**Analysis and optimization of Prime Number Sieve Algorithms**

Valentina Scarfi – 40182166  
Concurrent and Parallel Systems (SET10108)

**Abstract**

Three algorithms to generate prime numbers are analysed, having their baseline performances traced to identify which parallelization techniques can be applied and where in each algorithm. After this analysis, multiple techniques have been considered, but only two have been chosen and these are OpenMP and C++11 threads. These techniques improved the overall performances of the algorithm, but few problems occurred.

It is concluded that the best technique is OpenMP, which combines correctness of output and performances improvements with ease of use.

# **Introduction and Background**

**Prime Numbers** The algorithms that will be analysed generate prime numbers up to N. A prime number **p** is a positive integer that cannot be divided by any number other than 1 and itself [5].

**Prime Numbers Algorithms** Amongst many available algorithms to generate prime numbers, three Sieves have been chosen for this project: Eratosthenes [1] [2], Sundaram [3] and Atkin [4]. These differ from each other both for approach and complexity, which will be discussed in more detail in the next section.   
These algorithms will be used to generate prime numbers up to one billion and to write them to a file, from the smallest to the biggest.

**Project Objectives** The main purpose of this project is to analyse possible bottlenecks of each algorithm proposed and to apply appropriate parallelization techniques to optimize them and improve performances. Results will be then evaluated on different machines in terms of speedup, efficiency and correctness of output.

**Techniques** Several parallelization techniques have been considered in first place, but only two have been implemented, which are C++11 threads and OpenMP library. They both allow to parallelize a serial algorithm in different ways. These techniques will be described and discussed later in this document

# **Initial Analysis**

**Hardware** The all three Sieves have been tested on two different machines: PC at University (Games Lab) and Home PC.

Games Lab

* CPU – Intel i7-4790K @ 4.00GHz

4 hardware and 4 logical = 8 cores

* GPU – Nvidia GeForce GTX 980
* OS – Windows 10 Pro N 64 bit

Home

* CPU – Intel i5 760 @ 2.80GHz 4 hardware cores
* GPU – Nvidia GeForce GTX 960
* OS – Windows 10 Pro 64 bit

**Sieve of Eratosthenes** This Sieve is the easiest to implement amongst the three and the one with the lowest time complexity of (***Figure 1***). Having a vector of Booleans all set to true, the algorithm starts calculations from the first prime number, which is 2. It then checks the Booleans from 2 to *n*, and if the checked bool is true, it marks all multiple of the current number as false [2].

It then prints all prime numbers at position of a true Boolean.

**Sieve of Sundaram** Sundaram is an algorithm with notable complexity of (***Figure 1***). This Sieve only considers odd numbers, so, having a vector of integers all set to 1, it starts by halving the upper bound by 2. It then loops through all numbers from 1 to . For every number within this range, it finds all solutions to , where is the halved limit and and are the outer and inner loops. This can be also solved for [7].

It then marks all numbers at position .

If a number is marked as 0, it’s not prime number. Then, for each number at position , the prime is of the form .

Al primes are then printed.

