EECE 6520. Parallel & MP Architecture

Final Project Report

Image Edge Detection using Sobel Filter

Spring 2016

Group Members:

Sayali Vaidya

Deepak Bansal

Ganesh Kurapati

Bhavya Guntupally

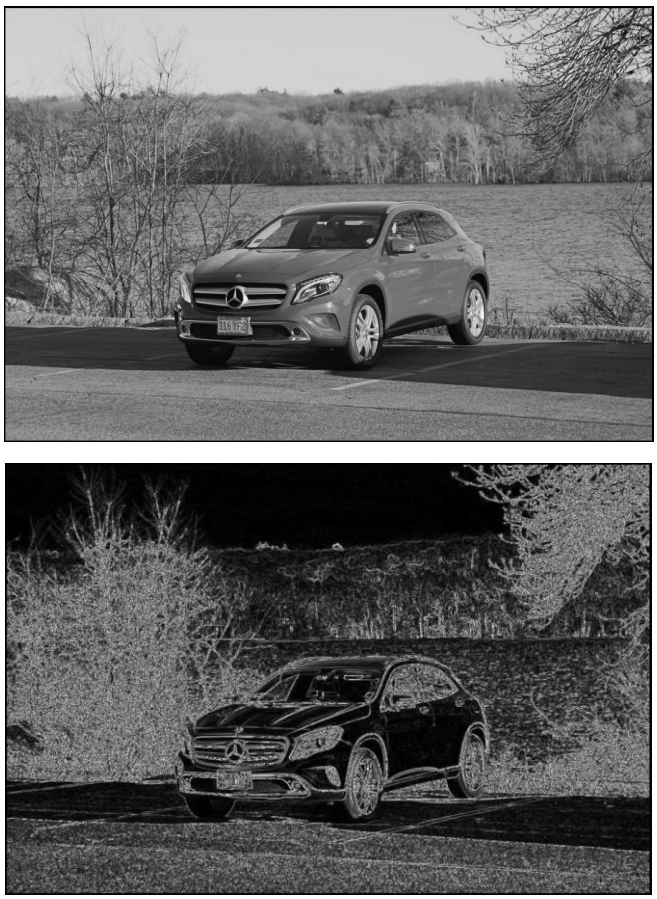
INDEX

Introduction:

The aim of this project was to parallelize the image edge detection process using Sobel filter. Sobel filters are one of the most commonly used methods to detect the edges of the image. In this two masks namely Horizontal and Vertical Masks are used to find the horizontal and vertical edges respectively. We have tried to parallelize the functioning of the Sobel filter by using two different methods namely: OpenMP and Pthreads.

Edge Detection:

It is a general name for a class of routines and techniques that operate on an image and results in a line drawing of the image. The lines represented changes in values such as cross sections of planes, intersections of planes, textures, lines, and colors, as well as differences in shading and textures. The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. Using some of the mathematical approaches, the edges can be detected. In this project, we are using sobel filter. Edge detection is one of the fundamental steps in image processing, image analysis, image pattern recognition, and computer vision techniques.



Edge detection sample

Image edge detection has application in various areas such extracting important objects from image, identifying tracking objects for computer vision applications, identifying objects and anomalies in medical imaging, reducing unwanted background data from images etc.

Sobel Filter:

Sobel operator is used in image processing and computer vision, particularly in edge detection algorithms where it creates an image emphasizing edges. It is also known as Sobel-Feldman operator.

It consists of two kernels (Masks) which detect horizontal and vertical changes in an image

- The 3x3 Sobel kernels are:

  Horizontal

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

  Vertical

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

Algorithm:

1. Read the ppm image into matrix form. Let image[x][y] represents the matrix of the image.
2. Convolution of image matrix with Horizontal mask

Horizontal Mask = $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

Image[x][y] * H[3][3] = temp1[x][y]

3. Convolution of image with vertical mask (transpose of the above matrix)

Vertical mask V =

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Image[x][y] * V[3][3] = temp2[x][y]

4. Resultant matrix = sqrt[(temp1)^2 + (temp2)^2]
   This resultant matrix represents the output image file. We need to convert this matrix to ppm image format at the end.

