LAB 1: C++ Threads

Implementation due **04/26/21 (Mon) - 11:55 pm**

PART A: Using Atomic

Implement a version of Dining Philosophers problem using C++ threads. Let's 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 structs) as number of threads and initialize them.
3. Each fork is a C++ Atomic, so each philosopher will have one.
4. Create N threads each representing a unique philosopher.
5. Philosopher will try grabbing the fork in front of them first (using Atomic Compare and Exchange). Again, remember each fork is of Atomic type.
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 one second long *sleep* function to represent eating of a particular course. After that drop both the forks.
8. Record the time when the meal was finished. (Look at the code skeleton for more info)
9. Now, it's time for next course, so repeat from step 5 till all 3-courses in the meal are done.
10. After joining all threads, print all 3 meal times for each philosopher.

PART B: Workload Balancing

Implement a parallel edge detection (using Roberts cross operator) using C++ threads. The algorithm takes a grayscale image as input and outputs an image with edge outlines.

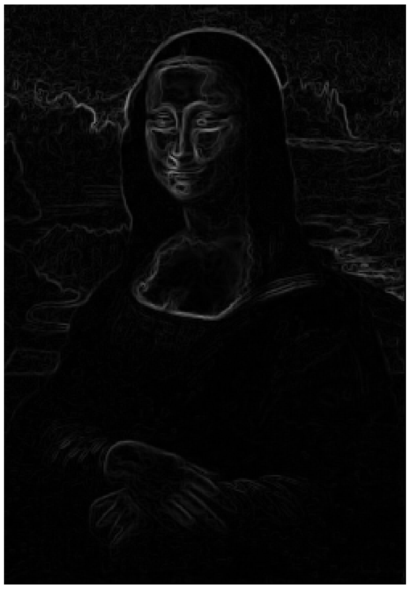
You will be provided with a skeleton program to get a head start with the solution. The solution must work as follows:

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 2x2 Roberts operators, one for each dimension, that convolve the image.
4. Implement the edge detection 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 (left) and its corresponding output (right) 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).

Edge Detection Algorithm (using Roberts cross operator):
The algorithm applies the 2x2 operator/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.

```
2x2 Mask for X-direction: [ +1, 0; 0, -1 ]
2x2 Mask for Y-direction: [  0, +1; -1, 0 ]
For each pixel(p) in the image (except image boundaries):
        Multiply the surrounding pixels of p(X,Y) with corresponding Mask values(I,J)
          for each dimension and add them together to compute the gradients (one for each dimension):
                grad_X += Image(X+I, Y+J) * maskX(I,J)
                grad_Y += Image(X+I, Y+J) * maskY(I,J)
        grad = sqrt( (grad_X * grad_X) + (grad_Y * grad_Y)  )   (NOTE: sqrt() in C++ library function)
        If grad < 0 then grad = 0. If grad > 255 then grad = 255.
        Write grad into the output image
```

Serial Algorithm Code:

```
/* 2x2 Roberts cross mask for X Dimension. */
maskX[0][0] = +1; maskX[0][1] = 0;
maskX[1][0] = 0; maskX[1][1] = -1;

/* 2x2 Roberts cross mask for Y Dimension. */
maskY[0][0] = 0; maskY[0][1] = +1;
```

```
maskY[1][0] = -1; maskY[1][1] =   0;


for( int x = 0; x < height; ++x ){
    for( int y = 0; y < width;  ++y ){
        grad_x = 0;
        grad_y = 0;
        /* For handling image boundaries */
        if( x == (height-1) || y == (width-1))
            grad = 0;
        else{
            /* Gradient calculation in X Dimension */
            for( int i = 0; i <= 1; i++ )  {
                for( int j = 0; j <= 1; j++ ){
                    grad_x += (inputImage[x+i][y+j] * maskX[i][j]);
                }
            }
            /* Gradient calculation in Y Dimension */
            for(i= 0; i<=1; i++)  {
                for(j= 0; j<=1; j++){
                    grad_y += (inputImage[x+i][y+j] * maskY[i][j]);
                }
            }
             /* Gradient magnitude */
             grad = (int) sqrt( (grad_x * grad_x) + (grad_y * grad_y) );
    }
    outputImage[x][y] = grad <= 255 ? grad : 255;
    }
}
```

