CMPT 360: Lab Assignment #4
Benchmarking Sieve of Eratosthenes

Brady Coles

last updated: January 3, 2018

Contents

1 Course Goals

This assignment fulfills the following goals:

- a group I language (C++)
- implemented on the Windows platform
- a group IV language (Ada)
- implemented on the Windows platform

2 Problem Description

The given problem is to use the Sieve of Eratosthenes (a prime number calculator) to benchmark and compare two compilers for different languages.

3 Results and Language Comparison

The Sieve of Eratosthenes can be optimized in many ways, but for the sake of benchmarking and simplicity, only a basic implementation was used. The implementation is nearly identical in each language, limited by language rules.

In each case, the program was made to calculate primes up to 10,000 and this was repeated 50,000 times, with the total time and average time per execution being reported.

The C++ compiler produced a program that took 3.73 s total, or 0.0746 ms per time calculating primes under 10,000.

The Ada compiler produced a program that took 4.58 s total, or 0.0915 ms per trial.

Both compilers produced incredibly fast code, and the difference between them was not large, with Ada only being about 20% slower, but even after running the program many times, the C++ program was consistently faster. Both languages compile all the way to machine code, so they both get the benefit of compile time optimizations. They both take advantage of strong typing rules to avoid extra run-time operations.

Note: The programs described in this document do not reset the arrays between runs of the sieve, which is not how the sieve is meant to be run, but run-time and post-conditions are unaffected since the algorithm is deterministic, and will work as long as the array starts with at least the primes set to the default value (false in my implementation). Therefore, it is still useful for benchmarking purposes.

4 Program Documentation

4.1 Problem Solution

The unoptimized version of Eratosthenes Sieve follows the following procedure.

1. Initialize a boolean array with max index the largest number to check
2. Starting at 2, if a number has not been marked 'not prime', mark every multiple of it 'not prime'.
3. Repeat step 2, incrementing the value each time until every number in the range has been checked.

Further optimizations, such as only representing odd numbers, improve the algorithm, but were not used.

4.2 Pseudocode

This is pseudocode implementation of the unoptimized Sieve of Eratosthenes.

```
bool primes[] = array [2..MaxNumber] with default value True
for i in [2..MaxNumber] do
    if primes[i] then
        for j from 2*i to MaxNumber by i do
            primes[j] = False
        end for
    end if
end for
```

4.3 Documentation

Each program could be used to find prime numbers, here is how.

4.3.1 C++

The C++ function `sieve(primes[], size)` accepts a boolean array, which it assumes begins all false, and a length value for the size of the array. It modifies the array in place, such that the false values in the array are primes (excluding 0 and 1).

4.3.2 Ada

The Ada program has a package called `Prime_Sieve` which contains a procedure `Sieve(Primes, Size)` and type `PrimeList`. An array of type `PrimeList`, with index starting at 1 and all values false must be passed into `Sieve`, along with a parameter indicating the size of the array. The procedure modifies the array in place, and when the procedure is done, all elements that are false are prime (excluding 1).

5 Program Listings

5.1 C++ Listing

```
1  /*
2   * Author: Brady Coles
3   * Lab Assignment #4
4   * Sieve of Eratosthenes Benchmark
5   */
6
7  #include<iostream>
8  #include<chrono>
9  using namespace std;
10 using namespace chrono;
11
12 // Calculate primes up to size, leaving primes as false in primes[]
13 void sieve(bool primes[], int size) {
14     for (int i = 2; i < size; i++) {
15         if (!(primes[i])) {
16             for (int j = i + i; j < size; j += i) {
17                 primes[j] = true;
18             }
19         }
20     }
21 }
```

```
22
23  // Times the execution of the Eratosthenes Sieve benchmark
24  int main() {
25
26      int const limit = 10000;
27      int const trials = 50000;
28
29      bool p[limit] = {};
30      steady_clock::time_point begin = steady_clock::now();
31      for (int i = 0; i < trials; i++) {
32          sieve(p, limit);
33      }
34      steady_clock::time_point end = steady_clock::now();
35
36      for (int i = 2; i < limit; i++) {
37          if (!(p[i])) cout << i << " ";
38      }
39      double time = duration_cast<microseconds>(end - begin).count() * 0.001;
40      cout << endl << "Time taken = " << time << "ms" << endl;
41      cout << "Per trial = " << time / trials << "ms" << endl;
42  }
```

5.2 Ada Listing

```
1   --
2   -- Author: Brady Coles
3   -- Lab Assignment #4
4   -- Sieve of Eratosthenes Benchmark
5   --
6   -- Spec file - prime_sieve.ads
7   --
8
9   package Prime_Sieve is
10     type PrimeList is array(Positive range <>) of Boolean;
11     procedure Sieve (Primes : IN OUT PrimeList;
12                      Size : Integer);
13  end Prime_Sieve;
```

```
1   --
2   -- Body file - prime_sieve.adb
3   --
4
5   package body Prime_Sieve is
6      procedure Sieve (Primes : IN OUT PrimeList;
7                       Size : Integer) is
8        j : Integer;
9      begin
10       for i in 2..Size loop
11          if not Primes(i) then
12             j := i + i ;
13             while j <= Size loop
14                Primes(j) := True;
15                j := j + i;
16             end loop;
17          end if;
```

```
      end loop;
   end Sieve;

end Prime_Sieve;
```

```
--
--Main program file - main.adb
--

with Ada.Text_IO; use Ada.Text_IO;
with Prime_Sieve; use Prime_Sieve;
with Ada.Real_Time; use Ada.Real_Time;

-- Times the execution of the Eratosthenes Sieve benchmark
procedure Main is
   Size : Integer := 10000;
   Primes : PrimeList(1..Size);
   Start_Time, End_Time : Time;
   Elapsed_Milliseconds : Float;
   Trials : Integer := 50000;
begin
   Put_Line("Brady Coles - Lab Assignment #4");

   Start_Time := Ada.Real_Time.Clock;
   -- Actual benchmark
   for i in 1..Trials loop
      Sieve(Primes, Size);
   end loop;


   End_Time := Ada.Real_Time.Clock;
   Elapsed_Milliseconds := Float( To_Duration( End_Time - Start_Time ) ) * 1000.0;

   -- Print primes to check validity
   for i in 2..Size loop
      if not Primes(i) then
         Put(Integer'Image(i));
      end if;
   end loop;

   Put_Line("");
   Put_Line(Float'Image(Elapsed_Milliseconds) & "ms");
   Put_Line(Float'Image(Elapsed_Milliseconds / Float(Trials)) & "ms per trial");

end Main;
```

6 I/O Listing

To show that the sieve's work, after the benchmark was done, it printed the prime numbers. Here are snippets of the output.

6.1 C++

```
2 3 5 7 11 13 17 19 23 29 31 37 41 ... 9941 9949 9967 9973
Time taken = 3729.65ms
Per trial = 0.074593ms
```

6.2 Ada

```
 2 3 5 7 11 13 17 19 23 29 31 37 41 ... 9941 9949 9967 9973
 4.57614E+03ms
 9.15228E-02ms per trial
```