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.

1. Introduction and Background

Prime Numbers The algorithms that will be analysed must 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, especially, correctness of output.

Possible 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

2. 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 $O(nlog(log(n)))$ (**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 $O(nlog(n))$ (**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 $limit\ /\ 2$. For every number within this range, it finds all solutions to $i + j + 2ij < m$, where m is the halved limit and i and j are the outer and inner loops. This can be also solved for $j < \frac{m-1}{2i+1}$ [7].
It then marks all numbers at position $i + j + 2ij = 0$.
If a number is marked as 0, it's not a prime number. Then, for each number at position i, the prime is of the form $2i + 1$.

All 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 $O(n/(log(log(n))))$ (**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 $\sqrt{limit}$, marking the numbers that meet certain conditions (always valid for $num \leq limit$):

- $num = 4x^2 + y^2$, if num % 12 is equal to 1 or 5, then the number is prime
- $num = 3x^2 + y^2$, if num % 12 is equal to 7, then the number is prime
- $num = 3x^2 - y^2$, if $x > y$ and num % 12 is equal to 11, then the number is prime

After this process, all the multiples of squares from 5 to $\sqrt{limit}$ are marked as non-prime [4].

The algorithm then is ready to print all data.

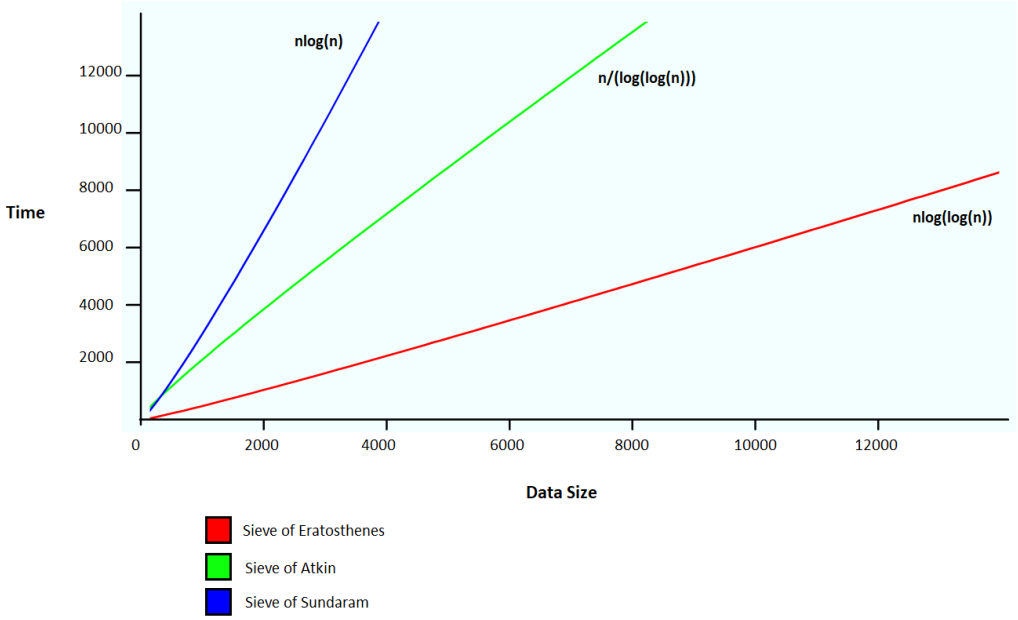

Figure 1: time complexity of each algorithm, generated with a graph generator [5]

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.

