Operating Systems Lab - CS 314

Rishit Saiya - 180010027, Assignment - 5

March 6, 2021

1 Part-1

In Part-1 of the assignment, 2 sequential transformations were made namely ***RGB to Grayscale*** & ***Edge Detection***. The idea and implemented is explained in following sub-sections:

1.1 RGB to Grayscale

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:

$$NI = (rData \times 0.2989) + (gData \times 0.5870) + (bData \times 0.1140)$$

where NI is the New Image variable.
According to this equation, Red has a contribution of about 30%, Green has a contribution 59% which is greater in all three colors and Blue has a contribution of only about 11%.

1.2 Edge Detection

For Edge Detection, Sobel Edge Detection Technique (Reference) was used. One kernel is simply the other rotated by 90°. These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these G_x and G_y). 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:

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

Often, this absolute magnitude is the only output the user sees, the two components of the gradient are conveniently computed and added in a single pass over the input image using the pseudo-convolution operator shown in Figure 1.

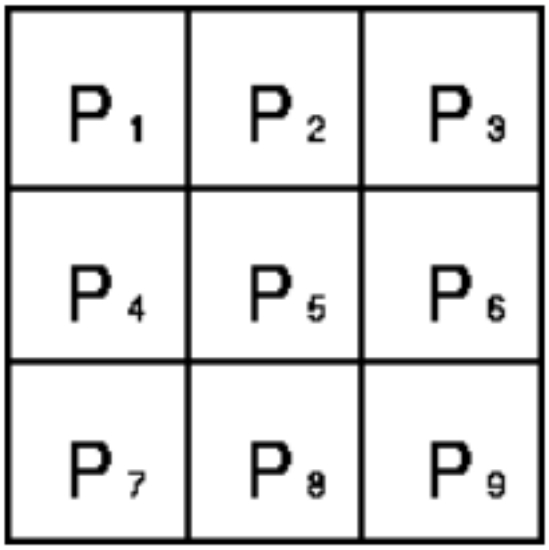

Figure 1: Pseudo-Convolution Kernels used compute approximate Gradient magnitudes efficiently

Using this kernel the approximate magnitude is given as follows:

$$|G| = |(P_1 + 2 \times P_2 + P_3) - (P_7 + 2 \times P_8 + P_9)| + |(P_3 + 2 \times P_6 + P_9) - (P_1 + 2 \times P_4 + P_7)|$$

2 Part-2

It is given that a processor is embedded with two cores. It is to be done by having the file read and T1 done on the first core, passing the transformed pixels to the other core, where T2 is performed on them, and then written to the output image file. The following subsections, different implementation using Synchronization Primitives is explained.

2.1 2.1.a

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself. In here, Synchronization had to be done using atomic operations.

```
// Initializes the atomicFlag to FALSE
atomic_flag atomicFlag2False = ATOMIC_FLAG_INIT;
```

```
// Bits intialized to check if the selected bit is correct
int goBack[2] = {0,0};
```

After using above logic and appropriate flags, we got same image.

2.2 2.1.b

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself. In here, Synchronization had to be done using Semaphores.

```
sem_t binarySemaphore;
sem_wait(&binarySemaphore);
sem_post(&binarySemaphore);
```

Using above mentioned semaphores, we implemented the Semaphore synchronization primitive.

2.3 2.2

T1 and T2 are performed by 2 different processes that communicate via shared memory.

```
int shmid = shmget(key, sizeof(struct pixel) * height * width, 0666 | IPC_CREAT);
sem_t *binarySemaphore = sem_open(semaphoreName, O_RDWR);
int shmid2 = shmget(limitKeyofT2, sizeof(int) * 2, 0666 | IPC_CREAT);
goBack = (int *)shmat(shmid2, NULL, 0);
```

Using above mentioned pieces of code, they were used in appropriate while loops to implement the IPC using Shared Memory.

2.4 2.3

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

```
int pipefds[2];
int pipeDesc1;

pipeDesc1 = pipe(pipefds);

if (pipeDesc1 == -1)
    perror("pipe");

int pipefds2[2];
```

```
int pipeDesc2;

pipeDesc2 = pipe(pipefds2);

if (pipeDesc2 == -1)
    perror("pipe");
```

After using such Pipe File descriptors and implementation, expected results were achieved.

3 Images & Results

Images of PP3 format were found on this reference. Figure 2 shows Input File 1. Figure 3 shows its corresponding output. Figure 4 shows Input File 2. Figure 5 shows its corresponding output.

Figure 2: Input File - 1

4 Proof of Correctness

The operations applied are as follows:

$$T1 \cdot [A]_{3\times3}$$

$$T2 \cdot [B]_{3\times3}$$

Here $[A]_{3\times3}$ and $[B]_{3\times3}$ during time of operation should never have common elements, otherwise partially modified values by T_j will be picked up by T_i which will result in wrong output. Furthermore, it was also checked using `diff` command.

In order to ensure that, this doesn't happen we have kept a size 2 array `goBack = {0,0}` available to both T1 and T2 functions as follows:

Figure 3: Output File - 1 - Post Transformations

Figure 4: Alternate Input File - 1

- In *2.1.a*, it is global, since global variables are accessible by threads.
- In *2.1.b*, it is global, since global variables are accessible by threads.
- In *2.2*, it is pushed to a separate Shared Memory (`shared_memory` in code) of (size(int) $\times$ 2) other than the pix_map.
- In *2.3*, it is pushed to separate pipe other than the `pix_map`.

5 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. But it isn't happening as expected.

It might be due to transformations we have used are such that critical section has very less computation so speed up obtained through parallelization is more than compensated by

Figure 5: Alternate Output File - 1 - Post Transformations

Program File	**Sync Primitives**	**Time Taken (in μ secs)**	
		Input_File1	**Input_File2**
part1.cpp	Sequentially	18742	9784112
part2_1a.cpp	Threads using Atomic Variables	28347	15017499
part2_1b.cpp	Threads using Semaphores	71039	37703140
part2_2.cpp	Threads using Shared Memory	43489	9181206
part2_3.cpp	Processes using Pipes	49957	18200153

Table 1: Analysis across 2 different images varying on size extremes

increased overhead of communication in other approaches.

Sequential approach is comparatively expensive than Parallel Threading using Atomic Variables and Semaphores. Furthermore, Shared Memory is more expensive than Sequentially because of extra writing and reading values to and from memory. Pipeline Approach is most expensive because sending `newarrayOfData` increases total run time in this case.

Thus, on a combined section, 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).

In order to make this theory concrete, delay loops were deliberately introduced in each transformation and observed was significant speed up in other approaches as compared in the case of sequential approach execution. Table 1 shows the time taken for various different program and corresponding synchronization primitives used.

`Input_File1` was taken same as above mentioned Figure 2. For `Input_File2`, a large file of around 250MB was used. Table 1 shows analysis on it.

PS: For Larger PPM files, Part 2_2 was run before any other parts because of limited Shared Memory Size and eventually would lead to Segmentation Fault.

6 Ease/Difficulty of Implementing/Debugging in Approaches

The approaches which involved threads were fairly clear to implement as they shared data segment so it was easy to implement constructs primitives like semaphores and atomic variables.

While approaches involving different processes were difficult to implement and debug because proper care had 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 complicated because in this case structure of shared memory is to be maintained by us unlike implementation using pipes.