University of Porto - Faculty of Engineering

Master in Informatics and Computing Engineering

Project Report

Sieve of Eratosthenes

Impact of Parallelism and Cache-Aware Programming

Miguel Rodrigues
up201906042@edu.fe.up.pt
Sérgio Estêvão
up201905680@edu.fe.up.pt

March 2023

Contents

List of Figures

1 Introduction

Sieve of Eratosthenes is an algorithm to generate prime numbers up to a given limit N. Prime numbers are natural numbers whose factors are 1 and itself. The origins of this algorithm date back to the early of 2^{nd} century BC.

Besides that, prime numbers have found their way out of the field of number theory into real-world applications, particularly in cryptography. They are used in the generation of public and private key pairs.

Given this, improving the time execution of this algorithm can be advantageous and there are several optimizations that can be applied. The main aim of this report is to demonstrate how the application of such optimizations impacts execution time.

Considering this, in the first stage, we will show the implementation of the basic sieve, i.e. the implementation of the algorithm as it was proposed. After that, we will demonstrate how it is possible to improve the running times, while considering relevant aspects such as locality and cache sizes.

After that, we will use multi-core processing to further improve the running times and the speedup achieved according to the number of cores. This is made using OpenMP API [1].

To close this report, there will be a section for conclusions where we will analyze the obtained results. This will also include a short discussion of how the concepts applied for each optimization can be used in other situations where computational performance is necessary.

2 Tools for performance measurements

In this section, we will provide a short briefing about each of the external tools we have used to develop the programs shown in the following sections. Another relevant note is that all the tests shown in this report were executed on a machine with a 5.15.0-60-generic Kernel, running an Intel® Core™ i7-9750H CPU (Coffee Lake) with 6 cores and 12 threads with Hyper-Threading @ 2.60 GHz.

2.1 PAPI

PAPI[1] [2], short for *Performance Application Programming Interface*, is a very useful and easy-to-use library to measure the performance of a given piece of code.

It provides counters that measure the number of cache misses - for both L1 and L2 - as well as the total number of clock cycles and instructions performed during the execution of a program.

2.2 OpenMP

OpenMP[2], is an API specification for parallel programming.

The API provides the convenience of not having to deal with thread management, which can get complicated very quickly. Therefore, such responsibility is passed to the compiler which implements the specification.

[1]Version 7.0.0.0 of PAPI was used.
[2]Version 4.5 of OpenMP, bundled with g++-12, was used.

The usage is pretty straightforward. In source code, `#pragma` statements allow a certain region to be executed in parallel. Also, when compiling the flag `-fopenmp` must be provided to enable this functionality.

3 Sequential Algorithms

In this section, we will show 3 different implementations of the sieve. The first will be the basic version, followed by the fast-marking of multiples versions, and last, the segmented version.

We will focus on the sequential implementation of the algorithm, each version has distinct degrees of optimizations and considerations regarding cache friendliness. We have opted to use C++20 for implementing the versions in the following subsections due to its compatibility with OpenMP and the functionality implemented by the C++ standard library. Moreover, it provides the opportunity for a more granular refinement of the program's behavior, and subsequently its performance.

3.1 Basic

The basic version of the sieve is the easiest to understand. It starts with the seed k and goes from k^2 up to N marking all the numbers that are divisible by k. The next step is to find the smallest non-marked element greater than k and make it the new seed. It just follows the algorithm as it was proposed.

Even though this is the most basic version, it already has several optimizations in place. In line 5, `sieve` represents a list of numbers up to N, only containing odds. Additionally, it is important to highlight the role of the `std::vector<bool>`. According to the standard [3], `std::vector<bool>` specification states that its members shall be packed in bits rather than bytes.

With this in mind, the combination of the 2 optimizations allows that this program uses 16 times less memory than what would be the initial expectation.