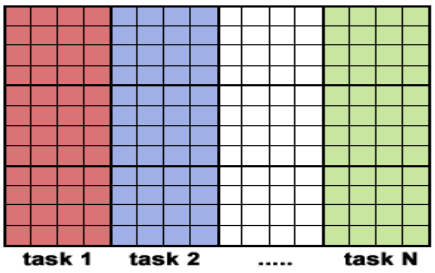
5. Covert matrix format of image into ppm format image. In our case the input and output files were by default in .ppm format.

6. Direction of the resultant gradient can be calculated as Angle = atan2(temp2, temp1) but this was not used in this project.

There are various masks available for edge detection, apart from the ones mentioned above and they can be used according to intensity, complexity of image pixels and application.

Parallelization using OpenMP and Pthreads:

In both the cases the image was divided into multiple stripes by height as follows:



Hence, if there are ‘ysize’ number of pixels in the width of the image then, every thread is processing (ysize/number_of_threads)*xsize number of pixels.

In serial execution, the single thread processes ysize*xsize number of threads.

Results:

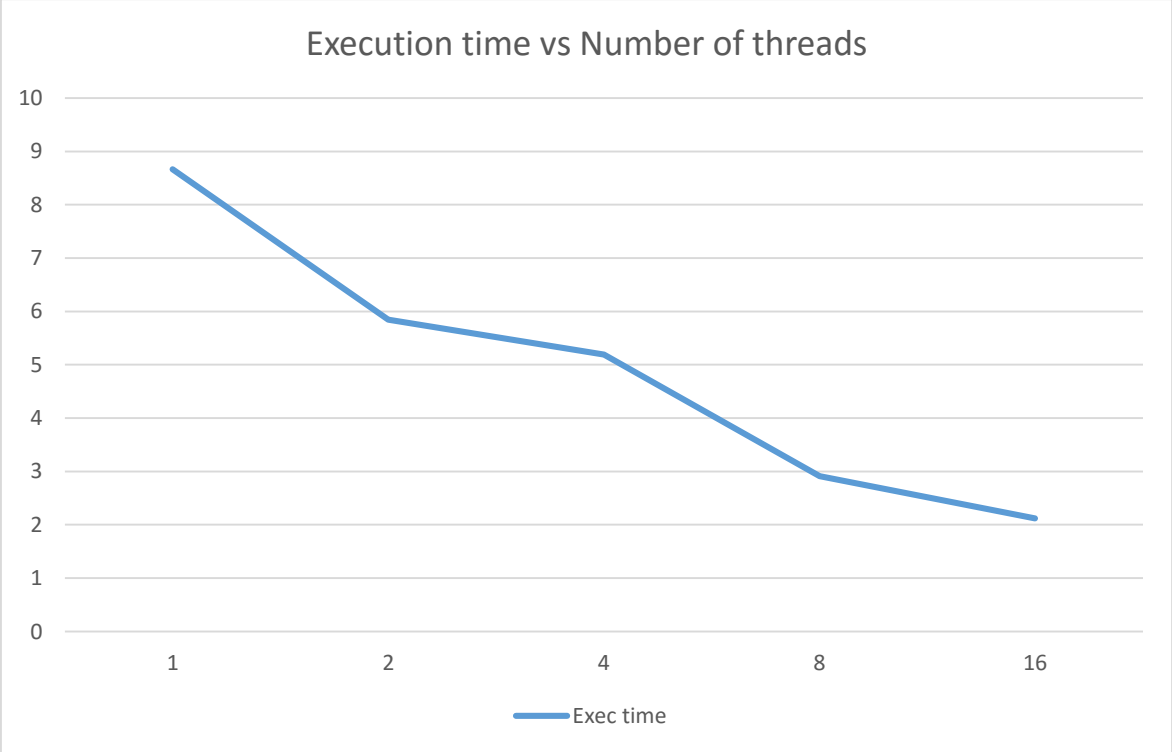
For a smaller image of the dimension: 5472 x 3648

Number of Threads	**OpenMP** (time in sec)	**Pthreads** (time in sec)
1	1.00343	1.00343
2	0.594289	1.296802
4	0.570645	1.268846
8	0.564268	1.250636
16	0.546827	1.242044

For a larger image of the dimension: 15500 x 7750

Number of Threads	**OpenMP** (time in sec)	**Pthreads** (time in sec)
1	8.66211	4.625901
2	5.84269	4.444238
4	5.18925	7.588105
8	2.91151	7.512653
16	2.12000	7.509531

