Filtro Mediana Processamento Paralelo

André Thiago Borghi Couto, *Graduando, UFES*

Resumo—A simple demonstration of computational power, is undoubtedly the edition of images, being able to be applied in several areas in our world, being the processes simple or complex. One of the most basic we can see in our daily life is the median filter, which consists of reducing values that are very different from your neighborhood, getting a cleaner and sharper image.

Index Terms—Computer Science, Parallel Programming, LaTeX, Threads, Ciência da Computação, Pragramação Paralela, OpenMP.

1 Introdução ao Problema

SEQUANCIALMENTE temos uma complexidade de código em

$$O(2^{3nm} + 2^{6n} + 3)$$

diretamente, desconsiderando as chamadas de funções internas, analisando a totalidade do código, temos os seguintes passos:

- Leitura da imagem;
- Cópia da imagem original;
- Cálculo da mediana, (centro da imagem)
- Cálculo da mediana, (Bordas da imagem)
- Cálculo da mediana, (cantos da imagem)
- Gravação da nova imagem;

Descrevendo os passos de cálculo de mediana, temos a seleção dos pixel em forma de janela, em volta do pixel analisado, logo obtendo 9 pixels no total, uma função que ordena esses pixels e retorna o pixel central, assim a imagem final tem o filtro aplicado. É relevante saber que as bordas e cantos são separados para evitar falha de segmentação. Um grande problema identificado nessa implementação é a forma de armazenamento da nova imagem, que a cada execução faz a gravação no disco, e sabemos que utiliza de varias execuções para chegar a um resultado bom, isso interfere diretamente no resultado, por fazer vários acessos no disco. Junto com os acessos a disco, temos o problema da cópia, que torna-se inútil quando estamos trabalhando na produção de uma nova imagem, sem a alteração da anterior.

2 Estratégia

Temos identificados os problemas na introdução, agora focamos em otimizá-los.

2.1 Leitura de disco e Cópia

Primeiramente procurei a retirada do problema da leitura em disco, com a opção de alocação da memória para a nova imagem:

```
    ...
// inde img eh a imagem a ser processada
Imagem *imgDest = (Imagem*)calloc(1,sizeof(Imagem));
imgDest->pixel =aloca(img->altura+2,img->largura+2);
imgDest->altura = img->altura;
imgDest->largura = img->largura;
imgDest->threshold = img->threshold;
    ...
```

Juntamente com uma modificação na função, para apenas ser salva ao final das operações, substituindo a função:

```
void removeRuidoMediana(char *arqOri,char *arqDes);
```

Pelo protótipo:

```
Imagem *removeRuidoMediana(Imagem *img);
```

Com isso reduzimos um dos loops, após isso o núcleo do código é o próximo passo, onde retiramos loops desnecessários, que são parte do algoritmo original, porém nada otimizado, ao fim podemos colocar todas as instruções de entrada dentro de apenas uma passada por toda a imagem, reduzindo a complexidade do código para $O(3nm)$.

2.2 Modularização

A substituição de parte do código por uma função que faz o cálculo da mediana, no caso só encapsulado.

```
calcMedia
(int ***im, int imax, int jmax, int k, int mmax){
    int m[9], n = 0;
    for (int i = imax - 1; i <= imax + 1; i++)
        for (int j = jmax - 1; j <= jmax + 1; j++)
            m[n++] = im[i][j][k];
    sort(m, m + mmax);
    return m[mmax / 2];
}
```

- *A.T.B. Couto é graduando em ciência da computação, na UFES, campus CEUNES, em São Mateus - ES.*
  E-mail: andreww.max@hotmail.com

Trabalho escrito em 19 de junho de 2019;

Comparando as as implementações temos além da redução das linhas de código, a redução do tempo e a legibilidade, em questão de tempo temos um speedup relacionado na tabela

