Operating Systems

Lab - CS 314

Balsher Singh – 180010008

LAB 5

Part-1

In Part-1 of the assignment, 2 sequential transformations were made namely ConvertGrayscale & HorizontalBlur. The idea and implemented is explained in following subsections:

1.1 ConvertGrayscale

A weighted average method was integrated for conversion of RGB image to Grayscale type. Since red has highest wavelengths of all the three colors, and green is the color that has not only less wavelength then red color but also green is the color that gives more soothing effect to the eyes. It means that we have to decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. So, the new equation that form is as follows:

values[p][q].red = (colour_blue * 0.114) + (colour_red * 0.299) + (colour_green * 0.587);

values[p][q].green = (colour_blue * 0.114) + (colour_red * 0.299) + (colour_green * 0.587);

values[p][q].blue = (colour_blue * 0.114) + (colour_red * 0.299) + (colour_green * 0.587);

where, values is the image matrix that is updated to create a new transformed Image.

According to this equation, Red has a contribution of about 29.9%, Green has a contribution 58.7% which is greater in all three colours and Blue has a contribution of only about 11.4%.

1.2 HorizontalBlur

Any kind of blur is essentially making every pixel more similar to those around it. In a horizontal blur, we can average each pixel with those in a specific direction, which gives the illusion of motion.

For this program, we will calculate each new pixels value as half of its original value. The other half is averaged from a fixed number of pixels to the right. This fixed number can be called the BLUR_AMOUNT. The larger the value, the more blurring that will occur.

This is done independently for the red, green, and blue components of each pixel.

This is expressed in the following algorithm:

- For each row of the image:
    - For each pixel in the row:
        - Set rr to be half the pixels red component.
        - Set gg to be half the pixels green component.
        - Set bb to be half the pixels blue component.
        - For ii from 1 up to BLUR_AMOUNT:

- Increment rr by R×0.5$BLUR_AMOUNT$R×0.5BLUR_AMOUNT, where RR is the red component of the pixel ii to the right of the current pixel.
- Increment gg by G×0.5$BLUR_AMOUNT$G×0.5BLUR_AMOUNT, where GG is the green component of the pixel ii to the right of the current pixel.
- Increment bb by B×0.5$BLUR_AMOUNT$B×0.5BLUR_AMOUNT, where BB is the blue component of the pixel ii to the right of the current pixel.

- Save r,g,br,g,b as the new color values for this pixel.

You must also ensure you don't access pixels off the bounds of the image. For example, the third pixel from the right should only average itself (half weight), and the next two (at a quarter weight each).

For this program, set the BLUR_AMOUNT to 50.

Input →

Output for all parts →

(Grayscale + Horizontal Blur/Motion Effect)

➢ **Proof of Correctness**

Method to prove correctness of ordering of Pixels:- Compare the output files directly using the diff command in Terminal. The command compares the output files, and tells about the file difference information if any i.e. command is :- diff correct_file.ppm output_file.ppm. I also printed the pixels on which each transformation was applied and it was observed that the second transformation was

only applied to the pixels on which first transformation was already applied thus proving correctness of implementation.

➢ **Run time & Speed up comparison**: -

Analysis and observation for run time and speedup comparison are given below: -

➢ Observation: - Observation table is given below which contains time taken by each approach to complete the transformation on an image of dimension 300 × 168

Program file	Program approach	Time Taken(μs)	
		File Size: 476 KB	File Size: 247588 KB
part1.cpp	Sequentially	33217	17340671
part2_a.cpp	Threads using Atomic	32505	17220299
part2_b.cpp	Threads using semaphore	32496	17246882
part2_2.cpp	Processes using Shared Memory	95205	19677030
part2_3.cpp	Processes using Pipes	85515	31154515

```
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part1
g++ part1.cpp
./a.out input.ppm output_part1.ppm
Time: 33217 microseconds
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_1
part2_1a  part2_1b
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_1a
g++ part2_1a.cpp  -lpthread
./a.out input.ppm output_part2_1a.ppm
Time: 32505 microseconds
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_1b
g++ part2_1b.cpp  -lpthread
./a.out input.ppm output_part2_1b.ppm
Time: 32496 microseconds
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_2
g++ part2_2.cpp  -lpthread
./a.out input.ppm output_part2_2.ppm
Time: 21567 microseconds
Time: 23809 microseconds
Time: 33919 microseconds
Time: 95205 microseconds
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_3
g++ part2_3.cpp  -lpthread
./a.out input.ppm output_part2_3.ppm
Time: 85515 microseconds
```

(With smaller file)

```
sher@DELL-G3:/mnt/c/Users/balsh/Downloads/180010008_lab5$ make part2_2
g++ part2_2.cpp  -lpthread
./a.out input.ppm output_part2_2.ppm
Time: 458913 microseconds
Time: 10201292 microseconds
Time: 15000747 microseconds
Time: 19677030 microseconds
```

(With larger file)

➢ **Analysis: -**

- The sequential program's approach should have taken more time than other approaches as it does everything sequentially whereas other approaches have multiple threads and processes which have some sort of parallelization in completing the required task.
- Sequential approach is more expensive than Parallel Threading using atomics and semaphores.
- Shared Memory is more than Sequentially because of extra writing and reading values to and from memory.
- Pipeline Approach is most expensive because in my T2, each pixel needs next 50 pixels' T1 completed so that it can process further and also, I am sending updated values from T1 to T2 in batches using pipeline, so that's why that sending and receiving time increases total run time in this case.
- Thus, on a combined we can see that Shared Memory approach is good but it only allows shared memory up to some limitations.
- Pipeline takes a lot time to transfer and read on other end if data is large.
- Atomic Locks and Semaphore Locks perform almost same (Semaphores better a little bit).

➢ **Ease/ difficulty of implementing/ debugging each approach:-** Approaches involving threads were fairly easy to implement as they shared data segment so it was easy to implement constructs like semaphores and atomic variables while approaches involving different processes were difficult to implement and debug because proper care was to be taken that correct values are passed by processes in correct order and they are properly received by other processes. Implementation through shared memory was specifically difficult because in this case structure of shared memory is to be maintained by us unlike implementation using pipes.