Speedup graph for best case scenario from our experiments:



Large sized image and OpenMP implementation

Conclusion:

In above results we can still see that the execution time is decreasing with increase in number of threads. Hence we may find better correlation in results if only image processing loops are timed. Also, we can see that the speedup is seen more on large sized image, we can conclude that this is a weakly scalable use case. In parallel processing, the thread creation mechanism itself take some finite time hence we can see that the speedup is significant on larger problem size where the thread creation time is negligibly smaller. Also, in pthreads and OpenMP both cases, the parallelization is limited by number of available hardware threads.

Future Scope:

There are multiple masks to try using the same code and by varying the kernel. The same approach can be combined with image sharpening filters to enhance quality of image if required. In parallelization space, there is scope for launching 2D nested pragma inside pragma OpenMP implementation.

References:

https://en.wikipedia.org/wiki/Sobel_operator

https://en.wikipedia.org/wiki/Edge_detection

http://www.cs.utah.edu/~mhall/cs6963s10/

Appendix: Codes

Serial Code-

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "string.h"
#include<iostream>
#include<time.h>
#include<math.h>

using namespace std;

unsigned int *read_ppm( char *filename, int * xsize, int * ysize, int *maxval )
{
 if ( !filename || filename[0] == '\0')
 {
  fprintf(stderr, "read_ppm but no file name\n");
  return NULL;  // fail
 }

 FILE *fp;

 fprintf(stderr, "read_ppm( %s )\n", filename);
 fp = fopen( filename, "rb");
 if (!fp)
 {
   fprintf(stderr, "read_ppm()    ERROR  file '%s' cannot be opened for reading\n", filename);
   return NULL; // fail
 }

 char chars[1024];
 //int num = read(fd, chars, 1000);
 int num = fread(chars, sizeof(char), 1000, fp);

 if (chars[0] != 'P' || chars[1] != '6')
 {
   fprintf(stderr, "Texture::Texture()    ERROR  file '%s' does not start with \"P6\"  I am expecting
   a binary PPM file\n", filename);
   return NULL;
 }

 unsigned int width, height, maxvalue;
```

```
char *ptr = chars+3; // P 6 newline
if (*ptr == '#') // comment line!
{
   ptr = 1 + strstr(ptr, "\n");
}

num = sscanf(ptr, "%d\n%d\n%d",  &width, &height, &maxvalue);
fprintf(stderr, "read %d things   width %d  height %d  maxval %d\n", num, width, height, maxvalue);
*xsize = width;
*ysize = height;
*maxval = maxvalue;

unsigned int *pic = (unsigned int *)malloc( width * height * sizeof(unsigned int));
if (!pic)
{
  fprintf(stderr, "read_ppm()  unable to allocate %d x %d unsigned ints for the picture\n", width,
  height);
  return NULL; // fail but return
}

// allocate buffer to read the rest of the file into
int bufsize =  3 * width * height * sizeof(unsigned char);
if ((*maxval) > 255) bufsize *= 2;
unsigned char *buf = (unsigned char *)malloc( bufsize );
if (!buf)
{
  fprintf(stderr, "read_ppm()  unable to allocate %d bytes of read buffer\n", bufsize);
  return NULL; // fail but return
}

char duh[80];
char *line = chars;

// find the start of the pixel data.   no doubt stupid
sprintf(duh, "%d\0", *xsize);
line = strstr(line, duh);
line += strlen(duh) + 1;

sprintf(duh, "%d\0", *ysize);
line = strstr(line, duh);
line += strlen(duh) + 1;

sprintf(duh, "%d\0", *maxval);
```

```
  line = strstr(line, duh);


  fprintf(stderr, "%s found at offset %d\n", duh, line - chars);
  line += strlen(duh) + 1;

  long offset = line - chars;
  fseek(fp, offset, SEEK_SET); // move to the correct offset
  long numread = fread(buf, sizeof(char), bufsize, fp);
  fprintf(stderr, "Texture %s   read %ld of %ld bytes\n", filename, numread, bufsize);

  fclose(fp);

  int pixels = (*xsize) * (*ysize);
  for (int i=0; i<pixels; i++) pic[i] = (int) buf[3*i];  // red channel

