INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS PUEBLA

MULTIPROCESADORES

GRUPO 1

PROF: EMANUEL TORRES

ACTIVIDAD 2.1

MULTI-THREADS

FERNANDO LÓPEZ RAMÍREZ - A07144620

PUEBLA, PUEBLA A 30 DE SEPTIEMBRE DE 2022

INTRODUCCIÓN

El multithreading es el resultado de una interacción entre el hardware y el software. Los programas y procesos se dividen en hilos individuales y la CPU los procesa en estas unidades más pequeñas. Se distingue entre el multithreading basado en hardware y el basado en software.

En este caso realizamos diferentes tipos de ejercicios con programación en paralelo utilizando la librería OpenMP y usando nueva funcionalidades como for, schedule, collapse, sections, etc.

DESARROLLO

Se intentó realizar la actividad que cumplía con los siguientes puntos:

- Pasar imagen a escala de grises.
- Implementar paralelización.
- Hacer uso de memoria dinámica con **malloc.**
- Girar horizontalmente la imagen.
- Girar verticalmente la imagen.

De los puntos anteriores solo logré los 3 primeros, de ahí en fuera los giros no me salieron (hasta ahora). Hasta el día de la clase no se me había ocurrido utilizar las otras distintas funcionalidades que OpenMP proporciona para el correcto funcionamiento y manejo de memoria y datos que utiliza el procesador.

Imagen 1. Esta es la primera imagen que obtuve en escala de grises SIN utilizar paralelización.

Imagen 2. Esta es la primera imagen que obtuve en escala de grises CON paralelización.

Podemos observar que baja la calidad de imagen usando paralelización por el manejo de memoria, ya que por el momento no es el mejor.

