Assignment 5

Operating Systems Lab, Group 12

Cheryala Rohith	190010012	190010012@iitdh.ac.in
Mohammad Sameer	190010024	190010024@iitdh.ac.in

Part 1

In this part, we have written a simple image processing application in C++ with RGB to Grey-Scale and Edge Detection transformations. These transformations were performed sequentially one after the other. A brief description of these image transformations is given below.

1.1 RGB to Grey-Scale

We have used the weighted method, also called luminosity method, where the colors red, green and blue are weighed according to their wavelengths. The formula used by us is as follows:

```
greyScale = (0.2989 * R) + (0.5870 * G) + (0.1140 * B)
```

where,

- `R` = Value of Red pixel
- `G` = Value of Green pixel
- `B` = Value of Blue pixel

and `greyScale`, is the new pixel value given to Red, Green and Blue value of output pixel.

1.2 Edge Detection

We have used the Sobel Edge Detection Technique for this transformation. The Sobel operator emphasizes regions of high spatial frequency that correspond to edges by performing a 2-D spatial gradient measurement on an image. It is most commonly used to calculate the estimated absolute gradient magnitude at each point in a grayscale picture input. The operator is made up of a pair of 3×3 convolution kernels as shown.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 1: Convolution Kernals

One kernel is just 90 degrees rotated from the other. These kernels, one for each of the two perpendicular orientations, are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid.

The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these Gx and Gy). These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The gradient magnitude is given by:

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

Part 2

In this part, we have to make the application run parallel transformations on the input image using various process/threads and various synchronization primitives. Let RGB

to Grey-Scale conversion be transformation T1 and Edge Detection be transformation T2. A brief description of then is given below.

2.1 a

In this part T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself while synchronization is done using atomic operations.

In this part, we have used `atomic_flag_test_and_set` and `atomic_flag_clear` to make the steps performed by transformations seem atomic. To ensure that the pixels were received as sent, in the sent order we have used a global array `greyProgress`, which keeps track of i and j coordinates till where the grey scale transformation has progressed.

The edge detection at $(i\ ,\ j)$ coordinates can only happen when grey scale has finished progress till $(i+1\ ,\ j+1)$ coordinates.

2.1 b

In this part T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself while synchronization is done using semaphores.

In this part, we have used `sem_wait(&semLock)` and `sem_post(&semLock)` to ensure synchronization. To ensure that the pixels were received as sent, in the sent order we have used a global array `greyProgress`, which keeps track of i and j coordinates till where the grey scale transformation has progressed.

The edge detection at $(i\ ,\ j)$ coordinates can only happen when grey scale has finished progress till $(i+1\ ,\ j+1)$ coordinates.

2.2

In this part T1 and T2 are performed by 2 different processes that communicate via shared memory while synchronization is done using semaphores.

In this part, 2 different shared memories were used. One for sharing the image pixels information and one for sharing the information related to `greyProgress`. We have used `sem_wait(&semLock)` and `sem_post(&semLock)` to ensure synchronization. To ensure that the pixels were received as sent, in the sent order we have used an array `greyProgress` via shared memory, which keeps track of i and j coordinates till where the grey scale transformation has progressed.

The edge detection at $(i\ ,\ j)$ coordinates can only happen when grey scale has finished progress till $(i+1\ ,\ j+1)$ coordinates.

2.3

In this part T1 and T2 are performed by 2 different processes that communicate via pipes.

In this part, a single pipe was used for sharing the image pixels information. Synchronization was taken care by kernel since we are using pipe. To ensure that the pixels were received as sent, in the sent order, we read whenever some new information was written into pipe and stored it into a temporary image array.

The edge detection at $(i\ ,\ j)$ coordinates can only happen when grey scale has finished progress till $(i+1\ ,\ j+1)$ coordinates. This has been ensured using simple `if` condition.

Note

For all the above parts we have run the `diff` command with the output image of part 1. All of those were outputs were found to be exactly identical to one another.

