Static Wheels in Fast Sieves

Mircea GHIDARCEA * **Decebal POPESCU** **

* *University POLITEHNICA of Bucharest - Computer Science Department, Splaiul Independentei 313, Romania (e-mail: mircea.ghidarcea@stud.acs.upb.ro).*
** *University POLITEHNICA of Bucharest - Computer Science Department, Splaiul Independentei 313, Romania (e-mail: decebal.popescu@upb.ro).*

Abstract: Prime numbers sieving is not anymore a topic of interest for the IT specialists and, besides some commendable open source efforts, practically nobody tries to advance the domain (Ghidarcea and Popescu, 2023) - despite all hardware progress, even today we are not capable to generate all primes up to 2^{64} in practical time .
This article deals with fast sieves based on the classic Sieve of Eratosthenes, introducing an innovative approach for using the static wheels that has the potential to surpass the fastest algorithms known so far. It then tries to generalize the new method and demonstrates its real potential by benchmarking it against the best performing sieve implementation we are ware of, the ultra-refined **Primesieve**, showing it can outperform it.

Keywords: Prime numbers sieving, Prime number generation, Algorithms, Algorithm optimization, Parallel algorithms.

1. INTRODUCTION

Sieve of Eratosthenes (SoE), presumably dating from the 3rd century BCE (Nicomachus, 1926), is still the most popular algorithm to generate all prime numbers in a contiguous set. As shown in (Crandall, Richard and Pomerance, Carl, 2005), the complexity of SoE is:

$$C(SoE) = O(N \ln(\ln(N)))$$

This is not a stellar complexity – there are algorithms which are linear, like (Mairson, 1977), (Pratt, 1977) or (Gries and Misra, 1978), and even sub-linear ones, like (Atkin and Bernstein, 2003). Yet, (Sorenson, 1991) shows why Sieve of Eratosthenes is still basically superior in practice to all other theoretically (sub)linear algorithms.

In this context, a lot of tricks are employed to reduce the huge number of computations involved by sieving for very large sets, and the Static Wheels technique is perhaps the best known such strategy. The basic idea underlying Static Wheels techniques is to eliminate from the sieving process set as many operations as possible before the sieving occurs:

- eliminate a large number of candidates as represented in the target set by eliminating entire classes of composites (i.e. eliminating even numbers or some other arithmetical pattern proved to generate composites, like 6k+[0,2,3,4]);
- in sieving, step over multiples of primes that are not in the target set or are known to be eliminated in previous sweeps by using certain patterns to generate the eligible candidates.

The origins of the Static Wheels technique as applied in prime sieving date very early, appearing with the first modern implementations of sieving algorithms: even the first sieving algorithm ever published, T.C.Wood's (Wood, 1961) and Chartres' 310 (Chartres, 1967), skipped over even numbers. Chartres' 311 (Charters, 1967) skips over multiples of 2 and 3 using a very smart and concise technique, thus implementing the $6k \pm 1$ pattern, and only two years later Singleton introduces the idea for the $30k+i$ pattern with his **357** article (Singleton, 1969b). It is easy to recognise these patterns as the first three wheels:

- $W(1) \Rightarrow 2k + i$, where $i \in \{1\}$
- $W(2) \Rightarrow 6k + i$, where $i \in \{1, 5\}$
- $W(3) \Rightarrow 30k+i$, where $i \in \{1, 7, 11, 13, 17, 19, 23, 29\}$.

The next big thing in wheels appeared in 1981 in a paper from Paul Pritchard, introducing probably the most beautiful prime sieving algorithm - Sieve of Pritchard (Pritchard, 1981). The algorithm was further clarified by the author in (Pritchard, 1982) with a detailed and intuitive explanation regarding how dynamic wheels work, thus throwing more light on the static wheels in general.

(Pritchard, 1983) is another meaningful and influential work, being the one that definitively formalizes the so called *static* or *fixed wheel* and proves the linear character of the sieve when a static wheel of order $k = max(i|P(i) \leqslant \sqrt{N})$ is used (where $P(i)$ is the Primorial for i, i.e. the product of the first i prime numbers). The results are somewhat refined in (Sorenson, 1990), but especially in (Sorenson, 1998) and (Dunten et al., 1996).

