SSC0903
Computação de Alto Desempenho
(2023-2) BCC Turma B

1º Trabalho Prático – OpenMP

Membros:

- Eduardo Garcia de Gáspari Valdejão, 11795676
- Henrico Lazuroz Moura de Almeida, 12543502
- Luca Gomes Urssi, 10425396
- Victor Lucas de Almeida Fernandes, 12675399

Profs. Responsáveis:

- Paulo Sergio Lopes de Souza
- Sarita Mazzini Bruschi

Github:

- https://github.com/lucaurssi/SSC0903-CAD

Indice:

Escolha do Algoritmo:

Escolhemos usar um filtro de suavização Gaussiano, onde a suavização tem pesos diferentes dependendo da relevância da posição em relação a posição que está sendo alterada. Depois buscamos pelos maiores e menores valores, achando um valor de cor que está na média dos valores alterados até então, e o utilizamos para binarizar a imagem em 1's e 0's.

Implementação Sequencial:

Começamos a implementação sequencial criando duas matrizes vazias (***calloc*** garante que são preenchidas com 0's), ***mat*** e ***new_mat***, onde ***mat*** é a imagem inicial que recebe valores aleatórios e ***new_mat*** é a imagem resultante do programa.

```
// create image matrix - initialized with zeros
int **mat = (int**)calloc (IMG_SIZE, sizeof (int*));
for (int i=0; i < IMG_SIZE; i++)
    mat[i] = (int*)calloc (IMG_SIZE, sizeof (int));

// create result image matrix
int **new_mat = (int**)calloc (IMG_SIZE, sizeof (int*));
for (int i=0; i < IMG_SIZE; i++)
    new_mat[i] = (int*)calloc (IMG_SIZE, sizeof (int));

// populating 'mat' with random values between 0-255
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        mat[i][j] = rand()%256;
```

Em seguida, pegamos o tempo inicial da região de processamento.

```
double wtime;
double total_time = 0;

for(int k=0; k<100; k++){
    wtime = omp_get_wtime();
```

O primeiro passo do processamento é o filtro gaussiano aplicado em cada posição da matriz.

```
// Apply Gaussian Smoothing
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        new_mat[i][j] = smoothing(mat, IMG_SIZE, i, j); // smooth one position at a time

//print_mat(new_mat, IMG_SIZE);
```

Na figura abaixo podemos ver os pesos do filtro que escolhemos, sendo ‘4’ a posição atual na matriz e outros valores as posições adjacentes. A função ***smoothing()*** adiciona os valores da região multiplicando pelos seus pesos respectivos, e no final divide por ***divisor***, um valor que muda no caso da posição atual da matriz estar em uma borda ou canto.

```
    Gaussian Smoothing:
    -smoothing filter with the following weights:
    1 2 1
    2 4 2
    1 2 1
*/
int smoothing(int **mat, int IMG_SIZE, int i, int j){
    int sum = 0;
    int divisor = 4;

    // first smoothing column
    // 1 x x
    // 2 x x
    // 1 x x
    if(i!=0){ // not in column 0
        divisor += 2;
        if(j!=0){
            sum+= mat[i-1][j-1]; // not in line 0
            divisor += 1;
        }
        sum+= mat[i-1][j] * 2;
        if(j!=IMG_SIZE-1) {
            sum+= mat[i-1][j+1]; // not at the last line
            divisor += 1;
        }
    }

    // middle smoothing column
```

```
    return sum/divisor;
}
```

O segundo passo do processamento é a busca pelo maior e o menor valor da matriz.

```
// Fiding the max & min
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        if(new_mat[i][j]>max) max = new_mat[i][j];
        else if(new_mat[i][j]<min) min = new_mat[i][j];

mean = (int)(max+min)/2;
```

Encontrado esses valores, calculamos ***mean*** de forma que podemos binarizar a imagem no passo a seguir.

```
// binary
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        if(new_mat[i][j] <= mean) new_mat[i][j] = 0;
        else new_mat[i][j] = 1;
```

