EE109 Lab – Need for Speed

1 Introduction
In this lab you will parallelize two pieces of code. The first is an image processing program to blur (smooth) or sharpen an image. We will guide you through this process. Then you will apply what you learned in lecture and in part 1 to parallelize a random simulation program to calculate the odds of winning the dice-game of craps. This lab should be performed individually on Vocaruem.

2 What you will learn
This lab exercise will help you apply your understanding of parallelization techniques and issues that affect correctness and performance.

1. Apply OpenMP parallel constructs in an attempt to parallelize a sequential software program to achieve the best possible parallel speedups.
2. Apply the private and reduction clauses correctly with appropriate variables
3. Generate speedup curves for a parallel program
4. Understand the issues with thread-safe functions.

3 Background Information and Notes
Part 1: Image Manipulation: In this lab we will give you code that performs blurring on an image. We have provided all the working code for you. But here is some background information for your own interest.

Tommy Trojan filtered with a blur filter

We can produce these images by taking the weighted average of nearby pixels. By defining the neighborhood and the weight of each neighbor, we arrive at a small 2D “kernel” that we will apply to each pixel in the image.
To arrive at these images we simply iterate over every pixel in the image, and for each pixel iterate over the NxN neighboring pixels and apply the weighted average based on the kernel weights. This process is called convolution. Realize that the process just described above yields many nested for loops as shown below:

```c
unsigned char iimage[SIZE][SIZE][RGB];
unsigned char oimage[SIZE][SIZE][RGB];
double kernel[N][N]
```

```
// Apply the NxN kernel to the SIZE x SIZE image
for(int y=N/2; y<SIZE+N/2; y++) // row of image
    for(int x=N/2; x<SIZE+N/2; x++) // column of image
        for(int k=0; k<RGB; k++) // Red/Green/Blue plane
            for(int i=0; i<N; i++) // row of kernel
                for(int j=0; j<N; j++) // col of kernel
                    oimage[y][x][k] +=
                        iimage[y+i-N/2][x+j-N/2][k]*kernel[i][j];
```

This is a perfect piece of code to parallelize as each output pixel is computed completely independent of each other (i.e. each output image pixel is depending on the input image which does not change and thus we need not worry about synchronization.)

Note: To apply the kernel (i.e. weighted average of neighbors) to the top and bottom rows and left-most and right-most columns would cause us to access pixels...
that don’t exist (i.e. out of bounds). To ensure this doesn’t happen we make a bigger 2D array to allow for extra padding rows and columns (like a frame around the actual image). We initialize those padding rows and columns to copies of the outer rows/columns with 0’s in the corners and then copy in the actual input image to be centered.

In part 1 of this lab you will perform simple parallelization of the large, nested loops that perform the convolution.

**Part 2 - Craps Game Simulation:** A simplified version of the craps dice game has the following rules:

1. The player rolls 2 dice.
2. If the sum of the dice is 7 or 11 the player wins. The game is over after the first roll.
3. If the sum of the dice is 2, 3, or 12 the player loses. The game is over after the first roll.
4. If the sum is any other number (besides 2, 3, 7, 11, or 12) then that value is known as the point number and play continues.
5. The player rolls the dice until...
   a. The sum of the dice is 7 in which case the player loses.
   b. The sum of the dice is the same as the point value in which case the player wins.

**We want to answer two questions:**

1. How many rolls does the average game last?
2. What are the odds of winning?
We can answer these analytically by applying probability and deriving and solving expressions. Or we can find it by simply simulating the game MANY times and see what the average results are. This is known as Monte-Carlo simulation. Thus we have provided working code to simulate MANY craps games and print the results. Your job will be to parallelize this code to achieve near linear speedup (i.e. 2x faster with 2 threads, and 4x faster with 4 threads).

**rand() and Thread safe functions:** To perform this simulation we will need to generate random numbers. Typically this would be done with `rand()` which is a function supplied by the C library. However, some functions (even some in the C library) maintain a hidden static/global variable [i.e. one copy of a variable that they modify each time the function is called]. The problem with this is when we introduce parallelism and multiple threads, we can unwittingly introduce bugs or flaws into the program due to the hidden race conditions that occur when those unsafe functions are executed in parallel. `rand()` is an example of such a function. It maintains a static ‘seed’ variable (that is not accessible to you) and modifies it on each call. Thus, when multithreading is introduced, we may not get a random (or pseudo-random) sequence because threads calling at the same time might see the same seed.

Luckily different compilers provide some different `rand()` functions that are thread safe. When functions are designed to be safe given multithreading, we often call them thread-safe or re-entrant functions. The C-library has a re-entrant version of `rand()` called:

```c
// reentrant rand
int rand_r(unsigned int* seed);
```

- You will declare a private seed variable per thread and pass a pointer to that variable. Instead of updating the hidden global variable like `rand()` does, `rand_r()` will use the value of seed and then update it.
- The return value of `rand_r` will be the random number you should use.

You will need to modify the craps code to use the `rand_r()` function as described next.

Declare `unsigned int rseed` OUTSIDE of the parallelized for loop and don’t initialize it.-Then, ensure `rseed` is a thread-private variable in your `#pragma omp parallel for`. This will replicate `rseed` and essentially provide garbage bits in each thread’s version but this works to our advantage since those values should be random and act as a random seed.

```c
#include <cstdlib>
unsigned int rseed;  // leave it uninitialized
// To get a random number call rand_r(&rseed)
  r = 1 + rand_r(&rseed)%6;
```

**Important:** `rseed` must be a separate variable per-thread!
**Recording time**: To have your program record the execution time of a code sequence we can use C library functions. Unfortunately, different OS'es versions of the C Library use slightly different constructs. Use the appropriate code below as a template.