  printf("Success in reading\n");
  return pic; // success
}

void write_ppm( char *filename, int xsize, int ysize, int maxval, int *pic)
{
  FILE *fp;
  int x,y;

  fp = fopen(filename, "w");
  if (!fp)
    {
     fprintf(stderr, "FAILED TO OPEN FILE '%s' for writing\n");
     exit(-1);
    }

  fprintf(fp, "P6\n");
  fprintf(fp,"%d %d\n%d\n", xsize, ysize, maxval);

  int numpix = xsize * ysize;
  for (int i=0; i<numpix; i++)
  {
    unsigned char uc = (unsigned char) pic[i];
    fprintf(fp, "%c%c%c", uc, uc, uc);
  }
  fclose(fp);

}
```

```
int main( int argc, char **argv )
{
        char *filename;
        filename = strdup( DEFAULT_FILENAME);

   if (argc == 3)
   { // filename
        filename = strdup( argv[1]);
   }
   if (argc == 2)
   {
    filename = strdup( argv[1]);
   }

fprintf(stderr, "file %s\n", filename);


int xsize, ysize, maxval;
unsigned int *pic = read_ppm( filename, &xsize, &ysize, &maxval );


int *result = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );
int *imagetmp1 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );
int *imagetmp2 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );

if (!result)
{
        fprintf(stderr, "sobel() unable to malloc %d bytes\n", 4*xsize * ysize * 3 * sizeof( int ));
        exit(-1); // fail
}

int r, c, magnitude, sum1, sum2;
clock_t begin, end;
double time_spent;
int *out = result;

for (int col=0; col<ysize; col++)
{
        for (int row=0; row<xsize; row++)
        {
                *out++ = 0;
        }
}
```

```
begin = clock();

for(r=1;r<ysize-1;r++)
{
        for(c=1;c<xsize-1;c++)
        {
                int offset = r*xsize + c;
                imagetmp1[r*xsize+c] = (pic[(r-1)*(xsize)+(c-1)])*(-1)  +  (pic[(r-1)*xsize+c])*(-2)
                + (pic[(r 1)*xsize+c+1])*(-1)  +  (pic[r*xsize+(c-1)])*(0)  +  (pic[r*xsize+c])*(0)
                + (pic[r*(xsize)+c+1])*(0) + (pic[(r+1)*(xsize)+(c-1)])*(1) + (pic[(r+1)*xsize+c])*(2)
                + (pic[(r+1)*xsize+c+1])*(1);
         }
 }

for(r=1;r<ysize-1;r++)
{
        for(c=1;c<xsize-1;c++)
        {
                int offset = r*xsize + c;
                imagetmp2[r*xsize+c] = pic[(r-1)*(xsize)+(c-1)]*(-1)  +  pic[(r-1)*xsize+c]*(0)  +
                pic[(r-1)*xsize+c+1]*(1)      +      pic[r*xsize+(c-1)]*(-2)      +      pic[r*xsize+c]*(0)      +
                pic[r*(xsize)+c+1]*(2)  +  pic[(r+1)*(xsize)+(c-1)]*(-1)  +  pic[(r+1)*xsize+  c]*(0)  +
                pic[(r+1)*xsize+c+1]*(1);
        }
  }

for(r=1;r<ysize-1;r++)
{
        for(c=1;c<xsize-1;c++)
        {
                result[r*xsize+c]=sqrt(imagetmp1[r*xsize+c]*imagetmp1[r*xsize+c]
                                + imagetmp2[r*xsize+c]*imagetmp2[r*xsize+c]);
                result[r*xsize+c] = ceil(result[r*xsize+c]);
         }
}

  end = clock();
  write_ppm( "/home1/03995/tg832943/project/result.ppm", xsize, ysize, 255, result);
  time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  cout<<"Time = "<<time_spent<<endl<<endl;
  fprintf(stderr, "done\n");

}
```

OpenMP Code-

```
/*
File: sobel_OMP.cpp
Purpose: Describes all the functions used to detect the horizontal and vertical images present in a .ppm
extension file by making use of the concepts of sobel filter
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "string.h"
#include<iostream>
#include<time.h>
#include<omp.h>
#include <math.h>

using namespace std;

unsigned int *read_ppm( char *filename, int * xsize, int * ysize, int *maxval )
{
 if ( !filename || filename[0] == '\0')
 {
        fprintf(stderr, "read_ppm but no file name\n");
        return NULL;  // fail
 }

