

Relatório do Mini-Exercício de Programação 2

Wellerson Prenholato de Jesus

30 de novembro de 2020

1 Descrição do problema

O problema consiste em utilizar uma versão otimizada e paralelizada do algoritmo de remoção de ruído em uma imagem utilizando o cálculo da mediana do conjunto de valores dos pixels vizinhos.

2 Otimização do Código Sequencial

As melhorias feitas para otimizar o código são mencionadas a seguir:

2.1 Controle no acesso ao arquivo

1. Minimização no número de leituras ao arquivo

No arquivo `main.cpp` fornecido pelo professor, todas as vezes que o método `removeRuidoMediana` é chamado, o carregamento da imagem é realizado. Como sabemos o acesso e o carregamento de arquivos é muito custoso computacionalmente falando. Dessa forma, a estratégia utilizada para otimizar essa etapa foi minimizar o número de leituras do arquivo, reduzindo de n vezes para apenas uma vez, lembrando que n é o número de chamadas do método `removeRuidoMediana`.

2. Minimização no número de escritas no arquivo

A escrita do arquivo segue a mesma ideia da leitura, é realizada n vezes, sendo n o número de chamadas do método `removeRuidoMediana`. O método utilizado para otimizar essa etapa, foi reduzir o número de escrita no arquivo para apenas uma vez após a execução de n vezes do método `removeRuidoMediana`.

Vale ressaltar que controle de leitura e escrita do arquivo é realizado na função `main` como é apresentando na imagem logo abaixo.

```
int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "usage: prog <in.ppm> <out.ppm> <n>\n");
        exit(1);
    }

    int n = atoi(argv[3]);

    double start_time = omp_get_wtime();

    //Realiza a Leitura da imagem
    Imagem *img = readPBM(argv[1]);

    removeRuidoMediana(img);
    for(int i=2; i<n; i++)
        removeRuidoMediana(img);

    //Salva imagem no arquivo destino
    salvaPBM(img, argv[2]);

    double end_time = omp_get_wtime();
    printf("%s\t%lf\n", argv[1], end_time - start_time);

    return 0;
}
```

Figura 1: Função main

2.2 Curiosidade

Na busca em otimizar o algoritmo ao máximo, uma ideia foi a mudança na ordem nos laços de repetição no método *removeRuidoMediana*, infelizmente não houve um resultado considerável.

3 Paralelização

Na paralelização, a interface de programação OpenMP foi utilizada, com seção paralela aplicada em alguns trechos do método *removeRuidoMediana* onde não havia dependência de dados, especificamente nas estruturas que envolvem a parte principal da imagem e as bordas.

A diretiva `#pragma omp parallel shared(copia, img) private(m,i,j,k)` foi inserida englobando os trechos que são referentes a parte principal da imagem e as bordas.

De maneira resumida o **shared** compartilha as variáveis por todos os threads ficando à responsabilidade do programador garantir o seu manuseio correto e o **private** que duplica as variáveis definidas nele em cada thread e o seu acesso passa a ser local (privado) em cada thread, depois disso a diretiva `#pragma omp for` é utilizada para paralelizar cada estrutura de repetição **for** dentro desse escopo.

Vale ressaltar que a estrutura que envolve os cantos da imagem está fora do escopo abordado acima, visto que o número de repetições dessa estrutura é pequeno, dessa forma a paralelização nessa estrutura acaba sendo desvantajosa.

4 Análise Comparativa

No gráfico podemos notar a diferença existente nos três métodos, sendo eles: o sequencial, sequencial otimizado e o otimizado e paralelizado.

