MC970 - Introdução à Programação Paralela - 2023.1

Projeto Final

Explorando o Paralelismo no Processamento de Imagens

Nome: Victor Rigatto
RA: 178068

Introdução

Com as tecnologias de diminuição dos transistores e o consequente aumento da sua quantidade dentro dos processadores, a Lei de Moore previu que a velocidade de processamento dobraria a cada 18 meses, com aproximadamente o dobro de transistores a cada período. Essa lei se manteve bastante fiel até por volta dos anos 2000. Enquanto era possível inserir mais transistores cada vez menores, a tensão também podia ser diminuída ou ao menos mantida, mantendo o consumo de energia e dissipação de calor sob controle.

A partir de 2008, já não era mais possível reduzir o tamanho dos componentes com a mesma facilidade, e a energia dissipada com o aumento da quantidade de transistores passou a ser uma limitação para o aumento da performance. A partir dessa limitação, a solução encontrada foi a criação de processadores multi núcleo e multi thread, que poderiam executar instruções em paralelo, o que efetivamente revolucionou o funcionamento das aplicações. Atualmente, estamos próximos de atingir o limite no tamanho de transistores, que já chegam ao tamanho de alguns átomos, o que torna impossível o controle da corrente para seu funcionamento. Dessa forma, como apresentado na disciplina e em palestras como a do professor David Patterson, uma das melhores e únicas soluções atuais para essa limitação é a programação paralela.

O processamento de imagens é uma área essencial da ciência da computação que desempenha um papel fundamental em diversos campos, abrangendo desde a medicina até a indústria e o entretenimento. Com o avanço da tecnologia e o surgimento de novos algoritmos para processamento paralelo, a manipulação de imagens se torna cada vez mais eficiente, e uma ferramenta indispensável para analisar, melhorar e extrair informações valiosas a partir de imagens digitais.

Tendo em vista seu amplo papel e aplicações, é esperado que o processamento de imagens se torne cada vez mais utilizado e, por consequência, imagens cada vez maiores e em maior quantidade estejam presentes. Essa maior demanda, inclusive em dispositivos móveis com processadores multi núcleo e baixo consumo de energia, pode ser escalada de forma eficiente através da programação paralela.

Neste trabalho, vamos explorar algumas aplicações da programação paralela no processamento de imagens utilizando o padrão OpenMP para CPUs.

Definição do Sistema

Foi utilizado um notebook padrão da Dell, livre de modificações como overclocks, e refrigeração adequada, com as seguintes especificações:

- Processador Intel Core i3-8130U
  2 núcleos, 4 threads, frequência base de 2.20 GHz
- Memória RAM 16 GB DDR4 2400 MHz
- Ubuntu 22.04 x64, executado em WSL2 no Windows 11 x64
- Compilador *GCC* 11.3.0 *OpenMP* 4.5 e *perf* 5.15.90.1

Definição do Projeto

A exploração das técnicas de paralelização e otimização foram realizadas utilizando-se a linguagem C. Os programas foram compilados utilizando o GCC com a flag *-fopenmp*, habilitando a sua utilização para o padrão OpenMP.

Todas as aplicações recebem como entrada o mesmo arquivo, *input.txt*, que pode ser gerado de forma aleatória através do programa *generate*. É possível escolher o tamanho da imagem ao executá-lo. Os dois primeiros valores do arquivo de entrada informam o tamanho da imagem em pixels. A seguir, cada linha representa a sequência dos pixels em formato RGB (cores representadas de 0, preto, a 255, branco). Na figura abaixo, a entrada representa uma imagem de tamanho 5x5. Um valor de 4096x4096 foi utilizado para os testes neste relatório, que causou um bom tempo de processamento.

```
5 5
180 199 111 238 85 123 19 250 75 50 123 72 229 177 240
132 120 87 237 16 93 42 240 104 66 165 166 48 225 181
108 150 124 219 132 209 86 151 203 161 201 70 234 175 248
218 51 112 49 32 129 142 74 113 246 140 22 157 189 248
82 41 142 206 4 18 159 90 170 106 252 115 176 230 34
```

```
Insira a largura da imagem em pixels:
4096
Insira a altura da imagem em pixels:
4096
Arquivo de entrada gerado: input.txt
```

Foram utilizados dois algoritmos de processamento de imagem, a Aplicação de Blur Gaussiano 3x3 e o Pontilhamento Floyd-Steinberg. Os respectivos arquivos para cada algoritmo estão descritos no README do repositório, assim como os passos para a sua execução.

As aplicações geram como saída arquivos específicos, *output_*.txt*, onde * recebe o nome da aplicação, e que representa os pixels em formato RGB da imagem processada.

A medição do tempo de execução foi feita através da função *omp_get_wtime()*, sendo estritamente relativa à execução do algoritmo e não leva em consideração a abertura e leitura dos arquivos ou qualquer outro processamento não relevante.

