Efficient Computation of the Partial Sieve Function

Kim Walisch

June 13, 2025

Abstract

We present a comprehensive overview of the partial sieve function $\phi(x, a)$, which counts the integers up to x that are not divisible by the first a primes and underlies all combinatorial prime counting algorithms. After recalling its historical origin and basic recursive definition, we survey the principal known optimizations and discuss their applicability ranges. We then introduce a new compressed cache of $\phi(i, j)$ results for small and medium (i, j) values that improves performance by more than an order of magnitude in practice.

1 Introduction

The partial sieve function $\phi(x, a)$ counts the numbers $\leq x$ that are not divisible by any of the first a primes. This function is sometimes also named "Legendre's sum" after the French mathematician Adrien-Marie Legendre who first studied it in the 19th century [1]. The partial sieve function is at the heart of all combinatorial prime counting algorithms. In fact, Legendre's prime counting function $\pi(x) = \pi(\sqrt{x}) + \phi\big(x, \pi(\sqrt{x})\big) - 1$ can be computed using solely the partial sieve function, whereas the other more advanced combinatorial algorithms require evaluating the partial sieve function as well as other formulas.

In the advanced combinatorial prime counting algorithms such as the Lagarias–Miller–Odlyzko algorithm [3] and the Deléglise–Rivat algorithm [4], the partial sieve function is mainly used as an auxiliary function during initialization. Therefore, the overall performance of these algorithms does not significantly depend on the execution speed of the partial sieve function.

However, with the advent of multicore CPUs, the partial sieve function has found another important use case within the combinatorial prime counting algorithms. For software programs to scale well on multicore CPUs it is crucial that the individual worker threads are independent of each other, so that the threads have exclusive access to the CPU's resources and in order to prevent that a thread has to wait idle for data from another thread. In 2001 Xavier Gourdon [5] devised a modification to the hard special leaves algorithm so that the computation can be split up into independent chunks. This modification relies on the partial sieve function for generating a lookup table of $\phi(x, i)$ results for $i \in [0, a]$. The Generate $\boldsymbol{\phi(x, i)}$ Lookup Table section contains more information on this subject.

Hence, now the partial sieve function's performance has become critical for parallel implementations of the combinatorial prime counting algorithms. This paper describes the many known optimizations that can be used to speed up the $\phi(x, a)$ computation and introduces a new optimization: a compressed cache of $\phi(i, j)$ results for small and medium (i, j) values, that was first implemented in the author's `primecount` C/C++ library and which improves performance by more than an order of magnitude in practice.

2 primecount's $\phi(x, a)$ implementation

In the author's `primecount` C/C++ library the partial sieve function is implemented in the file `phi.cpp` and the file `PhiTiny.cpp` contains auxiliary functions for computing $\phi(x, a)$ in constant time for small values of $a \leq 7$. The partial sieve function $\phi(x, a)$ is also part of `primecount`'s C/C++ API and is available in the command-line application via the `--phi` option.

How to compute $\phi(1000, 10)$ using the `primecount` command-line application:

```
$ primecount 1000 10 --phi
```

3 Recursive $\phi(x, a)$ formula

$$\phi(x, a) = \phi(x, a - 1) - \phi(\lfloor x/\text{prime}_a \rfloor, a - 1)$$

This is the main formula for the computation of the partial sieve function, it was first described by Legendre in his book "Théorie des nombres" in 1830 [1]. When implemented in a computer program, the above recursive $\phi(x, a)$ formula, with a set to $a = \pi(\sqrt{x})$, allows computing Legendre's prime counting function $\pi(x) = \pi(\sqrt{x}) + \phi\big(x, \pi(\sqrt{x})\big) - 1$ in $O(x)$ operations and $O(\sqrt{x}/\log x)$ space. Tomás Oliveira e Silva's paper [6] contains a simple C implementation of this formula:

```
static void phi(int x, int a, int sign)
{
loop:
    if(a == 0)
        sum += sign * x;
    else if(x < p[a])
        sum += sign;
    else
    {
        --a;
        phi(x / p[a], a, -sign);
        goto loop;
    }
}
```

4 Known $\phi(x, a)$ optimizations

There are many known optimizations that, when combined, can speed up the $\phi(x, a)$ computation by many orders of magnitude. We briefly summarize them here.

$$\boldsymbol{\phi(x, a) = \lfloor x/pp \rfloor \cdot \varphi(pp) + \phi(x \bmod pp, a)}$$

This formula allows computing $\phi(x, a)$ in $O(1)$ for small values of a, for example $a \leq 7$. $\varphi(n)$ is Euler's totient function and pp denotes the product of the first a primes: $pp = 2 \times 3 \times ... \times \text{prime}_a$.