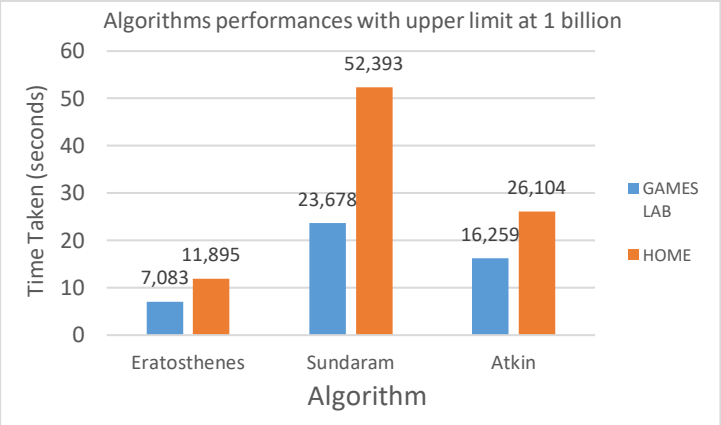

Figure 2: Baseline performances of serial algorithms

As **Figure 2** shows, timings are as expected, as the previous time complexity analysis pointed out. Eratosthenes is the fastest on both Lab and Home PCs, while Sundaram is the slowest, hitting an average of almost a minute on Home machine.

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**).**Figure 3:** Correlation table for the Sieve of Sundaram

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)

As ***Figure 3*** 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.

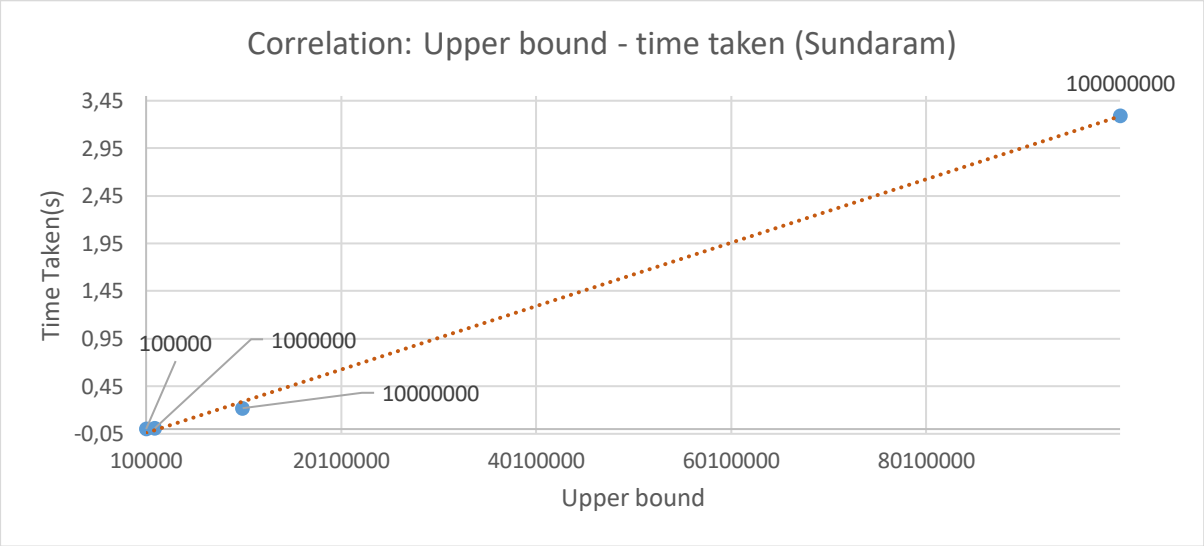

Figure 3: Correlation table for the Sieve of Sundaram

Bottlenecks and parallelization Analysis Interesting data has been gathered from the Visual Studio 2017 Performance Profiler as it is clear, from ***Figure 4***, that the base algorithm is only using one core, as the CPU usage is constantly between 10% and 20%.