O perfilador utilizado foi o nperf, e as estatísticas principais monitoradas foram:

cycles:u: Esse evento representa o número de ciclos de CPU executados pelo programa. O sufixo :u indica que ele mede o número de ciclos não interrompidos, excluindo os ciclos gastos em estados de baixa potência.

instructions:u: Esse evento mede o número de instruções executadas pelo programa. O sufixo :u indica que ele conta o número de instruções concluídas, excluindo instruções especulativas ou predições incorretas.

cache-misses: Esse evento rastreia o número de falhas de cache ocorridas pelo programa. Falhas de cache ocorrem quando dados ou instruções não são encontrados no cache e precisam ser buscados em um nível superior na hierarquia de memória.

L1-dcache-loads: Esse evento conta o número de leituras do cache L1 de dados. Ele mede a frequência com que o programa acessa dados no cache L1, que é o cache mais próximo e rápido.

L1-dcache-load-misses: Esse evento mede o número de falhas de leitura no cache L1 de dados. Ele representa os casos em que uma instrução de leitura precisa acessar dados que não estão presentes no cache L1 e precisam ser buscados em um nível superior na hierarquia de memória.

L1-dcache-stores: Esse evento rastreia o número de escritas no cache L1 de dados. Ele mede a frequência com que o programa escreve dados no cache L1.

dTLB-loads: Esse evento conta o número de leituras que acessam o Buffer de Tradução de Endereços de Dados (dTLB). O dTLB é responsável por traduzir endereços virtuais em endereços físicos.

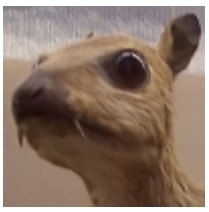
dTLB-load-misses: Esse evento mede o número de falhas de leitura no Buffer de Tradução de Endereços de Dados (dTLB). Uma falha de leitura ocorre quando a tradução de um endereço virtual não é encontrada no dTLB e requer uma busca na tabela de tradução correta.

Aplicação de Blur Gaussiano 3x3

O algoritmo de desfoque gaussiano 3x3 é uma técnica de processamento de imagens usada para suavizar uma imagem, reduzindo o ruído e as imperfeições. Ele utiliza uma matriz de convolução 3x3, também conhecida como kernel, para calcular a média ponderada dos pixels vizinhos de cada ponto da imagem original.

O kernel gaussiano 3x3 é projetado para aplicar um efeito de borramento suave, em que os pixels próximos têm um peso maior na média ponderada do que os pixels mais distantes. Essa ponderação é baseada na distribuição gaussiana, que segue uma curva em forma de sino.

O processo de aplicar o algoritmo de desfoque gaussiano 3x3 envolve deslizar o kernel sobre cada ponto da imagem original e calcular a média ponderada dos valores dos pixels sob o kernel. Em outras palavras, envolve a multiplicação de matrizes (entre a matriz do kernel e a dos pixels que formam a imagem). O resultado é uma nova imagem em que os detalhes finos são suavizados e o ruído é reduzido.



Normal

Blur Gaussiano 3x3

O processo de multiplicação de matrizes é um forte candidato a ser paralelizado. Sabemos, inclusive, que as GPUs são processadores muito especializados em realizar esse tipo de operação, porém neste trabalho vamos utilizar a CPU para realizá-lo.

```
void convolutionBlur(int *pixels, int width, int height) {
    int kernel[3][3] = { {1, 2, 1},
                         {2, 4, 2},
                         {1, 2, 1} };
    int kernelSize = 3;
    int radius = kernelSize / 2;

    int *blurredPixels = malloc(width * height * sizeof(int));

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int sum = 0;
            for (int ky = -radius; ky <= radius; ky++) {
                for (int kx = -radius; kx <= radius; kx++) {
                    int nx = x + kx;
                    int ny = y + ky;
                    if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                        int pixelIndex = ny * width + nx;
                        int kernelValue = kernel[ky + radius][kx + radius];
                        sum += pixels[pixelIndex] * kernelValue;
                    }
                }
            }
            int blurredPixelIndex = y * width + x;
            blurredPixels[blurredPixelIndex] = sum / 16; // Divide by the sum of kernel values
        }
```

O filtro de desfoque é aplicado pela função *convolutionBlur*, que utiliza uma matriz kernel 3x3 pré-definida para calcular a média ponderada dos valores dos pixels próximos de cada pixel da imagem.

```
for (int i = 0; i < width * height; i++) {
    pixels[i] = blurredPixels[i];
}
```

O resultado é armazenado em um novo array chamado *blurredPixels*. Em seguida, os pixels desfocados são copiados de volta para o array original *pixels*.