Yet, the dynamic wheels are now forgotten and the static wheels are only used in the form of W(3), which is practically always hard-coded in the modern Eratosthenes based fast sieve algorithms and is considered to be the optimal compromise between the performance advantages

Table 1. Wheels parameters

N	π(N)	Wheel values	Reduction factor	Maximum gap
1	2	1	2.00	2
2	6	2	3.00	4
3	30	8	3.75	6
4	210	48	4.38	10
5	2'310	480	4.81	14
6	30'303	5'760	5.21	22
7	510'510	92'160	5.54	26
8	9'699'690	1'658'880	5.84	34
9	223'092'870	36'495'360	6.11	40
10	6'469'693'230	1'021'870'080	6.36	46
11	200'560'490'130	30'656'102'400	6.54	474

brought by a wheel versus the overhead inflicted to the algorithms.

***NOTE 1**: Given that probably the most important characteristic of a sieve implementation is performance, the most appropriate language for such implementation is arguably C/C++ - thus, most of the algorithms described next were fully implemented in C++. All the timings for this paper were measured on a desktop computer with AMD processor Ryzen 9 7900X, Auto OC on.*

***NOTE 2**: All accompanying code used for the experiments and proof-of-concept implementations in this article can be consulted on* **GitHub** *at* `https://github.com/mirceag70/StaticWheels`.

2. STATIC WHEELS FUNDAMENTALS

In Table 1 we show the main parameters for the first eleven wheels. We can draw two main conclusions from these values:

- The reduction factor grows less and less with N, with diminished returns for bigger Ns; anyway, after $N = 10$ the number of wheel values is extremely large: due to the added overhead for diminished returns, it does not make much sense to even consider wheels greater than $N = 10$
- As the wheel is always parsed in order and gaps between consecutive values are lower than 255 for any practical wheel, we can represent the wheel only as increments in a vector of bytes. (*The gap is always even, so for N <= 7 one could keep it halved and compress two values per byte if it makes sense for some particular scenario*)

When we think about wheel generation, the first idea that comes to mind is some kind of brute force approach: the wheels seem quite small and simple, so we assume that there is not much work there - yet, for large wheels we are deeply mistaken, as we can see in Table 2 where we show the timings for single-threaded wheel generation.

Another intuitive alternative would be an optimized Sieve of Pritchard (SoP), which in theory is devised for fast wheel generation. We also added in the mix a classic sieving algorithm based on the *1bit 6k* technique we devised for optimized Mairson. As we can see, for the sensible range of wheels the *1bit 6k* algorithm is the fastest one - our optimized Sieve of Pritchard may take over at $N > 10$, but at that point we are out of the practical interval for wheels. Anyway, even past $N = 10$, a parallelized version of *1bit 6k* would be faster than Sieve of Pritchard which is very tough to parallelize. The timings for the parallel *1bit 6k* procedure are added in the table for reference. You can check the accompanying code for further details.

Table 2. Wheel generation times [ms]

N	Brute force	SoP	1bit 6k	1b6k parallel
6	1	0	0	-
7	1	1	1	-
8	15	13	12	4
9	369	318	270	31
10	14'137	9'013	7'779	805

Devising a basic algorithm that exploits a static wheel is pretty straightforward: while parsing for each the prime p all the sieving composite values `n = delta * p;`, we are using using only the revolving wheel values for the multiplier `delta` (the wheel values storage as increments comes in handy here).

The only tricky thing to do is to get the first multiple wheel value that is greater than p^2 as fast as possible, but, at least for the first segment, as long as we are using the wheel to generate the primes we are always in the right spot, as we can see in the accompanying code.

For the general case, as long as we iterate primes in order, the next wheel index multiplier for the next prime will always be one of the next indexes from the last one, thus some code gymnastics can speed up this task - see accompanying code for details. In Table 3 we have timings for the basic single threaded wheel based algorithm. The sweet spots are highlighted accordingly:

Table 3. Sieving times [ms] (incl. wheel generation)

Wheel	Limit (10^n)						
	5	*6*	*7*	*8*	*9*	*10*	*11*
W(2)*	0	1	10	91	1'400	20'949	239'183
W(3)	0	1	10	88	1'300	19'878	229'772
W(4)	0	1	9	82	1'209	17'674	200'644
W(5)	0	1	8	75	1'123	16'531	186'066
W(6)	1	2	8	72	1'072	15'665	175'578
W(7)	1	2	8	70	1'044	15'072	167'965
W(8)**	-	-	10	69	1'020	14'400	161'123
W(9)	-	-	37	95	1'029	14'006	156'829
W(10)	-	-	889	931	1'869	14'571	153'892

*For $N = 2$ we used the faster 1bit 6k algorithms which hard-codes W(2)
**For $N >= 8$ we used parallel wheel generation.