Por fim, pegamos o tempo final e subtraímos do tempo inicial, obtendo o tempo de execução da região de processamento. Como é necessário fazer vários testes para avaliar o desempenho, o código da área de processamento é repetido 100 e devolve uma média.

```
    wtime = omp_get_wtime() - wtime;
    total_time += wtime;
}
printf("Sequential= %.5f\n", total_time/100 );
```

Implementação Paralela com OpenMP:

Iniciamos nossas alterações englobando toda a área de processamento com uma diretiva ***parallel*** com ***T*** threads.

```
for(int k=0; k<100; k++){
    wtime = omp_get_wtime();
    #pragma omp parallel num_threads(T)
    {
```

Seguido de uma diretiva ***for*** na região da aplicação do filtro gaussiano. Inicialmente havíamos escolhido a diretiva ***simd*** mas o gcc da máquina testada fez questão de ignorar a diretiva com uma mensagem de warning que não conseguimos resolver.

```
// Apply Gaussian Smoothing
#pragma omp for
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        new_mat[i][j] = smoothing(mat, IMG_SIZE, i, j);
```

A seguida utilizamos de ***reduction*** duas vezes, uma para pegar o valor máximo e outra para o valor mínimo, em ambas não há conflito de região crítica pois ***reduction*** garante isso.

```
// Fiding the max & min
#pragma omp for reduction(max:max) reduction(min:min)
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        if(new_mat[i][j]>max) max = new_mat[i][j];
        else if(new_mat[i][j]<min) min = new_mat[i][j];
```

Como estamos dentro de uma diretiva ***parallel***, podemos garantir que apenas uma thread vai calcular a ***mean*** com a diretiva ***single.***

```
#pragma omp single
mean = (int)(max+min)/2;
```

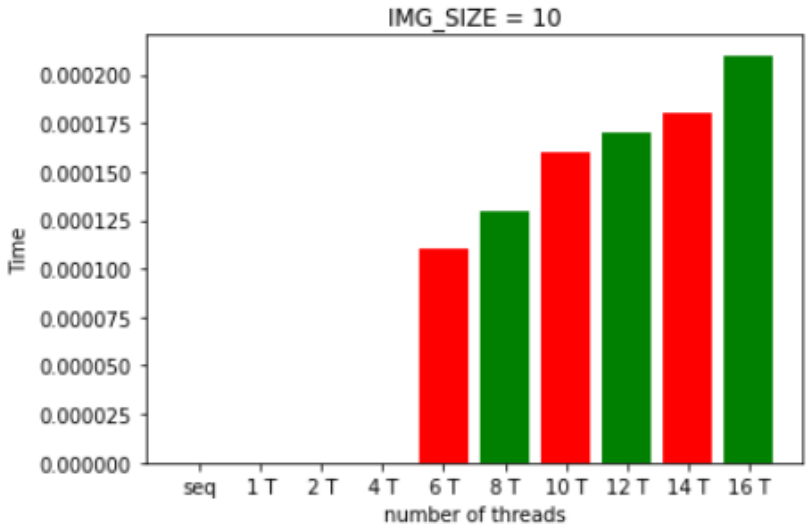
E por último outra diretiva ***for***.

```
// binary
#pragma omp for
for(int i=0; i<IMG_SIZE; i++)
    for(int j=0; j<IMG_SIZE; j++)
        if(new_mat[i][j] <= mean) new_mat[i][j] = 0;
        else new_mat[i][j] = 1;
```

