

Efficient Computation of the Easy Special Leaves in the Combinatorial Prime Counting Algorithms

Kim Walisch

July 26, 2026

Abstract

We present practical improvements to the computation of easy special leaves in combinatorial prime counting algorithms, including the Deléglise–Rivat and Gourdon algorithms. The methods improve cache efficiency, parallel scalability, load balancing, and the instruction-level parallelism of clustered easy leaves. We also apply Gourdon’s inversion equality a second time to transform and merge a remaining clustered contribution in Gourdon’s C formula into the existing sparse loop, without changing the asymptotic complexity. All improvements are implemented in the `primecount` C/C++ library. Benchmarks on modern multicore processors show substantial individual reductions in running time and executed instructions.

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 with $x \geq 10^{22}$, as the `SegmentedPrimePi[n]` lookup table becomes exceedingly large; for example, at $x = 10^{30}$ 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 the segment size of $\sqrt{x/y}$ suggested by Gourdon [3] is already considerably smaller than the segment size of y suggested by Deléglise-Rivat [2], it still uses too much memory 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 $x = 10^{30}$ the `SegmentedPrimePi[n]` lookup table is only approximately 2 MiB in `primesieve`. 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 $[low, low + \text{segment size}[$ 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.

4 Improving the Instruction-Level Parallelism of the Clustered Easy Leaves

The main advantage of the Deléglise-Rivat algorithm [2] over the Lagarias-Miller-Odlyzko algorithm [1] is its efficient treatment of the clustered easy leaves. Every easy special leaf is described by a pair of primes p and q , and for a fixed p the quantity $\pi(x/(pq))$ is a non-increasing step function of q . Deléglise-Rivat split the range of q at $\sqrt{x/p}$. Below that point consecutive leaves are almost always distinct, and each leaf must be evaluated on its own; these are the *sparse* easy leaves. Above it, i.e. for

$$q \in \left] \sqrt{x/p}, \min(x/p^2, y) \right],$$

we have $x/(pq) < \sqrt{x/p}$, so that long runs of consecutive primes q share one and the same value of $\pi(x/(pq))$. These are the *clustered* easy leaves [4], and they correspond to the W_3 and W_5 formulas of Deléglise-Rivat [2]. Rather than evaluating each of them individually, one determines how many consecutive leaves are identical and adds the contribution of the whole run in a single step. It is this mechanism that brings the running time of the Deléglise-Rivat algorithm down to $O(x^{2/3} \log^{-2} x)$, a factor of $\log x$ better than the $O(x^{2/3} \log^{-1} x)$ of the Lagarias-Miller-Odlyzko algorithm, at the cost of a comparable increase in memory usage [4].

Despite these theoretical benefits, the clustered easy leaves algorithm has a significant drawback on real hardware: it suffers from long instruction dependency chains, which limit instruction-level parallelism (ILP). The algorithm is implemented as a single loop that walks through the primes q one run at a time, and the index at which the next run begins can only be derived from the index of the current one, through a chain of two integer divisions and four dependent table lookups. Computing the sparse easy leaves individually, by contrast, has very good ILP, since consecutive leaves are independent of each other. The natural way to recover ILP would be to process several independent streams of prime pairs $(p_1, q_1), (p_2, q_2), \dots$ at the same time, but this is difficult to implement in practice, because the number of iterations is not the same for every stream.

A far simpler strategy achieves most of the benefit. For a single prime p , the range of q is traversed from both ends at once, with one descending and one ascending stream, until the two meet in the middle. The two streams are independent by construction, so the processor can overlap their dependency chains completely.

```
while (ilo <= ihi)
{
    // Descending stream
    int64_t xpq_hi = x / (p * primes[ihi]);
    int64_t pi_xpq_hi = pi[xpq_hi];
    int64_t phi_hi = pi_xpq_hi - b + 2;
    int64_t next_hi = x / (p * primes[pi_xpq_hi + 1]);
    int64_t ihi_new = max(pi[next_hi], ilo - 1);
    sum += phi_hi * (ihi - ihi_new);
    ihi = ihi_new;

    if (ilo > ihi)
        break;

    // Ascending stream
    int64_t xpq_lo = x / (p * primes[ilo]);
    int64_t pi_xpq_lo = pi[xpq_lo];
    int64_t phi_lo = pi_xpq_lo - b + 2;
    int64_t next_lo = x / (p * primes[pi_xpq_lo]);
    int64_t ilo_new = pi[min(next_lo, primes[ihi])] + 1;
    sum += phi_lo * (ilo_new - ilo);
    ilo = ilo_new;
}
```

This improvement was implemented in `primecount-8.3` (see [S2.easy.cpp](#)), where it improved the performance of the easy special leaves algorithm by a factor of up to 1.7 on an Apple Silicon M2 CPU from 2022.

