Convert Colour Image Into Gray-Scale Image
Final Project Report

Saurabh Kakade
School of Informatics, Computing & Cyber Systems
Northern Arizona University, Flagstaff, AZ, U.S.A.
sk2354@nau.edu

Figure 1: Colour Image

Figure 2: Gray Scale Image (Sequential)

***Abstract*—Colour image into gray scale image converter is a program which uses OpenMp method to convert every pixel of colour image into gray scale image. All the pixel conversion will be done in parallel to reduce the total time required by sequential method. The plotting of every pixel conversion will be seen on the newly created .bmp file.**

***Keywords*-Image processing, gray-scale, OpenMP**

I. Introduction

Conversion of a colour image (Figure 1) into a gray-scale image (figure 2) is computed in C language programming where conversion of every image pixel into gray scale pixels. This program uses OpemMp method to make some blocks of code execute in parallel and tries increase the overall performance as compared to sequential approach. All the 8 bits .bmp images are converted from colour to gray scale. A specified folder must contain the source colour image file where the output grey scale image will be generated. OpenMP parallel code has been applied and performance timing is measured. There are still some major and minor complications faced during the project execution for various reasons which is also discussed.

II. Without parallelism

Execution of such conversion program in sequential approach takes the longer time consumption as every pixel gets converted one after another. Even on great computational system, the sequential approach will not utilize the machine power efficiently. For smaller sized images the time consumption would rarely matter. But for larger images the conversion will be slower and time consuming. In order to prevent such lengthiest approach, program is converted into parallel programming via OpenMp concept.

III. Image Resolution

The input image to the program can have maximum resolution of 800x800 which is the limitation occurred later while developing the project. This is because of .bmp file having 24 bit depth supporting such resolution. This program is made for 8 and 24 bit depth of .bmp image format and not for 32bit depth.

IV. Background

Till now there are very few attempts made on image processing in C language and found no projects as such to compare which performs colour image transformation

```
for(int i =0;i<imgSize;i++)
{
    buffer[i][0] = getc(fIn); //red
    buffer[i][1] = getc(fIn); //green
    buffer[i][2] = getc(fIn); //blue
    int temp =0;

    //grey scale conversion constants
    temp = (buffer[i][0]*0.3 ) +
           (buffer[i][1]*0.59) +
            (buffer[i][2]*0.11);

    putc(temp,fOut);
    putc(temp,fOut);
    putc(temp,fOut);
}
```

Figure 3: Sequential approach FOR LOOP

into grey scale image using any parallel programming techniques specially OpenMP. Most of such transformation is done in other languages with more efficiency and accuracy with parallel programming.

V. Approach to the problem

After understanding the image processing concept, C language program was made with sequential approach. The main FOR-LOOP used for image conversion is shown is the figure 3. Such FOR-LOOP is then transformed into parallel programming with OpenMP technique. The transformed FOR-LOOP in OpenMP is shown in figure 4. The getch() and putch() file operations were kept in critical section because of three dimension of every pixel and they cannot be made into parallel. Efforts were made to remove the critical section but race condition was taking place. In this image processing, race condition means output image getting more random pixels as shown is figure 5. Also we need to convert three dimensions of RGB at a time in every pixel conversion. Improvement from the previous approach figure 6 is made.

VI. Other Example with improvised code

Consider another image as shown in figure 7 to crosscheck the validity. This is done with improvised OpenMP approach from figure 4. This colour image is transformed to grey scale image as shown in figure 8. All the timings are calculated from this image transformation.

VII. Actual Performance

Consider figure 9 for sequential performance of the program. It shows better performance in without com-

```
#pragma omp parallel for private(i) shared(buffer) schedule(static)
    for(i =0;i<imgSize;i++)
    {
        #pragma omp critical
        {
            buffer[i][0] = getc(fIn); //red
            buffer[i][1] = getc(fIn); //green
            buffer[i][2] = getc(fIn); //blue
        }
            temp =0;
            temp = (buffer[i][0]*0.3 ) +
                   (buffer[i][1]*0.59) +
                    (buffer[i][2]*0.11);

        #pragma omp critical
        {
            putc(temp,fOut);
            putc(temp,fOut);
            putc(temp,fOut);
        }
    }
```

Figure 4: OpenMP FOR LOOP

piler optimisation mode. It performs faster due to less computation. Total four threads were used as programming computer has four hardware cores on system. Now consider figure 10, where the performance of both compiler optimisation and without it is shown. Due to less computation, the performance is less than sequential. Also in order to tackle race conditions, two critical operations are used which causes more time. As this program is for 24 bit depth image, only resolution up to 800x800 = 64000 pixels are transformed. For such lower computation, multi threads are showing non efficiency in results. But if we develop 32 bit .bmp image processing which has resolution higher than 1000x1000, then this program will show great time efficiency. After executing this program for several times for higher resolution, the drawback to limited resolution came forward and then the major limitation of this project was discovered. Implementation for 32bit .bmp image format was also made but it took long time and dint worked successfully. So this program was kept as basic project.

VIII. Specific Challenges with updates

- Resolution of larger image size can be detected via this program but unable to perform image processing task. It leads to error of segmentation fault. Status: higher resolution needs 32 bit .bmp image format which needs to be implemented more which is the solution found.
- For smaller images, performance of parallel code take greater time than sequential approach. In order to check performance for larger images, larger image must execute successfully which is not yet achieved. Status: True and still the same issues.
- Multiple sub formats are available for .bmp format. Inculcating every format will be extra time consuming. So for this project, we will stick to only basic format.