Paralelização da Aplicação de Blur Gaussiano 3x3

```
void convolutionBlur(int *pixels, int width, int height) {
    int kernel[3][3] = { {1, 2, 1},
                         {2, 4, 2},
                         {1, 2, 1} };
    int kernelSize = 3;
    int radius = kernelSize / 2;

    int *blurredPixels = malloc(width * height * sizeof(int));

    #pragma omp parallel for collapse(2)
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int sum = 0;
            for (int ky = -radius; ky <= radius; ky++) {
                for (int kx = -radius; kx <= radius; kx++) {
                    int nx = x + kx;
                    int ny = y + ky;
                    if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                        int pixelIndex = ny * width + nx;
                        int kernelValue = kernel[ky + radius][kx + radius];
                        sum += pixels[pixelIndex] * kernelValue;
                    }
                }
            }
            int blurredPixelIndex = y * width + x;
            blurredPixels[blurredPixelIndex] = sum / 16; // Divide by the sum of kernel values
```

Na função *convolutionBlur*, a paralelização é realizada usando a diretiva ***#pragma omp parallel for collapse(2)***. Essa diretiva indica que os loops aninhados seguintes devem ser executados em paralelo. A cláusula *collapse(2)* especifica que os dois loops (um para percorrer a altura e outro para percorrer a largura da imagem) devem ser combinados em um único loop para execução paralela. Isso permite que várias threads trabalhem em porções diferentes da imagem simultaneamente, melhorando o desempenho. Ao paralelizar os loops aninhados, cada thread é atribuída a um subconjunto dos pixels da imagem para processamento. As iterações dos loops são divididas entre as threads, e cada thread realiza o cálculo do desfoque independentemente. Os arrays *sum* e *blurredPixels* são compartilhados entre as threads, já que cada thread atualiza sua parte respectiva dessas variáveis.

Na segunda região paralela, a diretiva ***#pragma omp parallel for*** foi usada para paralelizar o loop que copia os pixels borrados do array *blurredPixels* de volta para o array original *pixels*. Essa paralelização permite que várias threads copiem os pixels simultaneamente, melhorando o desempenho geral. A utilização dessas diretivas foi possível pois, nesse caso, não foram identificados riscos de condição de corrida por dependências de variáveis, que fosse necessário tratar com regiões críticas ou modificações atômicas, por exemplo.

Execução

A execução da versão serial e paralelizada foi realizada junto ao nperf.

Serial:

```
victor@DESKTOP-HB517L1:~/Studio/MC970/Projeto/Final$ ./perf stat --repeat 3 -e cycles:u,instructions:u,cache-misses:u,L1-dcach
e-loads:u,L1-dcache-load-misses:u,L1-dcache-stores,dTLB-loads,dTLB-load-misses ./blur_serial
Arquivo de saída: output_blur_serial.txt
Tempo de execução do Blur: 1.084110
Arquivo de saída: output_blur_serial.txt
Tempo de execução do Blur: 0.965501
Arquivo de saída: output_blur_serial.txt
Tempo de execução do Blur: 1.038095

 Performance counter stats for './blur_serial' (3 runs):

     11600538008      cycles:u                                                ( +-  1.47% )  (62.36%)
     31540069297      instructions:u            #    2.73  insn per cycle     ( +-  0.52% )  (74.87%)
        10516792      cache-misses:u                                          ( +-  2.60% )  (74.87%)
      9852176832      L1-dcache-loads                                         ( +-  0.14% )  (74.88%)
        11300771      L1-dcache-load-misses     #    0.11% of all L1-dcache accesses  ( +-  3.26% )  (74.88%)
      4869433039      L1-dcache-stores                                        ( +-  0.19% )  (75.14%)
      9918835100      dTLB-loads                                              ( +-  0.20% )  (50.25%)
          306066      dTLB-load-misses          #    0.00% of all dTLB cache accesses  ( +-  8.08% )  (49.98%)

           3.7679 +- 0.0613 seconds time elapsed  ( +-  1.63% )
```

O tempo médio de execução serial em 3 rodadas foi de **1,029235** segundos.

Paralelizada:

```
victor@DESKTOP-HB517L1:~/Studio/MC970/Projeto/Final$ ./perf stat --repeat 3 -e cycles:u,instructions:u,cache-misses:u,L1-dcach
e-loads:u,L1-dcache-load-misses:u,L1-dcache-stores,dTLB-loads,dTLB-load-misses ./blur_parallel
Arquivo de saída: output_blur_parallel.txt
Tempo de execução do Blur: 0.482521
Arquivo de saída: output_blur_parallel.txt
Tempo de execução do Blur: 0.608394
Arquivo de saída: output_blur_parallel.txt
Tempo de execução do Blur: 0.719011

 Performance counter stats for './blur_parallel' (3 runs):

     13900324069      cycles:u                                                ( +-  1.24% )  (62.80%)
     31902960443      instructions:u            #    2.25  insn per cycle     ( +-  0.43% )  (75.42%)
        11573164      cache-misses:u                                          ( +-  5.87% )  (75.11%)
     10071056237      L1-dcache-loads                                         ( +-  0.28% )  (75.29%)
        11219464      L1-dcache-load-misses     #    0.11% of all L1-dcache accesses  ( +-  1.60% )  (75.15%)
      4880582447      L1-dcache-stores                                        ( +-  0.06% )  (75.13%)
     10077219623      dTLB-loads                                              ( +-  0.41% )  (49.80%)
          379769      dTLB-load-misses          #    0.00% of all dTLB cache accesses  ( +-  6.73% )  (50.08%)

           3.4028 +- 0.0873 seconds time elapsed  ( +-  2.57% )
```