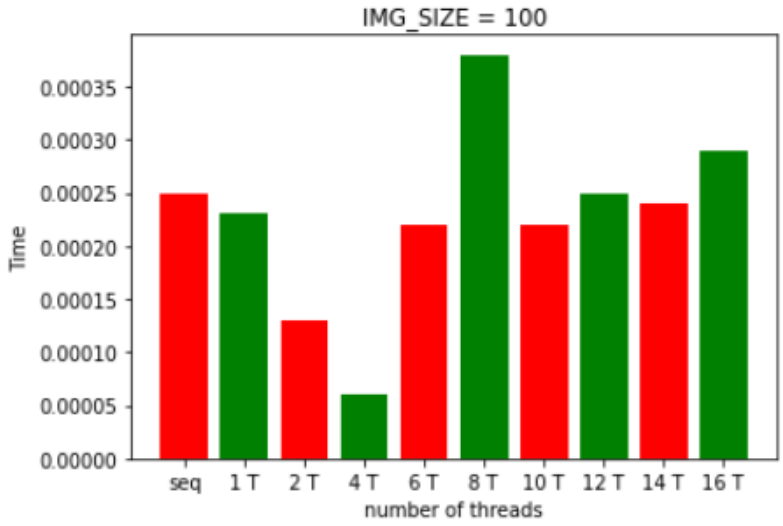
Experimentação e análise dos resultados:

Utilizamos a VM da semcomp no computador do laboratório do bloco 6 para testar nosso código, a VM tem 4 núcleos.

Decidimos testar o código mudando o tamanho da imagem e o número de threads disponibilizados. Nota-se que em cada teste nosso programa roda 100 vezes e tira uma média. O primeiro tamanho de imagem escolhido foi 10x10, onde podemos observar que o custo de paralelizar é maior do que o ganho de desempenho.



No segundo teste utilizamos o tamanho 100x100, onde tivemos os primeiros ganhos de desempenho com até 4 threads, mas de 6 threads ou mais, podemos observar que o custo de comunicação ainda está alto, concluímos que isso ocorre pois há uma necessidade das threads disputarem pelos recursos dos processadores.



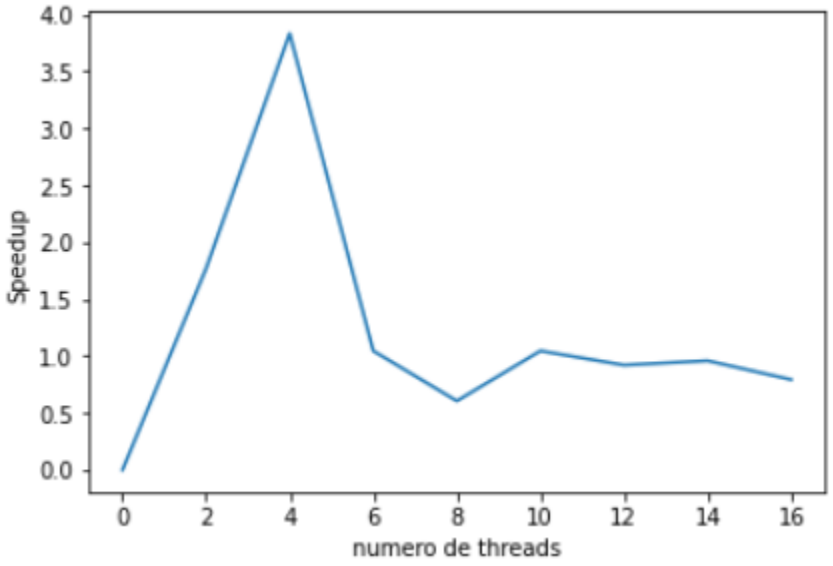
Podemos começar a avaliar o desempenho a partir desse tamanho de imagem.

Dado que em nossos testes o tempo sequencial e o tempo paralelo com uma thread são bem similares, vamos utilizar o tempo paralelo com uma thread para fazer os cálculos.

