Computer Architecture and Organization - DA – 2

Program on image processing that is, to process and blur images, based on OPENMP. The program takes in a BMP image file and outputs a blurred image.

Serialized/Normal code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bmp.h"
//we hardcode the image dimensions here
int imgwidth = 512;
int imgheight = 512;
int size = 512*512;
int pix_num = 255;

void filewrite(unsigned char* d, int width, int height){
    struct bmp_id id;
    id.magic1 = 0x42;
    id.magic2 = 0x4D;

    struct bmp_header header;
    header.file_size = width*height+54 + 2;
    header.pixel_offset = 1078;

    struct bmp_dib_header head;
    head.header_size = 40;
    head.width = width;
    head.height = height;
```

```
    head.num_planes = 1;
    head.bit_pr_pixel = 8;
    head.compress_type = 0;
    head.data_size = width*height;
    head.hres = 0;
    head.vres = 0;
    head.num_colors = 256;
    head.num_imp_colors = 0;

    char padding[2];

    unsigned char* arr = (unsigned char*)malloc(1024);
    for(int c= 0; c < 256; c++){
        arr[c*4] = (unsigned char) c;
        arr[c*4+1] = (unsigned char) c;
        arr[c*4+2] = (unsigned char) c;
        arr[c*4+3] = 0;
    }

    FILE* fp = fopen("out.bmp", "w+");
    fwrite((void*)&id, 1, 2, fp);
    fwrite((void*)&header, 1, 12, fp);
    fwrite((void*)&head, 1, 40, fp);
    fwrite((void*)arr, 1, 1024, fp);
    fwrite((void*)d, 1, width*height, fp);
    fwrite((void*)&padding,1,2,fp);
    fclose(fp);
}
```

```
unsigned char* fileread(char* filename){

    FILE* fp = fopen(filename, "rb");

    int width, height, offset;

    fseek(fp, 18, SEEK_SET);
    fread(&width, 4, 1, fp);
    fseek(fp, 22, SEEK_SET);
    fread(&height, 4, 1, fp);
    fseek(fp, 10, SEEK_SET);
    fread(&offset, 4, 1, fp);
    unsigned char* d = (unsigned char*)malloc(sizeof(unsigned char)*height*width);

    fseek(fp, offset, SEEK_SET);  // to move the file pointer by our offset value.
    fread(d, sizeof(unsigned char), height*width, fp);
    fclose(fp);   //close file
    return d;
}

void flip_horizontal( unsigned char *d, unsigned int cols, unsigned int rows ) {
    unsigned int left = 0;
    unsigned int right = cols;
    for(int r = 0; r < rows; r++){
        while(left != right && right > left){
            int temp = d[r * cols + left];
            d[(r * cols) + left] = d[(r * cols) + cols - right];
```

```
            d[(r * cols) + cols - right] = temp;
            right--;
            left++;
        }
    }
}
int main(int argc, char** argv){
    int thread_num = atoi(argv[2]);
    unsigned char* image = fileread(argv[1]);
    unsigned char* new_img = malloc(sizeof(unsigned char) * size);

            for(int i=0;i<size;i++)
            {
                    printf("%d\n ",image[i]);
            }
   int* invert = (int*)calloc(sizeof(int), pix_num);
    for(int i = 0; i < size; i++){
        invert[image[i]]++;
                    //printf("%d\n",invert[image[i]]);
    }
    float* new_temp = (float*)calloc(sizeof(float), pix_num);
    for(int i = 0; i < pix_num; i++){
        for(int j = 0; j < i+1; j++){
            new_temp[i] += pix_num*((float)invert[j])/(size);
        }
    }
    for(int i = 0; i < size; i++){
        new_img[i] = new_temp[image[i]];   //storing the new image
```

