Nathan Bush
007463099

CSE-5250, Spring 2023
Midterm Assignment: Benchmarking Performance of the Sieve of Eratosthenes

Instead of a midterm exam, you will be given a midterm project, where you compare performance of the sieve of Eratosthenes with varying amounts of threads/processes and different sizes of workloads.

This assignment will consist of these parts:

1. **The multi-threaded sieve of Eratosthenes program (30pts)**

   Code included below, and also as an .cpp file uploaded alongside this report:

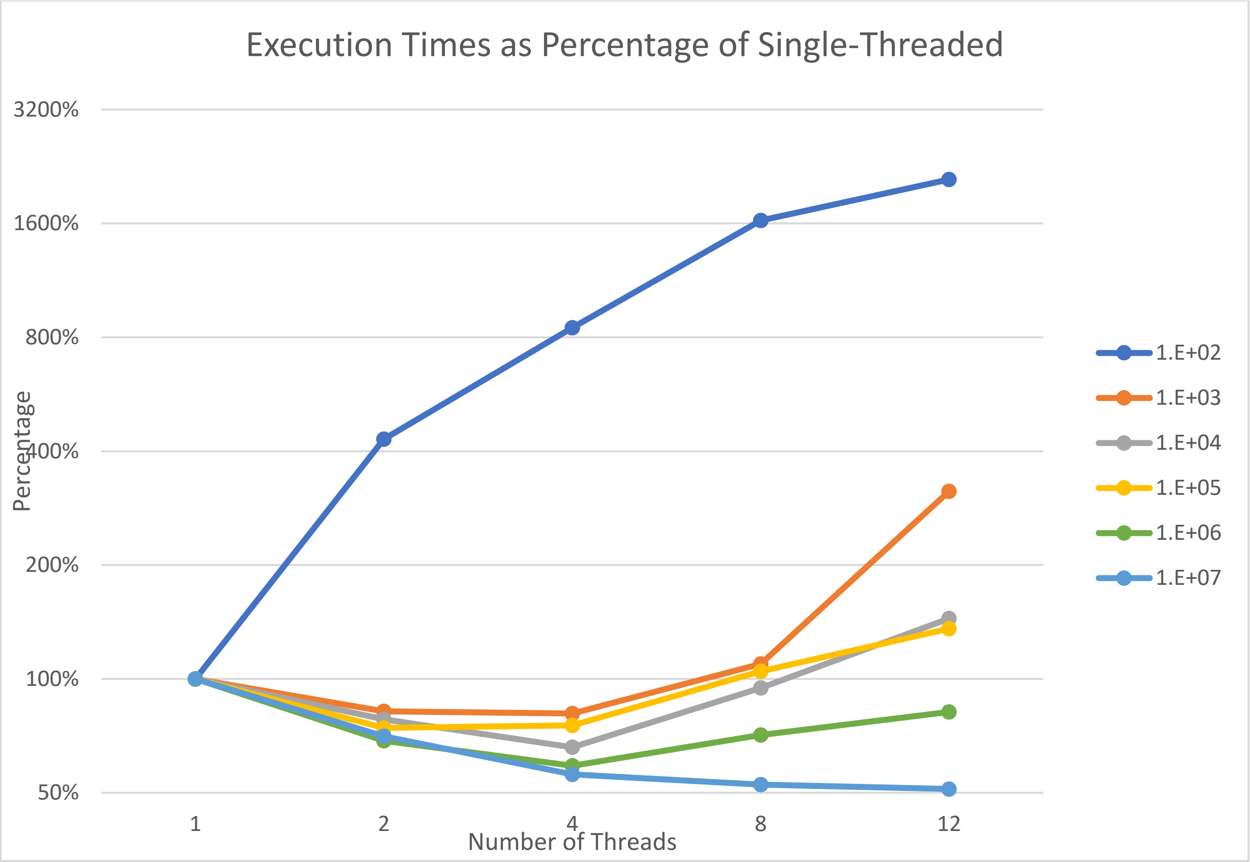
```
// CSE5250Midterm.cpp - Nathan Bush

#include <iostream>
#include <vector>
#include <cmath>
#include <chrono>
#include <omp.h>
#include <fstream>
#include <numeric>

int sieve(int, int);
std::ofstream file;

int main()
{
    int num_executions = 10;
    std::vector<int> thread_counts{ 1, 2, 4, 8, 12 };
    std::vector<int> search_maximums{ 100, 1000, 10000, 100000, 1000000, 10000000 };
    std::vector<int> times(num_executions, 0);

    file.open("output.txt");

    for (int thread_count : thread_counts) {
        for (int search_max : search_maximums) {
            std::cout << "Running Sieve of Eratosthenes with " << thread_count << "
thread/s up to a max value of " << search_max << ".\n\n";
            file << "Running Sieve of Eratosthenes with " << thread_count << " thread/s
up to a max value of " << search_max << ".\n\n";

            // run the sieve calculation for num_executions iterations, recording
execution time
            for (int i = 0; i < num_executions; i++) {
                std::cout << "\tExecution #" << i + 1 << ": ";
                file << "\tExecution #" << i + 1 << ": ";
                times[i] = sieve(thread_count, search_max);
            }

            // calculate and report average execution time
            int avg_time = std::accumulate(times.begin(), times.end(), 0) / times.size();
            std::cout << "The average execution time after " << num_executions << "
executions is " << avg_time << " microseconds.\n";
            std::cout << "----------------------\n\n";
            file << "The average execution time after " << num_executions << " executions
is " << avg_time << " microseconds.\n";
            file << "----------------------\n\n";
        }
    }
}


int sieve(int thread_count, int search_max) {
    // start timer
    auto start = std::chrono::high_resolution_clock::now();

    int sqrt_search_max = (int)sqrt(search_max);

    omp_set_num_threads(thread_count);

    std::vector<int> numbers(search_max, 0);

#pragma omp parallel for
    // initialize vector with all integers from 1 - search_max
    for (int i = 0; i < search_max; i++) {
        numbers[i] = (i + 1);
    }

    // 1 doesn't count, set the value to 0
    numbers[0] = 0;

    // perform sieve calculations
    for (int i = 0; i < sqrt_search_max; i++) {
        int current_prime = numbers[i];
        if (current_prime != 0) {
#pragma omp parallel for
            for (int j = i; j < search_max; j += current_prime) {
                int current_number = numbers[j];
                if (current_number != current_prime) {
                    numbers[j] = 0;
                }
            }
        }
    }

    // stop clock and calculate execution time
    auto stop = std::chrono::high_resolution_clock::now();
    auto exec_time = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

    // report results
    std::cout << "Found prime numbers up to " << search_max << " in " <<
exec_time.count() << " microseconds.\n\t";
    file << "Found prime numbers up to " << search_max << " in " << exec_time.count() <<
" microseconds.\n\t";
    for (int i = 0; i < 100; i++) {
        if (numbers[i] != 0) {
            std::cout << numbers[i] << " ";
            file << numbers[i] << " ";
        }
    }
    std::cout << std::endl << std::endl;
    file << std::endl << std::endl;

    return exec_time.count();
       }
```

