Faculty of Arts & Sciences
Department of Computer Science
CMPS 270 – Software Construction
Fall 2023 – Assignment 3

Counting Ones

In this assignment you will implement a program that counts and prints the total number of **ones** in an array of integers. The integers in the array are randomly generated with values between 0 and 5. Here's a sequential implementation of the program that you can use to evaluate your program's correctness:

```
int *array;
int length;
int count;

int count1s ()
{
   int i;
   count = 0;
   for (i=0; i<length; i++)
    {
       if (array[i] == 1)
       {
          count++;
       }
    }
    return count;
}
```

Your task is to experiment with different versions of a parallel implementation of the above program. You will follow a shared memory model using Posix threads. Your first task is to write code that generates an array of random integers with values between 0 and 5 (inclusive). You are required to submit a report along with the code. In your report, you should experiment with arrays of different sizes, namely 1,000, 1,000,000, and 1,000,000,000 integers. You should also experiment with a different number of threads, namely 1, 2, 4, and 32. Your report should include tables/plots showing how the runtime of the program differs as a function of the size of the array/number of threads. Start your report with a summary of your findings. The runtime of the program should not include the time it takes to generate the array.

You may follow this tutorial to time the relevant section of your C program:
https://www.geeksforgeeks.org/how-to-measure-time-taken-by-a-program-in-c/

Race Condition (call your program count_race.c)

Start by implementing a parallel version that divides the array across the available threads. Your program should accept the number of threads as an argument. Figure 1 illustrates an example of how the array might be distributed across threads.

Note that a simple parallel implementation includes a single *count* variable that gets updated by the different threads, and is thus an incorrect implementation. Run this version of the program 100 times and report the number of times the program returns the right answer.

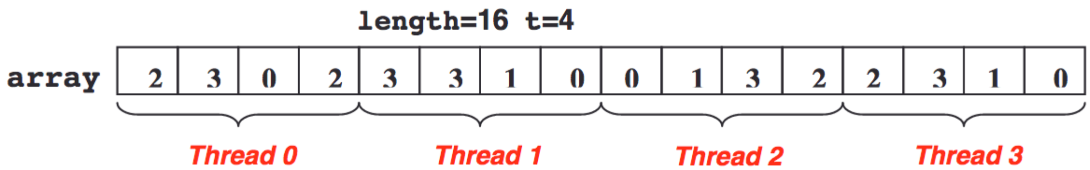

Figure 1: An example of an array being distributed across threads.

Mutex (call your program count_mutex.c)

One of the ways to deal with race conditions is by using a locking mechanism that allows *mutual exclusion*, such as a *mutex*. Write a program that fixes the race condition by using a mutex to guard the critical section in the code. Perform the same experiments to ensure your program's correctness. In addition to the experiments and the runtime graphs, include your analysis and insight of the observed results.

Private Counts (call your program count_private.c)

Another way to deal with race conditions is by using private variables that are not shared across threads, and syncing the results after the threads are done with their local workload. Implement a third version of this program that introduces an array of private counts that will be accessed privately by the corresponding threads. This version should not use a mutex. Repeat the same experiments and include your analysis. Do you notice any performance improvement? Why or why not?

Padding Caches (call your program count_cache.c)

A cache is a special memory that is much faster to access than the computer's main memory or storage. The cache is made up of multiple levels, some of which (e.g. L1) reside on the CPU chip itself. The computer's operating system deploys many tactics to make use of this special memory to optimize programs' performance. As we should have established by now, writing parallel programs and successfully optimizing performance is not an easy task. One of the things a programmer should be attentive to is the memory usage and bottlenecks. When a core updates a variable residing on its corresponding cache, the operating system proceeds to update the same variable on the rest of the caches corresponding to the different cores to ensure coherency. In a phenomenon called false sharing, variables that are only updated by one core may reside on the cache lines of other cores as well. This leads to extra work being done when one core updates the value of that variable. To avoid this extra work, we can make the variables occupy the entirety of the core's cache line. To do that, determine the size of the L1 cache on your computer, and create a struct that includes an integer which will be used as each thread's private count, in addition to a dummy *char* array with a size that fills the remaining bytes on the cache line. Update the private counts array to use instances of this struct. Repeat the previous experiments and include the results and analysis in your report.

In addition to your GitHub submission, add your programs to one folder: username.a03.zip (or .rar) and submit it to moodle.