```
    }
    filewrite(image, imgwidth, imgheight);
}
```

Parallelized code:

Pragmas are used to parallelize the for loops while reading the pixels from the image and while shifting them to produce a blur effect.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include "bmp.h"

const int imgwidth = 1080;
const int imgheight = 720;
const int size = 720*720;
const int pix_num = 255;

void filewrite(unsigned char* d, int width, int height){
    struct bmp_id id;
    id.magic1 = 0x42;
    id.magic2 = 0x4D;

    struct bmp_header header;
    header.file_size = width*height+54 + 2;
    header.pixel_offset = 1078;

    struct bmp_dib_header head;
    head.header_size = 40;
```

```
    head.width = width;
    head.height = height;
    head.num_planes = 1;
    head.bit_pr_pixel = 8;
    head.compress_type = 0;
    head.data_size = width*height;
    head.hres = 0;
    head.vres = 0;
    head.num_colors = 256;
    head.num_imp_colors = 0;
    char padding[2];
    unsigned char* arr = (unsigned char*)malloc(1024);
    for(int c= 0; c < 256; c++){
        arr[c*4] = (unsigned char) c;
        arr[c*4+1] = (unsigned char) c;
        arr[c*4+2] = (unsigned char) c;
        arr[c*4+3] = 0;
    }
    FILE* fp = fopen("out.bmp", "w+");
    fwrite((void*)&id, 1, 2, fp);
    fwrite((void*)&header, 1, 12, fp);
    fwrite((void*)&head, 1, 40, fp);
    fwrite((void*)arr, 1, 1024, fp);
    fwrite((void*)d, 1, width*height, fp);
    fwrite((void*)&padding,1,2,fp);
    fclose(fp);
}
```

```
unsigned char* fileread(char* filename){
    FILE* fp = fopen(filename, "rb");
    int width, height, offset;
    fseek(fp, 18, SEEK_SET);
    fread(&width, 4, 1, fp);
    fseek(fp, 22, SEEK_SET);
    fread(&height, 4, 1, fp);
    fseek(fp, 10, SEEK_SET);
    fread(&offset, 4, 1, fp);
    unsigned char* d = (unsigned char*)malloc(sizeof(unsigned char)*height*width);
    fseek(fp, offset, SEEK_SET);
    fread(d, sizeof(unsigned char), height*width, fp);
    fclose(fp);
    return d;
}

void flip_horizontal( unsigned char *d, unsigned int cols, unsigned int rows ) {
    unsigned int left = 0;
    unsigned int right = cols;

    for(int r = 0; r < rows; r++){
        while(left != right && right > left){
            int temp = d[r * cols + left];
            d[(r * cols) + left] = d[(r * cols) + cols - right];
            d[(r * cols) + cols - right] = temp;
            right--;
            left++;
        }
```

```
    }
}


int main(int argc, char** argv){

    if(argc != 3){
        printf("Useage: %s image thread_num\n", argv[0]);
        exit(-1);
    }
    int thread_num = atoi(argv[2]);

    unsigned char* image = fileread(argv[1]);
    unsigned char* new_img = malloc(sizeof(unsigned char) * size);

    int* invert = (int*)calloc(sizeof(int), pix_num);
            int image_val;
#pragma omp parallel for num_threads(thread_num), private(image_val)
    for(int i = 0; i < size; i++){
        image_val = image[i];
#pragma omp critical
        invert[image_val]++;
    }
    float* new_temp = (float*)calloc(sizeof(float), pix_num);
    int i,j;
#pragma omp parallel for num_threads(thread_num) schedule(static,1),private(j)
    for(i = 0; i < pix_num; i++){
        for(j = 0; j < i+1; j++){
```

```
                        #pragma omp atomic
            new_temp[i] += pix_num*((float)invert[j])/(size);
        }
    }
#pragma omp parallel for num_threads(thread_num)
    for(int i = 0; i < size; i++){
        new_img[i] = new_temp[image[i]];
    }
    filewrite(new_img, imgwidth, imgheight);
}
```

Programming language and hardware:

C is the programming language used and OPENMP is used to parallelize the program. CPU used is Intel i7-6500u which runs at a base frequency of 2.50Ghz and is a quad core CPU.

The unique header files used are bmp,h and omp.h. The bmp library allows us to write and make changes to our bmp image file with in-built read and write functions. This library helps us to manipulate the image and blue the image by shifting the pixels.

Omp.h is the OPENMP header file which allows us to parallelize the program .We mention the parallelizable sections of our code by using the keyword pragma openmp parallel. This will fork additional threads to help us carry out the intensive task much more effectively.

Details:

The program that is being parallelized is the bmp image file blur er. It is used to remove the outlier pixels by correcting them by replacing the wrong pixel values with the outer normal pixel values.This program however, does this in a very rough manner so excessive blur is seen. It uses a very nice technique to corrects pixels with the help of the bmp.h header file and manipulates the pixels according to the offset value that we provide it. We must initially hardcode the image width, breadth and resolution so that the output image is not corrupted.

We first run the serial code on an image of resolution 512 x 512, with just one thread.

```
vijay@DESKTOP-HQTN1KO:~/temp/CAO$ time ./a.out test1.bmp 1

real    0m0.007s
user    0m0.004s
sys     0m0.000s
```

Then we compile the parallelized code with the following GCC command to ensure that openmp is invoked.(-fopenmp flag is added).

Gcc -fopenmp omp.c

```
vijay@DESKTOP-HQTN1KO:~/temp/CAO$ time ./a.out test1.bmp 1

real    0m0.009s
user    0m0.008s
sys     0m0.001s
```

With 3 threads,

```
vijay@DESKTOP-HQTN1KO:~/temp/CAO$ time ./a.out test1.bmp 3

real    0m0.027s
user    0m0.063s
sys     0m0.001s
```

With 4 threads,

```
vijay@DESKTOP-HQTN1KO:~/temp/CAO$ time ./a.out test1.bmp 4

real    0m0.026s
user    0m0.086s
sys     0m0.001s
```

Instead of observing a reduction in time, we see an increase as the number of threads increase and this is due to the number of context switches that happen when we use more threads and parallel code. Also, as we go through every pixel, leads to very poor cache management.

The effects are felt only when very high-resolution images are passed.