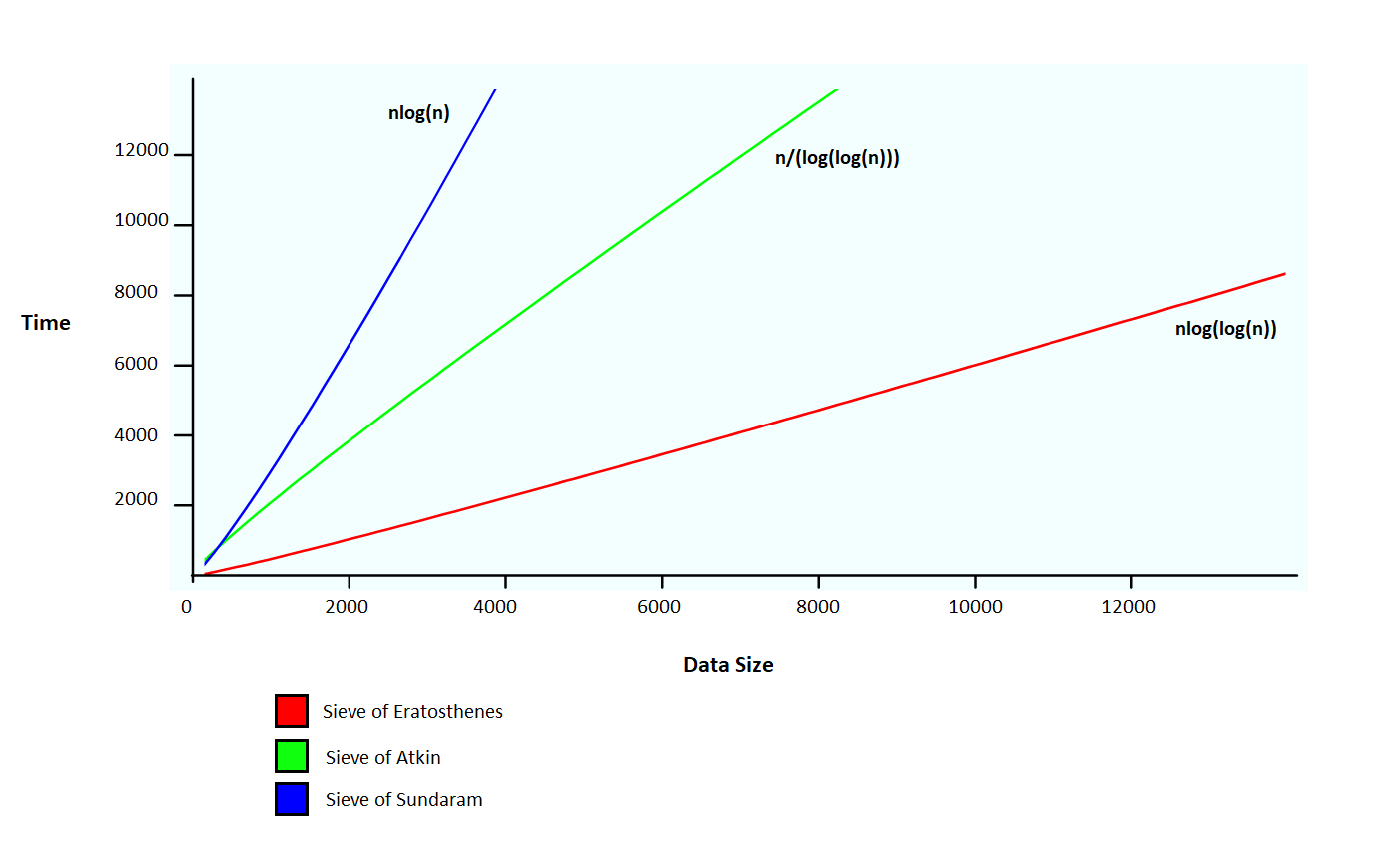
**Sieve of Atkin** This algorithm is the hardest to implement amongst the three and the one with a good time complexity (***Figure 1***).

Starting with a vector of false Booleans, the ones at position 2 and 3 are marked as true. Then, the algorithm loops twice from 1 to , marking the numbers that meet certain conditions (always valid for ):

* , if is equal to 1 or 5, then the number is prime
* , if is equal to 7, then the number is prime
* , if and is equal to 11, then the number is prime

After this process, all the multiples of squares from 5 to are marked as non-prime [4].

The algorithm then is ready to print all data.



**Figure 1:** time complexity of each algorithm, generated with a graph generator [5]

**First Analysis** Baseline performances of the three algorithms has been evaluated and recorded accordingly to identify possible optimizable bottlenecks. Each run of each algorithm has been timed and results averaged to get overall serial performances. The timings don’t include any write-to-file statement or vector initialization as these processes are the same for each algorithm.   
For accuracy and data consistency, all the tests are run in Release mode x64.