- P = 4 processadores
- Speedup(Sp) = Tseq / Tpar

Tseq(Tpar1) = 0.00023
Tpar2 = 0.00013
Tpar4 = 0.00006

Sp2 = 0.00023 / 0.00013 = 1.76923
Sp4 = 0.00023 / 0.00006 = 3.83333

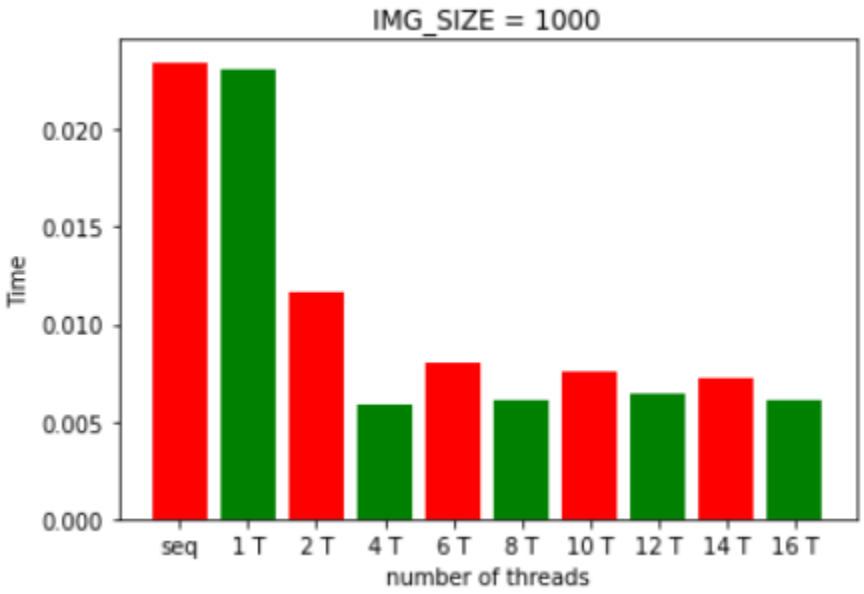


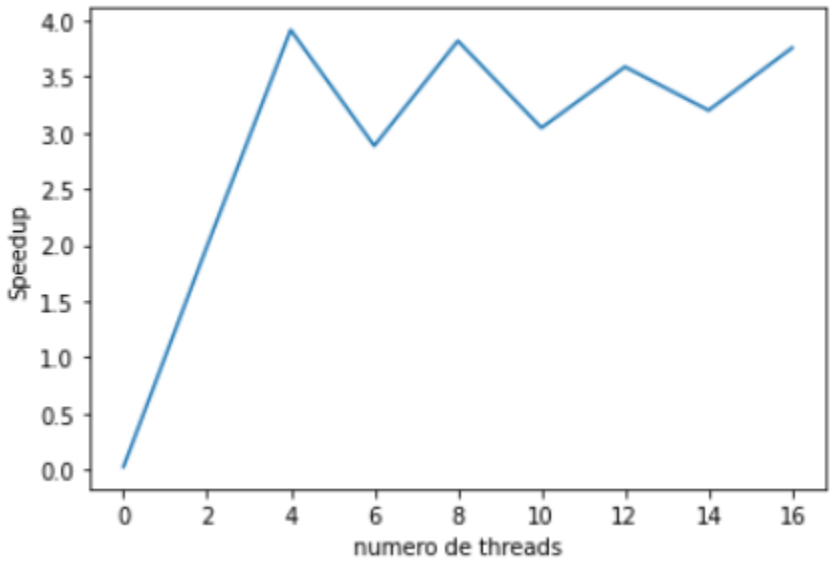
No caso ocorre uma grande perda de desempenho a partir de 6 threads, como esperado.