Input and Output Images

To get the `ppm` file format from other format we have the following command:

```
convert image.jpg -compress none result.ppm
```

We have used the following image as the input to our application.

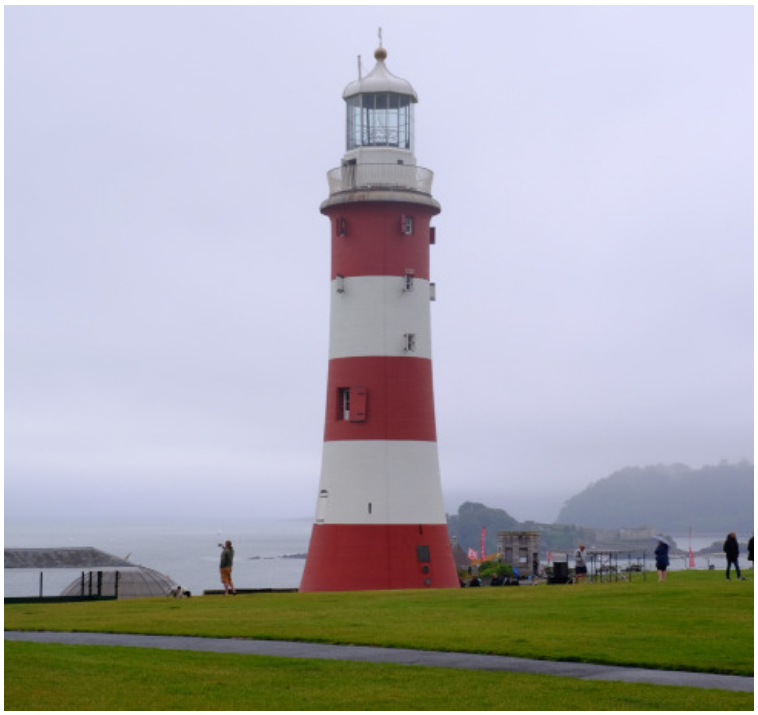

Figure 2: Input Image

Below is the greyscale transformation of the input image.

Figure 3: Greyscale Image

Below is the image that has been through the greyscale and edge detection transformation.

Figure 4: Output Image

Results and Discussion

The below table shows the run time of various versions of the application.

Program File	Method	Time Taken (in μ seconds)
part1.cpp	Sequential	127260
part2_1a.cpp	Threads using atomic operations	167956
part2_1b.cpp	Threads using semaphores	370271
part2_2.cpp	Processes using shared memory and semaphores	186055
part2_3.cpp	Processes using pipes	227165

- We have expected parallel implementation of transformations in part 2 would give better results than the part 1's sequential implementation. But from the run time observation that clearly isn't the case.
- The sequential implementation of part 1 ran in the least time of all. From this we drew a conclusion that the compensation of time that parallelism provided in part 2 was deprived by the synchronization and overhead of communication.

- A general observation was that threads running in parallel had easier method of communication via process' address space, making them run a bit faster compared to parallel processes.
- While running parallel processes, the part where pipes were used comparatively took more time. This can be attributed to the fact that pipes invoke kernel and mode switch, leading to greater overhead.
- The threads with semaphores had highest run time, which was not expected. Out intuition was that compared to spin wait, doing sleep and wake up in semaphores was expensive, leading to more run time.
- So, overall in this case the critical section was had very less computation. So going for a parallel transformation proved to be more expensive than the sequential implementation of transformations.

Ease/difficulty of implementation/debugging

- Overall the part 1 was the easiest to implement due to sequential transformations.
- In part 2, synchronization and making sure correct pixels were available was difficult to implement.
- In part 2_1, threads had advantage of communication via same address space, making their implementation a bit easier.
- Implementing pipes was a bit easier compared to shared memory which need manual synchronization.
- The shared memory implementation was particularly hard to both implement and debug due to the shape of memory. We needed a 2d array like memory, but instead we had to make use of a 1d array of shared memory.