**Correlation** In order to determine correlation and relationship between size of data processed and time taken, lower data sizes than one billion have been recorded to gather the correlation coefficient, which is 1, or very close, for all the three algorithms (***Table 1***).

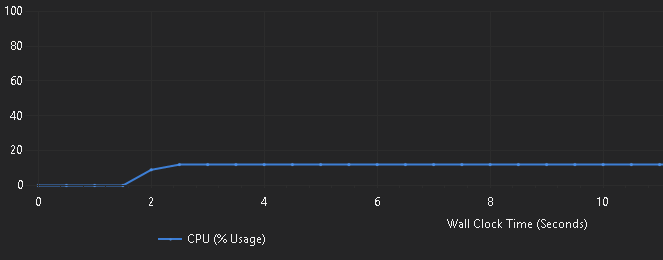
|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm:** | Eratosthenes | Sundaram | Atkin |
| Upper Bound (data size) | Time (s) | Time (s) | Time (s) |
| 100000 | 0.00031 | 0.000575 | 0.000646 |
| 1000000 | 0.0032456 | 0.00885 | 0.006087 |
| 10000000 | 0.0431964 | 0.219325 | 0.072176 |
| 100000000 | 0.836 | 3.294 | 1.547 |
| Correlation | 0.999 | 1.000 | 0.999 |

**Table 1:** table showing upper bound (data size) – time taken correlation based for the three Algorithms (Home only)

**Figure 2:** Correlation table for the Sieve of Sundaram

As ***Figure 2*** shows, in the case of the Sundaram algorithm, there is a perfect linear correlation between the time taken and the size of data to be processed. The correlation is also linear for Atkin and Eratosthenes.

**Bottlenecks and parallelization Analysis** Interesting data has been gathered from the Visual Studio 2017 Performance Profiler as it is clear, from ***Figure 3***, that the base algorithm is only using one core, as the CPU usage is constantly between 10% and 20%.   
On a 8 cores machine (Games Lab) the CPU usage for a serial application should exactly be in this range as a single working thread reaches the 12,5% of the total CPU usage.   
The Profiler also shows where in the code the CPU spent most of the time (***Figure 4***).



**Figure 3:** CPU Usage for Eratosthenes (Games Lab)



**Figure 4:** Inclusive Samples, in percentage, for all three algorithms

While for Eratosthenes and Sundaram it is clear that the CPU spent most of its time executing code in the inner loop of the algorithms, having Inclusive Samples higher than 80% in specific parts of code (in red), for Atkin it is not so obvious: as ***Figure 4*** shows, Inclusive Samples percentage is spread across the algorithm, having the highest hit at 34%.   
This analysis through the Performance Profile is very useful as it shows that the nested loops are certainly the bottlenecks of these algorithms, therefore the approaches used will focus on these parts of the code.

# **Methodology**

**From the analysis to the approach** As the initial analysis shown, all three algorithms have nested *for loops* which make the overall time complexity grow and the CPU spends most of the time executing calculations within them. For this reason, the preferred approaches to be implemented are OpenMP and C++11 manual threads, as they both are very suitable for nested loops, especially the first one.

**Other approaches** Excluding the already cited ones, other approaches have been considered on a first stage. One of this is SIMD, which was initially considered as one of the possible parallelization solution. This technique, however, is not suitable for the type of calculations within the algorithms. In fact, SIMD would be a good approach for problems focused on numeric calculations, such as additions. This approach wouldn’t then suit all three algorithms, having as exception only a part of Atkin, as few calculations are performed within the loop.

**OpenMP** The first technique successfully implemented is the OpenMP API. It has been chosen as it focuses on *for loops* with its *parallel for* statement, splitting the workload of the loop amongst the available threads. As already discussed in the previous section, the loops are the main bottleneck of these algorithms.

Even if OpenMP is relatively easy and quick to implement, few attempts have been tried to find the best solution that combined an acceptable speedup with the correct output.