- Eficiencia(Ep) = Sp / P

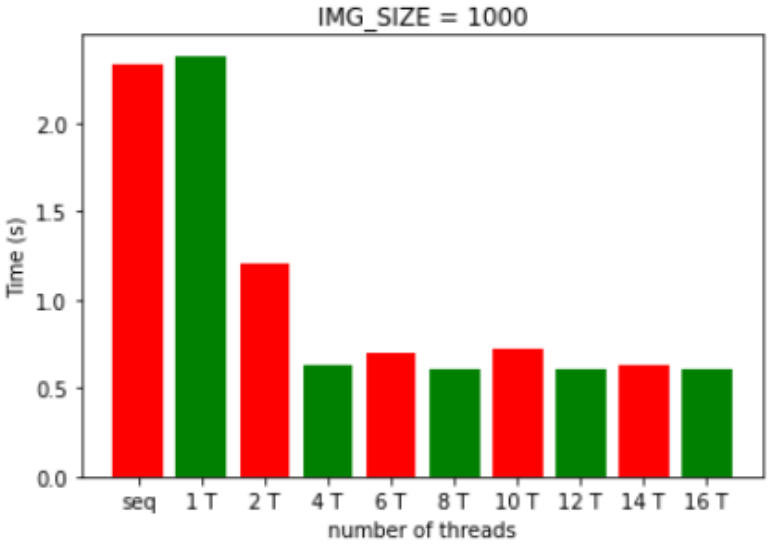
Ep2 = 0.44231
Ep4 = 0.95833

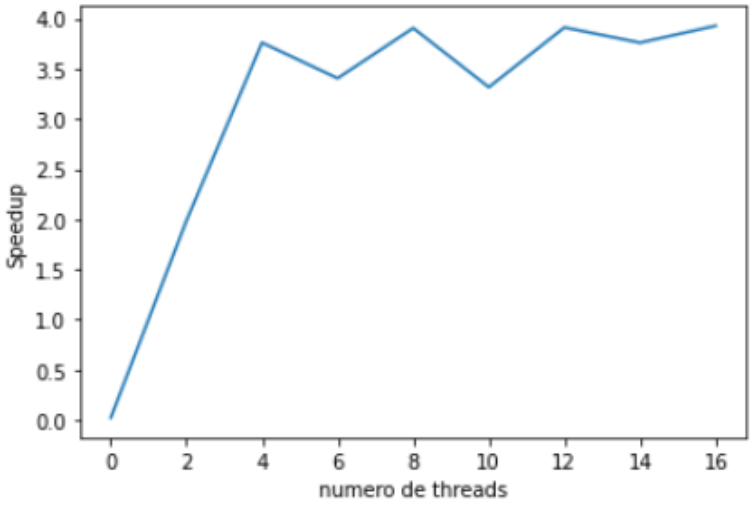
No terceiro teste utilizamos uma imagem de 100x100. Neste teste podemos observar e avaliar melhor nosso código pois a entrada é grande o suficiente para demonstrar ganhos em desempenho em todos os testes paralelos.

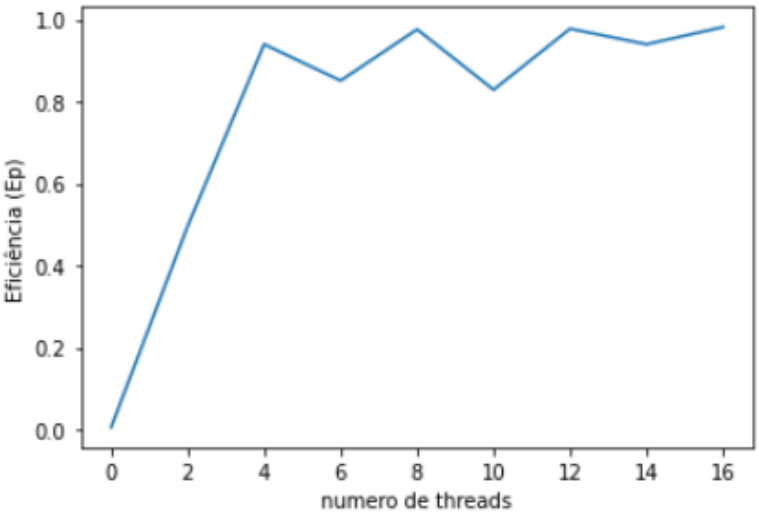




E nosso último teste que levou vários minutos para executar.







Como observado nos gráficos acima, podemos ver que nosso código tem uma eficiência acima de 0.8 e seu speedup na máquina testada com 4 núcleos se aproxima de 4 mas nunca chega, portanto pode ser classificado como *sublinear*.

Podemos notar nos gráficos de tempo por número de thread com IMG SIZE 100 e 1000, que quando o número de threads utilizado é um múltiplo de 4, o tempo é um pouco de execução é menor, imaginamos que isso ocorra pois a divisão de threads por processador custe menos.

O ‘gargalo’ visível em nossos testes é o número de processadores da máquina utilizada.