Prime Number Sieve Parallelization

Matt Newbigging

Edinburgh Napier Univserity

Concurrent and Parallel Systems (SET10108)

November 2017

Abstract

This report discusses an approach into evaluating the performance of three prime number generation algorithms in an attempt to improve the proficiency of each using parallel techniques.

Each algorithm will be analyzed to determine points in which a parallel approach may be implemented in order to obtain an increase in efficiency and speedup. Points of interest, particularly when attempting parallelization, will be for loops and methods found in the program code.

The approaches to be taken in this project include a manual application of threads and implementations using the OpenMP API. These methods will be discussed in this report and applied to certain areas of the algorithm; resulting differences in performance will be recorded and compared.

Results and conclusions drawn based on the implemented methods will be framed in terms of achieved speedup and efficiency, with the most successful approaches being compared against results from the other algorithms to determine the best used method in this project.

1 Introduction and Background

There exist several methods in generating prime numbers up to a given limit. Three such methods to be implemented in this project include the Sieves of Eratosthenes [4], Sundaram [1] and Atkin [2] dating 250BC, 1934 and 2004 respectively. Each of the three sieves takes a test limit value as an input, and output a list of prime numbers in order to a file. They use a list of boolean values the size of the test limit, initially all set to either true or false. After this list is set up, the algorithm is run which affects non-prime values by flipping the bool found at the appropriate place in the bools list according to the algorithm formula. Once complete, the bools list is converted to a more readable list of integers and written to a file. Since these steps are common in all three of the chosen algorithms, and since the setup and results gathering is not part of any sieve, only the middle part of each implementation that deals with each sieve algorithm method will be timed.

2 Initial Analysis

Testing and implementation will be carried out in Visual Studio 2017 on lab computers, the specifications of which are:

- CPU - i7-4790K @ 4.00Ghz, 8 cores
- RAM - 16GB ddr3
- GPU - Nvidia GeForce GTX 980
- OS - Windows 10 64 bit

Following research and implementation of the three Sieve methods, it became apparent that each algorithm has only one variable that affects it; the input of a test limit value, which is the number that the algorithm generates prime numbers up to. In order to view how this variable affects performance, baseline statistics were gathered with four increasing test limit values. Each algorithm was ran ten times at the four different test limit values, with the results averaged to give a more accurate reading. As mentioned, the algorithm

setup and results gathering will not be timed; only the time taken to execute the sieve will be measured, with times shown in milliseconds. These statistics allow each algorithm to be analyzed to give an idea of run time dependence on the input, and therefore determine its complexity.

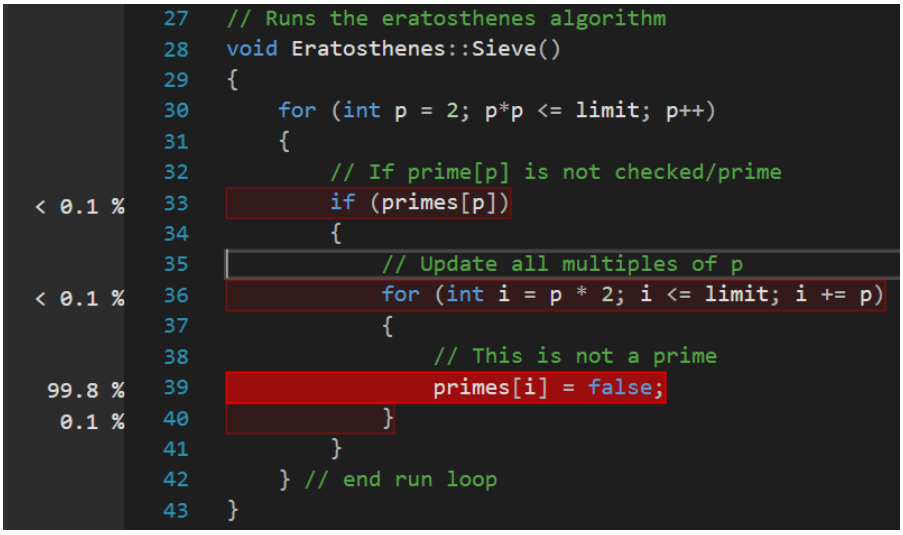
Further to finding the overall performance of each algorithm in terms of run time, the implementation itself will be scrutinized using a performance profiler to determine the most time consuming lines of code. This will provide an insight into the specific areas of logic in which a parallel approach could be taken. When aiming to improve parallel performance of a serial implementation, one looks to find loops of logic within the code. This makes any while or for loops valuable targets to attempt parallelization.