Some of the attempts include:

* Using *parallel for* on the outer loop, inner loop or both
* Changing scheduling (KEEP GOING HEREEEEEE)

**Algorithm 2:** OpenMP usage

1. //Using OpenMP declaring r as private and scheduling the work statically
2. #pragma omp parallel for private(r) schedule(static)

As *Algorithm 2* shows, *vec r* has been declared as private. This is because, by default, all variables within the loop are shared. In this case, having *vec r* as shared, causes an incorrect output image: a shared *vec r* can be overwritten by other threads that are using the variable.

**Multi-threading** The first technique that has been applied is C++11 thread feature that allows to retrieve the available threads, which are limited by the machine hardware. Using mutexes and lock guards combined with threads is necessary to prevent that more than one thread operates with the resource at the same time.

Threads have been created assigning a start and an end for each, which is the dimension of the image divided by the number of available threads (*Algorithm 1*). This has been implemented to guarantee an even workload to each thread.

**Algorithm 1:** algorithm showing threads retrieval and workload splitting

1. //Retrieving available threads
2. auto num\_threads = thread::hardware\_concurrency();
3. //Splitting dimension of the image by available threads
4. auto range = dimension / num\_threads;

Two approaches have been tried when creating threads: using all available threads to perform the required calculations and using all the available threads minus the last one which has been replaced by the main thread.

No difference in timings has been found, thus the second approach has been used for next tests.

# **Results and Discussion**

**Tests** Several tests have been run using each technique. As *Table 5* shows, the improvement in terms of average time of execution is significant.

Speedup and hardware efficiency, based on the number of spheres in the image, have been calculated to evaluate the effective result of each implemented feature (*Table 6*).

**Games Lab Results** From the table presented (*Table 5*, *Table 6*), a good speedup and efficiency has been achieved with all the implementations. Efficiency is slightly lower than 1, which is a good result as 1 would be a perfect efficiency. Speedup achieved is also good, since a value equal to 4 (number of cores) would be a linear speedup, which is rare to obtain. The best result has been achieved by OpenMP with a dynamic scheduling. An effective speedup of 72,51% has been reached by this implementation, followed by manual threads with a speedup of 71,94% and OpenMP with static scheduling with 71,54%.

**Table 5:** graph showing difference of timings between original serial algorithm and improved algorithm using different techniques (Games Lab)

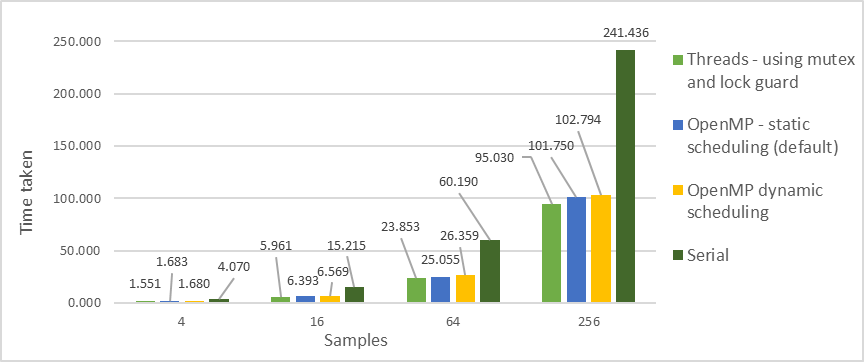
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Speedup of each technique** | | | | | **Efficiency of each technique**  **(p = 4)** | | |
|  | **Algorithm** | | | | **Algorithm** | | |
| **Technique** |  | **Erato** | **Sund** | **Atkin** | **Erato** | **Sund** | **Atkin** |
| **Threads** | 1.509 | 1.584 | 1.739 | 0.377 | 0.396 | 0.435 |
| **OpenMP** | 0.393 | 24.307 | 1.729 | 0.098 | 6.077 | 0.432 |

**Table 3:** speedup and hardware efficiency for each technique, considering different numbers of spheres. A 400x400 image with 256 samples per pixel has been processed in the Games Lab to obtain these results.