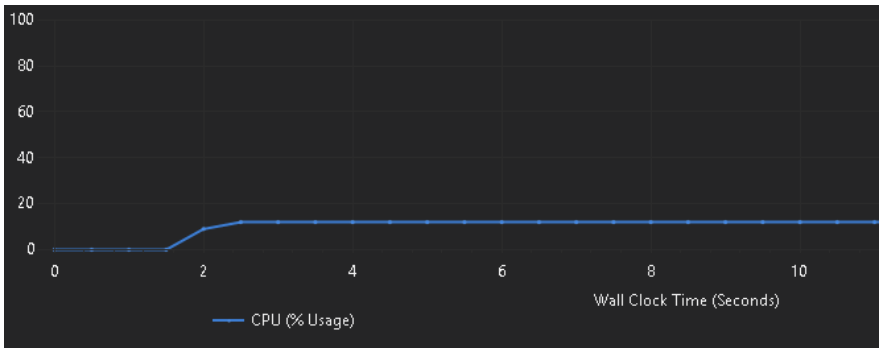

Figure 4: CPU Usage for Eratosthenes (Games Lab)

On an 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 5***).

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 5*** shows, Inclusive Samples percentage is spread across the algorithm, having the

highest hit at 34%.

Figure 5: Inclusive Samples, in percentage, for all three algorithms

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.

3. 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 arithmetic calculations, such as additions or multiplications. 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 such as static with chunks defined or dynamic
- Defining loop indexes as private or shared.

After multiple OpenMP approaches have been tried, the chosen ones differ for each algorithm: for Eratosthenes and Atkin OpenMP has been used for the inner loop, keeping the default schedule, which is static; for Sundaram, on the other hand, the outer loop has been optimized and a static scheduling with chunks of 10 has been defined (***Figure 6***).

Atkin

```
    // Looping through x and y, finding numbers that meet the conditi
    for (int x = 1; x <= lim; x++)
    {
        //Parallelizing inner loop --> outer loop causes wrong output
#pragma omp parallel for num_threads(nThreads)
        for (int y = 1; y <= lim; y++)
        {
            // First condition to be met
            // Having x = 1, y = 2 --> 4 * 1 * 1 + 2 * 2 = 8
            int num = (4 * x * x) + (y * y);

            // if num = 8, this condition won't be met, as 8 % 12 = 
            // if num = 13 for x = 1, y = 3, condition will be met a
            if (num <= n && (num % 12 == 1 || num % 12 == 5))
            {
                isPrime[num] = true;
            }

            // If previous conditions are not met, this will be chec
            num = (3 * x * x) + (y * y);
            if (num <= n && (num % 12 == 7))
            {
```

Eratosthenes

```
    for (int i = 2; i * i <= n; i++)
    {
        // If prime[i] is true (unchanged), then
        // If prime[i] is false, it has been mar
        if (isPrime[i])
        {
#pragma omp parallel for num_threads(nThreads)
            // Looping through multiples of i an
            for (int j = i * 2; j <= n; j += i)
            {
                isPrime[j] = false;
            }
        }
    }
```

```
#pragma omp parallel for num_threads(nThreads) schedule(static, 10)
    for (int i = 1; i < n; i++)
    {
        // from i + j + 2 * i * j <= nNew that can be solved as  j 
        int l = (n - i) / (2 * i + 1);

        for (int j = i; j <= l; j++)
        {
            // numbers of the form i + j + 2 * i * j are not prime
            // E.g. i = 1, j = 2
            // 1 + 2 + 2 * 1 * 2 = 7
            // the boolean at isPrime[7] will be marked as 0
            isPrime[i + j + 2 * i * j] = 0;
        }
    }
```

Sundaram

Figure 6: OpenMP applied to all algorithms

This difference in the approach comes with output problems arose when defining chunks for Eratosthenes and Atkin, which didn't happen with Sundaram and, since having chunks defined improved performances (as will be discussed in the next section), it is the only algorithm that has a different scheduling approach.

Multi-threading The second technique implemented is C++11 thread feature. This technique has been chosen for its ability of split workload amongst multiple threads manually, which suits the proposed problems well, as already discussed.

Like OpenMP, multiple attempts have been tried before finding the solution that suited the most each algorithm. More attempts have been tried for Eratosthenes, since none of the loops gave expected results after being threaded.