The following displays for each algorithm baseline statistics, analysis thereof, measurements of code performance and likely places to parralelize.

Sieve of Eratosthenes

Test limit	Time(milliseconds)
500,000	0.899
1,000,000	1.956
500,000,000	3629.2
1,000,000,000	7834

The above table depicts the time taken to complete the Sieve of Eratosthenes algorithm in its original, serial implementation at the same four test limit values. It can be noted from these results that as the test limit value increases, so too does the execution time giving a direct correlation between them. The arithmetic complexity of this algorithm is given as O(n log log n).

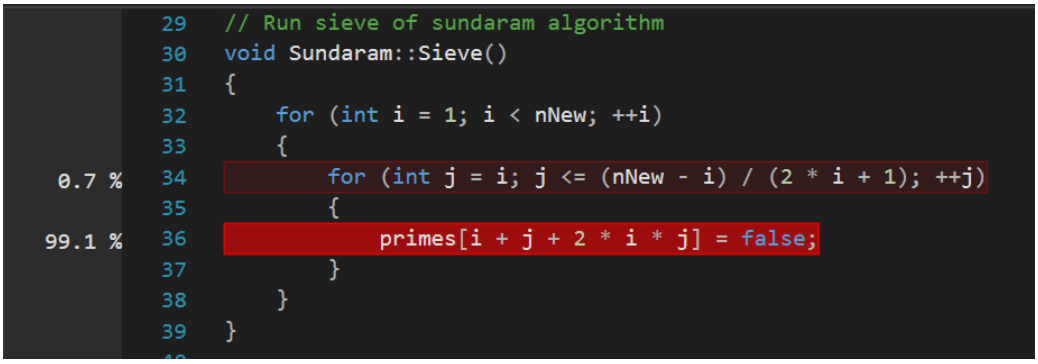


The Visual Studio profiler tool was used to give the above readings on the code involved in the Eratosthenes sieve function. It outlines how much time each line of code makes up for the total execution time. As is evident, the majority of time is spent accessing the primes list at index i, with the inner for loop calculations taking up the remaining time. As mentioned above, for loops are the first port of call in attempting to execute instructions in parallel in order to improve performance. As such, both for loops in the Eratosthenes sieve function will be targeted for parallel improvements.

Sieve of Sundaram

Test limit	Time(milliseconds)
500,000	2.788
1,000,000	5.938
500,000,000	7673.2
1,000,000,000	16839.5

The above table displays execution times for the Sieve of Sundaram algorithm in its original, serial implementation at the four different test limit values. The results illustrate that execution times increase as the test limit values increase; a direct correlation. The arithmetic complexity of this algorithm is given as O(n log n).