For smaller ranges it is visible how for $N > 7$, even with parallel wheel generation, the overhead imposed by the wheel drastically hamper the performance. Nevertheless, the sieving time is lowered, as seen if we deduce the wheel generation time from the total - in scenarios where repeated sievings may use the same wheel, the result may justify the overhead. On the other hand, the performance gains become really significant for large ranges, from 10^9 and above, where the improvements go beyond 30%.

As a final demonstration for the power of static wheels, in Table 4 we have a comparison between the original mobile sieve, as devised by Charters (1967) and optimized by Singleton (1969a) (check out (Ghidarcea and Popescu, 2023) for details), and our modern implementation of the front-wave sieve using the W(9) wheel: the wheel based version achieves almost triple the performance (see accompanying code for this optimized sieve):

Table 4. Times for Chartres/Singleton algorithm improved with W(9) [ms]

Limit (10^n)	*6*	*7*	*8*	*9*	*10*
Singleton 356	9	109	1'266	14'519	162'777
W(9)	335	365	718	5'068	57'707

3. PARALLEL CACHE INTENSIVE ALGORITHMS

Singleton's **357** algorithm (Singleton, 1969b), the grandfather of modern fast sieves, introduces two fundamental ideas to modern sieving algorithms:

- sieve individual chunks of the target set instead of the whole interval (the sieving is done using pre-generated root primes);
- use a static wheel, i.e. W(3) - this wheel has two big advantages:
  - is small enough to be easily hard-coded;
  - implements the 1bit buffer very efficiently.

Modern implementations use chunks small enough to fit inside the L1/L2 cache - contemporary CPUs have very intelligent prefetch algorithm, so most of the times L2 size is the one to be observed. As said, the original **357** algorithm was already built to work only on a small window of the sieve at once - reducing this window to fit within L2 and experimenting a bit with it will give you the optimal value for the buffer/chunk.

This window size will not match exactly the L1/L2 sizes as there are also other data to be cached and the CPU will optimize data access within all memory layers so that this buffers will seem greater than they should be arithmetically.

Moreover, there is some overhead associated with every window, so the optimal buffer size may slightly vary also with the size of the targeted domain: for example, in our case, for smaller domains (up to a couple of billions) the optimal buffer was somewhere around 250k × 1byte vector, whereas at tens of billions it was around 450k - the heavier the overhead, the larger the differences.

Table 5. Buffer size [kB] related to sieving time [ms] (up to 1 respectively 5 billions)

Buffer	**1billion**	**5billions**
1	2'556	17'962
5	1'321	8'222
10	1'061	6'460
50	780	4'236
90	765	3'960
100	754	3'901
110	755	3'898
125	756	3'954
150	750	3'830
175	748	3'792
200	748	3'785
250	748	3'809
300	751	3'798
350	748	3'795
400	750	3'751
450	749	3'757
500	753	3'755
550	753	3'750
600	755	3'770
675	756	3'775
750	762	3'780
850	763	3'820
950	771	3'868
1'000	776	3'839
1'100	777	3'855
1'200	772	3'911
1'300	783	3'903
1'400	789	3'903
1'500	844	3'920
5'000	850	4'252
10'000	1'010	4'301
20'000	1'367	5'180
30'000	1'614	6'926
40'000	1'614	8'569
50'000	1'929	9'990
60'000	2'181	11'330

There are 6 distinct zones that can be identified in the Table 5 [1] (of course, there are no exact boundaries for each range, as the exact point where data caching stop compensating for overhead loss is impossible to calculate and somewhat subjective to establish empirically):

(1) *Red* – the overhead incurred by the very small buffer is very bad
(2) *Yellow* – cache efficiency starts to compensated for the overhead
(3) *Green* – the sweet spot: the overhead is optimally spent for cache efficiency
(4) *Yellow* – L2 misses start to cancel L1/L2 efficiency
(5) *Blue* – L2 is trashed, but L3 manage to maintain some level of efficiency
(6) *Red* – the cache misses are the most important factor that affects the efficiency.

Nevertheless, we can see that L1 is not really a factor anymore for modern computers, as our green zones are clearly situates in the hundreds of kilobytes range, specific to L2. In our concrete case – AMD 7900x – the 32k L1 data cache is irrelevant, but the 1MB L2 cache per core clearly marks the spot (the L1 is of course instrumental under the

[1] We chose to mark down in the table the small statistical aberrations as measured, as they do not alter our overall conclusions.

hood, but the prefetching logic obscures it from our direct observation): the second yellow range, where performance starts again to deteriorate, is centered around that 1MB range.

The relative performance plateau of the blue zone is situated more or less within the range of L3 cache, that is 32MB per die here (64MB for the whole CPU).