 FILE *fp;

 printf("Reading file: %s\n", filename);
 fp = fopen( filename, "rb");
 if (!fp)
 {
        fprintf(stderr, "read_ppm()    ERROR  file '%s' cannot be opened for reading\n", filename);
        return NULL; // fail
 }

 //read the bytes of the input file and collect them in an array named: chars
 char chars[16384];
 int num = fread(chars, sizeof(char), 1000, fp);
```

```
if (chars[0] != 'P' || chars[1] != '6')
{
        fprintf(stderr, "Texture::Texture()    ERROR  file '%s' does not start with \"P6\"  I am expecting a
         binary PPM file\n", filename);
        return NULL;
}

unsigned int width, height, maxvalue;

//This will calculate and determine the width, height and the maximum value present in the input file
char *ptr = chars+3; // P 6 newline

if (*ptr == '#') // comment line!
{
        ptr = 1 + strstr(ptr, "\n");
 }

 num = sscanf(ptr, "%d\n%d\n%d",  &width, &height, &maxvalue);

 printf("Read %d things >> width: %d, height: %d, maxval: %d\n\n", num, width, height, maxvalue);
 *xsize = width;
 *ysize = height;
 *maxval = maxvalue;

 unsigned int *pic = (unsigned int *)malloc( width * height * sizeof(unsigned int));
 if (!pic)
 {
        fprintf(stderr, "read_ppm()  unable to allocate %d x %d unsigned ints for the picture\n", width,
        height);
        return NULL; // fail but return
  }

  // allocate buffer to read the rest of the file into
  int bufsize =  3 * width * height * sizeof(unsigned char);
  if ((*maxval) > 255) bufsize *= 2;
  unsigned char *buf = (unsigned char *)malloc( bufsize );
  if (!buf)
  {
        fprintf(stderr, "read_ppm()  unable to allocate %d bytes of read buffer\n", bufsize);
        return NULL; // fail but return
  }

  char duh[80];
  char *line = chars;
```

```
  //All the below sprintf's are adding the values present in the xsize, ysize, maxval pointers and storing
them into an array called duh, this also adds a null character at the end
  sprintf(duh, "%d", *xsize);
  sprintf(duh, "%c", '\0');
  line = strstr(line, duh);
  line += strlen(duh) + 1;

  sprintf(duh, "%d", *ysize);
  sprintf(duh, "%c", '\0');
  line = strstr(line, duh);
  line += strlen(duh) + 1;

  sprintf(duh, "%d", *maxval);
  sprintf(duh, "%c", '\0');
  line = strstr(line, duh);

  line += strlen(duh) + 1;

  long offset = line - chars;
  fseek(fp, offset, SEEK_SET); // move to the correct offset
  long numread = fread(buf, sizeof(char), bufsize, fp);
  printf("Texture: %s Read %ld of %d bytes\n\n", filename, numread, bufsize);
  fclose(fp);

  int pixels = (*xsize) * (*ysize);
  for (int i=0; i<pixels; i++)
    pic[i] = (int) buf[3*i];  // red channel

  return pic; // success
}

void write_ppm( char *filename, int xsize, int ysize, int maxval, int *pic)
{
  FILE *fp;
  int x,y;

  //open the file to write the values in it, the file in our case will be the final file containing the output
  fp = fopen(filename, "w");
  if (!fp)
  {
          fprintf(stderr, "FAILED TO OPEN FILE '%s' for writing\n", filename);
          exit(-1);
   }
```