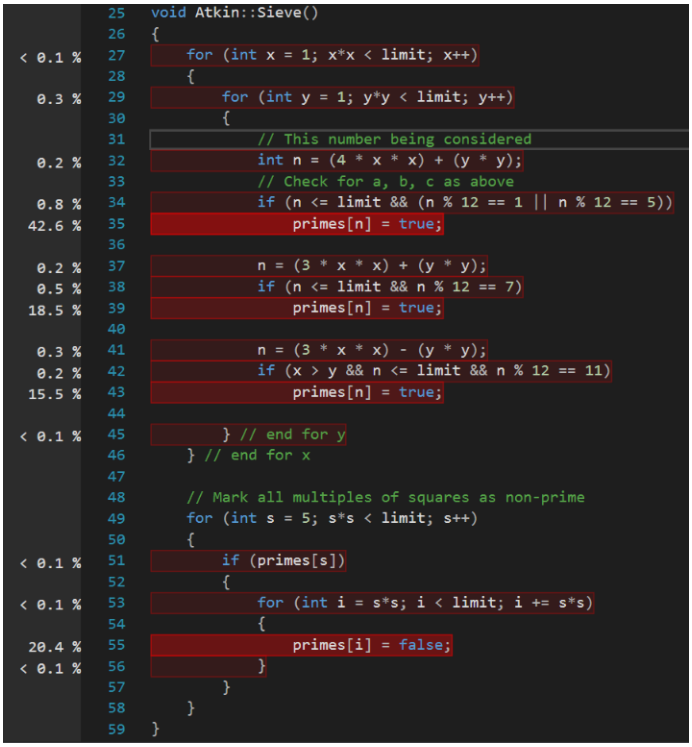


Using the Visual Studio profiler tool gave the above readings on the code involved in the Sundaram sieve function. This shows similar results to Eratosthenes in that accessing the primes list makes up for most of the execution time, with the remainder falling to the inner for loop calculations. Both for loops within the Sundaram sieve function mark worthwhile points for parallel improvements.

Sieve of Atkin

Test limit	Time(milliseconds)
500,000	1.846
1,000,000	3.955
500,000,000	8413
1,000,000,000	19124.5

The above table shows run times in completing the Sieve of Atkin algorithm in its original, serial implementation at the four different test limit values. As with the preceding algorithms, there is a direct correlation between test limit and run time; the higher the test limit the longer the run time. The arithmetic complexity of this algorithm is given as O(n/log log n).



```
        25  void Atkin::Sieve()
        26  {
< 0.1 %  27      for (int x = 1; x*x < limit; x++)
        28      {
 0.3 %  29          for (int y = 1; y*y < limit; y++)
        30          {
        31              // This number being considered
 0.2 %  32              int n = (4 * x * x) + (y * y);
        33              // Check for a, b, c as above
 0.8 %  34              if (n <= limit && (n % 12 == 1 || n % 12 == 5))
42.6 %  35                  primes[n] = true;
        36
 0.2 %  37              n = (3 * x * x) + (y * y);
 0.5 %  38              if (n <= limit && n % 12 == 7)
18.5 %  39                  primes[n] = true;
        40
 0.3 %  41              n = (3 * x * x) - (y * y);
 0.2 %  42              if (x > y && n <= limit && n % 12 == 11)
15.5 %  43                  primes[n] = true;
        44
< 0.1 %  45          } // end for y
        46      } // end for x
        47
        48      // Mark all multiples of squares as non-prime
        49      for (int s = 5; s*s < limit; s++)
        50      {
< 0.1 %  51          if (primes[s])
        52          {
< 0.1 %  53              for (int i = s*s; i < limit; i += s*s)
        54              {
20.4 %  55                  primes[i] = false;
< 0.1 %  56              }
        57          }
        58      }
        59  }
```

In profiling the Atkin sieve function, a trend can be noted in that accessing the primes list takes the most amount of time. The modulus calculations come in second in that regard, with for loop calculations costing the least amount of time by comparison. This function features two groups of nested for loops, which make for sensible targets for parallel improvements.

The profiler results for the three algorithms each exhibit a bottleneck in the same area; accessing the primes list. Since this occurs within two nested for loops in each case, it would be ideal to have this execute in parallel. Therefore, having the total number of for loop iterations carried out in equal amounts simultaneously should reduce overall execution times; this approach, then, chooses not to improve the speed of accessing the primes list, but make parallel the accessing of it.

3 Methodology

The general approach will adopt a test control that ensures the test limit variable remains the same across each algorithm; the assigned limit of 1 billion. The value being investigated is the total execution time for each sieve function at this test limit. The parallel approaches

implemented across all three algorithms will be run 10 times in release mode with results averaged to give a more accurate reading. Measures of speedup and efficiency will be taken in comparing parallel approaches to the original baseline performance of each algorithm.

Technical approaches implemented in this project include the use of OpenMP and manual threads, described below. They were chosen due to the nature of the sieve algorithms; they each feature nested for loops which multithreading and OpenMP lend themselves to parallelizing well. Other available parallel approaches, such as using GPU programming with CUDA, were deemed unfit for use with prime number algorithms. This is due to the fact that the sieve algorithms used are largely data dependent, making it difficult to section tasks and have them carried out independently of each other as one data step requires the completion of the previous in order to prevent unnecessary computation and ensure correct results.