In `primecount-8.3`, the author also implemented a second mechanism (see [AC.cpp](#)) for increasing instruction-level parallelism: We measure the average number of consecutive identical easy special leaves, and once this average has fallen below a certain threshold (6 in `primecount`'s implementation), we disable the clustered easy leaves algorithm for the rest of the segment and compute the remaining easy leaves individually. This way the clustered easy leaves algorithm is used while it is beneficial, i.e. while there are many consecutive identical easy special leaves, and disabled once it becomes inefficient, i.e. once there are only very few consecutive identical easy special leaves.

5 Gourdon's Inversion Equality

The two mitigations of the previous section improve the performance of the clustered easy leaves considerably, but they cannot remove the underlying problem: the dependency between consecutive loop iterations, which limits instruction-level parallelism. It would therefore be preferable to do away with the clustered easy leaves altogether. Simply computing them one at a time, using the algorithm employed for the sparse easy leaves, is not an option: this would restore the instruction-level parallelism but deteriorate the runtime complexity, since it is precisely the clustering that saves the factor of $\log x$ discussed above.

Gourdon [3] found a way to achieve this. The starting point is the inversion equality

$$\sum_{\alpha < q \leq \beta} \pi\left(\frac{z}{q}\right) = \pi(\beta) \pi\left(\frac{z}{\beta}\right) - \pi(\alpha) \pi\left(\frac{z}{\alpha}\right) + \sum_{\frac{z}{\beta} < q \leq \frac{z}{\alpha}} \pi\left(\frac{z}{q}\right), \quad (1)$$

where q runs over the primes and $0 < \alpha \leq \beta$. It is the hyperbola method applied to the pairs of primes (q, r) with $qr \leq z$: those with $r \leq z/\beta$ satisfy $qr \leq z$ for every q in the range and hence form a rectangle that is counted in closed form, while the remaining pairs are enumerated by r rather than by q . Applying the equality with $z = x/p$, $\alpha = \sqrt{x/p}$ and $\beta = \min(x/p^2, y)$, that is to exactly the range of clustered easy leaves of the previous section, replaces the inner sum of the W_3 and W_5 formulas by two products of π values, which cost a handful of table lookups, plus a sum over

$$q \in \left] \frac{x}{p\beta}, \sqrt{x/p} \right].$$

This range lies entirely below $\sqrt{x/p}$, i.e. inside the sparse region, so its leaves are again independent of each other and can be computed with the same loop, and the same good instruction-level parallelism, as the sparse easy leaves. Most of the leaves of the W_3 and W_5 formulas are thereby never touched at all. Nothing is lost in return, because the new sum has $\pi(\sqrt{x/p}) - \pi(x/(p\beta)) \leq \pi(\sqrt{x/p})$ terms, which is precisely the bound that Deléglise-Rivat give for the number of iterations of the clustered easy leaves algorithm [2]. The runtime complexity of the easy special leaves is therefore unchanged.

6 A Second Application of Gourdon's Inversion Equality

Gourdon applies (1) to both V_1 and V_2 , thereby avoiding the explicit computation of the clustered easy leaves corresponding to the W_3 and W_5 terms of the Deléglise-Rivat algorithm. Although Gourdon mentions only W_3 explicitly, his inversion of V_2 treats W_5 in the same way. The resulting reflected sums are merged with the sparse easy-leaf terms in Gourdon's A formula, so the clustered easy leaves do not require a separate computation. This is the main reason Gourdon's algorithm can outperform Deléglise-Rivat in practice; avoiding the clustered easy leaves algorithm's long dependency chains provides a secondary instruction-level-parallelism benefit, while the asymptotic complexity remains unchanged.

However, clustered easy leaves remain in the part of Gourdon's C formula with $\sqrt{z} < p \leq x^*$. In this range the easy special leaves correspond to pairs of primes p and q . For fixed p , put $t = x/p$

and define

$$q_{\text{low}} = \max\left(\sqrt{t}, \frac{t}{p^2}\right).$$

The clustered easy leaves in this part of C satisfy $q_{\text{low}} < q \leq y$. This interval can be nonempty only if $y > x^{3/8}$. Oliveira e Silva showed that the asymptotically optimal order is $y = \Theta(x^{1/3} \log^3 x)$ [4], so the interval eventually disappears. Under primecount's tuning model, however, it remains present at least up to $x = 10^{100}$. Gourdon's published C formula evaluates these terms individually.