Guidelines to Parallelize the Code:

- Two of the command line parameters are: Number of Threads and Number of Chunks.
- Number of Threads signify the total threads to be created by the program.
- Number of Chunks signify the number of equal size partitions of the input images.
  - For example: if Input image has 100 rows (i.e. height), and Number of Chunks is 4, then each chunk will have 25 rows each.
  - If for the same image, Number of Chunks is 6, then there will 5 partitions of 17 rows each (rounding up 100 divided by 6) and one last partition for the remaining rows, i.e., 15 rows.
- Scheduling of chunks on the threads:
  - We will do dynamic allocation of chunks to the threads.
  - Since this is dynamically decided (at execution), each thread should track if there are any chunk that hasn't been processed and fetch one. (HINT: Mutex and a global variable)
  - Result of dynamic scheduling is that on each execution different threads will get different chunks. This happens because creation of threads and processing of chunks can take different time for different threads.
  - For example: if there are 6 chunks and 4 threads.
    - First Execution: Thread 1 processes chunk 1, 5; Thread 2 processes chunk 2; Thread 3 processes chunk 3, 6; Thread 4 processes chunk 4.
    - Second Execution: Thread 1 processes chunk 2; Thread 2 processes chunk 3, 6; Thread 3 processes chunk 1, 5; Thread 4 processes chunk 4.
- Please check your code for Data Races. (More about it below and in the discussion section)

NOTE:

- Skeletons and the dataset for the lab are available on the Canvas.
- Usage of C++ Threads, Atomic, Compare and Exchange (CAS) will be shown in the discussion. Till then work on implementing the serial version of the problems for Part A and Part B. The code skeletons and the algorithms/code described above must be enough to get the serial versions running.

Point Breakdown:

Part A: Implementation -> 50 pts

Part B: Implementation -> 50 pts

Compiling and Running Part A:

To compile you must use gcc/8.2.0 available from the module system.

```
$ module load gcc/8.2.0
```

To compile use the `–std=c++11 –pthread` flags like so:

```
$ g++ -std=c++11 –pthread ImplementationA.cpp –o ImplementationA
```

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

```
$ g++ -Wall -std=c++11 –pthread ImplementationA.cpp –o
ImplementationA
```

Once compiled you must run it on the command line and pass it four parameters:

```
$ ./ImplementationA <Philosophers #>
```

Compiling and Running Part B:

To compile you must use gcc/8.2.0 available from the module system.

```
$ module load gcc/8.2.0
```

To compile use the `–std=c++11 –pthread` flags like so:

```
$ g++ -std=c++11 –pthread ImplementationB.cpp –o ImplementationB
```

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

```
$ g++ -Wall -std=c++11 –pthread ImplementationB.cpp –o
ImplementationB
```

To **check for data races** in your code, use the following flags (more about them in the discussion):

```
$ g++ -Wall -std=c++11 –pthread ImplementationB.cpp –o
ImplementationB        -fsanitize=thread -fPIE -pie
```

If there are any possible data races detected in the code, they will be shown as warnings on execution.

Once compiled you must run it on the command line and pass it four parameters:

```
$ ./ImplementationB <Input_Image_Path> <Output_Image_Path> <# of Threads> <# of Chunks>
```

Submit the following TWO files only via Canvas (Assignment "Lab1"**):** *YOU **MUST** USE the file names below*

1. Implementation of your C program for Part A: ImplementationA.cpp
2. Implementation of your C++ program for Part B: ImplementationB.cpp

USEFUL LINKS:

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

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

C++ Thread Sanitizer for detecting Data Races:
https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

C++ Atomic: https://en.cppreference.com/w/cpp/atomic/atomic

C++ Atomic Compare and Exchange (CAS):
https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange

https://tonywearme.wordpress.com/2014/08/15/understand-stdatomiccompare_exchange_weak-in-c11/