O tempo médio de execução paralela em 3 rodadas foi de **0,603308** segundos.

Podemos calcular o speedup da seguinte forma:

$$speedup = \frac{\text{sequential running time}}{\text{parallel running time}}$$

Portanto, o speedup foi de **1,7059**. Este resultado está dentro da lei de Amdahl, considerando aproximadamente que 90% do tempo de execução deve ter sido utilizado para o algoritmo, e os outros 10% para operações como leitura e escrita de arquivos.

$$speedup \leq \frac{1}{(1 - P_\sigma) + \frac{P_\sigma}{n}}$$

Nesse caso, o speedup máximo que poderíamos atingir seria de 3,077 para o sistema utilizado, com *n* = *4* threads.

O perfilamento mostrou que a abordagem utilizada não resultou em uma grande quantidade de cache-misses, o que afetaria negativamente a velocidade de execução. Observação adicional: No trabalho "*01-OmpFor-MatMul*" dos laboratórios de OpenMP, tivemos a tarefa de paralelizar uma multiplicação de matrizes. Nesse laboratório, a paralelização gerou um grande aumento de cache-misses, o que tornou o paralelo mais lento que o serial. Nesse caso, a solução foi utilizar a transposta para multiplicação da matriz, o que otimizou o funcionamento da hierarquia de memória. Apesar de não ser o caso neste algoritmo, o resultado desse laboratório foi interessante:

```
Performance counter stats for 'build/serial tests/5.in' (2 runs):

     27032707905      cycles:u                                      ( +-  0.65% )  (62.47%)
     30784549815      instructions:u            #    1.15  insn per cycle         ( +-  0.24% )  (74.95%)
       318590980      cache-misses:u                                ( +-  0.34% )  (74.90%)
      6804049481      L1-dcache-loads                               ( +-  0.13% )  (75.06%)
      4384646996      L1-dcache-load-misses     #   64.36% of all L1-dcache accesses  ( +-  0.02% )  (75.12%)
        50997509      L1-dcache-stores                              ( +-  0.29% )  (75.00%)
      6825162966      dTLB-loads                                    ( +-  0.08% )  (49.97%)
       785040110      dTLB-load-misses          #   11.51% of all dTLB cache accesses  ( +-  1.04% )  (49.94%)

       8.2034 +- 0.0402 seconds time elapsed  ( +-  0.49% )

Performance counter stats for 'build/parallel tests/5.in' (2 runs):

     14751365674      cycles:u                                      ( +-  0.09% )  (62.53%)
     27406240248      instructions:u            #    1.86  insn per cycle         ( +-  0.06% )  (74.90%)
       170441566      cache-misses:u                                ( +-  5.77% )  (75.12%)
      6828245621      L1-dcache-loads                               ( +-  0.10% )  (74.88%)
       188913339      L1-dcache-load-misses     #    2.77% of all L1-dcache accesses  ( +-  1.16% )  (75.23%)
        54802750      L1-dcache-stores                              ( +-  1.51% )  (75.34%)
      6864069782      dTLB-loads                                    ( +-  0.16% )  (49.66%)
         3360910      dTLB-load-misses          #    0.05% of all dTLB cache accesses  ( +-  0.07% )  (50.11%)

       1.2314 +- 0.0133 seconds time elapsed  ( +-  1.08% )
```

Pontilhamento de Floyd-Steinberg

É um algoritmo de dithering, processo de simulação de cores intermediárias em imagens com uma paleta limitada. Por exemplo, pode ser utilizado para converter imagens para o formato GIF, que possui uma restrição de no máximo 256 cores.

A ideia central do pontilhamento de Floyd-Steinberg é propagar o erro de quantização gerado ao converter um tom contínuo para um tom de preto e branco ou para uma cor da paleta limitada (como foi o caso nessa aplicação) para os pixels vizinhos não processados. Dessa forma, o algoritmo tenta distribuir o erro de maneira mais uniforme ao longo da imagem, resultando em uma representação visualmente mais agradável.