The implementation in `primecount-8.6` instead uses the clustered easy leaves algorithm presented in Section 4, which can reduce their computation time by a constant factor. A further improvement, first implemented in `primecount-8.7` (see [AC.cpp](#)), is to apply Gourdon's inversion equality (1) once more, with t in place of z , $\alpha = q_{\text{low}}$, and $\beta = y$. When $q_{\text{low}} < y$, this gives

$$\begin{aligned} \sum_{q_{\text{low}} < q \leq y} \left(\pi\left(\frac{t}{q}\right) - \pi(p) + 2 \right) &= \pi\left(\frac{t}{y}\right) \pi(y) - \pi\left(\frac{t}{q_{\text{low}}}\right) \pi(q_{\text{low}}) \\ &+ \sum_{t/y < r \leq t/q_{\text{low}}} \pi\left(\frac{t}{r}\right) - (\pi(p) - 2)(\pi(y) - \pi(q_{\text{low}})), \end{aligned}$$

where q and r run over the primes; the interval of the r -sum is the reflected range.

For fixed p , the ordinary sparse loop processes the primes

$$\max\left(\frac{t}{p^2}, p\right) < r \leq \sqrt{t},$$

whereas the reflected sum runs over

$$\frac{t}{y} < r \leq \frac{t}{q_{\text{low}}}.$$

The reflected interval is contained in the sparse interval. Indeed, $p > \sqrt{z}$ and $y \leq z$ give $p^2 > y$, and hence $t/p^2 < t/y$. Moreover, $p \leq x^* \leq \sqrt{x/y}$ implies $p \leq t/y$. Thus the lower endpoint t/y of the reflected interval is not smaller than either lower bound of the sparse interval. At the other end, $q_{\text{low}} \geq \sqrt{t}$ implies $t/q_{\text{low}} \leq \sqrt{t}$. Therefore every prime in the reflected range is also processed as an ordinary sparse easy leaf.

For each such prime r , the ordinary contribution $\pi(t/r) - \pi(p) + 2$ and the reflected contribution $\pi(t/r)$ are evaluated together as $2\pi(t/r) - \pi(p) + 2$. The algorithm therefore adds the boundary correction once for each p whose clustered interval is nonempty and evaluates the sparse range with a single loop. Each reflected term reuses the division and π lookup already required for the corresponding sparse easy leaf, eliminating a separate pass over the reflected range.

For the benchmarks, *Original* denotes Gourdon's original C algorithm, *Clustered* the clustered easy leaves implementation of `primecount-8.6` described in Section 4, and *Reflected* the `primecount-8.7` algorithm above, which merges the reflected terms into the sparse loop. We disabled the A contribution in three otherwise identical binaries so that only C was computed. The benchmarks used all available cores of an Intel Core Ultra 245K and an Apple Silicon M2. The first three timing columns report elapsed seconds, and the last two give the speedup of *Reflected* over *Original* and *Clustered*.

Intel Core Ultra 245K

x	Original	Clustered	Reflected	vs. Original	vs. Clustered
10^{17}	0.075	0.076	0.048	$1.56\times$	$1.58\times$
10^{18}	0.298	0.298	0.171	$1.74\times$	$1.74\times$
10^{19}	1.389	1.403	0.760	$1.83\times$	$1.85\times$
10^{20}	6.161	6.115	3.080	$2.00\times$	$1.99\times$
10^{21}	26.728	26.269	12.983	$2.06\times$	$2.02\times$
10^{22}	121.397	118.915	54.523	$2.23\times$	$2.18\times$

Apple Silicon M2

x	Original	Clustered	Reflected	vs. Original	vs. Clustered
10^{17}	0.191	0.195	0.128	$1.49\times$	$1.52\times$
10^{18}	0.760	0.776	0.454	$1.67\times$	$1.71\times$
10^{19}	3.287	3.398	1.787	$1.84\times$	$1.90\times$
10^{20}	15.265	15.673	7.639	$2.00\times$	$2.05\times$
10^{21}	72.948	74.088	34.761	$2.10\times$	$2.13\times$
10^{22}	360.667	352.961	166.754	$2.16\times$	$2.12\times$

The *Clustered* implementation provides little or no improvement over the *Original C* algorithm in these benchmarks, whereas the *Reflected* algorithm is substantially faster than both. On both processors, its speedup grows from about $1.5\times$ at $x = 10^{17}$ to more than $2.1\times$ at $x = 10^{22}$. This consistent behavior across x86-64 and ARM64 indicates that the improvement is not specific to one processor architecture. None of the three implementations changes Gourdon’s asymptotic complexity.

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.
- [5] K. Walisch, *primecount: Fast C/C++ prime counting function library*, Version 8.3, 2025. Available at: <https://github.com/kimwalisch/primecount>