The larger green zone for the 1billion limit vs the one for 5billions is explained by the size of extra-data that is managed for the larger limit and which has to be cached and also processed in the same boundaries with added overhead, thus less and less flexibility: for the lower values the overhead takes over faster, while for the larger values the caching of the other data (mainly the list of root primes and their current multiples) is hampered earlier.

Anyway, the main conclusion to be taken from here is that cache intensity will provide us a factor of at least 3x performance gain, as demonstrated here, and is instrumental for any good algorithm and especially for sieving.

3.1 Classic fast sieve

The fastest contemporary implementations of prime sieves are based on the observations and ideas presented previously – cache intensive 1bit small windows sieve with hard-coded W3. You can see in Table 6 a comparison between timings of **357** and our initial attempt to implement the fast W3 cached sieve – as visible from the values there, the overhead involved in the basic manipulation of the wheel practically cancels the performance gains.

The performance of our sieve, although deliberately designed for cache intensity and benefiting from a superior wheel, is only marginally better than **357** – a perfect testimonial for the inspiration behind the good old **357** algorithm. The academic level implementation which can be consulted in the accompanying code is a pretty good exemplification for the basics of a standard modern sieve.

This basic ideas were perpetuated with small improvements until 2001 when **Tomás Oliveira e Silva** introduced the **bucket sieve** algorithm (Oliveira e Silva, 2015): in this variant the root primes are distributed in buckets so that all and only those primes that have multiples during current interval/window/chunk are in placed the current bucket.

The proof-of-concept code, demonstrating that the overhead implied by bucket management is really well spent for some major performance gain, is also published there (the code includes several other typical optimizations for fast sieves).

The main issue for this proof-of-concept implementation is that it does not really generate the primes, but counts them with very clever byte level operations that are much faster than the actual prime generation – the performance of such sieve implementations is somewhat misleading but, unfortunately, many other similar implementations perpetuate this practice. In order to allow for direct comparison we had to modify our code also to count primes instead of generating them. The results can be seen in Table 7 – as one can see when comparing to the ones in Table 6, the prime values generation is quite taxing.

We know that W(3) is only 10-15% faster than a normal sieve, and the timings here show that the W(3) is less efficient than the bucket technique, even if it still manages to keep within the same range of performance.

After several years Tomás Oliveira published a new, improved version of the bucket sieve, with many new optimizations added to the code: the new timings are in Table 8. The new proof-of-concept code is power-of-two oriented, and we decided to keep it that way for minimal alteration of the code – still, we can interpolate 50-75% performance gain with the new, refined code, versus his initial proof-of-concept.

At this moment, the pinnacle of fast sieving is represented by implementations using this approach: cache intensive hard-coded W(3) with bucket sieve. The greatest overhead cost for these sieves comes from

- buckets management;
- iterating wheel values;
- finding and marking the correct bit in the *1bit* compressed buffers.

If we can reduce this hassle the algorithm may benefit greatly – of course, the best way to reduce something is to avoid it altogether.

3.2 New fast sieve

In the classic W(3) approach each byte codifies all the 8 values of the wheel, one bit for each value: for every bit i in byte k the codified candidate value n is

$$n_i = 30 \cdot k + r_i$$

where $r_i \in \{1, 7, 11, 13, 17, 19, 23, 29\}$ (the values of W(3)) – both k and r_i are here variable and dependant on the current step i in the wheel/byte.

Similar, for each root prime `p` the multiplier `delta` to obtain the composite `n = delta * p` to be sieved out jumps from one revolving wheel value to the next, resulting in some complicated code gymnastics involving lookup tables and other intricate optimization techniques (see Tomas Oliveira's code for some inspired examples).

Our idea is to **process each wheel value r independently**.

First, we codify all candidate values corresponding to the current wheel value r on their own contiguous buffer, independent of the others: this will eliminate most of the overhead required for bit discrimination relative to r - thus, the new formula for the candidate n is

$$n_i = 240 \cdot k + 30 \cdot i + r$$

where r is now fixed per each of the eight sieve buffers and i sweeps monotonously the interval [0..7]. This split buffer approach is depicted in figure 1, where the classic stacked buffer approach is on the left.