**Home Results** The results for speedup and efficiency obtained from home machine (*Table 7*) are very different compared to Games Lab’s ones. This is because of the very different hardware, as one is a 1st generation and the second is a 4th generation Intel processor. The speedup and efficiency obtained at home are positive, but not as good as the ones obtained in the Games Lab. Also, manual threads results are better than OpenMP with a dynamic schedule. The effective speedup achieved at home by the best technique, which is C++11 multi-threading, is of a 60,64%. This can still be considered a good result, as the gain in time, especially considering the samples per pixel (*Table 8*), is positive and still significant.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Speedup of each technique** | | | | | **Efficiency of each technique**  **(p = 4)** | | |
|  | **Algorithm** | | | | **Algorithm** | | |
| **Technique** |  | **Erato** | **Sund** | **Atkin** | **Erato** | **Sund** | **Atkin** |
| **Threads** | 1.284 | 1.552 | 1.419 | 0.321 | 0.388 | 0.355 |
| **OpenMP** | 0.973 | 1.022 | 1.388 | 0.243 | 0.255 | 0.347 |

**Table 7:** speedup and hardware efficiency for each technique, considering different numbers of spheres. A 400x400 image with 256 samples per pixel has been processed at Home to obtain these results.



**Table 8:** graph showing difference of timings between original serial algorithm and improved algorithm using different techniques with different samples per pixel on an image 400x400 at Home.

The CPU usage, for all the different techniques, hits an average of 75% with peaks near 80%. From the Performance Profiler this can be clearly seen, although the CPU usage in this case seems to oscillate more than the serial algorithm usage (*Figure 3*)



**Figure 5:** threads, OMP static and OMP dynamic CPU usage at Home

# **Conclusion**

**Threads and dynamic OpenMP** As the results show, Games Lab and Home performances, with the techniques implemented, are quite different. Especially when it comes to compare threads and OpenMP with a dynamic schedule: the second is the best one in the Games Lab, while the first has better results at Home. This might happen because of the hardware differences already discussed: dynamic schedule in OpenMP could cause some overhead at Home because of the fewer cores available compared to the ones in the Games Lab.

**Real improvement** Analysing again *Table 5* and *Table 8* it is evident how all the techniques used have levelled up the huge difference in performance between the smallest sample per pixel and sphere number analysed, 4 and 9 respectively, and the greatest number of these considered, 256 and 20. While in the serial algorithm an image with 20 spheres was taking almost 30 seconds to be created more than the one with only 9 spheres, the parallelized algorithm has a difference of 5 to 8 seconds to perform the same calculation.

**Final considerations** The results obtained by the techniques implemented can be considered satisfactory. Even though there are differences in the results of each implementation, these are not enough significant to prefer a technique over the others in terms of performance obtained.

On the other hand, OpenMP is the way easier to implement than threads, as the workload is split and managed directly by the API, while using manual threads implies the use of mutexes and guards to guarantee a proper functioning of the whole system.

**References**

1. *Sieve of Eratosthenes* (used for Serial and OpenMP) from: <http://www.geeksforgeeks.org/sieve-of-eratosthenes/>
2. *Sieve of Eratosthenes* (used for Threads) from: <http://www.algolist.net/Algorithms/Number_theoretic/Sieve_of_Eratosthenes>
3. *Sieve of Sundaram* from: <http://www.sanfoundry.com/cpp-program-generate-prime-numbers-between-given-range-using-sieve-sundaram/>
4. *Sieve of Atkin* from: <http://www.sanfoundry.com/cpp-program-implement-sieve-atkins/>
5. *Prime Number* from: <http://mathworld.wolfram.com/PrimeNumber.html>
6. *Graphing Calculator* from: <https://my.hrw.com/math06_07/nsmedia/tools/Graph_Calculator/graphCalc.html>
7. *Sieve of Sundaram* (explanation) from: <https://luckytoilet.wordpress.com/2010/04/18/the-sieve-of-sundaram/>