The use of this formula requires initializing a lookup table of $\phi(i, a)$ results for $i \in [0, pp[$, hence the lookup table has a size of pp. The German astronomer Ernst Meissel was the first who used this formula for the computation of the number of primes below 1 billion at the end of the 19th century. This formula is also present in Lehmer's paper from 1959 [2] and is described in more detail in most of the other combinatorial prime counting function papers. In `primecount` this formula is implemented in `PhiTiny.cpp`.

$$\boldsymbol{\phi(x, a)} = \begin{cases} \boldsymbol{\lfloor x/pp \rfloor \cdot \varphi(pp) + \phi(x \bmod pp, a),} & \boldsymbol{x \bmod pp \leq pp/2,} \\ \boldsymbol{\lfloor x/pp \rfloor \cdot \varphi(pp) + \varphi(pp) - \phi(pp - 1 - x \bmod pp, a),} & \boldsymbol{x \bmod pp > pp/2} \end{cases}$$

In the formulas above, pp corresponds to the product of the first a primes: $pp = 2 \times 3 \times \cdots \times \text{prime}_a$, and $\phi(n)$ is Euler's totient function. If it is not possible to compute $\phi(x, a)$ in $O(1)$ using the formula from the previous paragraph, these formulas can be used to avoid computing $\phi(x, a)$ where x may be large, and instead compute $\phi(x \bmod pp, a)$ or $\phi(pp-1-x \bmod pp, a)$ where $x \bmod pp$ and $pp - 1 - x \bmod pp$ may be orders of magnitude smaller than x. We have tested these formulas in the `primecount` C/C++ library; however, they did not provide a measurable speedup in practice. The main issue with these formulas is that they are only useful for relatively large values of x and they are limited to small values of a because they involve the product of the first a primes, which grows relatively quickly. In computer programs that use 64-bit integers these formulas can be used for $a \leq 16$. These formulas are partially described in R. P. Leopold's paper [7].

Stop recursion at c instead of 1

Using the formula $\phi(x, a) = \lfloor x/pp \rfloor \cdot \varphi(pp) + \phi(x \bmod pp, a)$ it is possible to compute $\phi(x, c)$ in $O(1)$ for small values of c, for example $c \leq 7$. Using this formula we can stop recursion at c instead of 1 in the main recursive $\phi(x, a)$ formula and simply increase the sum by $\phi(x, c)$. This optimization is generally useful. However, it only provides a minor speedup.

Calculate all $\phi(\lfloor x/\text{prime}_i \rfloor, i - 1) = 1$ upfront in $O(1)$

Once $\phi(\lfloor x/\text{prime}_i \rfloor, i - 1) = 1$ occurs in the main recursive $\phi(x, a)$ formula all subsequent $\phi(\lfloor x/\text{prime}_j \rfloor, j - 1)$ computations with $j \in]i, a]$ will also be 1. Generally $\phi(\lfloor x/\text{prime}_i \rfloor, i - 1) = 1$ if $(\lfloor x/\text{prime}_i \rfloor \leq \text{prime}_{i-1})$. Hence, instead of computing $\phi(\lfloor x/\text{prime}_j \rfloor, j - 1)$ individually for all $j \in]i, a]$, we can simply increase the sum by $a - i$. This optimization is very important as it allows to terminate recursion early in the main recursive $\phi(x, a)$ formula and thereby considerably reduce the number of operations used by the algorithm.

$$\boldsymbol{\text{if}(a \geq \pi(\sqrt{x}))} \quad \boldsymbol{\phi(x, a) = \pi(x) - a + 1}$$

This formula also allows computing $\phi(x, a)$ in $O(1)$ provided that a is relatively large and x is relatively small. If $a \geq \pi(\sqrt{x})$ then $\phi(x, a)$ counts the number of primes $\leq x$, minus the first a primes, plus the number 1. The use of this formula requires using a $\pi(x)$ lookup table of size x. In order to reduce the memory usage, it is best to use a compressed $\pi(x)$ lookup table such as `primecount`'s PiTable.hpp. The use of a lookup table makes this formula unsuitable for computing $\phi(x, a)$ for large values of x due to its excessive memory requirement. However, for large values of x we can compute the $\pi(x)$ part of this formula using a prime counting function implementation in $O(x^{\frac{2}{3}})$ or less instead of using a lookup table which uses much less memory.

$$\textbf{if}(\pi(\sqrt[3]{x}) \leq a < \pi(\sqrt{x}))\quad \phi(x, a) = \pi(x) + \mathrm{P}_2(x, a) - a + 1$$

$\mathrm{P}_2(x, a)$ corresponds to the second partial sieve function, it counts the numbers $\leq x$ that have exactly two prime factors, each exceeding the a-th prime. If $\pi(\sqrt[4]{x}) \leq a < \pi(\sqrt[3]{x})$ then one needs to add the $\mathrm{P}_3(x, a)$ term, i.e. $\phi(x, a) = \pi(x) + \mathrm{P}_2(x, a) + \mathrm{P}_3(x, a) - a + 1$. The formulas from this paragraph are not yet being used in the author's `primecount` C/C++ library, even though we expect that their use could significantly speed up some $\phi(x, a)$ computations.