2. OpenMP benchmarks (80pts)

Tables and graph representing benchmark data are included below. Also included alongside this report is an output.txt file showing the raw data collected by the above program. All average times were calculated using 10 trials, and individual run data can be found in output.txt.

Average Execution Time(microseconds) w/ 10 trials each					
Search Max	Number of Threads				
	1	2	4	8	12
1.E+02	**23**	99	195	375	**481**
1.E+03	179	147	**145**	196	**561**
1.E+04	1723	1347	**1136**	1631	**2488**
1.E+05	18364	**13629**	13845	19242	**24935**
1.E+06	**204643**	140448	**120588**	145546	167500
1.E+07	**2255981**	1590966	1262379	1185660	**1154220**

Execution Times as Percentage of Single-Threaded					
Search Max	Number of Threads				
	1	2	4	8	12
1.E+02	**100%**	430%	848%	1630%	**2091%**
1.E+03	100%	82%	**81%**	109%	**313%**
1.E+04	100%	78%	**66%**	95%	**144%**
1.E+05	100%	**74%**	75%	105%	**136%**
1.E+06	**100%**	69%	**59%**	71%	82%
1.E+07	**100%**	71%	56%	53%	**51%**



3. **Answer these questions (20pts; you can be as short or as detailed as you like):**
   a. **Where do you think the "sweet spot" is for thread count? Alternatively, at what point does the addition of more threads result in diminishing returns?**

   The most optimal thread count on my computer in this scenario appears to be 4 threads. 4 threads actually produced the fastest average execution time for 3 out of 6 values for search max, and essentially tied on 1e5 test. Basically, outside of the extreme cases of only 100 and 10,000,000 search maxes, 4 threads was effectively the top performer. Adding more than 4 threads resulted in more overhead from context switching that outweighed the benefit of parallelization.

   b. **Where do you think the "sweet spot" is for workload size? Alternatively, at what maximum value (max value to search for prime numbers) does it become beneficial to run the sieve in parallel?**

   A benefit to parallelization can be seen with as low as a search max of 1,000 given that the number of threads utilized is not larger than 4. We don't see a benefit to parallelization across all thread counts until the search max increases as large as 1,000,000. That search max also resulted in the point at which 4 threads began to experience diminished returns. Given that 4 threads was concluded to be optimal in a), then a search max of 1e6 seems to be an optimal candidate for parallelization.