Kyle MacMillan
Programming Assignment 1
October 1, 2018

Description of Program:
Problem 1 revolved around testing the outputs of a given circuit to see which outputs produced a 1. Of the 65,536 possible inputs only 9 produced an output of 1:

```
0110111110011001
1110111110011001
1110111111011001
0110111111011001
1110111110111001
0110111110111001
1010111110011001
1010111111011001
1010111110111001
```

There were no tricks to get it to work, simply ensure your unsigned integers don't roll over and it performs out of the box.

I ran it 200 times to get an average runtime. For dynamic it took:

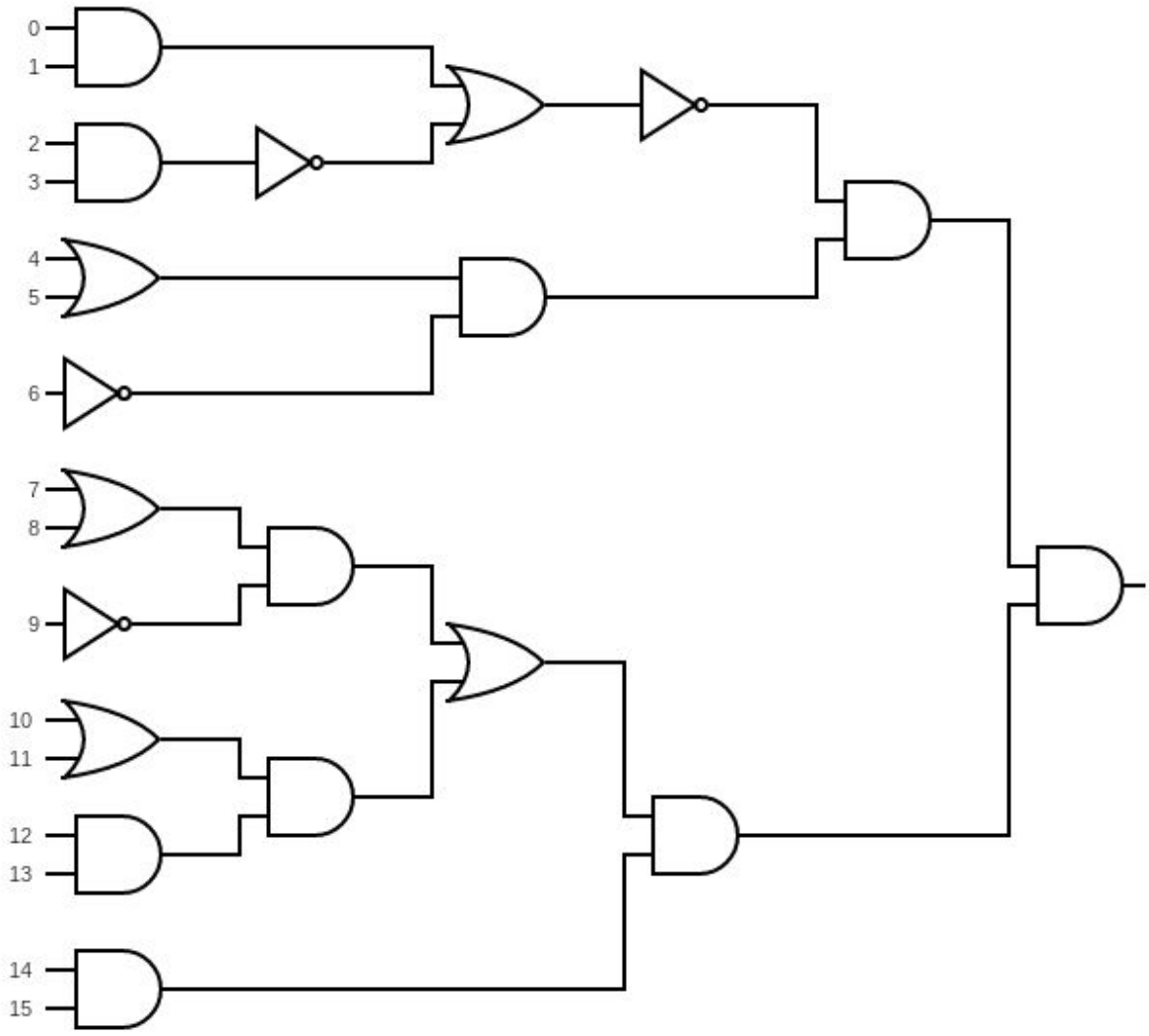
```
Found 9 valid combinations in 1.703534 ms.
```

For static it took:

```
Found 9 valid combinations in 0.809210 ms.
```

The reason for the disparity is that all of the work is exactly the same, so static, being optimized for constant work, is more performant. These times were obtained by not printing out the result in each ***check_circuit*** function. You want to skip printouts if checking timings.