5 New $\phi(x, a)$ optimization

Due to the recursive nature of the main $\phi(x, a)$ formula the same values of $\phi(i, j)$ are calculated over and over again, this is especially the case for small and medium values of (i, j). The formula $\phi(x, a) = \lfloor x/pp \rfloor \cdot \varphi(pp) + \phi(x \bmod pp, a)$ can be used to avoid recursion, however it is limited to small values of $a \leq c$ with c being a small constant, for example $c = 7$. The formula $\phi(x, a) = \pi(x) - a + 1$ can also be used to calculate $\phi(x, a)$ in $O(1)$, however it is limited to large values of $a \geq \pi(\sqrt{x})$. Hence, there is currently no known optimization for computing $\phi(x, a)$ for small and medium values of $a \in]c, \pi(\sqrt{x})[$.

The new optimization that we have devised is a $\phi(i, j)$ cache for small and medium values of (i, j), for example $i \leq \sqrt{x}$ and $j \leq 100$. The more $\phi(i, j)$ results are cached, the fewer recursive calls occur in the main $\phi(x, a)$ formula and the faster the computation runs. On the other hand we are memory constrained, we cannot cache everything, and ideally our $\phi(i, j)$ cache should fit into the CPU's fast cache memory. Hence, the main goal of our cache is to store as many $\phi(i, j)$ results as possible using as little memory as possible.

The densest data structure for storing the count of numbers $\leq n$ that are not divisible by any of the first a primes that we are aware of is a bit array. If a bit is set, this means that the corresponding number is not divisible by any of the first a primes. The 8 bits of each byte correspond to the offsets $\{1, 7, 11, 13, 17, 19, 23, 29\}$, hence each byte represents an interval of size 30. However, to determine the count of numbers $\leq n$ that are not divisible by any of the first a primes, we need to iterate over the bit array and count all set bits that correspond to numbers $\leq n$. This means that the access time of our cache would be $O(n)$, which is inefficient. Therefore we introduce a second array which contains the count of set bits in the first array $\leq$ the current index. Using these two arrays we can now count the numbers $\leq n$ that are not divisible by any of the first a primes in $O(1)$ operations. Note that the two arrays may be interleaved which makes our cache slightly more memory efficient. Below is the corresponding code from the author's `primecount` C/C++ library:

```
int64_t phi_cache(uint64_t x, uint64_t a)
{
    // We devide by 240 instead of 30 here because our
    // arrays use the uint64_t type instead of uint8_t
    uint64_t count = sieve_[a][x / 240].count;
    uint64_t bits = sieve_[a][x / 240].bits;
    uint64_t bitmask = unset_larger_[x % 240];
    return count + popcnt64(bits & bitmask);
}
```

Before the $\phi(i, j)$ cache can be used, it needs to be initialized. The cache can be initialized using a modified version of the sieve of Eratosthenes algorithm. In the `primecount` C/C++ library the cache is lazily initialized during the execution of the main recursive $\phi(x, a)$ formula. Whenever a new $\phi(x, i)$ calculation is started we first check whether that result is not yet present in the cache and if (x, i) meet the caching criteria. If these conditions apply the cache will be filled up to (x, i), so that all values $\leq (x, i)$ will be cached. In the first part of this algorithm we unset the bits that correspond to numbers that are divisible by the i-th prime. When sieving has finished, we proceed to the second part of the algorithm where we count all set bits (below each index) in the first array and store that count in the second array. Below is the corresponding code from the `phi.cpp` file of the `primecount` C/C++ library:

```
// Cache phi(x, i) results with: x <= max_x && i <= min(a, max_a).
// Eratosthenes-like sieving algorithm that removes the first a primes
// and their multiples from the sieve array. Additionally this
// algorithm counts the numbers that are not divisible by any of the
// first a primes after sieving has completed. After sieving and
// counting has finished phi(x, a) results can be retrieved from the
// cache in O(1) using the phi_cache(x, a) function.
//
void init_cache(uint64_t x, uint64_t a)
{
    // Each bit in the sieve array corresponds to an integer that
    // is not divisible by 2, 3 and 5. The 8 bits of each byte
    // correspond to the offsets { 1, 7, 11, 13, 17, 19, 23, 29 }.
    sieve_[3].resize(max_x_size_, sieve_t{0, ~0ull});

    for (uint64_t i = 4; i <= a; i++)
    {
        // Initalize phi(x, i) with phi(x, i - 1)
        sieve_[i] = sieve_[i - 1];

        // Remove prime[i] and its multiples
        for (uint64_t n = primes_[i]; n <= max_x_; n += primes_[i] * 2)
            sieve_[i][n / 240].bits &= unset_bit_[n % 240];

        // Fill an array with the cumulative 1 bit counts.
        // sieve[i][j] contains the count of numbers < j * 240 that
        // are not divisible by any of the first i primes.
        uint64_t count = 0;
        for (auto& sieve : sieve_[i])
        {
            sieve.count = count;
            count += popcnt64(sieve.bits);
        }
    }
}
```