Threads have been implemented having the main logic loop in a method. After retrieving the threads available, using *thread::hardware_concurrency()* function, each thread is created calculating a range by splitting the current $limit$ amongst the threads and setting the start point according to what each algorithm requires (e.g. Eratosthenes starts from 2, while Atkin and Sundaram start from 1). The end point is set to $previousStart + range$. After a thread of the separated method has been created and pushed to a vector, start is set to $endPoint + 1$. When all the available threads have been created, these are joined together using the function *t.join().*

An example of this implementation can be seen in ***Figure 7***.

```
// Rounding square root of n up to the nearest integer
int lim = (int)ceil(sqrt(n));

auto range = lim / nThreads;

int startPoint = 1;
int endPoint = 1;

//Loop for creating threads based on a start and an end point
for (int i = 1; i <= nThreads; i++)
{
    //endPoint is the previous start point plus the range
    endPoint = startPoint + range;

    //Creating and pushing Threads into a vector
    allThreads1.push_back(thread(&Thread::threadedAtkin, this, startPoint, endPoint, ref(isPrime)));
    //setting start at end of previous thread + 1
    //this is because in the main loop of the algorithm the x is <= end.
    startPoint = endPoint + 1;
}

for (auto &t : allThreads1)
{
    t.join();
}
```

Figure 7: Example of threaded Sieve of Atkin

4. Results and Discussion

Tests Multiple runs have been performed, using both techniques on each algorithm, recording results and averaging times. Output files containing prime numbers have been verified accordingly with different practical tests, such as line counting, file size comparison and random check at same row position across different files.

Although, few problems arose with the output for threaded Eratosthenes as the algorithms outputs 1 KB more than it should. Even with multiple attempts to readapt the implementation, the problem has not been fixed.

Games Lab Results As can be seen in ***Table 2*** and ***Figure 8***, a modest improvement in performances has been achieved with both techniques.

Efficiency, on the other hand, doesn't reach the full power with any technique, as each core is working at less than half of its maximum capability, considering a value of 1 as a perfect efficiency. Speedup, that is perfectly linear when it is equal to 4, is acceptable for both techniques on all algorithms, excluding threaded Sundaram: this Sieve hasn't had any positive speedup, but it slowed down by 2,65% compared to serial. The best result has been achieved using OpenMP which reached an effective speedup of 43,18% for Atkin, 36,85% for Sundaram and 33,72% for Eratosthenes.

Threaded achieved a modest speedup for Atkin, which increased by 42,17%. Eratosthenes, although the output values are not correct by a small amount of numbers, also increased its speed by a small 10,63%.

Speedup of each technique					**Efficiency (p = 4)**		
	Algorithm				**Algorithm**		
Technique		**Erato**	**Sund**	**Atkin**	**Erato**	**Sund**	**Atkin**
	OpenMP	1,509	1,584	1,739	0,377	0,396	0,435
	Threads	1,119	0,974	1,729	0,280	0,244	0,432

Table 2: Speedup and Efficiency of both techniques on each algorithm, calculating prime numbers up to 1 billion (Games Lab)

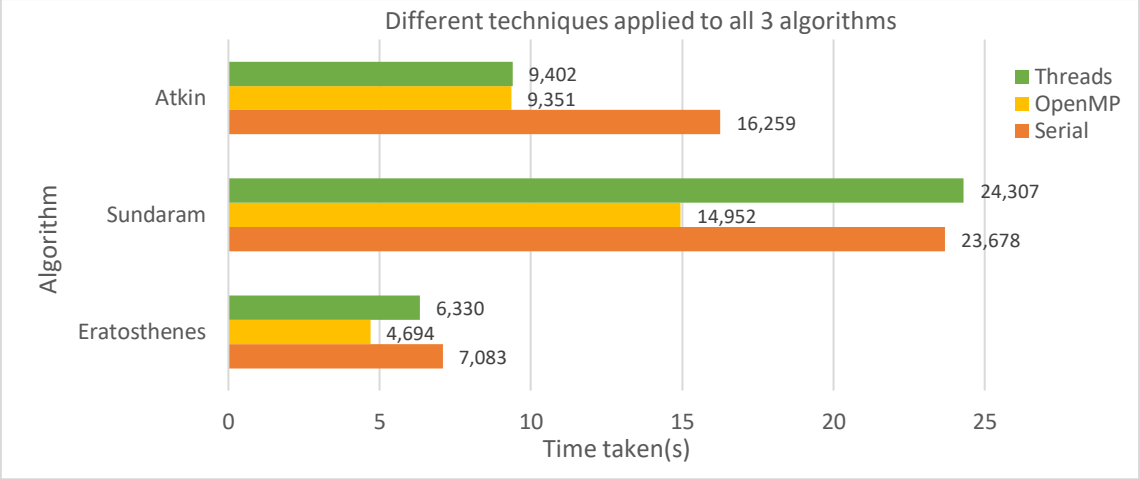

Figure 8: Final timings for both techniques on each algorithm, calculating prime numbers up to 1 billion (Games Lab)

Home Results The results obtained from home machine (***Table 3*** and ***Figure 9***) follow the same pattern of Games Lab's ones, except that in this case threaded Eratosthenes has been slowed down by 2,82%, while Sundaram achieved a very tiny speedup of 2,11%. OpenMP is still the best technique achieved, with a maximum speedup of 35,55% with the Sieve of Sundaram.

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

Table 3: Speedup and Efficiency of both techniques on each algorithm, calculating prime numbers up to 1 billion (Home)

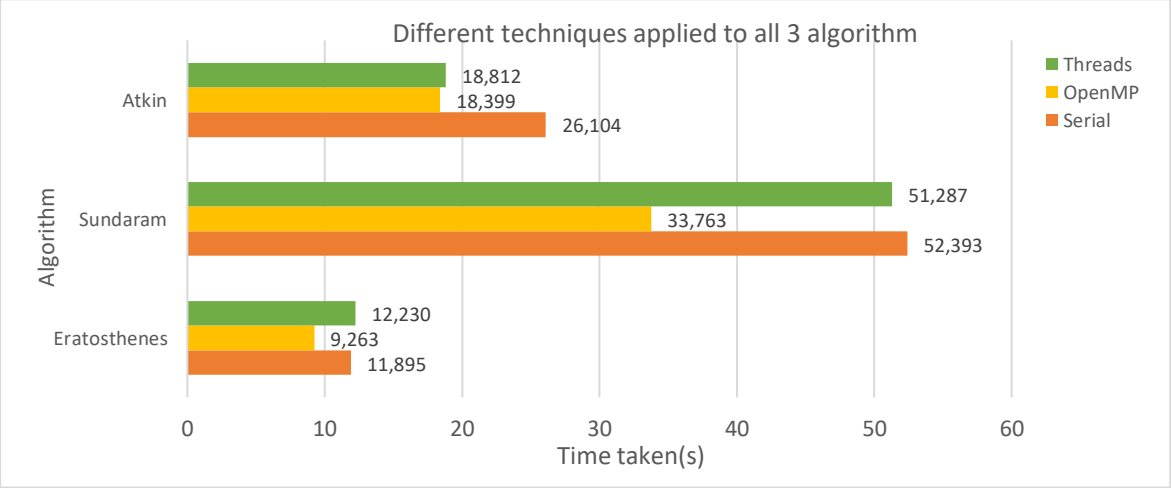

Figure 9: Final timings for both techniques on each algorithm, calculating prime numbers up to 1 billion (Home)

CPU Usage The Performance Profiler (***Figure 10***) shows how differently each technique influence the CPU usage: while for OpenMP CPU usage constantly reached more than 80% when executing main loops, threads produce an inconstant behaviour with peaks higher than 80% and slowing down as the current run approaches the end.

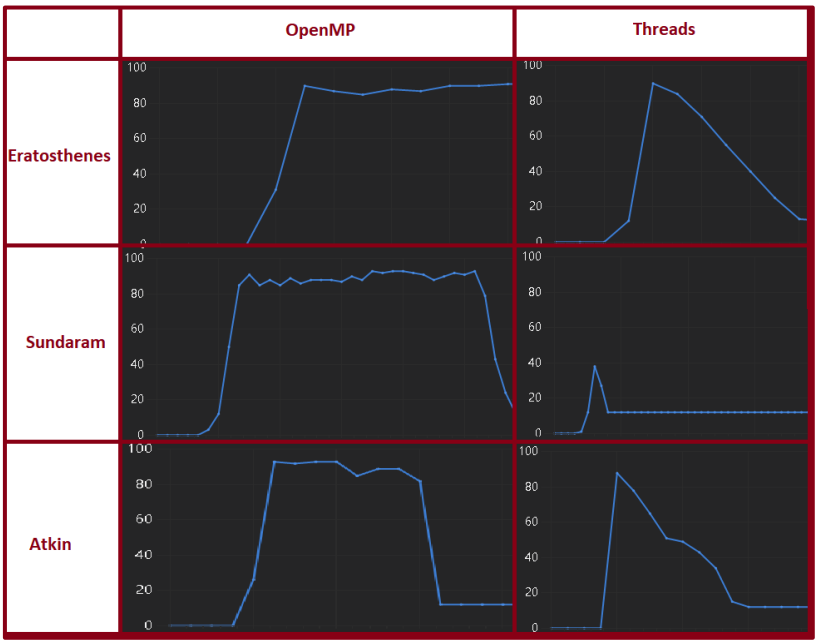

Figure 10: Final Performance Profile CPU usage for both techniques on each algorithm

From ***Figure 10*** is it clear that Threaded Sundaram is highly inefficient compared to other algorithms using the same technique, as CPU usage doesn't go higher than 40%.

5. Conclusion

Techniques After considering different possible techniques, the Initial Analysis pointed out which parts of the code could've been parallelized. As the nested *for loops* were taking the highest CPU usage and execution time, techniques tailored to work with loops have been prioritized.

OpenMP and manual thread have been then implemented, trying different approaches to combine a possible speedup with output correctness.

This purpose has been achieved with OpenMP, after few attempts, but threads leaded to problems such as wrong output and overall slow down of execution time. Even with several attempts, trying different combinations of threads, these problems haven't been fixed.

Modest improvement Looking back at ***Figure 8*** and ***Figure 9***, it can be easily noticed that OpenMP and threads results can be equally compared only when applied

to Atkin, as they both achieved a similar speedup, even though OpenMP is still slightly faster.

For the other algorithms, OpenMP is definitely faster than threads, as these, in few cases, even slowed down by few seconds the overall algorithm runtime.

Moreover, OpenMP achieved with only few combinations attempts a correct output, while threads took much more effort, not even fully achieving this purpose in the case of the Sieve of Eratosthenes.

The overall speedup obtained by OpenMP can be considered fairly satisfactory, even though it is clear, looking at ***Table 2*** and ***Table 3***, that the efficiency achieved doesn't correspond to the full potential obtainable.

Final considerations and future work After having analysed all the results, it can be concluded that OpenMP, related to these implementations of prime Sieves, must be preferred over manual threads as it always produces a correct output, which is the main focus of the problem presented in this document.

Different techniques can be applied to parallelize these three Sieves, such as OpenCL or CUDA. The second one will be probably preferred over the first one as it is simpler to use.

SIMD could be reconsidered, in combination with OpenMP, to optimize those few arithmetic calculations in the loops of the presented Sieves.

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/