Description of Algorithms & Libraries Used:
No additional libraries beyond -fopenmp were required but I utilized an extra include: <inttypes.h> because they provide standardization in integer naming. It was necessary to come up with a circuit (algorithm) for the graduate portion of the problem so I created the following:



Description of Functions & Program Structure:
The program begins in main and some initial variables are set. Depending on if the user compiled with the -DGRAD flag or not it will either run ***check_circuit*** or ***grad_circuit***. Timing of these will be performed, though for optimal timing performance the person compiling should comment out the printouts from ***check_circuit*** and ***grad_circuit***.

check_circuit and ***grad_circuit*** are the same code, only differing in the if block. ***check_circuit*** is based on the circuit provided and ***grad_circuit*** is a custom-built circuit for the graduate portion of the assignment. Each function sets up an array based on the integer z that was passed to it. From there it runs the bitstring of ***z***, saved in ***v[]*** against the circuit's if block. If the block is true it will return a 1, otherwise a zero. The

number of “hits” are stored in a sum to be printed out after the program is done. Upon completion of testing the aforementioned sum is printed and the program exits.

EXTRACT_BIT was rewritten to be an inline function because #defines are dangerous if not fully understood and should be avoided whenever possible. Here is a good link detailing #define: https://stackoverflow.com/a/321173

Description of Testing & Verification Process:
Testing of problem 1 was rather straightforward. I partook in peer review to see if our numbers matched, and we were all in agreement. Initially I struggled because of a wraparound issue regarding static but that was resolved. Oddly enough if you have an unsigned integer overflow dynamic won’t infinite loop but static will. I presume this has to do with differences in how they are under the hood. For timing verification I added an extra loop and ran it for 200 iterations, then divided the time by the number of runs to obtain the speed I noted above. For the ***grad_circuit*** I walked the if block 4 times to ensure there were no issues and I added extra parenthesis to reduce confusion.

Description of Program:
Problem 2 was familiar as I'd done the Sieve of Eratosthenes in the past. I attempted to follow the steps you listed but I could not get OMP to beat out sequential running I seriously spent about 4 hours trying to make it work. I made numerous attempts to rewrite the code in the same manner you had described but I could not get it to run fast with OMP at all (it would run without OMP just fine). In order to get it to work I had to resort to an optimized version.

This resulted in a great improvement!
Without OMP where n = 1,000,000:

```
 78470: 999563 999599 999611 999613 999623 999631 999653 999667 999671 999683
 78480: 999721 999727 999749 999763 999769 999773 999809 999853 999863 999883
 78490: 999907 999917 999931 999953 999959 999961 999979 999983
Elapsed time = 5.982519 ms
```

With OMP where n = 1,000,000 with dynamic scheduling:

```
 78470: 999563 999599 999611 999613 999623 999631 999653 999667 999671 999683
 78480: 999721 999727 999749 999763 999769 999773 999809 999853 999863 999883
 78490: 999907 999917 999931 999953 999959 999961 999979 999983
Elapsed time = 1.270703 ms
```

This was performed over 200 runs to average out any inconsistencies and was performed on an overclocked machine.
Static scheduling was significantly slower than dynamic:

```
 78470: 999563 999599 999611 999613 999623 999631 999653 999667 999671 999683
 78480: 999721 999727 999749 999763 999769 999773 999809 999853 999863 999883
 78490: 999907 999917 999931 999953 999959 999961 999979 999983
Elapsed time = 4.802790 ms
```

This is due to the inner for loop being a dynamic workload. I ramped n up to 100,000,000 and it takes 0.4163446 seconds to find all primes with OMP enabled, and 1.18152 seconds without OMP. That was interesting because OMP offered around a 4.7 times speedup when n was 1,000,000 but only 2.838 times speedup when n was 100,000,000.

Description of Algorithms & Libraries Used:
I utilized the math library to take the square root of ***n***. This greatly increases the speed at which the algorithm runs. The algorithm is as follows:

```
Set all primes to true
for (uint32_t i = 2; i < size; ++i){
        if (primes[i]){
        for (uint32_t j = i * i; j < n; j += i){
                        primes[j] = false;
                }
                }
        }
```

Description of Functions & Program Structure:
main() is the primary function and where all initial variables are checked and set, to include allocating memory for the prime array. If the user inputs an incorrect amount of variables or if they make n > 10,000,000 the user will get hit with the ***Usage()*** function. Usage prints out how to use the program and forces an exit. If there were no issues with input the program kicks into the OMP section if compiled with -DOMP or the #else section if not. The OMP section and non-OMP section are the same except OMP implementation. After finding all of the primes the ***Printout()*** function is called to give a nicely formatted printout of the prime numbers that were found and how long it took to run. Unlike program one there is no need to suppress the printout because the printout is performed after all timing, and collection of primes does not "cost" anything. The primes array is then freed and the program exits.

Description of Testing & Verification Process:
Testing was done by reducing the size of n to 10 at first, then 100. The program output was verified against online tables of prime numbers. Many sequential versions of the sieve were constructed but this one was settled on due to how well it works with OMP. Other sieves worked sequentially but when implemented with OMP were extremely slow or did not compile.

Description of Submission:
This submission contains the code for all parts of Programming Assignment #1. The repository can be found here:
https://github.com/macattackftw/HighPerfComputing/tree/master/Prog1

Problem 1 is contained in Prog1_1.c and Problem 2 is contained in Prog1_2.c. There is a Makefile which can be used to make p1 and p2 (problem 1 and 2 respectively) and you can make clean to clean them up. There is also grad_circuit.png and grad_circuit.svg which were used for the graduate problem. The circuit diagram was built using: https://www.circuit-diagram.org/editor/