According to our benchmarks, the cache as implemented above speeds up `primecount`'s $\phi(x, a)$ implementation by more than an order of magnitude. Based on our empirical tests, caching $\phi(x, a)$ results for $a \leq 100$ provides the best performance. As mentioned earlier, smaller values of (x, a) are accessed much more frequently than larger values. Hence, caching $\phi(x, a)$ results with $a > 100$ leads to diminishing returns and often increases CPU cache misses, which deteriorates performance. We also limit the size of the cache to around 16 megabytes in the `primecount` C/C++ library, which corresponds roughly to the size of the L3 cache in many of today's CPUs.

6 Generate $\phi(x, i)$ Lookup Table

In 2001 Xavier Gourdon devised a modification to the hard special leaves algorithm so that the computation can be split up into independent chunks. This modification is particularly useful for parallelizing the hard special leaves algorithm, since it avoids the need for frequent thread synchronization. This modification relies on the partial sieve function for generating a lookup table of $\phi(x, i)$ results for $i \in [0, a]$. The idea of the algorithm is described very briefly at the end of Gourdon's paper [5] and it is also described in some more detail in Douglas Staple's paper [8], however no pseudocode is provided in either paper.

Computing $\phi(x, i)$ individually for all $i \in [0, a]$ would be far too slow. However, by taking advantage of the recursive nature of the main formula $\phi(x, a) = \phi(x, a-1) - \phi(\lfloor x/\text{prime}_a \rfloor, a-1)$, we can actually generate a lookup table of $\phi(x, i)$ results for $i \in [0, a]$ in the same asymptotic time as computing just $\phi(x, a)$. We first compute $\phi(x, 0)$, next we compute $\phi(x, 1)$ and reuse the $\phi(x, 0)$ result we have computed previously. Then we compute $\phi(x, 2)$ and reuse our previous $\phi(x, 1)$ result and so forth. The code below shows how this algorithm can be implemented using a simple for loop. The `phi_recursive(x / primes[i], i - 1)` part needs to be computed using the main recursive $\phi(x, a)$ formula in combination with the optimizations described in this paper.

```
int* phi = new int[a + 1];
phi[0] = x;

// Fill an array with phi(x, i) results
for (int i = 1; i <= a; i++)
    phi[i] = phi[i-1] - phi_recursive(x / primes[i], i - 1);
```

References

[1] A. M. Legendre, *Théorie des nombres*, 3rd ed., Paris, 1830.

[2] D. H. Lehmer, *On the exact number of primes less than a given limit*, Illinois J. Math., 3 (1959), pp. 381–388.

[3] J. C. Lagarias, V. S. Miller, and A. M. Odlyzko, *Computing $\pi(x)$: The Meissel–Lehmer method*, Math. Comp., 44 (1985), pp. 537–560.

[4] M. Deléglise and J. Rivat, *Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko Method*, Math. Comp., 65 (1996), pp. 235–245.

[5] X. Gourdon, *Computation of $\pi(x)$: Improvements to the Meissel, Lehmer, Lagarias, Miller, Odlyzko, Deléglise and Rivat method*, Feb. 15, 2001.

[6] T. Oliveira e Silva, *Computing $\pi(x)$: The combinatorial method*, Revista do DETUA, 4(6) (2006), pp. 759–768.

[7] R. P. Leopold, *On Methods for Computing $\pi(x)$*, 2007.

[8] D. B. Staple, *The combinatorial algorithm for computing pi(x)*, Master of Science Thesis, Dalhousie University Halifax, Nova Scotia, August 2015.

[9] K. Walisch, *primecount: Fast C/C++ prime counting function library*, Version 7.19, 2025. Available at: https://github.com/kimwalisch/primecount