O processo começa pixel a pixel na imagem original. Cada pixel é avaliado e seu valor é arredondado para o tom mais próximo na paleta de cores. O erro entre o valor original e o valor arredondado é calculado e propagado para os pixels vizinhos não processados usando um conjunto específico de pesos. Os pesos utilizados indicam quanto do erro de quantização deve ser distribuído para os pixels vizinhos.

a) Imagem original b) Pontilhado ordenado c) Pontilhado de Floyd-Steinberg

```
void floydSteinbergDithering(int *pixels, int width, int height) {
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int pixelIndex = y * width + x;
            int oldPixel = pixels[pixelIndex];
            int newPixel = oldPixel > 128 ? 255 : 0;
            pixels[pixelIndex] = newPixel;

            int quantError = oldPixel - newPixel;

            if (x + 1 < width)
                pixels[pixelIndex + 1] += quantError * 7 / 16;
            if (x > 0 && y + 1 < height)
                pixels[pixelIndex + width - 1] += quantError * 3 / 16;
            if (y + 1 < height)
                pixels[pixelIndex + width] += quantError * 5 / 16;
            if (x + 1 < width && y + 1 < height)
                pixels[pixelIndex + width + 1] += quantError * 1 / 16;
        }
    }
}
```

O algoritmo de Floyd-Steinberg é implementado pela função "*floydSteinbergDithering*". Ela percorre todos os pixels da imagem e, para cada pixel, calcula um novo valor com base no valor original do pixel. Se o valor original for maior que 128, o novo valor será 255 (cor branca); caso contrário, o novo valor será 0 (cor preta). Essa etapa cria a imagem dithered, com apenas duas cores.

O algoritmo também calcula o erro de quantização, que é a diferença entre o valor original e o novo valor do pixel. Esse erro é distribuído para os pixels vizinhos ponderadamente. Os pixels vizinhos são atualizados de acordo com os coeficientes de difusão de erro definidos pelo algoritmo de Floyd-Steinberg: 7/16 para o pixel à direita, 3/16 para o pixel inferior esquerdo, 5/16 para o pixel abaixo e 1/16 para o pixel inferior direito.

Paralelização do Pontilhamento de Floyd-Steinberg - Versão 1

A paralelização nesse algoritmo é um desafio, pois a sua natureza consiste na modificação de um pixel que depende de um pixel anterior. Essa dependência de dados traz uma limitação.

```
void floydSteinbergDithering(int *pixels, int width, int height) {
    #pragma omp parallel for collapse(2)
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int pixelIndex = y * width + x;
            int oldPixel = pixels[pixelIndex];
            int newPixel = oldPixel > 128 ? 255 : 0;
            pixels[pixelIndex] = newPixel;

            int quantError = oldPixel - newPixel;

            // Calculate private variables for each thread
            int localQuantError = quantError;

            #pragma omp critical
            {
                if (x + 1 < width)
                    pixels[pixelIndex + 1] += localQuantError * 7 / 16;
                if (x > 0 && y + 1 < height)
                    pixels[pixelIndex + width - 1] += localQuantError * 3 / 16;
                if (y + 1 < height)
                    pixels[pixelIndex + width] += localQuantError * 5 / 16;
                if (x + 1 < width && y + 1 < height)
                    pixels[pixelIndex + width + 1] += localQuantError * 1 / 16;
            }
        }
    }
}
```

Na primeira versão da paralelização, a diretiva ***#pragma omp parallel for collapse(2)*** é usada para paralelizar o loop externo (referente ao eixo y, a altura da imagem) e o loop interno (referente ao eixo x, a largura da imagem). Essa diretiva indica que as iterações dos loops podem ser executadas em paralelo. Para cada pixel, o valor antigo do pixel é comparado com um valor limiar (128 neste caso) para determinar se o novo pixel será preto (0) ou branco (255). O valor do pixel é atualizado no array de *pixels*.

A região crítica ***#pragma omp critical*** é utilizada para garantir que as atualizações dos pixels vizinhos sejam feitas corretamente, pois existe dependência entre eles. Dentro da

região crítica, os pixels vizinhos são atualizados usando as fórmulas de difusão de erro de quantização de Floyd-Steinberg.

Como vamos observar, essa primeira versão terá um desempenho até mesmo pior ao da versão serial. Isso ocorre devido à necessidade da região crítica, o que causa um bloqueio da execução paralela nos threads.

Paralelização do Pontilhamento de Floyd-Steinberg - Versão 2

Para tentar resolver esse problema, foi utilizada uma abordagem diferente.

A abordagem consiste em dividir os pixels em blocos trapezóides, de forma a delimitar as regiões de pixels que podem ser processadas paralelamente, ou seja, que não possuem dependência. Na figura abaixo, a região sombreada mostra quais pixels precisam ter sido finalizados antes da região "*ECFG*" ser iniciada. Devido ao padrão de distribuição de erros, o pixel do canto inferior direito (no vértice *F*) pode ser processado somente depois que todos os pixels restantes forem processados. Se o triângulo "*CDE*" faz parte do vizinho bloco trapezoidal, então o paralelogramo "*ECFG*" define o bloco atual de pixels que precisa ser pontilhado.