Another optimization is present in line 10, where composite numbers are marked without branching operations, easing the load on the branch predictor.

```
1  static constexpr std::uint64_t N = 2 << 25;
2
3  int main(void)
4  {
5      std::vector<bool> sieve(N >> 1, true);
6      std::uint64_t k = 3;
7
8      do {
9          for (std::uint64_t i = k * k; i < N; i += 2)
10             sieve[i >> 1] = sieve[i >> 1] && (i % k != 0);
11
12         do {
13             k += 2;
14         } while (k * k <= N && !sieve[k >> 1]);
15     } while (k * k <= N);
16
17     const auto computed = std::count(sieve.cbegin(), sieve.cend(), true);
18     cpa::sieve_of_erastothenes::report<N>(computed);
19     return 0;
20 }
```

3.2 Multiples

The multiples version is built on top of the basic version. The unique catch here is that seed's multiples from k^2 up to N are marked as composites.

The improvement here happens because of the efficiency associated with the operations. In modern processors, addition and multiplication are much more efficient than subtraction and division, respectively. In the previous version the usage of the modulo operator, which can be computed using a `div` instruction, creates a significant overhead when compared with this version.

Even though this version offers an improvement over the previous versions it is far from ideal. Despite using more efficient operations it still lacks in what regards cache-friendliness, especially, for large values of N.

```
1  static constexpr std::uint64_t N = 2 << 25;
2
3  int main(void)
4  {
5      std::vector<bool> sieve(N >> 1, true);
6      std::uint64_t k = 3;
7
8      do {
9          for (std::uint64_t i = k * k; i < N; i += 2 * k)
10             sieve[i >> 1] = false;
11
12         do {
13             k += 2;
14         } while (k * k <= N && !sieve[k >> 1]);
15     } while (k * k <= N);
16
17     const auto computed = std::count(sieve.cbegin(), sieve.cend(), true);
18     cpa::sieve_of_erastothenes::report<N>(computed);
19     return 0;
20 }
```

3.3 Segmented

Considering this, there is still room for improving the performance of the sieve. The Segmented version takes a slightly different approach when compared with the previous. In this version, the main catch is that primes in a given range are computed once, i.e. loops are being reordered. The ranges are obtained by diving the list into blocks - that, ideally, fit in a line of cache. This is critical to keep the list of numbers in the cache and improve locality.

Given this, the approach here is to compute a set of initializing seeds. Since the search space for each seed starts from k^2, it means that there is no such seed bigger than $\sqrt{N}$. Also, it is important to recall that the concentration of prime numbers is higher near 0, therefore most of the seeds are present on the first block.

The next step, after calculating the initial seeds, is to determine what is the smallest seed's multiple bigger than the lower bound of the range being processed, i.e. the index in the processed block. Our implementation uses a table to keep track of this information. Following that, all the seeds' multiples in the block are marked as composites, similar to the multiplication version.

Concerning the technical optimizations, it is important to highlight the usage of `std::bitset`.

In contrast to the `std::vector<bool>`, `std::bitset` is statically allocated, thus its size must be known at compile-time. Our implementation tries to match the size of the block with the size of the L1 data cache[3], see line 7.

```
1  static constexpr std::uint64_t N = 2 << 25;
2  static constexpr std::uint64_t L1D_CACHE_SIZE = 32 * 1024 * 8;
3
4  int main(void)
5  {
6      constexpr auto n_sqrt = static_cast<std::uint64_t>(std::sqrt(N));
7      constexpr auto segment_size = std::max(n_sqrt, L1D_CACHE_SIZE);
8      std::uint64_t computed = 0, k = 3;
9
10     std::bitset<segment_size> sieve;
11     std::vector<std::pair<std::uint64_t, std::uint64_t>> seeds;
12     std::bitset<(n_sqrt >> 1) + 1> is_prime;
13     is_prime.set();
14
15     for (std::uint64_t low = 0; low <= N; low += segment_size) {
16         const auto high = std::min(N, low + segment_size - 1);
17
18         for (; k * k <= high; k += 2) {
19             if (!is_prime[k >> 1])
20                 continue;
21
22             for (std::uint64_t i = k * k; i <= n_sqrt; i += 2 * k)
23                 is_prime[i >> 1] = false;
24             seeds.emplace_back(k, k * k - low);
25         }
26
27         sieve.set();
28
29         for (auto& [seed, i] : seeds) {
30             for (const auto incr = 2 * seed; i < segment_size; i += incr)
31                 sieve[i] = false;
32             i -= segment_size;
33         }
34
35         for (auto i = low + 1; i <= high; i += 2)
36             computed += sieve[i - low];
37     }
38
39     cpa::sieve_of_erastothenes::report<N>(computed);
40     return 0;
41 }
```