Second, we now compute the multiples n for each root prime p much simpler. Let us take two such multiples:

$$n_1 \cdot p = 30 \cdot k_1 + r$$
$$n_2 \cdot p = 30 \cdot k_2 + r$$

Thus

$$(n_2 - n_1) \cdot p = 30 \cdot (k_2 - k_1)$$

Table 6. Times for basic cache intensive algorithms [ms]

Limit (10^n)	*6*	*7*	*8*	*9*	*10*	*11*	*12*
357	2	9	83	817	8'304	87'160	-
W(3)	7	13	77	749	7'662	77'207	-
W(3) *2T*	7	11	53	517	5'581	59'502	-
New W(3)	11	14	50	416	3'908	38'153	384'981
New W(3) *2T*	11	14	40	273	2'398	22'168	221'699
Fast W(3) LP *8T*	23	25	27	49	301	2'904	27'919

Fig. 1. Split buffer approach

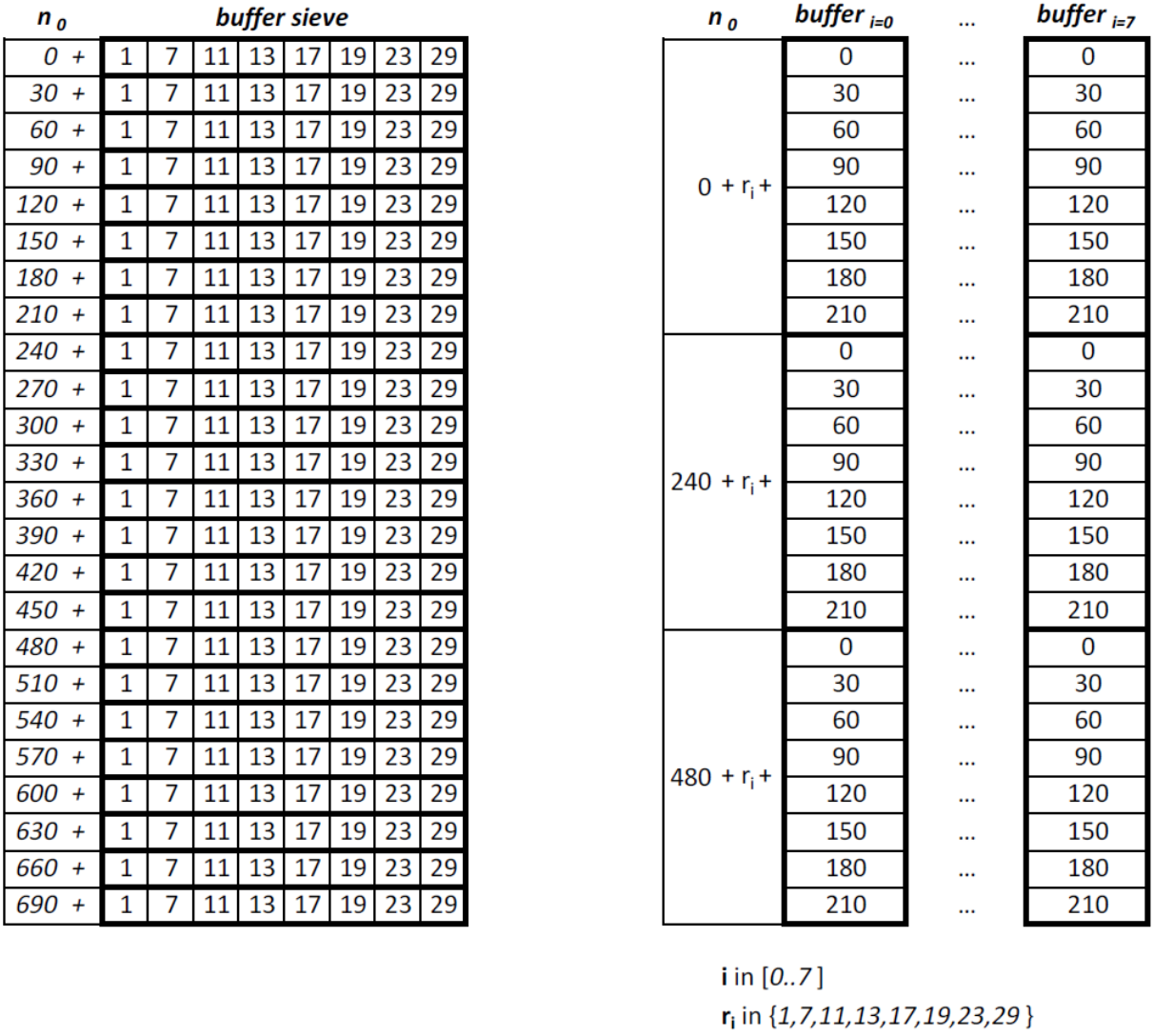

Table 7. Times for advanced cached algorithms [ms]

Limit (10^n)	*8*	*9*	*10*	*11*	*12*
Buckets (*PoC*)	42	430	4'663	53'581	-
New W(3) (*count only*)	50	519	5'526	60'215	-

While p is prime and all values are integers, it is necessary that 30 exactly divides $(n_2 - n_1) \Rightarrow$ the smallest value for $n_2 - n_1 = 30 \Rightarrow$ the smallest value for $k_2 - k_1 = p$. So, for each of the sieves corresponding to r_i, we have now the formula to advance to the next bit for each root prime p : just advance with p bits – no more divisions, modulus, multiplications, lookup tables and conditional statements.