```
 fprintf(fp, "P6\n");
 fprintf(fp,"%d %d\n%d\n", xsize, ysize, maxval);

 int numpix = xsize * ysize;
 for (int i=0; i<numpix; i++)
 {
        unsigned char uc = (unsigned char) pic[i];
        fprintf(fp, "%c%c%c", uc, uc, uc);
 }
 fclose(fp);
}

int main( int argc, char **argv )
{
 clock_t begin, end;
 double stime=omp_get_wtime();

 int nthreads = atoi(argv[1]);
 char *filename;

 filename = strdup( argv[2]);
 printf("Input Filename: %s\n", filename);

 printf("Number of Threads Working:%d\n", nthreads);

 int xsize, ysize, maxval;
 unsigned int *pic = read_ppm( filename, &xsize, &ysize, &maxval );

 int *result1 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );
 int *result2 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );
 int *result = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );

 if (!result)
 {
        fprintf(stderr, "sobel() unable to malloc %lu bytes\n", 4*xsize * ysize * 3 * sizeof( int ));
        exit(-1); // fail
 }

 int i, j, magnitude, sum1, sum2;

 double time_spent;
 int *out = result;
 int tid;
```

```
 for (int col=0; col<ysize; col++)
 {
        for (int row=0; row<xsize; row++)
        {
                *out++ = 0;
        }
}

#pragma omp parallel num_threads(nthreads) shared(pic,nthreads) private(tid,i,j)
{
   tid = omp_get_thread_num();

   for (i = (tid*(ysize)/nthreads)+1;  i <= (tid+1)*(ysize)/nthreads; i++)
   {
        for (j = 1; j < xsize -1; j++)
        {
                result1[i*xsize+j] = (pic[(i-1)*(xsize)+(j-1)])*(-1) + (pic[(i-1)*xsize+j])*(-2) +
                (pic[(i-1)*xsize+j+1])*(-1)    +    (pic[i*xsize+(j-1)])*(0)    +    (pic[i*xsize+j])*(0)    +
                (pic[i*(xsize)+j+1])*(0)  +  (pic[(i+1)*(xsize)+(j-1)])*(1)  +  (pic[(i+1)*xsize+  j])*(2)  +
                (pic[(i+1)*xsize+j+1])*(1);
        }
   }
 }

#pragma omp parallel num_threads(nthreads) shared(pic,nthreads) private(tid,i,j)
 {
   tid = omp_get_thread_num();

   for (i = (tid*(ysize)/nthreads)+1;  i <= (tid+1)*(ysize)/nthreads; i++)
  {
     for (j = 1; j < xsize -1; j++)
    {
        result2[i*xsize+j] = pic[(i-1)*(xsize)+(j-1)]*(-1) + pic[(i-1)*xsize+j]*(0) + pic[(i-1)*xsize+j+1]*(1) +
        pic[i*xsize+(j-1)]*(-2) + pic[i*xsize+j]*(0) + pic[i*(xsize)+j+1]*(2) + pic[(i+1)*(xsize)+(j-1)]*(-1) +
        pic[(i+1)*xsize+ j]*(0) + pic[(i+1)*xsize+j+1]*(1);
    }
   }
 }

#pragma omp parallel num_threads(nthreads) shared(nthreads) private(tid,i,j)
 {
   tid = omp_get_thread_num();
```

```
  for (i = (tid*(ysize)/nthreads)+1;  i <= (tid+1)*(ysize)/nthreads; i++)
 {
        for (j = 1; j < xsize -1; j++)
        {
              result[i*xsize+j] = sqrt(result1[i*xsize+j]*result1[i*xsize+j]
                                    + result2[i*xsize+j]*result2[i*xsize+j]);
        }
  }
 }

 double etime=omp_get_wtime();
 write_ppm((char *)"/home/kapeed/Documents/deepak/result.ppm", xsize, ysize, 255, result);

 double total=etime-stime;

 printf("Task Successfully Completed\n");
 cout<<"Total Time Taken="<<total<<endl;

}
```

Pthreads Code-

```
/*
File: sobelp.cpp
Purpose: Describes all the functions used to detect the horizontal and vertical images present in a .ppm
extension file by making use of the concepts of sobel filter
*/

#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdint.h>
#include <fcntl.h>
#include "string.h"
#include "unistd.h"

//This is the default case when no file name is given
#define DEFAULT_FILENAME "/home/kapeed/Documents/deepak/cali.ppm"

void *processing(void* rank);
```

```
unsigned int *read_ppm( char *filename, int * xsize, int * ysize, int *maxval )
{
 if ( !filename || filename[0] == '\0')
 {
  fprintf(stderr, "read_ppm but no file name\n");
  return NULL;  // fail
 }

 FILE *fp;

 printf("Reading file: %s\n", filename);
 fp = fopen( filename, "rb");
 if (!fp)
 {
    fprintf(stderr, "read_ppm()    ERROR  file '%s' cannot be opened for reading\n", filename);
    return NULL; // fail
 }

 char chars[100];

 //read the bytes of the input file and collect them in an array named: chars
 int num = fread(chars, sizeof(char), 100, fp);

 if (chars[0] != 'P' || chars[1] != '6')
 {
    fprintf(stderr, "Texture::Texture()    ERROR  file '%s' does not start with \"P6\"  I am expecting
    a binary PPM file\n", filename);
    return NULL;
 }

 unsigned int width, height, maxvalue;

 //This will calculate and determine the width, height and the maximum value present in the input file
 char *ptr = chars+3; // P 6 newline
 if (*ptr == '#') // comment line!
   ptr = 1 + strstr(ptr, "\n");

 num = sscanf(ptr, "%d\n%d\n%d",  &width, &height, &maxvalue);

 printf("Read %d things >> width: %d, height: %d, maxval: %d\n\n", num, width, height, maxvalue);
 *xsize = width;
 *ysize = height;
 *maxval = maxvalue;
```

```
unsigned int *pic = (unsigned int *)malloc( width * height * sizeof(unsigned int));
if (!pic)
{
        fprintf(stderr, "read_ppm()  unable to allocate %d x %d unsigned ints for the picture\n", width,
        height);
        return NULL; // fail but return
}

// allocate buffer to read the rest of the file into
int bufsize =  3 * width * height * sizeof(unsigned char);
if ((*maxval) > 255) bufsize *= 2;
unsigned char *buf = (unsigned char *)malloc( bufsize );
if (!buf)
{
        fprintf(stderr, "read_ppm()  unable to allocate %d bytes of read buffer\n", bufsize);
        return NULL; // fail but return
}

char duh[80];
char *line = chars;

//All the below sprintf's are adding the values present in the xsize, ysize, maxval pointers and storing
them into an array called duh, this also adds a null character at the end
sprintf(duh, "%d", *xsize);
sprintf(duh, "%c", '\0');
line = strstr(line, duh);
line += strlen(duh) + 1;

sprintf(duh, "%d", *ysize);
sprintf(duh, "%c", '\0');
line = strstr(line, duh);
line += strlen(duh) + 1;

sprintf(duh, "%d", *maxval);
sprintf(duh, "%c", '\0');
line = strstr(line, duh);

line += strlen(duh) + 1;

long offset = line - chars;
fseek(fp, offset, SEEK_SET); // move to the correct offset
long numread = fread(buf, sizeof(char), bufsize, fp);
printf("Texture: %s Read %ld of %d bytes\n\n", filename, numread, bufsize);
fclose(fp);
```

```
 int pixels = (*xsize) * (*ysize);
 for (int i=0; i<pixels; i++)
   pic[i] = (int) buf[3*i];  // red channel

 return pic; // success
}

void write_ppm( char *filename, int xsize, int ysize, int maxval, int *pic)
{
 FILE *fp;
 int x,y;

 //open the file to write the values in it, the file in our case will be the final file containing the output
 fp = fopen(filename, "w");
 if (!fp)
  {
    fprintf(stderr, "FAILED TO OPEN FILE '%s' for writing\n", filename);
    exit(-1);
  }

 fprintf(fp, "P6\n");
 fprintf(fp,"%d %d\n%d\n", xsize, ysize, maxval);

 int numpix = xsize * ysize;
 for (int i=0; i<numpix; i++)
 {
          unsigned char uc = (unsigned char) pic[i];
          fprintf(fp, "%c%c%c", uc, uc, uc);
 }
 fclose(fp);
}

int xsize, ysize, maxval;
unsigned int *pic;
int *result = NULL;
int *imagetmp1 = NULL;
int *imagetmp2 = NULL;
int thread_count;

int main( int argc, char **argv )
{
 pthread_t* thread_handles;
 char *filename;
```

```
 char path[] = "/home/kapeed/Documents/deepak/result.ppm";
 filename = strdup( DEFAULT_FILENAME);

 if (argc == 3)
 { // filename
        thread_count=atoi(argv[1]);
        filename = strdup( argv[2]);
 }

 if (argc == 2)
  thread_count = atoi(argv[1]);

 printf("Input Filename: %s\n\n", filename);