Por outro lado, essa abordagem pode criar uma sobrecarga de criação de encadeamento, assim como muito tempo pode ser gasto na troca de contexto. Há também uma questão da granularidade, para definir o tamanho dos blocos. Ainda assim, a execução dessa abordagem mostrou-se ser bastante eficiente.

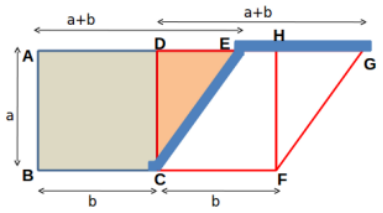


Figure 5. For the block-based CPU algorithm, consider two blocks ABCE and DCFG. The triangle DEC is the overlapping portion between these two blocks. If we have already processed block ABCE, then block DCFG needs to process only the parallelogram ECFG. We just need the error values along EC, EG (blue strips) and one error value along BC (blue block).

```c
#define BLOCK_SIZE 16

typedef struct {
    int row;
    int col;
} PrimalBlock;

void getPrimalBlock(int M, int N, int a, int b, int itr, PrimalBlock *primalBlock) {
    if (itr > 0 && itr <= N / b) {
        primalBlock->row = 1;
        primalBlock->col = itr;
    } else {
        primalBlock->row = ceil((itr - (N / b)) / 2.0);
        primalBlock->col = (N / b) - ((itr - (N / b)) % 2);
    }
}
```

É definida uma constante *BLOCK_SIZE* com valor 16, que representa o tamanho do bloco de pixels a ser processado, e uma estrutura *PrimalBlock* que contém as coordenadas de um bloco primário.

```
void getPrimalBlock(int M, int N, int a, int b, int itr, PrimalBlock *primalBlock) {
    if (itr > 0 && itr <= N / b) {
        primalBlock->row = 1;
        primalBlock->col = itr;
    } else {
        primalBlock->row = ceil((itr - (N / b)) / 2.0);
        primalBlock->col = (N / b) - ((itr - (N / b)) % 2);
    }
}
```

A função *getPrimalBlock* é definida para calcular as coordenadas do bloco primário com base nos parâmetros fornecidos (*M*, *N*, *a*, *b* e *itr*). Essa função é utilizada para determinar quais blocos primários cada thread irá processar.

```
void ditherBlock(int row, int col, int *pixels, int width, int height) {
    int xStart = col * BLOCK_SIZE;
    int yStart = row * BLOCK_SIZE;
    int xEnd = xStart + BLOCK_SIZE;
    int yEnd = yStart + BLOCK_SIZE;

    for (int y = yStart; y < yEnd && y < height; y++) {
        for (int x = xStart; x < xEnd && x < width; x++) {
            if (x >= 0 && y >= 0) {
                int pixelIndex = y * width + x;
                int oldPixel = pixels[pixelIndex];
                int newPixel = oldPixel > 128 ? 255 : 0;
                pixels[pixelIndex] = newPixel;

                int quantError = oldPixel - newPixel;

                if (x + 1 < width)
                    pixels[pixelIndex + 1] += quantError * 7 / 16;
                if (x > 0 && y + 1 < height)
                    pixels[pixelIndex + width - 1] += quantError * 3 / 16;
                if (y + 1 < height)
                    pixels[pixelIndex + width] += quantError * 5 / 16;
                if (x + 1 < width && y + 1 < height)
                    pixels[pixelIndex + width + 1] += quantError * 1 / 16;
            }
        }
    }
}
```

A função ditherBlock é definida para realizar o dithering em um bloco de pixels específico. Ela recebe as coordenadas do bloco, um ponteiro para o array de pixels, bem como a largura e altura da imagem.

```
t1 = omp_get_wtime();
#pragma omp parallel
{
int numThreads = omp_get_num_threads();
int threadId = omp_get_thread_num();
int totalBlocks = (numIterations + numThreads - 1) / numThreads;  // Update calculation
int start = threadId * totalBlocks;
int end = (threadId + 1) * totalBlocks;
if (end > numIterations) {
    end = numIterations;
}

for (int i = start; i < end; i++) {
    PrimalBlock primalBlock;
    getPrimalBlock(M, N, a, b, i, &primalBlock);

    for (int j = 0; j < totalBlocks; j++) {
        int row = primalBlock.row - j;
        int col = primalBlock.col - 2 * j;
        ditherBlock(row, col, pixels, width, height);
    }
}
}
t1 = omp_get_wtime() - t1;
```

M é definido como a altura da imagem, *N* como a largura da imagem, *a* e *b* como o tamanho do bloco. A variável *numIterations* é calculada com base nas dimensões da imagem e do tamanho do bloco. Essa variável representa o número total de iterações que o algoritmo de dithering irá executar.