This code will return the number of microseconds elapsed from some fixed timepoint (e.g. since Jan. 1, 2000, 12:00:00 a.m.)

**Unix/Linux:**
```c
#include <time.h>
#include <sys/time.h>

double usec()
{
  struct timeval t0;
  gettimeofday(&t0, NULL);
  return ((double)(t0.tv_sec*1e6 + t0.tv_usec));
}
```

Using this function, we can get a timestamp (in microseconds) at the time the function is called. By getting a timestamp before we start our work and once it is done, then we can take the difference to determine how long the work took. See the example below.

```c
double t0, t1;
// Start timer
  t0 = usec();

/* Work that you want to time */
  t1 = usec();
  cout << "Duration: " << t1-t0 << " us" << endl;
```

**Parallel Performance and I/O**: While a processor may have multiple cores, the I/O system (keyboard input, File I/O, console output) often only allows sequential access and hurts parallel performance, not to mention any time spent waiting for the user to provide input. Thus, you should not add `cout` statements or other I/O in the area of code that that you are trying to optimize.

**Compiling OpenMP**: When using gcc/g++ with OpenMP, you must enable support via the option: `-fopenmp` to the command line.

```bash
$ g++ -fopenmp filter.cpp -o filter
$ g++ -fopenmp craps.cpp -o craps
```
4 Procedure

Setup
1. Login to Vocareum and start the **Lab 10 – Parallelism** lab and start the OMP part of the lab.
2. In your work area should be two files:
   a. filter.cpp
   b. craps.cpp
3. To edit these files you can use the built-in Vocareum editor or copy and paste back and forth to your favorite editor on your own PC.

Part 1:
4. Edit the **filter.cpp** file. We will only concern ourselves with the `convolve()` function since that is where nearly all the time is spent. In the `convolve()` function first parallelize the convolution loops by adding the line:

   ```
   #pragma omp parallel for
   ```

   Just above the 5 nested loop that starts off:

   ```
   for(int y=N/2;y<SIZE+N/2;y++)
   ```

5. With this simple addition we can likely get good parallel performance. Save the file and then in the terminal (bottom half) of the Vocareum workspace, compile your code by running the command:

   ```
   g++ -fopenmp filter.cpp bmplib.cpp -o filter
   ```

6. Then run the program with 1 thread so we can get a base amount of time. We’ll run it first with a 5x5 kernel

   ```
   ./filter Tommy.bmp 5 out.bmp 1
   ```

   If you refresh the work area on the left side of the browser and click on out.bmp you should see a blurred image appear. However, let us focus on time. **Record the time it took to run with 1 thread (you will enter this in your report).**

7. Now run with 2 threads and then 4 threads and record the times **(you will enter these times in the report)**. Is the speedup fairly linear? Run the program several times for each value of threads (1, 2, 4) and pick a representative time (i.e. sometime other background tasks affect a single program's execution time so run it a few time to get a sense for what the representative answer is.)

8. Repeat your runs (of 1, 2, and 4 threads for a 3x3 kernel by running the program with an argument of 3 in place of the 5 above and then repeat for 1, 2, and 4 threads for a 7x7 kernel). After this you should have 9 recorded times (1, 2, and 4 threads for 3x3, 5x5, and 7x7 kernels). Create a table you’ll submit in your
report that shows the time for each combination. Do you get near linear speedup for all cases or just some? Why?

<table>
<thead>
<tr>
<th># of Threads</th>
<th>Time for 3x3 kernel</th>
<th>Time for 5x5 kernel</th>
<th>Time for 7x7 kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9. We can try to improve a bit more. Add a #pragma omp parallel for statement to the last set of for loops in the convolve() that clamps the output if it is outside the range of 0 to 255. Recompile your code and rerun with 1, 2, and 4 threads for a 7x7 kernel. Do you see much of a difference? Why?

**Part 2:**

10. Edit the craps.cpp file. This part is open-ended and up to you to find the correct parallel implementation.

11. Compile and run the program to get a feel for how it works. The command below will compile and run 100,000 simulations of the game with 1 thread:

   ```
g++ -fopenmp craps.cpp -o craps
./craps 1000000 1
```

12. Remove all the calls to rand() and convert them to rand_r() using the rseed variable. See the background notes earlier in this document for how to do this. To ensure it is working you should get nearly the same results for 2 or 4 threads as with 1 thread.

13. Your job at this point is to parallelize the for loop that runs ‘totalgames’ times. Do this by adding an appropriate:

   ```
   #pragma omp parallel for
   ```

   However, you must think about what variables should be private and what variables you want to perform reductions on. **YOU MAY NOT CHANGE ANY OF THE DECLARATION OR MOVE THE CODE AROUND.** Just look at the code and consider what needs to be **private** and what should be **reduced**. Doing this correctly should yield near-linear speedup for 2 (for sure) and 4 threads (likely for larger number of games simulated).
14. Keep testing your code by compiling and running it with 1, 2, or 4 threads until you get it working (results close to the same as with 1 threads is a good indicator it is working) and yielding linear speedup. Record your speedup for various threads and number of total games to simulate.

<table>
<thead>
<tr>
<th># of Threads</th>
<th>1,000,000 Games</th>
<th>500,000 Games</th>
<th>100,000 Games</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Lab Report
Include the following items in your lab report:
1. Submit your final versions of the .cpp files.
2. Update and submit the report.txt file that contains the results of your experiments and some questions.