3.1 OpenMP

OpenMP is an API to support shared memory concurrency; multi-processing. It utilises threads, but abstracts this from the developer. To implement this approach, the developer need only target individual for loops within the program by adding the necessary #pragma line above the loop, with any additional required parameter options set.

The motivation to use OpenMP in this project is due to the fact that it is a simple, powerful tool that allows great results as regards execution times at little cost in terms of developer input. Furthermore, OpenMP works by targeting loops of code which each algorithm features.

3.2 Threads

With the arrival of the C++11 standard in 2011 [3], and with it threads, developers were given tools to create multithreaded applications without the need for third party extensions. A thread can be thought of as a worker. When executing instructions laid out in program code, multiple threads, or workers, can be called upon to carry out the instructions simultaneously.

The use of manual threads can achieve a greater speedup than using OpenMP since threads have no API overhead. In order to achieve this, however, the developer requires a thorough understanding of the code being made parallel; code must be made threadsafe. This approach is being used in the project optimistically; with the outlook that it should be possible to achieve greater speedup using threads manually rather than relying on a more automated approach as with OpenMP. Threads will be used to carry out logic contained within for loops found in each sieve method.

4 Results and Discussion

Findings of the project are given in the following section, to include results of the multithreaded and OpenMP approaches in all three algorithms. Measures of speedup and efficiency were taken to accompany the resulting execution times of the parallel implementations. The equations for each are shown below; speedup:

$$S = \frac{T_{serial}}{T_{parallel}} \tag{1}$$

Where T_{serial} is the time taken to run the program without parallelization, and $T_{parallel}$ is the time taken to run the parallel version of the program. The efficiency formula:

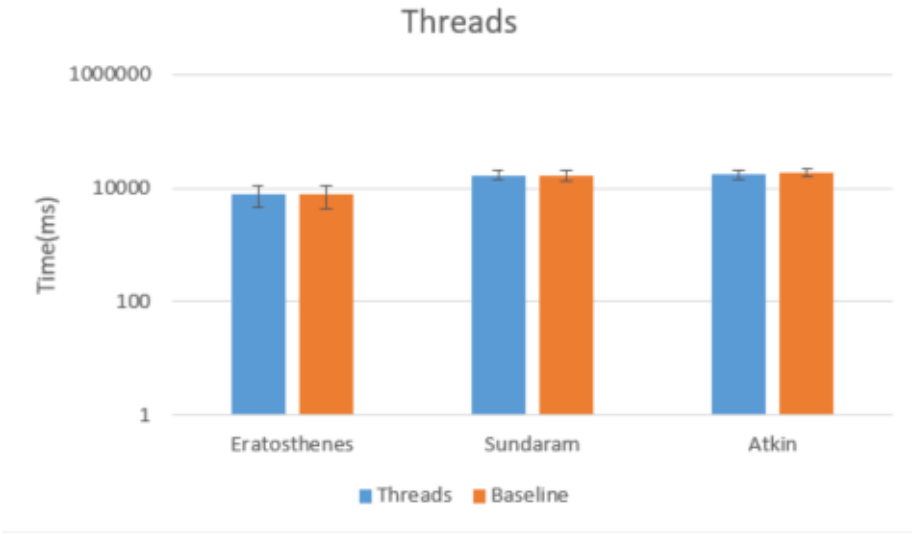
$$E = \frac{T_{serial}}{p \cdot T_{parallel}} \tag{2}$$

Where p is the number of processors used, or threads in this case.

The following table displays resulting run times in milliseconds of the multithreaded parallel approach against the original baseline performance, maintaining a test limit of one billion:

Time(ms)	Eratosthenes	Sundaram	Atkin
Threads	7704.9	17001.1	17200.5
Baseline	7834	16839.5	19124.5

The chart below accompanies the above table of figures, depicting multithreading results for the three sieves:



The results show minimal error margins, with similar figures for all three algorithms; Eratosthenes was the fastest, followed by Sundaram and Atkin. Speedup and efficiency measures for this approach are as follows:

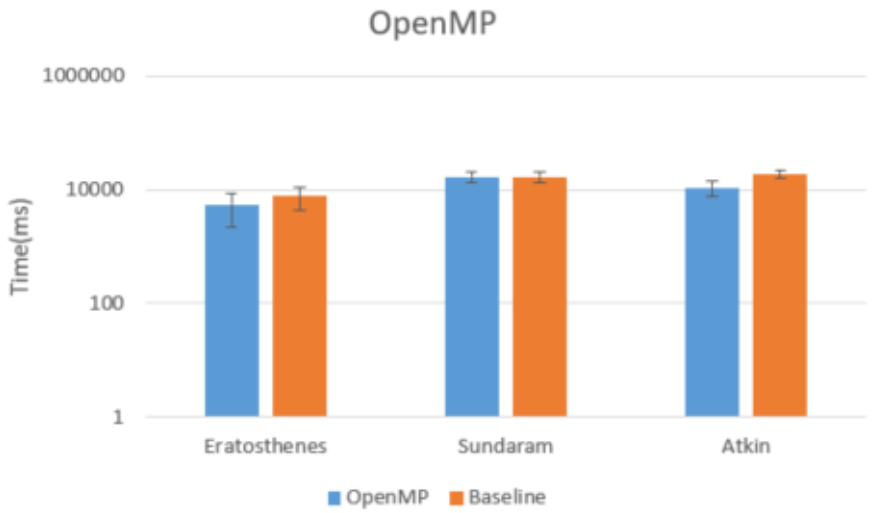
Threads	Eratosthenes	Sundaram	Atkin
Speedup	1.017	0.99	1.111
Efficiency	0.127	0.124	0.139

As is apparent from the above table, the multithreaded approach affected the Atkin algorithm most positively, achieving the greatest values for speedup and efficiency in this area. Sundaram saw a slight, albeit negligible, negative effect in terms of speedup.

The following table displays resulting run times in milliseconds of the OpenMP parallel approach against the original baseline performance, maintaining a test limit of one billion:

Time(ms)	Eratosthenes	Sundaram	Atkin
OpenMP	5457	16791.8	10795.3
Baseline	7834	16839.5	19124.5

The chart below illustrates the above table of results for the OpenMP approach in all three sieves:



Atkin achieved the greatest positive change to run time, followed by Eratosthenes. Sundaram received a negligible difference under the OpenMP implementation. Speedup and efficiency for this approach are as follows:

OpenMP	Eratosthenes	Sundaram	Atkin
Speedup	1.436	1.003	1.772
Efficiency	0.179	0.125	0.221

As the figures show, Atkin benefited most from the OpenMP parallelization, followed by Eratosthenes while it had little effect on Sundaram.

5 Conclusion

Throughout the project, three prime number generation algorithms were sourced and implemented in order to generate primes in order up to one billion. A baseline set of performance

statistics was gathered, experimenting with the input variable to understand its effect on run times; the greater the input the longer the execution time. Two types of parallel approach were then implemented, threads and OpenMP, affecting for loops within the logic of each algorithm with resulting execution times measured against the baseline. It was ensured that all parallel implementations returned a correct result. The nature of each sieve method, particularly Eratosthenes, made it difficult to parallelize in that the current data step being calculated relied on the completion of the previous in order to prevent unnecessary repeated processing and ensure correct data.

The results gathered define a clear winner as regards parallel approach; OpenMP gave the highest returns in terms of speedup and efficiency compared to that of threads for two of the three implemented algorithms. The least positively affected run time was for the Sundaram algorithm, which evidently does not lend itself well to parallelization as the results prove with there being a negligible positive impact in the OpenMP approach and a slight negative impact in the multithreaded approach. It it certainly worth noting that the efficacy of a multithreaded approach largely depends on the quality of implementation; the difficulty involved simply does not compare to that of OpenMP. This is a major consideration to make when attempting to gain parallel performance, which frames performance gains against effort expended in implementation.

From the work undertaken and results thereof, this project concludes that OpenMP provides the biggest return in terms of parallel performance at the least effort in implementation for the three sieves of Eratosthenes, Sundaram and Atkin.

References

[1] Andrew Baxter. *Sundaram's Sieve*. URL: http://banach.millersville.edu/~bob/math478/History/Sundaram.html.

[2] Alfonso L. Castaño. *Sieve of Atkin*. URL: https://fylux.github.io/2017/03/16/Sieve-Of-Atkin/.

[3] CPP. *Thread*. URL: https://goo.gl/HzSF5t.

[4] Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. URL: https://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf.