# LAB 1: Pthreads and C++ Threads

Design due **01/31/17 (Tuesday) - 11:59 pm**

Implementation due **02/07/17 (Tuesday) - 11:59 pm**

## PART A ( Pthreads using C) :

Implement a version of Dining Philosophers problem using pthreads. Lets say there are N philosophers. They sat down at a circular table for a 3 course meal. There will be N forks on the table and each philosopher needs two forks to eat a course of the meal. (Hope you see the problem here!). That is only some philosophers can eat at a particular instance. Philosophers will eat one course at a time i.e. whoever grabs two forks eat the course and put both forks back on table for their neighbors to pick up. Each philosopher can pick the fork in front of them or the one on their right.

The algorithm must follow these rules:

1. Read command line arguments to get number of threads.
2. Create as many philosophers (using C structs ) as number of threads and initialize them.
3. Each fork is a mutex, so there will be an array of pthread_mutex.
4. Create N threads each representing a unique philosopher.
5. Philosopher will try grabbing the fork in front of them first. Again, remember each fork is an element in mutex array.
6. Once they have that one fork, they will try for the one on the right. If the one on the right is unavailable, then drop the one in hand and repeat. (They can't eat with just one, so let it go!)
7. Once a philosopher has both forks, use sleep(1) function to represent eating of a particular course. After that drop both the forks.
8. Now, its time for next course, so repeat from step 5 till all 3 course in the meal are done. Do a pthread_exit at this point.

## PART B ( C++ Threads ):

Implement a Sobel filter edge detection algorithm using C++ threads. The algorithm takes a grayscale image as input and outputs an image with edge outlines.

You will be provided with a skeleton program to get a head start with the solution. The solution must work as follows:

1. Read command line arguments to fetch names of the input and output image (.pgm images).
2. Read the image file into a 2-D array.
3. Generate the two 3x3 Sobel masks, one for each dimension.
4. Implement the Sobel filter algorithm that process the entire image (You must be able to expose parallelism here!).
    a. Use dynamic scheduling by distributing parts of the image to different threads [More about this in discussion]
5. Generate the output image.

A sample input and its corresponding output is presented below:

The PGM images are 2-D matrix with each element representing a pixel. Each pixel has a value from 0 to 255.
In the file, when opened as text, first line represents the format 'P2'. Second line defines the size of the image in pixels(ex: 250x360). Third line represent shade range (255 in our test cases).

Sobel filter algorithm:
The algorithm applies the 3x3 mask to the neighboring pixels such that the pixels with values closer in magnitude are mapped to values around 0 that represents black and the pixels with significant differences in magnitude are mapped to values around 255 i.e. white.

3x3 Mask for X-direction: [ -1 0 1; -2 0 2; -1 0 1 ]
3x3 Mask for Y-direction: [ 1 2 1; 0 0 0; -1 -2 -1]
For each pixel(p) in the image (except image boundaries):
Multiply and Add the surrounding pixels of p(X,Y) i.e. center being p itself ( total of 9, same as the mask ) with corresponding Mask values(I,J) for each dimension:
-1 <= I <= 1; -1 <= J <= 1
sumX += Image(X+I, Y+J)*MaskX(I,J)
sumY += Image(X+I, Y+J)*MaskY(I,J)
Sum = |sumX| + |sumY| (NOTE: |num| = abs(num) in C++ library function)
If Sum < 0 then Sum = 0. If Sum > 255 then Sum = 255.
Write Sum into the output image

**Serial Algorithm:**

```
/* 3x3 Sobel mask for X Dimension. */
maskX[0][0] = -1; maskX[0][1] = 0; maskX[0][2] = 1;
maskX[1][0] = -2; maskX[1][1] = 0; maskX[1][2] = 2;
maskX[2][0] = -1; maskX[2][1] = 0; maskX[2][2] = 1;

/* 3x3 Sobel mask for Y Dimension. */
maskY[0][0] =  1; maskY[0][1] =  2; maskY[0][2] =  1;
maskY[1][0] =  0; maskY[1][1] =  0; maskY[1][2] =  0;
maskY[2][0] = -1; maskY[2][1] = -2; maskY[2][2] = -1;

for( int x = 0; x < height; ++x ){
   for( int y = 0; y < width;  ++y ){
      sumx = 0;
      sumy = 0;
      /* For handling image boundaries */
      if( x == 0 || x == (height-1) || y == 0 || y == (width-1))
         sum = 0;
      else{
         /* Gradient calculation in X Dimension */
```

```
        for( int i = -1; i <= 1; i++ )  {
           for( int j = -1; j <= 1; j++ ){
              sumx += (inputImage[x+i][y+j] * maskX[i+1][j+1]);
           }
        }
        /* Gradient calculation in Y Dimension */
        for(i=-1; i<=1; i++)  {
           for(j=-1; j<=1; j++){
              sumy += (inputImage[x+i][y+j] * maskY[i+1][j+1]);
           }
        }
         /* Gradient magnitude */
         sum = (abs(sumx) + abs(sumy));
   }
   outputImage[x][y] = (0 <= sum <= 255);
   }
}
```

**NOTE: Skeletons and the dataset for part A will be updated soon. Work on the Designs and Implementing from scratch till then.**

**Point Breakdown:**

Part A: Design -> 5pts, Implementation -> 45pts

Part B: Design -> 5pts, Implementation -> 45pts

**For Design section**, write a pseudo-algorithm of your proposed implementation. This design must be detailed enough to show how you are going to expose parallelism and not verbose like the descriptions that are provided above.

**NOTE:** C++ Threads and Pthreads and their usage will be discussed in the discussions.

**Compiling and Running:**

To compile you must use gcc/5.2.0 available from the module system.
To compile use the `-std=c++11 -pthread` flags like so:

```
$ g++ -std=c++11 -pthread lab1.cpp -o LAB1
```

It is highly recommended to use the compilation flag `-Wall` to detect errors early, like so:

```
$ g++ -Wall -std=c++11 -pthread lab1.cpp -o LAB1

For Pthread part, switch g++ to gcc and a CPP file to C file.
```

Once compiled (for example to the program LAB1) you must run it on the command line and pass it two parameters:

```
$ ./LAB1 <Input_Image_Path> <Output_Image_Path>
```

**Submit via EEE (**dropbox "Lab1") **in 3 parts:** *YOU **MUST** USE the file names below*

1. Part A and B design, e.g. high-level pseudo code algorithm, as files DesignA.txt and DesignB.txt
2. Implementation of your C/C++ program: Implementation.c/cpp

**USEFUL LINKS:**

https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/

(good examples)

http://thispointer.com/c-11-multithreading-part-1-three-different-ways-to-create-threads/

(complete tutorial, in particular parts 1 - 5)