3.4 Results

The following plots show the results obtained for the 3 different single-core versions. The results displayed below were obtained by calculating the average between 3 distinct executions for each algorithm and each limit N (2^N). Due to the long execution times, some of the benchmarks for the basic algorithm executions for higher limit values (between 2^{30} and 2^{32}) were estimated using an exponential regression.

[3]The value used in our implementation is 32 KiB - the L1 cache size of an Intel® Core™ i7-9750H

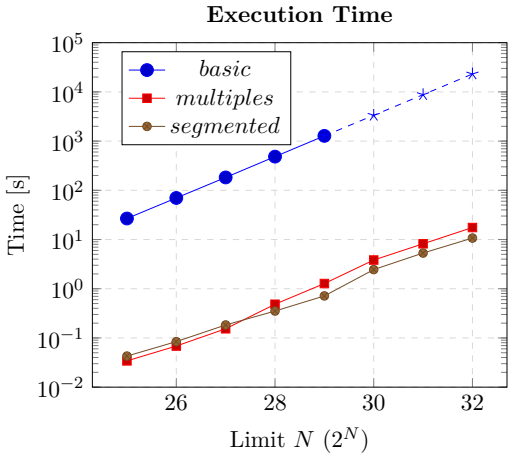
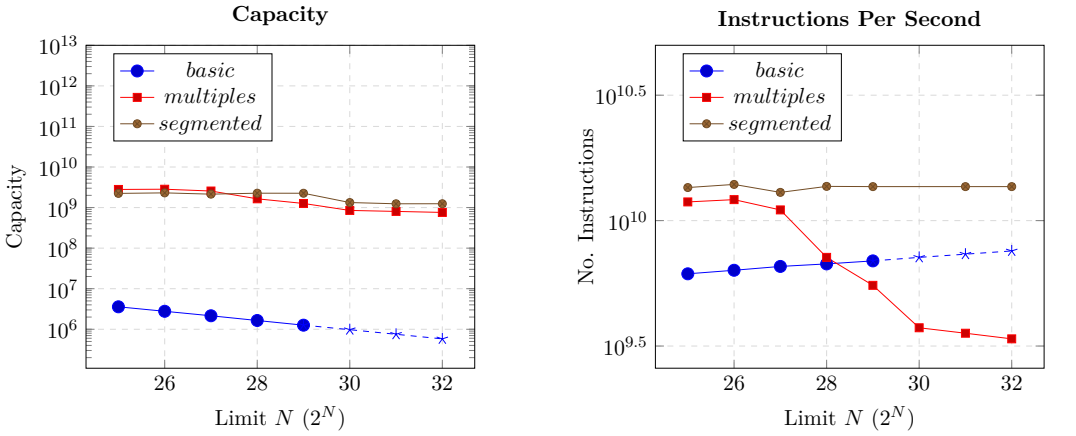


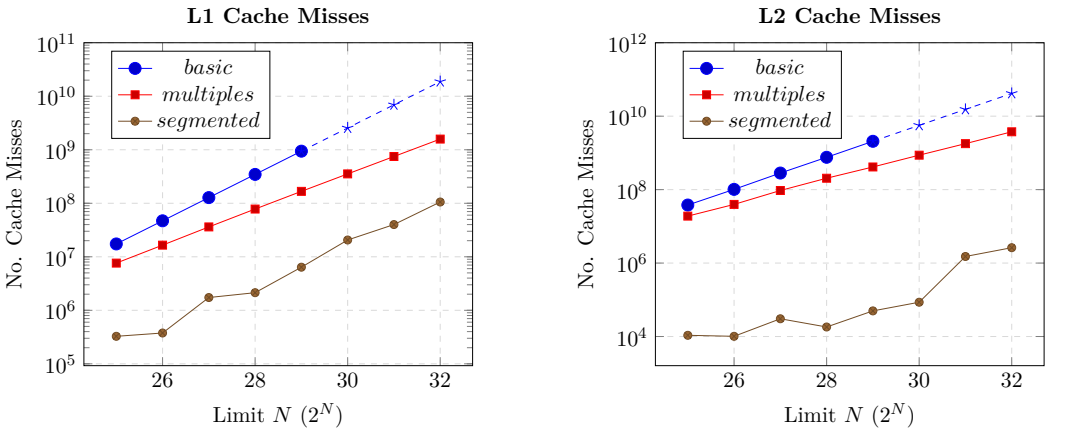
Figure 1: Execution Time for the Sieve Of Eratosthenes



(a) Capacity

(b) Instructions executed in each second

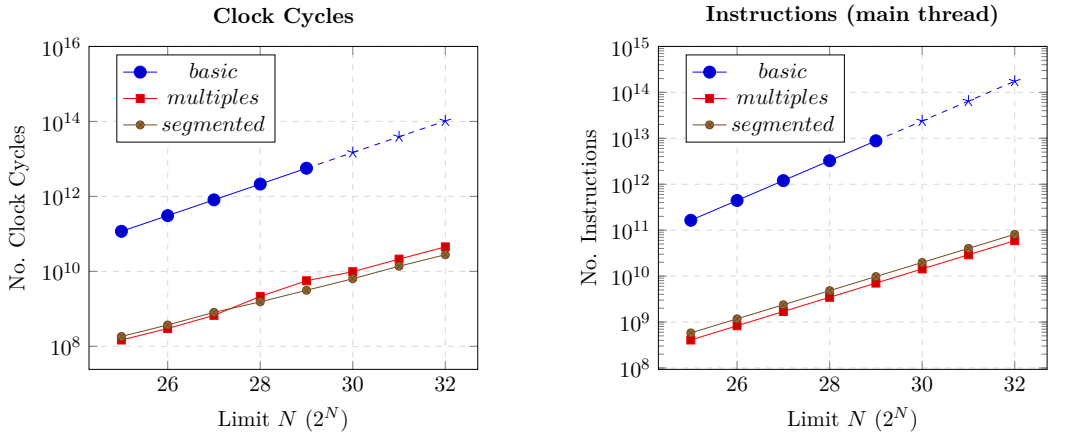
Figure 2: Scalability evaluation for the different Sieve of Eratosthenes executions



(a) Cache level 1 data misses

(b) Cache level 2 data misses

Figure 3: Cache data misses during the different Sieve of Eratosthenes executions



(a) Total of clock cycles completed

(b) Total number of instructions executed

Figure 4: Absolute metrics for the different Sieve of Eratosthenes executions

4 Parallel Algorithms

In spite of being an algorithm of sequential nature, the sieve of Eratosthenes can, up to a certain degree, be parallelized. Obviously, for that to be a reality there are several small adjustments that must be implemented, but those let us extract more performance from a machine's processor.

4.1 Simple `parallel for` loop

The first of the parallel implementations is the simplest. Due to its simplicity and cleverness, this was, on average, the fastest implementation. Essentially, this version is built on top of the segmented version shown in the previous section. However, there are some differences common to all parallel versions that were introduced here when compared with the single-core segmented version.

Here all the seeds up to $\sqrt{N}$ are precomputed by the main thread. Then the scheduling of each iteration, or range of processing, is statically determined by OpenMP. Also, it is relevant to highlight that the list of numbers of a given range must be a private variable to avoid false sharing in the shared region.

Another, important adjustment here is the need for a function that is able to determine what is the smallest multiple greater than the lower bound of the range being processed. Contrary to what was done in the single-core version where the index was initialized and then updated block after block, in the parallel versions such flow is no longer possible.

To overcome this data dependency problem we must calculate the indexes prior to processing the range in question. In our implementation, the lambda function `segment_index` is responsible to perform this task.

```cpp
static constexpr std::uint64_t N = 2 << 25;
static constexpr std::uint64_t L1D_CACHE_SIZE = 32 * 1024 * 8;
static constexpr std::uint8_t NUM_THREADS = 12;

int main(void)
{
    constexpr auto n_sqrt = static_cast<std::uint64_t>(std::sqrt(N));
    constexpr auto segment_size = std::max(n_sqrt, L1D_CACHE_SIZE);
    constexpr auto segment_index = []<typename T>(const T& k, const T& m) {
        return k * k > m ? k * k - m : (2 * k) - ((m - k) % (2 * k));
    };

    std::uint64_t computed = 0;
    std::bitset<segment_size> sieve;
    std::bitset<(n_sqrt >> 1) + 1> is_prime;
    is_prime.set();

    for (std::uint64_t k = 3; k * k <= N; k += 2) {
        if (!is_prime[k >> 1])
            continue;

        for (auto i = k * k; i <= n_sqrt; i += 2 * k)
            is_prime[i >> 1] = false;
    }

    #pragma omp parallel for num_threads(NUM_THREADS) private(sieve) reduction(+:computed)
    for (std::uint64_t low = 0; low <= N; low += segment_size) {
        const auto high = std::min(N, low + segment_size - 1);

        std::vector<std::pair<std::uint64_t, std::uint64_t>> seeds;
        for (std::uint64_t k = 3; k * k < high; k += 2) {
            if (is_prime[k >> 1])
                seeds.emplace_back(k, segment_index(k, low));
        }

        sieve.set();

        for (auto& [seed, i] : seeds) {
            for (const auto incr = 2 * seed; i < segment_size; i += incr)
                sieve[i] = false;
        }

        for (auto it = low + 1; it <= high; it += 2)
            computed += sieve[it - low];
    }

    cpa::sieve_of_erastothenes::report<N>(computed);
    return 0;
}
```

4.2 SPMD

The version shown below is the SPMD version. SPMD stands for *Single Program Multiple Data* and it represents a technique to employ parallelism. It can be used both in shared and distributed memory models.

A characteristic of this model is that it is often needed to know either how many threads are available

as the ids associated with those threads. Having these aspects in consideration, OpenMP provides the `#pragma parallel` instruction offering the possibility of assembling a team composed of a custom number of threads. Besides that, the OpenMP API specifies `int omp_get_num_threads(void)` and `int omp_get_thread_num(void)`. These functions return the number of threads on a given team and the associated id to each of those threads, respectively.

This is exactly what our implementation uses, see lines 29 to 31. Here, the id of a thread is used to determine the initial processing segment. Upon processing a segment, it will know that the next segment to be processed is `NUM_THREADS` segments away. In other words, a cyclic distribution of processing is performed among threads. This technique is very often used when employing SPMD.

```
1  static constexpr std::uint64_t N = 2 << 25;
2  static constexpr std::uint64_t L1D_CACHE_SIZE = 32 * 1024 * 8;
3  static constexpr std::uint8_t NUM_THREADS = 12;
4
5  int main(void)
6  {
7      constexpr auto n_sqrt = static_cast<std::uint64_t>(std::sqrt(N));
8      constexpr auto segment_size = std::max(n_sqrt, L1D_CACHE_SIZE);
9      constexpr auto segment_index = []<typename T>(const T& k, const T& m) {
10         return k * k > m ? k * k - m : (2 * k) - ((m - k) % (2 * k));
11     };
12
13     std::bitset<segment_size> sieve;
14     std::bitset<(n_sqrt >> 1) + 1> is_prime;
15     is_prime.set();
16     std::uint64_t computed = 0;
17
18     for (std::uint64_t k = 3; k * k <= N; k += 2) {
19         if (!is_prime[k >> 1])
20             continue;
21
22         for (auto i = k * k; i <= n_sqrt; i += 2 * k)
23             is_prime[i >> 1] = false;
24     }
25
26     #pragma omp parallel num_threads(NUM_THREADS) private(sieve) reduction(+:computed)
27     {
28         const int thread_num = omp_get_thread_num();
29         constexpr auto increment = NUM_THREADS * segment_size;
30         const auto begin = segment_size * thread_num;
31
32         for (std::uint64_t low = begin; low <= N; low += increment) {
33             const auto high = std::min(N, low + segment_size - 1);
34             std::vector<std::pair<std::uint64_t, std::uint64_t>> seeds;
35
36             for (std::uint64_t k = 3; k * k < high; k += 2) {
37                 if (is_prime[k >> 1])
38                     seeds.emplace_back(k, segment_index(k, low));
39             }
40
41             sieve.set();
42
43             for (auto& [seed, i] : seeds) {
44                 for (const auto incr = 2 * seed; i < segment_size; i += incr)
45                     sieve[i] = false;
46             }
```

```
47
48              for (auto it = low + 1; it <= high; it += 2)
49                  computed += sieve[it - low];
50          }
51      }
52
53      cpa::sieve_of_erastothenes::report<N>(computed);
54      return 0;
55  }
```

4.3 Tasks

The last parallel version made use of the task construct. Tasks are a form of functional parallelism opposed to the data parallelism seen in the previous parallel versions.

Fortunately, with the introduction of OpenMP version 4.5, the `taskloop` construct simplifies the whole process of dealing with tasks, hence the minimal changes when comparing this version and the initial parallel version.

Nevertheless, this simplicity does not translate into faster execution times. In fact, the overhead derived from the management of tasks makes this program as slow as the sequential segmented implementation regardless of the number of tasks.

Such slowness might be explained by several factors. The first of those has to do with our own implementation. This happens because we already try to match the size of the numbers list to the size of the cache. For larger lists, tasks can be the useful paradigm that benefits memory locality. Another consideration is that tasks can bring benefits over static scheduling when iterations are not uniform regarding execution time. However, that is not the case.

```
1   static constexpr std::uint64_t N = 2 << 25;
2   static constexpr std::uint64_t L1D_CACHE_SIZE = 32 * 1024 * 8;
3   static constexpr std::uint8_t NUM_THREADS = 12;
4
5   int main(void)
6   {
7       constexpr auto n_sqrt = static_cast<std::uint64_t>(std::sqrt(N));
8       constexpr auto segment_size = std::max(n_sqrt, L1D_CACHE_SIZE);
9       constexpr auto segment_index = []<typename T>(const T& k, const T& m) {
10          return k * k > m ? k * k - m : (2 * k) - ((m - k) % (2 * k));
11      };
12
13      std::uint64_t computed = 0;
14      std::bitset<segment_size> sieve;
15      std::bitset<(n_sqrt >> 1) + 1> is_prime;
16      is_prime.set();
17
18      for (std::uint64_t k = 3; k * k <= N; k += 2) {
19          if (!is_prime[k >> 1])
20              continue;
21
22          for (auto i = k * k; i <= n_sqrt; i += 2 * k)
23              is_prime[i >> 1] = false;
24      }
25
26      #pragma omp taskloop num_tasks(12) private(sieve) reduction(+:computed)
27      for (std::uint64_t low = 0; low <= N; low += segment_size) {
```

```
28          const auto high = std::min(N, low + segment_size - 1);
29
30          std::vector<std::pair<std::uint64_t, std::uint64_t>> seeds;
31          for (std::uint64_t k = 3; k * k < high; k += 2) {
32              if (is_prime[k >> 1])
33                  seeds.emplace_back(k, segment_index(k, low));
34          }
35
36          sieve.set();
37
38          for (auto& [seed, i] : seeds) {
39              for (const auto incr = 2 * seed; i < segment_size; i += incr)
40                  sieve[i] = false;
41          }
42
43          for (auto it = low + 1; it <= high; it += 2)
44              computed += sieve[it - low];
45      }
46
47      cpa::sieve_of_erastothenes::report<N>(computed);
48      return 0;
49  }
```

4.4 Results

The next set of plots shows the performance results of the 3 multi-core versions calculating the prime numbers until 2^{32}. These results were obtained by calculating the average between 3 distinct executions for each algorithm with different numbers of threads/tasks.

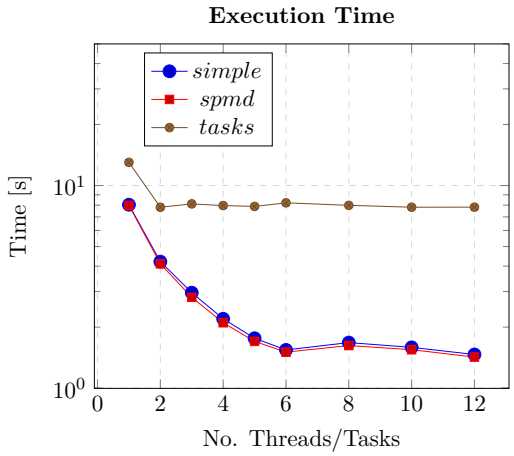
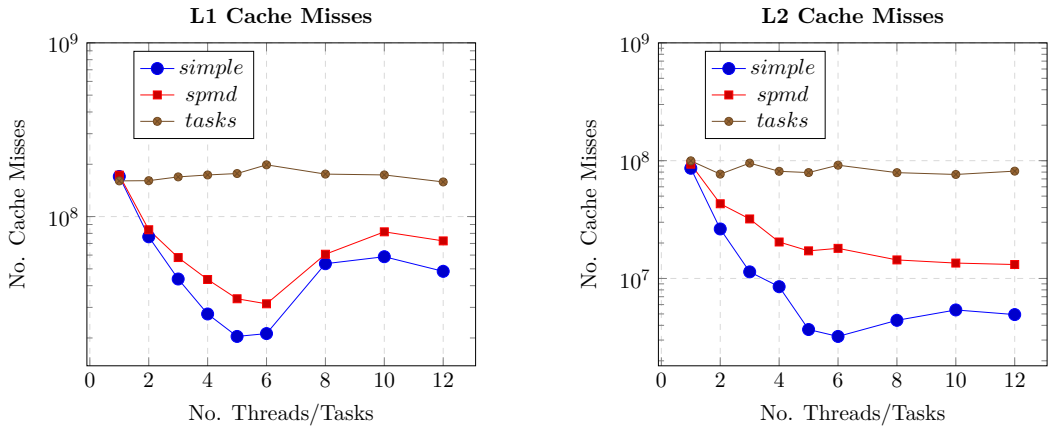


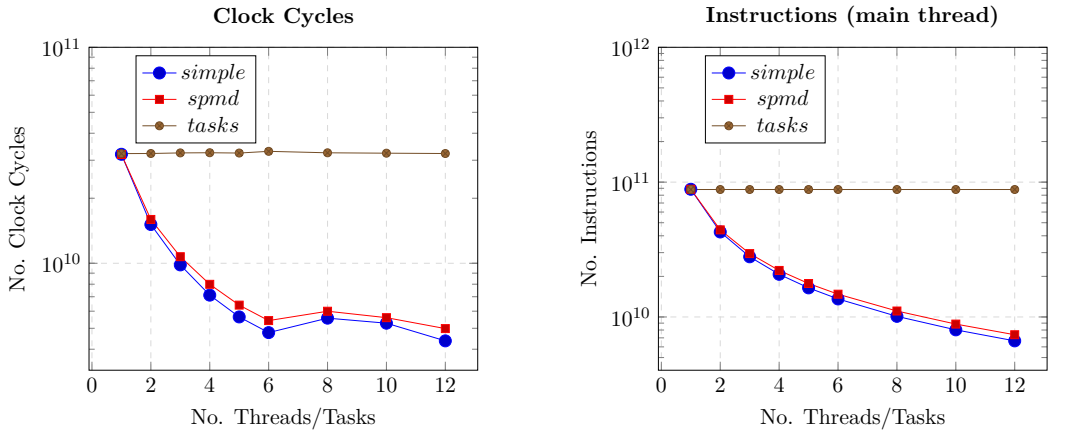
Figure 5: Execution Time for the Sieve Of Eratosthenes



(a) Cache level 1 data misses

(b) Cache level 2 data misses

Figure 6: Cache data misses during the different Sieve of Eratosthenes executions



(a) Total of clock cycles completed

(b) Total number of instructions executed

Figure 7: Absolute metrics for the different Sieve of Eratosthenes executions

5 Analysis

In this section, we perform a brief analysis and draw several conclusions over the results collected from the execution of the programs displayed in the previous section.

5.1 Sequential versions

As expected, in the sequential algorithms, a cache-wise approach and further parallelization of the algorithm significantly reduced the execution time.

Regarding the first phase of this experiment which involved the improvement and testing of various single-core approaches to the sieve of Eratosthenes, the segmented and multiples outperformed the basic version.

In *figure 1* it is possible to ascertain that both the multiples and segmented outperformed the basic version by almost 3 orders of magnitude. This can be justified by the number of L1 and L2 cache

misses during the execution, which implies that the program needed to refresh the invalid contents in the cache much more frequently. This confirms that the heuristics that regard the cache behavior, did in fact impact the performance of the sieve of Eratosthenes.

In *figure 2a*, it is noticeable that the capacity of the multiples version decreases in the higher limit tests. This can be justified by the decreased amount of instruction per second the algorithm was able to perform when finding all the prime numbers to a limit of 2^{27} or higher. The most suitable explanation for this phenomenon has to do with the fact that the list of numbers no longer fits in the L3 cache[4]. Comparing the segmented and multiples version, which had similar results, we concluded that segmented ended up being the ideal version as it showed to be more scalable, having more consistent performance and capacity along all measurements. It is also important to pinpoint the lower number of cache misses as seen in the *figure 3*, the constant space usage regarding the upper limit is vital to achieve maximum performance.

5.2 Parallel versions

Considering this, the segmented version was the one used to implement and test the parallel versions.

Regarding time execution, as seen in *figure 5*, the simple `parallel for` and SPMD versions had similar results and outperformed the Tasks version. The reason for this is the additional overhead involved with the management and distribution of tasks by OpenMP. Even though, SPMD and simple `parallel for` versions are almost equivalent, the latter ended up having fewer cache misses along with all the executions for different numbers of active threads.

By analyzing all figures it is possible to conclude that the results using 6 threads and 12 were similar. An important remark is that these 2 values were where the maximum performance was achieved. The explanation for this has to do with the fact that the machine in which executions are performed has 6 physical cores with 12 threads in hyper-threading. Other values of threads seem to introduce some kind of overhead that is, naturally, undesired.

Finally, concerning the speedup achieved, the maximum practical speedup obtained was 5.6, in the SPMD execution with 12 threads. We can see that the speedup achieved is close to the theoretical maximum speedup, which is 100, based on Amdahl's law. This theoretical value was obtained by measuring the number of operations dedicated to the sequential code and calculating its percentage, followed by the percentage of parallelizable code, which was approximately 99%. This suggests that a high portion of the program is parallelizable and that there is still much more room for improvement due to the high theoretical speedup. With more processor cores the program could have better execution times.

6 Conclusions

Throughout this project there are several key concepts that must be retained.

The fact is that modern processors follow certain heuristics that must be known by the developers who seek to extract the maximum from the hardware. Understanding how caches behave in general proved to be crucial to achieve the such considerable improvements concerning execution times. Not only that, but the multi-core paradigm versus the single-core paradigm is game changer when it comes to boost the performance while being less power hungry.

[4] 12 MB (shared) for the Intel® Core™ i7-9750H

The optimizations applied throughout the different versions are a good overview of some of techniques that are usually employed to achieve better performance and greater efficiency.

Even though putting optimizations in place on existing code bases is an expensive and demanding effort, the software industry knows that it can increase the value of a its products. Therefore, it is relatively safe to assume that the demand for faster and efficient software is greater than has ever been.

References

[1] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," 2015.

[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009* (M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, eds.), (Berlin, Heidelberg), pp. 157–173, Springer Berlin Heidelberg, 2010.

[3] C. S. Committee, "Iso international standard iso/iec 14882: 2020, programming language c++," geneva, switzerland: International organization for standardization (iso)," tech. rep., Tech. Rep., 2020, http://www. openstd. org/jtc1/sc22/wg21, 2020.