Efficient computation of the easy special leaves in the combinatorial prime counting algorithms

Kim Walisch

June 6, 2025

Abstract

We present practical enhancements for computing the easy special leaves, a key set of formulas used in many combinatorial prime counting algorithms, including the algorithms of Lagarias–Miller–Odlyzko, Deléglise–Rivat, and Gourdon. We describe how to improve the cache efficiency of the algorithm and how to parallelize the algorithm so that it scales well on modern multicore processors. Both these improvements have been implemented in the author's `primecount` C/C++ library, resulting in a more than twofold increase in execution speed when running on a dual-socket HPC server with 192 CPU cores.

1 Introduction

In the combinatorial prime counting algorithms, the computation of the special leaves is the computationally most expensive task. To speed up that computation, Lagarias-Miller-Odlyzko [1] have split the special leaves into *easy* special leaves and *hard* special leaves. The contribution of each easy special leaf can be computed in $O(1)$ using a $\pi(x)$ lookup table, whereas the contribution of each hard special leaf requires evaluating the partial sieve function $\phi(x, a)$ and therefore cannot be computed in $O(1)$.

In the Deléglise-Rivat [2] and Gourdon [3] prime counting algorithms (which are based on the Lagarias–Miller–Odlyzko algorithm), the computation of the easy special leaves requires looking up the number of primes $\leq n$ with $n < \sqrt{x}$. Since a `PrimePi[n]` lookup table of size $\sqrt{x}$ is much too large to be practical, Deléglise-Rivat [2] have suggested segmenting the interval $[0, \sqrt{x}[$ using a segment size of y ($\sim \sqrt[3]{x} \log^3 x$). Hence, instead of using a `PrimePi[n]` lookup table of size $\sqrt{x}$, one uses a `SegmentedPrimePi[n]` lookup table of size y, which also returns the number of primes $\leq n$ but requires that n lie within the current segment $[low, low + y[$. This approach was used in the author's `primecount` C/C++ library up to version 6.4. However, this segment size causes severe scaling issues for large computations above 10^{22} as the `SegmentedPrimePi[n]` lookup table becomes exceedingly large, for example, at 10^{30} its size was 137 GiB in `primecount`. For this reason, Gourdon [3] suggested using a smaller segment size of $\sqrt{x/y}$, which is orders of magnitude smaller and generally a good practical improvement.

Here are links to `primecount`'s `PiTable` and `SegmentedPiTable` implementations.

2 Improving the Cache Efficiency

The `SegmentedPrimePi[n]` lookup table is accessed very frequently in the computation of the easy special leaves (about once for each leaf) and these memory accesses are non-sequential. Therefore, it is important that the `SegmentedPrimePi[n]` lookup table fits into the CPU's fast cache memory. Although Gourdon's smaller segment size already reduces memory pressure substantially, it is still too large for new record computations. For this reason, we suggest using an even smaller segment size of $\sqrt[4]{x}$ for the computation of the easy special leaves. With a segment size of $\sqrt[4]{x}$, the `SegmentedPrimePi[n]` lookup table fits into the CPU's cache even for record computations; for example, at 10^{30} the `SegmentedPrimePi[n]` lookup table is only approximately 2 MiB in `primecount`. A segment size of $\sqrt[4]{x}$ does not deteriorate the runtime complexity of the algorithm because the segmented sieve of Eratosthenes, which is used to initialize the `SegmentedPrimePi[n]` lookup table, has the same runtime complexity as the unsegmented sieve of Eratosthenes, provided that the segment size is not smaller than the square root of the total sieving distance. In our case, the total sieving distance is $\sqrt{x}$, so the required segment size is at least $\sqrt{\sqrt{x}} = \sqrt[4]{x}$.

Note that Deléglise-Rivat [2] split the easy special leaves into many formulas and suggest using segmentation only for the two formulas that require looking up the number of primes $< \sqrt{x}$. All other formulas (which only need to look up the number of primes $\leq y$) should be computed without segmentation. However, since a `PrimePi[n]` lookup table of size y is much too large to fit into the CPU's cache, and since `PrimePi[n]` is accessed in random order, we recommend segmenting all easy special leaf formulas that are computationally expensive using a segment size of $\sqrt[4]{x}$ in order to improve performance. To minimize programmer effort, it is best to sieve the interval $[0, \sqrt{x}[$ only once and compute all easy special leaf formulas within that same sieve.