Other advantages of this approach:

- the eight times increase of the window span will drastically reduce the impact of a bucket strategy to the point where the cost of the added bucket overhead becomes comparable to the associated gain;
- each of the 8 buffers are completely independent, thus allowing for parallel sieving without any synchronization or other special care.

Of course, something has to give:

- the logic required to determine the initial position of each root prime is marginally more complex;

Table 8. Times for advanced cached algorithms [ms]

Limit (2^n)	*26*	*28*	*30*	*32*	*34*	*36*	*37*	*38*
Buckets (*fast*)	16	64	272	1'142	4'786	20'270	-	-
New W(3) (*count only*)	11	37	149	611	2'492	10'404	21'233	42'948

Table 9. Generalized Wheel timings [ms]

Limit **(10^n)**	**primesieve**	**generalized wheel**				
		3	*4*	*5*	*6*	*7*
11	743	1'481	1'099	920	871	1'190
12	9'177	15'284	12'085	10'218	**9'024**	9'252
13	117'605	-	-	112'730	**100'197**	**96'627**
14	1'298'232	-	-	-	1'331'153	**1'244'144**

- there are 8 distinct sets of auxiliary data for root prime positioning that have to be maintained.

Nevertheless, the gains far outweigh the costs, as exemplified by the timings in Table 6. Moreover, as seen in Table 8, even when compared with the academic single-threaded optimized implementation of the most advanced algorithm to date - Tomas Oliveira's bucket sieve - our algorithm, although benefiting from much less code optimization and refinement, outpaces it with quite a good margin.

One can object that the bucket implementation we are comparing against does not benefit from the W(3) wheel - we argue that a similar level of optimization applied to the new algorithm will benefit the New W(3) code significantly more than W(3) applied to bucket sieve, as we plan to demonstrate further.

For further reference and analysis, in Table 6 we compiled the timings for the full New W(3) sieve (including full prime values generation) and added a version of New W(3) *2T* where the prime generation is delegated to a separate thread.

Also, you will find there timings for Fast W(3) *8T*, a light parallel version for the New W(3) with the native 8 threads (each processing independently one of the 8 buffers) - the performance there includes again full prime generation and speaks for itself. Furthermore, adding just one extra thread for parallel counting (as exemplified in New W(3) *2T*), may increase performance with another 20-30%.

For supplementary clarifications check the accompanying code which includes implementations for all the new sieve variants introduced and referred here.

3.3 Wheel generalization

Looking closer at the idea behind our improved W(3) algorithm introduced in previous chapter (3.1) we sense that we can extend it to any wheel beyond W(3): for any N, we should be able to devise an algorithm that can sieve individually and independently a number of subsieves equal with the number of elements in the wheel W(N) - such an algorithm may achieve several objectives at once:

- significantly larger reduction in sieving effort, proportional with the wheel's reduction factor;
- greater native parallelism of the process due to the bigger number of distinct, independent buffers;
- practical elimination of the need for bucket sieve tactics because a larger wheel will determine an even larger window span.

We've already seen how to generate large wheels quite fast or how to iterate multiples of root primes on split buffers with minimal complexity. Still, the somewhat crude solution employed by us in New W(3) for initial positioning of each root prime multiple at the beginning of sieving for each segment can work decently for W(3), which has only 8 values, but, as seen in Table 1, the number of values grows exponentially with larger wheels: W(6) already has 5'760 values, and W(7) no less than 92'160. At this point we have to analyze which are the possible combinations involved when generating the composites, so we need get back at the theory behind static wheels and make some theoretical considerations, starting with a very short review:

(a) the product of the first n prime numbers is called a **Primorial** (P):

$$P(n) = \prod_{i=1}^{n} p_i \quad (\leftarrow p_1 = 2,\, p_2 = 3, \ldots ;\, p_i\, is\, prime).$$

(b) if an integer δ is coprime with any p_i, then $P(n) + \delta$ is also coprime with any p_i.

