GONZALEZ, R. E.; WOODS, R. C.. **Digital image processing:** Electrical and Computer Engineering Series. 2.ed. United Kingdom: Addison/wesley, 2007.

_____. **Processamento digital de imagens. 3.ed** São Paulo: Pearson/Prentice Hall, 2010.

HONGJUN, L. CHING, Y. S. **A novel Non-local means image denoising method based on grey theory**, Pattern Recognition, Vol.49, p. 237-248, 2016.

JAQUIE, K. R. L.. **Extensão da ferramenta de apoio à programação paralela (F.A.P.P.) para ambientes paralelos virtuais.** 2009. 152 f. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional) – Instituto de Matemática e Computação da Universidade de São Paulo - USP, São Paulo, 2009.

LEÃO, A. C. **Gerenciamento de cores para imagens digitais.** 2008. 135 f. Dissertação (Mestrado em Artes Visuais) - Escola de Belas Artes da Universidade Federal de Minas Gerais - UFMG, Belo Horizonte, 2008.

LOPES, Luana de L.; DE MELO, Francisco R. **Processamento Digital De Imagens Para Uma Análise Quantitativa De Sementes De Feijão**. In: Anais do Congresso de Ensino, Pesquisa e Extensão da UEG (CEPE). 2018.

MARENGONI, M.; STRINGHINI, D.. High level computer vision using opencv. In: SIBGRAPI CONFERENCE ON, 24. 2011, Maceió. **Proceedings at Graphics, Patterns and Images Tutorials (SIBGRAPI-T)**. Maceió: IEEE, 2011. p. 11-24.

MARONGIU, A.; BURGIO, P.; BENINI, L.. Supporting OpenMP on a multi-cluster embedded MPSoC. In: Microprocessors and Microsystems, 8. 2011, Amsterdam. **Proceedings at Microprocess Microsyst.** Amsterdam: Elsevier Science Publishers, 2011, p. 668-682.

OLIVEIRA, L. F. de et al. A. Segmentação de Imagens com Fundo Azul utilizando a multiplicação dos Canais HSV. In: VI WORKSHOP DE VISÃO COMPUTACIONAL, 6, 2010, Presidente Prudente. **Anais do VI Workshop de Visão Computacional.** Presidente Prudente: Fct/Unesp, 2010. p. 1 - 6.

PADUA, D. A. **Encyclopedia of Parallel Computing.** 1.ed. Berlin: Springer, 2011.

PILLA, M. L.; SANTOS, R. R. dos; CAVALHEIRO, G. G. H.. Introdução à programação em arquiteturas multicore. In: IX ESCOLA REGIONAL DE ALTO DESEMPENHO (ERAD), 9. 2009, Porto Alegre. **Anais da IX Escola Regional de Alto Desempenho (ERAD).** Porto Alegre - Rio Grande do Sul: UCS, 2009, p. 72-102.

PLATANIOTIS, K. N.; VENETSANOPOULOS, A. N.. **Color Image Processing and Applications.** Berlin: Springer, 2008.

PRAJAPATI, H.B.; VIJ, S.K.. Analytical study of parallel and distributed image processing. In: International Conference on Image Information Processing (ICIIP), 182. 2011, Waknaghat. **Proceedings at Image Information Processing (ICIIP-2011).** Waknaghat/India: IEEE, 2011. p. 1-6.

REINDERS, J.. **Intel threading building blocks.** Sebastopol, USA: O'reilly & Associates, 2007.

ROMAN, F. R.. **Desenvolvimento de Customizações para o Sistema Visual Advance 3.0.** Joinville: Udesc, 2010.

SANTOS, P. V. **Metodologia para análise de imagens de baixa resolução, para definição de MUB (Mapa Urbano Básico) para apoio às concessionárias de distribuição**. 2018. 78 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Goiás, Goiânia, 2018.

SCHWARZROCK, Janaina. **Adaptação dinâmica do número de threads em aplicações paralelas openMP para otimizar EDP em sistemas embarcados**. 2018.

SCURI, A. E.. **Fundamentos da Imagem Digital.** Rio de Janeiro: Tecgraf/PUC-Rio, 2007.

SONG, C.; SUDIRMAN, S.; MERABTI, M.. A robust region-adaptive dual image watermarking technique. **Journal Of Visual Communication And Image Representation**, v. 23, n. 3, p.549-568, 2012.

STPICZYNSKI, Przemysław. **Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus**. The Journal of Supercomputing, v. 74, n. 4, p. 1461-1472, 2018.

THAKUR, Y. S.; MAURYA, Ajay. **Comparison and Analysis of Different Denoising Techniques in Image Processing**. 2017.

Submissão: 04/07/2018

Aceito: 13/08/2018