 //assigning memory to the variables
 pic = read_ppm((char *) filename, &xsize, &ysize, &maxval );
 result = (int*) malloc( xsize * ysize * 3 * sizeof( int) );
 imagetmp1 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );
 imagetmp2 = (int*) malloc( xsize * ysize * 3 * sizeof( int ) );

 if (!result)
 {
  fprintf(stderr, "sobel() unable to malloc %ld bytes\n", 4*xsize * ysize * 3 * sizeof( int ));
  exit(-1); // fail
 }

 int magnitude, sum1, sum2;
 clock_t begin, end;
 double time_spent;
 int *out = result;

 for (int col=0; col<ysize; col++)
 {
        for (int row=0; row<xsize; row++)
        {
                *out++ = 0;
        }
}

 thread_handles = (pthread_t*)malloc(thread_count*sizeof(pthread_t));

 //begin the clock
 begin = clock();
```

```
 printf("Number of threads working: %d\n", thread_count);

 for(long i=0;i<thread_count;i++)
  pthread_create(&thread_handles[i], NULL, processing, (void*)i);

 for(long i=0;i<thread_count;i++)
  pthread_join(thread_handles[i], NULL);

 //end the clock
 end = clock();
 write_ppm( (char *) path, xsize, ysize, 255, result);

 //calculate the execution time
 time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

 printf("Task Successfully Completed\n");

 printf("\nTotal Time Taken = %f\n",time_spent);
}

void *processing(void* rank)
{
 int r, c;
 int my_q,my_r;
 long first_r,last_r,temp;
 my_q = ysize/thread_count;
 my_r = ysize%thread_count;

 //This is the method by which we will figure out till what value should each thread use the data passed into it.
 if((long)rank< my_r)
  {
   temp=my_q + 1;
   first_r=(long)rank * temp;
  }
 else
  {
   temp=my_q;
   first_r=(long)rank*temp + my_r;
  }
 last_r = first_r + temp;

 /*In our case we are dividing the according to its height, i.e., if the height is of 100 bytes and the number of threads working on it are 4 then each thread will work on 100/4 = 25 bytes*/
```

```
 //The below loops are for applying the horizontal mask
 for(r=first_r+1;r<=last_r-1;r++)
 {
        for(c=1;c<xsize-1;c++)
        {
                int offset = r*xsize + c;
                imagetmp1[r*xsize+c] = (pic[(r-1)*(xsize)+(c-1)])*(-1) + (pic[(r-1)*xsize+c])*(-2)
                + (pic[(r-1)*xsize+c+1])*(-1) +(pic[r*xsize+(c-1)])*(0) + (pic[r*xsize+c])*(0)
                + (pic[r*(xsize)+c+1])*(0) + (pic[(r+1)*(xsize)+(c-1)])*(1) + (pic[(r+1)*xsize+ c])*(2)
                + (pic[(r+1)*xsize+c+1])*(1);
  }
}

 //The below loops are for applying the vertical mask
 for(r=first_r+1;r<=last_r-1;r++)
 {
        for(c=1;c<xsize-1;c++)
        {
                int offset = r*xsize + c;
                imagetmp2[r*xsize+c] = pic[(r-1)*(xsize)+(c-1)]*(-1) + pic[(r-1)*xsize+c]*(0)
                + pic[(r-1)*xsize+c+1]*(1) + pic[r*xsize+(c-1)]*(-2) + pic[r*xsize+c]*(0)
                + pic[r*(xsize)+c+1]*(2) + pic[(r+1)*(xsize)+(c-1)]*(-1) + pic[(r+1)*xsize+ c]*(0)
                + pic[(r+1)*xsize+c+1]*(1);
  }
}

 //The below loop will find the resultant image after combining the values obtained from applying the
horizontal and vertical loops
 for(r=first_r;r<last_r;r++)
{
  for(c=1;c<xsize-1;c++)
  {
        result[r*xsize+c] = sqrt(imagetmp1[r*xsize+c]*imagetmp1[r*xsize+c]
                        + imagetmp2[r*xsize+c]*imagetmp2[r*xsize+c]);
        result[r*xsize+c] = ceil(result[r*xsize+c]);
  }
 }
}
```