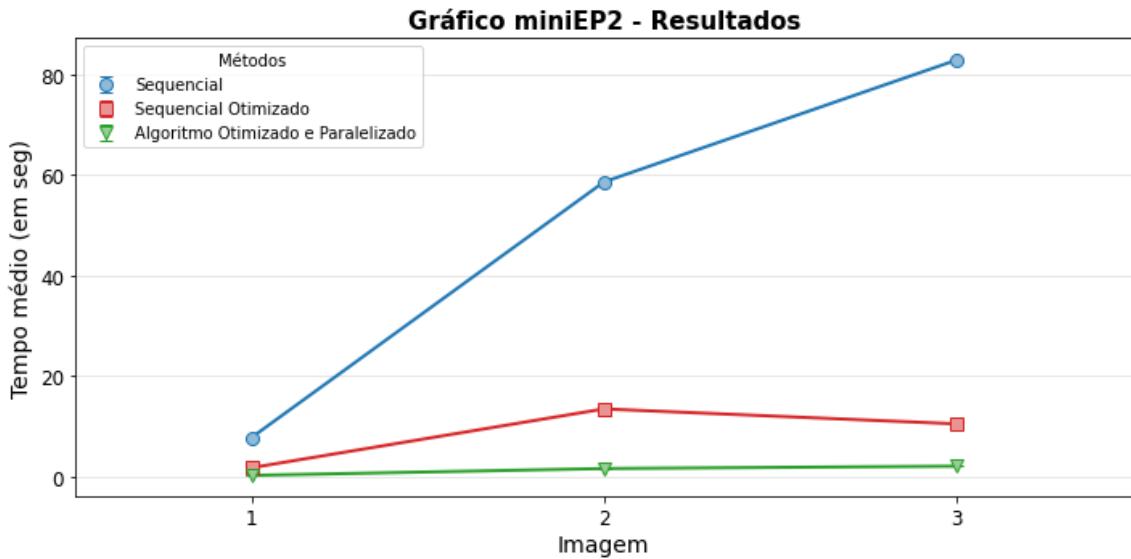


Figura 2: Gráfico comparativo com o tempo médio de execução para cada imagem

A figura 2 foi desenvolvida com a execução de cada método 10 vezes e a partir dos resultados encontrados nessas execuções uma média de tempo foi extraída.

4.1 Observações relacionadas a Figura 2

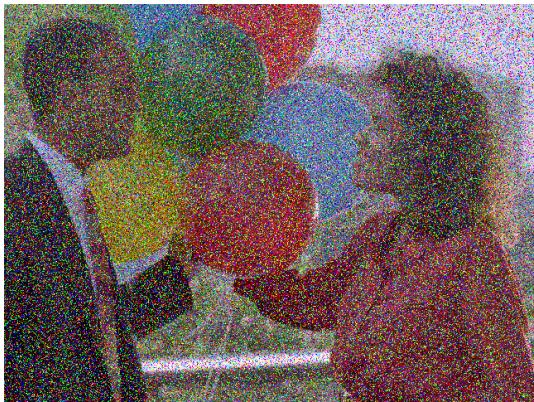
1. Na imagem 1 o tempo médio sequencial é aproximadamente 7.75 seg, em relação ao método otimizado paralelizado tivemos um SpeedUp de 33.70 e uma eficiência de aproximadamente 140%.

2. Na imagem 2 o tempo médio sequencial é aproximadamente 59 seg, em relação ao método otimizado paralelizado tivemos um SpeedUp de 36.90 e uma eficiência de aproximadamente 153%.
3. Na imagem 3 o tempo médio sequencial é aproximadamente 83 seg, em relação ao método otimizado paralelizado tivemos um SpeedUp de 39.52 e uma eficiência de aproximadamente 164%.

É bom frisar que o algoritmo paralelizado não possui identificação no número de threads, logo ele utiliza o número máximo de threads disponíveis no sistema, normalmente é 24.

5 Imagens Originais e Resultantes

Logo abaixo as imagens mostram como eram antes e como ficaram após a aplicação do algoritmo de remoção de ruído.



(a) Imagem 1 Original



(b) Imagem 1 Resultante



(c) Imagem 2 Original



(d) Imagem 2 Resultante



(e) Imagem 3 Original



(f) Imagem 3 Resultante

6 Ambiente de desenvolvimento

O processo de desenvolvimento e execução dos testes foi realizado no ambiente [Intel DevCloud](#).

Todos os testes foram executados utilizando o job:

```
qsub -l nodes=1:gpu:ppn=2 -d . miniEP2.sh
```

Cada job submetido está ligado a um método, e cada método é executado 10 vezes em cada imagem.