A diretiva **#pragma omp parallel** é utilizada para iniciar a região paralela. Todas as variáveis definidas dentro desta região serão compartilhadas entre as threads. As variáveis *numThreads*, *threadId*, *totalBlocks*, *start* e *end* são definidas para controlar o trabalho a ser realizado por cada thread. A função **omp_get_num_threads()** retorna o número de threads disponíveis, **omp_get_thread_num()** retorna o ID da thread atual.

Um loop é executado a partir da variável *start* até *end* para processar os blocos primários. Cada thread é responsável por um conjunto de blocos primários. Dentro do loop externo, a função *getPrimalBlock* é chamada para calcular as coordenadas do bloco primário atual.

Em seguida, um loop interno é executado para processar os blocos secundários. O número de iterações é igual ao número de blocos secundários em cada bloco primário.
Para cada bloco secundário, as coordenadas do bloco são calculadas subtraindo um valor apropriado das coordenadas do bloco primário. Finalmente, a função *ditherBlock* é chamada para aplicar o dithering a esse bloco.

Execução
Serial:

```
victor@DESKTOP-HB517L1:~/Studio/MC970/Projeto/Final$ ./perf stat --repeat 3 -e cycles:u,instructions:u,cache-misses:u,L1-dcach
e-loads:u,L1-dcache-load-misses:u,L1-dcache-stores,dTLB-loads,dTLB-load-misses ./floyd_serial
Arquivo de saída: output_floyd_serial.txt
Tempo de execução do Floyd-Steinberg Serial: 0.443825
Arquivo de saída: output_floyd_serial.txt
Tempo de execução do Floyd-Steinberg Serial: 0.446250
Arquivo de saída: output_floyd_serial.txt
Tempo de execução do Floyd-Steinberg Serial: 0.442431

 Performance counter stats for './floyd_serial' (3 runs):

      10093271972      cycles:u                                                ( +-  0.93% )  (62.50%)
      24853028317      instructions:u            #    2.46  insn per cycle     ( +-  0.17% )  (74.99%)
          5196344      cache-misses:u                                          ( +-  1.69% )  (75.00%)
       6205755466      L1-dcache-loads                                         ( +-  0.16% )  (75.00%)
          5539820      L1-dcache-load-misses     #    0.09% of all L1-dcache accesses  ( +-  1.81% )  (75.01%)
       3882504651      L1-dcache-stores                                        ( +-  0.29% )  (75.01%)
       6257975356      dTLB-loads                                              ( +-  0.27% )  (50.00%)
           294978      dTLB-load-misses          #    0.00% of all dTLB cache accesses  ( +-  6.33% )  (49.99%)

          3.4899 +- 0.0634 seconds time elapsed  ( +-  1.82% )
```

Tempo médio de execução serial em 3 rodadas foi de **0,444168** segundos.

Versão Paralelizada 1:

```
victor@DESKTOP-HB517L1:~/Studio/MC970/Projeto/Final$ ./perf stat --repeat 3 -e cycles:u,instructions:u,cache-misses:u,L1-dcach
e-loads:u,L1-dcache-load-misses:u,L1-dcache-stores,dTLB-loads,dTLB-load-misses ./floyd_parallel_1
Arquivo de saída: output_floyd_parallel_1.txt
Tempo de execução do Floyd-Steinberg Paralelo 1: 1.990814
Arquivo de saída: output_floyd_parallel_1.txt
Tempo de execução do Floyd-Steinberg Paralelo 1: 1.821337
Arquivo de saída: output_floyd_parallel_1.txt
Tempo de execução do Floyd-Steinberg Paralelo 1: 1.786220

 Performance counter stats for './floyd_parallel_1' (3 runs):

      29573447477      cycles:u                                                ( +-  3.48% )  (62.64%)
      26590591058      instructions:u            #    0.86  insn per cycle     ( +-  0.17% )  (75.08%)
          6012527      cache-misses:u                                          ( +-  0.75% )  (74.95%)
       6571894554      L1-dcache-loads                                         ( +-  0.10% )  (74.88%)
         73229283      L1-dcache-load-misses     #    1.11% of all L1-dcache accesses  ( +-  4.30% )  (74.99%)
       4024844804      L1-dcache-stores                                        ( +-  0.06% )  (74.95%)
       6582663795      dTLB-loads                                              ( +-  0.47% )  (50.25%)
           346953      dTLB-load-misses          #    0.01% of all dTLB cache accesses  ( +-  6.51% )  (50.17%)

          4.812 +- 0.135 seconds time elapsed  ( +-  2.80% )
```

Tempo médio de execução paralela ver. 1 em 3 rodadas foi de **1,868790** segundos.