CÓDIGO.

```
// Include libraries
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
// Define number of threads
#define NUM_THREADS 1

int main()
{
    // Set num of threads
    omp_set_num_threads(NUM_THREADS);

    // Declare pointers for image & new image
    FILE *image, *outputImage;

    // Open original image
    image = fopen("original_kitten.bmp", "rb");
    // image = fopen("sample.bmp", "rb");
    // Create new image
```

```
    outputImage = fopen("inverted_kitten_img/kitten_2.bmp", "wb");
    // outputImage = fopen("inverted_kitten_img/sample_2.bmp", "wb");

    // Declare width & height of the image
    long width;
    long height;
    // Define counter h
    int counter = 0, h = 0;
    int newLimit = 0, newStart = 0;
    // Declare RGB pixel chars
    unsigned char r, g, b;
    // Array for the first 54 header's line
    unsigned char xx[54];

    // Read first 54 header's line and move img memory counter
    for (int i = 0; i < 54; i++) {
        // Get header line from original img
        xx[i] = fgetc(image);
        // Set header line to new img
        fputc(xx[i], outputImage);
    }

    // Calculate the width & height of the original image
    height = (long)xx[20]*65536 + (long)xx[19]*256 + (long)xx[18] + 1;
    width = (long)xx[24]*65536 + (long)xx[23]*256 + (long)xx[22] + 1;

    // Print width & height
    printf("Img Width: %ld px\n", width);
    printf("Img Height: %ld px\n", height);

    // Calculate image size
    long imageSize = height * width * 3;

    // Print image size
    printf("Image size: %ld px\n", imageSize);

    // Measure initial time (t1)
    const double startTime = omp_get_wtime();
    double t1 = omp_get_wtime();

    // Define pointers of malloc
    // 2782080
    unsigned char* arrImgInput = (unsigned char*)malloc(2782080*sizeof(unsigned
char));
                        unsigned      char*      arrImgOutput      =      (unsigned
char*)malloc(2782080*sizeof(unsigned char));
    unsigned char pixel;
```

```
    int k = 0;
    while(!feof(image)) {
        // Get RGB values as received: B, G, R
        *(arrImgInput + k) = fgetc(image);
        *(arrImgOutput + k) = *(arrImgInput + k);
        k++;
    }

    #pragma omp parallel
    {
        // Start process with parallelism
        #pragma omp for
            for (int j = 0; j < imageSize; j+=3)
            {
                b = *(arrImgInput + j); // Blue
                g = *(arrImgInput + j + 1); // Green
                r = *(arrImgInput + j + 2); // Red

                // Create new pixel in gray scale
                pixel = 0.21*r + 0.72*g + 0.07*b;

                *(arrImgInput + j) = pixel;
                *(arrImgInput + j + 1) = pixel;
                *(arrImgInput + j + 2) = pixel;

                counter += 3;

                if (counter == width*3) {
                    j += 2;
                    counter = 0;
                }
            }


        #pragma omp for schedule(dynamic)
            for (int j = 0; j < imageSize; j+=3)
            {
                *(arrImgOutput + j) = *(arrImgInput + j);
                *(arrImgOutput + j + 1) = *(arrImgInput + j + 1);
                *(arrImgOutput + j + 2) = *(arrImgInput + j + 2);
            }
            // for (int row = 1; row <= height; row++)
            // {
            //     for (int subPx = 1; subPx <= width * 3; subPx++)
            //     {
              //          // arrImgOutput[h] = arrImgInput[(width * 3 * row) -
subPx];
```

```
            //          *(arrImgOutput + h) = *(arrImgInput + (subPx * width *
row));
            //          *(arrImgOutput + h + 1) = *(arrImgInput + (subPx * width
* row) + 1);
            //          *(arrImgOutput + h + 2) = *(arrImgInput + (subPx * width
* row) + 2);
            //          h+=3;
            //      }
            // }


        #pragma omp schedule(dynamic)
            for (int m = 0; m < imageSize; m+=3)
            {
                fputc(*(arrImgOutput + m), outputImage);
                fputc(*(arrImgOutput + m + 1), outputImage);
                fputc(*(arrImgOutput + m + 2), outputImage);
            }
    }

    // printf("\nCounter: %d\n", counter);
    printf("\nH: %d\n", h);

    // Measure final time (t2)
    double t2 = omp_get_wtime();

    // Calculate final time
    double finalTime = t2 - t1;

    // Print number of threads
    printf("Threads: %d\n", NUM_THREADS);
    // Print how much time did it take
    printf("It took (%lf) seconds\n", finalTime);

    // Close original image
    fclose(image);
    // Close new image
    fclose(outputImage);

    // End program execution
    return 0;
}

/*
    MALLOC:
    // Moving through malloc
    for (int j = 0; j < imageSize ; j++) {
        m[i] = *(p + i)
```

```
    }
*/

/*
    EXECUTE:
    gcc -fopenmp .\Actividad_2_1.c
    .\a.exe
*/

/*
    TASKS:
    + Gray scale.
    + Parallelism using "pragma for".
    / Use of malloc to calculate width & height and to flip image.
    - Horizontal flip (ask option with scanf)
    - Vertical flip (ask option with scanf)
*/

/*

    // Tremendo algoritmo para voltear filas

    original:
    1 2 3 4 5
    6 7 8 9 10

    resultado:
    5 4 3 2 1
    10 9 8 7 6

    int width = 5, height = 5;
    int factor = 0, h = 0;
    int newLimit = 0, newStart = 0;

    for (int k = 0; k < height; k++) {
        if (k == 0) printf("Pragma #2\n");
        factor = k + 1;
        newStart = (width * factor * 3) - 1;
        newLimit = width * k;
        for (int l = newStart; l >= newLimit; l--) {
            arrImgOutput[h] = arrImgInput[l];
            h++;
        }
    }
*/
```

CONCLUSIÓN

Por el momento puedo concluir que debí haber investigado y experimentado más funcionalidades que la librería OpenMP nos ofrece ya que me quedé estancado utilizando únicamente “for”. También, el saber utilizar los apuntadores es muy importante ya que si no se hace un buen uso de ellos, podemos realizar un mal manejo de memoria.

REFERENCIAS

Multithreading: más potencia para los procesadores. (n.d.). IONOS Digitalguide. Retrieved September 30, 2022, from https://www.ionos.mx/digitalguide/servidores/know-how/explicacion-del-multithreading/