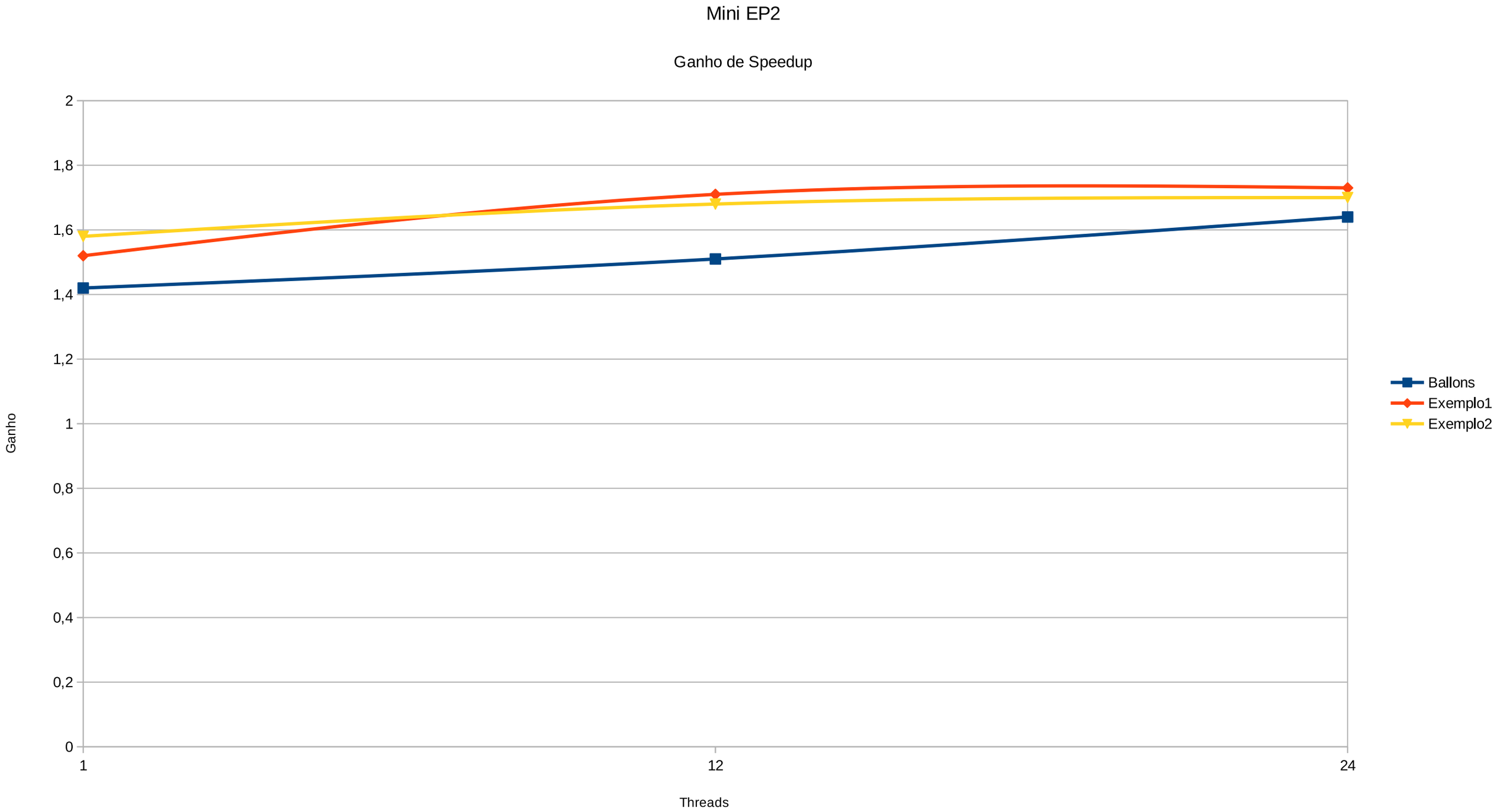
Original	Otimizado	12 Threads	24 Threads
1,9012	1,42	1,51	1,64
14,79767	1,52	1,71	1,73
11,46187	1,58	1,68	1,70

Então podemos considerar que a otimização fez uma grande diferença no desempenho, em média com as modificações que fiz, resultaram em 1.5 em speedup. Continuando com as threads observei a não utilização de 100% das funções do processador, onde devido a múltiplas trocas simples, os cálculos curtos de média, criaram um gargalo onde cada thread tem um pico de processamento. Assim, a otimização foi considerável, porém a biblioteca Openmp, não nos permite ver especificamente o que está acontecendo, então não tive como nívelar a utilização do poder de processamento.

3 Testes e Execuções

Dentre os testes que foram feitos, tive grandes picos de desempenho, e em analises no programa "top", observei por curiosidade uma parte de cada submissão de job, e em todos o "no" estava sendo utilizado por outro processo grande, (como já havíamos visto em um teste da aula), não tenho certeza se interferiu no desempenho dos teste, pois em testes locais no meu pc obtive ganhos próximos, considerando 1, 2, 3 e 4 threads respectivamente. Podemos ver o gráfico com o resultado do teste final em anexo.

André Thiago Borghi Couto atualmente cursa graduação em Engenharia da Computação (2016/1) na Universidade Federal do Espírito Santo. Possui conhecimento em desenvolvimento de software, banco de dados, projetos em Arduino, design gráfico e tem interesse nos temas de automação em Arduino, inteligência artificial, redes de computadores e processamento de imagens.


Mini EP2
Ganho de Speedup
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Ganho
1
12
24
Threads
Ballons
Exemplo1
Exemplo2

Mini EP2

Comparativo entre código Original e os Otimizados