Extra care needs to be used when segmenting the formulas that compute consecutive identical easy leaves more efficiently, sometimes these leaves are called *clustered easy leaves* [4]. In the Deléglise–Rivat algorithm, the W_3 and W_5 formulas compute clustered easy leaves. These formulas need to access `PrimePi[n]` values with $n \leq y$, but some of these accesses (specifically, those that compute how many consecutive leaves are identical) may fall outside of the current segment $[low,\ low + segment\ size[$. For those latter accesses, we suggest using a `PrimePi[n]` lookup table of size y instead of the `SegmentedPrimePi[n]` lookup table. Note that it is still important for performance to segment the clustered easy leaves, since there is a proportionally large number of them and their computation is relatively expensive.

3 Parallel Computation and Load-Balancing

So far, we have focused on improving the cache efficiency of the computation of the easy special leaves. Now we will have a look at how to parallelize the computation of the easy special leaves so that the algorithm scales well on modern multicore CPUs. In general, an algorithm scales well on multicore CPUs if it satisfies the following three properties:

- Each thread operates only on its own small chunk of memory that fits into the private L1 and/or L2 cache of the corresponding CPU core.
- All threads are independent of each other (i.e. they require no synchronization).
- The work is evenly distributed among all threads in order to avoid load imbalance.

A tiny segment size of $\sqrt[4]{x}$ already satisfies the first property. To achieve thread independence, we have borrowed an idea from Gourdon [3]: Each thread is assigned a different segment, which it processes exclusively. At the start of each new segment $[low,\ low + segment\ size[$ each thread computes $\pi(low)$ using a prime counting function implementation in $O(low^{\frac{2}{3}})$ or better. The result of

$\pi(low)$ is required to initialize that thread's own `SegmentedPrimePi[n]` lookup table for the current segment. With this extra $\pi(low)$ initialization step, all threads become independent of each other. This parallel algorithm was first implemented in `primecount-7.0` (see `SegmentedPiTable.cpp` and `AC.cpp`). It improved performance by more than a factor of two for $x = 10^{23}$ on the author's dual-socket AMD EPYC server with 192 CPU cores compared to `primecount-6.4`, which used a larger segment size and required frequent thread synchronization. It is important to ensure that the additional pre-computations do not deteriorate the overall runtime complexity of the algorithm. When sieving up to $\sqrt{x}$ with a segment size of $\sqrt[4]{x}$, there are exactly $\sqrt[4]{x}$ segments. For each segment, we must compute $\pi(low)$ with $low < \sqrt{x}$. Hence, in total these extra $\pi(low)$ computations have a runtime complexity of less than $O(\sqrt{x}^{\frac{2}{3}} \cdot \sqrt[4]{x}) = O(x^{\frac{7}{12}})$ which does not worsen the overall runtime complexity of the algorithm.

Lastly, we must ensure that the work is distributed evenly among all threads. The easy special leaves are distributed very unevenly: most of the leaves are located below y ($\sim \sqrt[3]{x}\log^3 x$) whereas above y the number of leaves slowly decreases and they become more and more sparse as they approach $\sqrt{x}$. Therefore, it is essential that the region below y is evenly distributed among the threads. Empirically, a tiny segment size of $\sqrt[4]{x}$ achieves near-perfect balance even on servers with many CPU cores (for example, the author's dual-socket AMD EPYC server with 192 CPU cores). Using a larger segment size, such as $\sqrt[3]{x}$ or y, leads to significant load imbalance (i.e. some threads receive much more work than others and continue computing long after most threads have finished), which severely degrades performance, especially on systems with a large number of CPU cores. Above y, there are far fewer easy special leaves, so one can gradually increase the number of segments per thread to reduce the $\pi(low)$ pre-computation overhead.

References

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

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

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

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