Figure 5: Race Conditions from older OpenMP approach

```
FILE *fIn = fopen("images/lena_color.bmp","rb");
FILE  *fOut = fopen("images/lena_gray.bmp","wb");

#pragma omp parallel private(i) shared(buffer)
{
        #pragma omp for nowait
        for(int i =0;i<imgSize;i++)
        {
                buffer[i][0] = getc(fIn);     //red
                buffer[i][1] = getc(fIn);     //green
                buffer[i][2] = getc(fIn);     //blue
                int temp =0;

                temp = (buffer[i][0]*0.3 )+(buffer[i][1]*0.59)+(buffer[i][2]*0.11);
                //grey scale conversion

                putc(temp,fOut);
                putc(temp,fOut);
                putc(temp,fOut);
        }
}

printf("Success!\n");
```

Figure 6: Older Approach for FOR-LOOP

Figure 7: New Image Colour

IX. Challenges/Status

- Check given file type for .bmp and output error for other files. Status: Achieved.
- File path needs to be mentioned for image file and output file. Status: Achieved and will be mentioned inside program.
- Program works for .bmp format but various other popular formats like .png and .jpg must be inculcated in future. Status: Only basic .bmp format is taken into consideration for this project. After successful completing the first .bmp format, other formats will be added in code.
- Crosscheck input and output files and check for any missed processed pixel as sanity check. Status: Its accurate and program processes properly each and every pixels.

X. Checklist/Status

- Find the number of pixels present in an image by multiplying coordinates of image (x,y). Status: Achieved.
- With OpenMP code, number of pixels will be distributed between threads to perform gray scale conversion function. Status: Achieved.
- Each thread will perform read (getc), convert the pixel in gray scale and write it to new file (putc). Status: Done.
- With the help of nowait clause, threads wont wait for each other to finish their task and will cross the barrier. Status: Nowait clause was not required in this program.
- Check for various other image formats other than .bmp like .jpg and .png Status: After completing of basic format .bmp, other formats will be added.
- Check for load balancing between threads and try to achieve the best results with the help of omp-get-wtime(). Status: Threads are showing load imbalances because of lower computation in program.
- To check performance of any size image, outer loop will iterate over image vs. inner loop of gray scale conversion will be analysed. Status: Only single loop is used.
- Compare input image resolution with output solution as sanity check. Status: For small resolution images, the program works perfect.

Figure 8: Grey Scale Image from OpenMP

```
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ gcc -o3 main.c -fopenmp -o main
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ ./main

Height: 800
Width: 800
Number of pixels: 640000
Success!
Elapsed time in OpenMP sequential: 0.074
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ gcc main.c -fopenmp -o main
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ ./main

Height: 800
Width: 800
Number of pixels: 640000
Success!
Elapsed time in OpenMP sequential: 0.052
saurabh@saurabh-desktop:~/RGB_To_Grayscale$
```

Figure 9: Sequential Output

XI. Milestones/Status

- Process any image size. Status: only below 800x800 resolution is working.
- Achieve lowest execution time. Status: Program of 32bit .bmp image format will achieve the lowest computation time as the for loop will be same.
- Perform best load balancing. Status: It was not able to check for lower image size.
- No threads should wait idle. Status: Static scheduling was used as computational was regular and repetitive.
- All pixels should be processed. Status: Achieved.

XII. Preliminary Ideas/Status

- Compute and find RGB attributes of a pixel and convert each attribute to gray scale. Status: Achieved.
- Value of Red attribute of a pixel should be multiplied by value 0.3, similarly green by 0.59 and blue by 0.11. Status: Achieved.
- Reading and writing outside files of type .bmp in a program and performing image processing. Status: Other formats will be added once basic .bmp format works for larger resolution files.
- Convert only a section of code which performs image processing to gray scale into parallel programming. Status: Achieved.

```
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ gcc -o3 main.c -fopenmp -o main
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ ./main

Height: 800
Width: 800
Number of pixels: 640000
Success!
Elapsed time in OpenMP sequential: 0.264
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ gcc main.c -fopenmp -o main
saurabh@saurabh-desktop:~/RGB_To_Grayscale$ ./main

Height: 800
Width: 800
Number of pixels: 640000
Success!
Elapsed time in OpenMP sequential: 0.225
saurabh@saurabh-desktop:~/RGB_To_Grayscale$
```

Figure 10: OpenMP Output

XIII. Things learned from this project

- Always perform parallel programming where complexity of computation is very high.
- Sometimes for lower computation, sequential approach performs better than parallel programming.
- Learnt the idea behind image processing and how parallel computation can be helpful in achieving better time performance.
- Avoid critical condition clause as mush as possible as it creates unnecessary locks and unlocks for N amount of iterations.
- Parallel programming efficiency in image processing can be only achieved when resolution of the image is way higher or transformation processing is complex.
- File operations in OpenMP programming needs to be handled carefully as some functions were not performing accurate.
- Understanding image formats and their execution in C language.
- Got to know where to apply parallel programming in real life applications and how can be achieved by changing image sub formats for better performances.
- OpenMP programming is far better than compared to pthreads or semaphores as it takes less programming ideology.

XIV. Conclusion

From this project it is clearly seen that parallel programming for image transformation can be achieved only when the resolution is higher than 1000x1000. This can be achieved from 32bit .bpm image formats. For resolution in 24bit .bmp image format, the parallel programming efficiency was not achieved perfectly. Sequential approach for such low resolution showed

better result. But for higher images, sequential approach will be not efficient and here the OpenMP will play a vital role in achieving the performance.