(c) a wheel *W(N)* is made from all the natural numbers lower than $P(N)$ that are not divisible by any of the first N primes:

$$W(N) = \{r \,|\, 1 \leqslant r \leqslant P(N) \,\&\, (r, P(N)) = 1\}.$$

(d) we say that the **Length** of wheel $W(N)$ is the primorial $P(N)$. The main ideas behind the static wheels are:

- any prime can be represented by a wheel: $p = P \cdot k + r$, thus reducing the set of candidates;
- any multiplier used to mark composites is also in the form $n = P \cdot k + r$, thus reducing the number of markings.

Given the above, we can express any composite c in relationship with two values of a wheel:

$$c = (P \cdot k_p + r_p) \cdot (P \cdot k_n + r_n) = P \cdot k_c + r_p \cdot r_n$$
$$\Rightarrow c \equiv (r_p \cdot r_n) \bmod P$$

But, because $(r, P) = 1$ for any r in wheel W, any $(r_p \cdot r_n) \bmod P$ must also be a member of wheel W (otherwise either r_p or r_n is not coprime with P). Moreover, it can be verified that for each r_p and r_k there is exactly one r_n so that $(r_p \cdot r_n) \bmod P = r_k$. As r_p is fixed per each root prime p and r_k is fixed per each buffer k, it becomes clear that all multipliers for p are in the form $n = P \cdot k + r_n$

where r_n is again fixed for each pair (p, k) - the practical problem is now to predetermine r_n for each r_p and r_k.

Of course, a good starting point is to compute the symmetrical moduli matrix R where $r_{ij} = (r_i \cdot r_j) \bmod P$, and from here to derive the r_n factor matrix. Our New W(3) implementation hard-codes the factor matrix as follows:

```
uint8 w3[8] = {1, 7, 11, 13, 17, 19, 23, 29};

//w3_pattern[i][j] = (w3[i]*w3[j]) mod 30
uint8 w3_pattern[8][8] =
    {{ 1,  7, 11, 13, 17, 19, 23, 29},
     { 7, 19, 17,  1, 29, 13, 11, 23},
     {11, 17,  1, 23,  7, 29, 13, 19},
     {13,  1, 23, 19, 11,  7, 29, 17},
     {17, 29,  7, 11, 19, 23,  1, 13},
     {19, 13, 29,  7, 23,  1, 17, 11},
     {23, 11, 13, 29,  1, 17, 19,  7},
     {29, 23, 19, 17, 13, 11,  7,  1} };

//(w3[i] * w3_factors[i][j]) mod 30 = w3[j];
uint8 w3_factors[8][8] =
    {{ 1,  7, 11, 13, 17, 19, 23, 29},
     {13,  1, 23, 19, 11,  7, 29, 17},
     {11, 17,  1, 23,  7, 29, 13, 19},
     { 7, 19, 17,  1, 29, 13, 11, 23},
     {23, 11, 13, 29,  1, 17, 19,  7},
     {19, 13, 29,  7, 23,  1, 17, 11},
     {17, 29,  7, 11, 19, 23,  1, 13},
     {29, 23, 19, 17, 13, 11,  7,  1} };
```

To speed up significantly the process we use this very important observation derived from studying the moduli and factor matrices: **the lines in the factor matrix are the same with the ones in the moduli matrix, just in different order and interchangeable** – this was verified experimentally by us for $N \leq 7$ and we assume that the statement holds for every N, but presently we do not have a mathematical demonstration for this conjecture.

Here we begin to see the practical limitations of this approach: the matrix already has $92'160^2 \simeq 8.5$ billion elements for N=7 and over 2.75 trillion for N=8 - we can conclude here that W(7) is the largest wheel that is practically manageable with the precomputed factor matrix technique. In the accompanying code you can find a relatively efficient implementation for in-place fast computing of the W(7) factor matrix from the moduli matrix in just under 3 seconds on our reference AMD 7900X CPU.

To demonstrate the strength of the new algorithm we created a fully parallelized academic implementation of the generalized wheel in two versions: one for N in [3..6] and one for N=7, and we compared the timings of our relative crude and unrefined code with Kim Walisch's **Primesieve** (Walisch, 2023), an extremely optimized version of W(3) bucket sieve. Table 9 contains comparison between our generalized wheel and **Primesieve**, proving the potential of the new algorithm.