Versão Paralelizada 2:

```
victor@DESKTOP-HB517L1:~/Studio/MC970/Projeto/Final$ ./perf stat --repeat 3 -e cycles:u,instructions:u,cache-misses:u,L1-dcach
e-loads:u,L1-dcache-load-misses:u,L1-dcache-stores,dTLB-loads,dTLB-load-misses ./floyd_parallel_2
Arquivo de saída: output_floyd_parallel_2.txt
Tempo de execução do Floyd-Steinberg Paralelo 2: 0.232868
Arquivo de saída: output_floyd_parallel_2.txt
Tempo de execução do Floyd-Steinberg Paralelo 2: 0.203036
Arquivo de saída: output_floyd_parallel_2.txt
Tempo de execução do Floyd-Steinberg Paralelo 2: 0.211619

 Performance counter stats for './floyd_parallel_2' (3 runs):

      10142346025      cycles:u                                                ( +-  1.27% )  (61.83%)
      24707252638      instructions:u            #    2.43  insn per cycle     ( +-  0.43% )  (74.66%)
          7903631      cache-misses:u                                          ( +-  7.40% )  (75.04%)
       6260283830      L1-dcache-loads                                         ( +-  0.46% )  (75.70%)
          7269080      L1-dcache-load-misses     #    0.12% of all L1-dcache accesses  ( +-  3.59% )  (76.25%)
       3869190598      L1-dcache-stores                                        ( +-  0.23% )  (75.64%)
       6189631921      dTLB-loads                                              ( +-  0.99% )  (48.85%)
           940299      dTLB-load-misses          #    0.01% of all dTLB cache accesses  ( +-  6.57% )  (49.12%)

          3.653 +- 0.382 seconds time elapsed  ( +- 10.46% )
```

Tempo médio de execução paralela ver. 2 em 3 rodadas foi de **0,215841** segundos.

De maneira análoga ao Blur Gaussiano, podemos calcular o speedup da seguinte forma:

$$speedup = \frac{\text{sequential running time}}{\text{parallel running time}}$$

Portanto, o speedup foi de **0,2376** para a versão 1, portanto mais lento que a serial, e de **2,0578** para a versão 2, mais rápida que a serial. Este resultado também está dentro da lei de Amdahl, considerando aproximadamente que 90% do tempo de execução deve ter sido utilizado para o algoritmo, e os outros 10% para operações como leitura e escrita de arquivos.

$$speedup \leq \frac{1}{(1 - P_\sigma) + \frac{P_\sigma}{n}}$$

Nesse caso, o speedup máximo que poderíamos atingir seria de 3,077 para o sistema utilizado, com *n = 4* threads.

O perfilamento mostra que a abordagem da paralelização na versão 1, a versão ineficiente, também causou um aumento nos misses da cache L1 de dados (*L1-dcache-load-misses*), o que também coincide com o desempenho ruim.

Conclusão:

Foi possível entender que a programação paralela é extremamente poderosa e necessária para resolver a maior parte das tarefas computacionais (que poderiam levar até centenas de anos para serem concluídas), especialmente nos dias atuais, e que paralelizar não é tarefa simples. Dentre os muitos desafios encontrados neste projeto, assim como nos laboratórios da disciplina, podemos destacar:

Problemas da decomposição: identificar corretamente as partes do programa que podem ser executadas simultaneamente de forma independente, de modo a garantir um uso eficiente dos recursos paralelos disponíveis.
Problemas da sincronização: gerenciar a comunicação e coordenação entre as diferentes threads ou processos paralelos, garantindo que compartilhem corretamente os dados e evitem conflitos de acesso (condições de corrida).
As dependências de dados: como lidar com as dependências entre as diferentes partes do programa, garantindo que os dados necessários estejam disponíveis antes de serem utilizados, o que pode requerer técnicas de sincronização e controle de fluxo.
A granularidade: encontrar um equilíbrio adequado entre o tamanho das tarefas paralelas e o overhead de comunicação e coordenação. Tarefas muito pequenas podem resultar em uma sobrecarga excessiva, enquanto tarefas muito grandes podem levar a um uso ineficiente dos recursos paralelos.
Overhead de comunicação: lidar com o custo adicional de comunicação entre os processos ou threads paralelos, que pode consumir recursos e diminuir o desempenho, especialmente em sistemas distribuídos.

Referências:

https://en.wikipedia.org/wiki/Kernel_(image_processing), Acesso em: 19 jun. 2023.[1]

http://www.brendangregg.com/perf.html Acesso em: 19 jun. 2023.[2]

https://en.wikipedia.org/wiki/Floyd–Steinberg_dithering, Acesso em: 23 jun. 2023.[3]

https://imisra.github.io/projects/dithering/HybridDithering.pdf, Acesso em: 26 jun. 2023.[4]