We have to stress out that **Primesieve** was at *version v11.1* at the moment of writing this article (august 2023) and benefits from many years of intensive code optimization and refinement, so the better timings above obtained with our proof-of-concept crude code are all the more significant for the potential of the new algorithm. There are at least several optimization paths that could be implemented:

- generate root primes and associated helper data only for the LIMIT at hand
- improve positioning algorithm and procedure; positioning is not at all cache intensive, it can be very well distributed over several threads
- use a separate thread for counting
- use an optimized free-fall approach to strike out composites - the accompanying code already contains the core of the free-fall procedure (see *Bit reset pattern + list init* section of methods at the end of *FreeGen.cpp* in accompanying code), but the positioning and hand-over details have to be optimized in order to outperform the basic approach.

An even more ambitious plan would be to avoid the factor matrix altogether and come up with a faster positioning method that would allow for very fast on-the-fly generation of that r_n used in positioning so that the initial full computation overhead of the factor matrix would not be required anymore, thus allowing for N=8 and even greater.

4. CONCLUSIONS

In this paper we analyzed some practical aspects regarding the exploitation of static wheels for primes sieving and proposed a new generalized method to approach the algorithms based on such wheels. We have shown both the advantages and practical limitations of the wheels, accompanied by exemplifying code that demonstrate such basic techniques and pertinent data supporting out conclusions that wheels do benefit the sieving, but wheels above N=10 become impractical.

We presented a brief history of the most efficient sieving algorithm to date and we analyzed the way in which cache intensity influence the performance. We also introduced a new approach to the sieving process that achieves superior performance in practice when applied to the basic implementation of state-of-the-art cache intensive fast sieve algorithms.

Finally, we tried to generalize the method exposed previously. We analyzed some theoretical aspects regarding the new method and built a proof-of-concept level implementation with excellent performance. We benchmarked our method against the best performing implementation we know at the moment of any sieving algorithm – Kim Walish's **Primesieve** – and showed it to be on par or outperform it for bigger values, although our proof-of-concept code is by far less optimized than **Primesieve**.

ACKNOWLEDGEMENTS

We express our gratitude to professors Nirvana POPESCU, Emil SLUSANSCHI and Vlad CIOBANU from University

POLITEHNICA of Bucharest, Computer Science Department, for their invaluable guidance and advice – their input was decisive for the quality of this paper.

CRediT [2] **author statement** - ***Mircea Ghidarcea***: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – Original & Editing, Visualization; ***Decebal Popescu***: Validation, Supervision, Project administration, Writing - Review

---

Funding: This research did not receive any specific grant from public, commercial or not-for-profit funding agencies.

Conflicts of Interest: The authors declare no conflict of interest.

REFERENCES

Atkin, A.O.L. and Bernstein, D.J. (2003). Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246), 1023–1030. doi:10.1090/S0025-5718-03-01501-1.

Charters, B.A. (1967). Algorithm 311: Prime number generator 2. *Communications of the ACM*, 10(9), 570. doi:10.1145/363566.363692.

Chartres, B.A. (1967). Algorithm 310: Prime number generator 1. *Communications of the ACM*, 10(9), 569. doi:10.1145/363566.363689.

Crandall, Richard and Pomerance, Carl (2005). *Prime Numbers: A Computational Perspective*. Springer-Verlag, New York. doi:10.1007/0-387-28979-8.

Dunten, B., Jones, J., and Sorenson, J. (1996). A space-efficient fast prime number sieve. *Information Processing Letters*, 59(2), 79–84. doi:10.1016/0020-0190(96)00099-3.

Ghidarcea, M. and Popescu, D. (2023). Prime Numbers Sieving - a Systematic Review with Performance Analysis. doi:10.2139/ssrn.4536720.

Gries, D. and Misra, J. (1978). A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12), 999–1003. doi:10.1145/359657.359660.

Mairson, H.G. (1977). Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20(9), 664–669. doi:10.1145/359810.359838.

Nicomachus (1926). *Introduction to Arithmetic*. Studies. Humanistic Series. Macmillan.

Oliveira e Silva, T. (2015). Fast implementation of the segmented sieve of Eratosthenes. https://sweet.ua.pt/tos/software/prime_sieve.html.

Pratt, V.R. (1977). CGOL - an Algebraic Notation For MACLISP users. *Working paper, MIT AI Lab, Cambridge*.

Pritchard, P. (1981). A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1), 18–23. doi:10.1145/358527.358540.

Pritchard, P. (1982). Explaining the wheel sieve. *Acta Informatica*, 17(4). doi:10.1007/BF00264164.

Pritchard, P. (1983). Fast compact prime number sieves (among others). *Journal of Algorithms*, 4(4), 332–344. doi:10.1016/0196-6774(83)90014-7.

Singleton, R.C. (1969a). Algorithm 356: A prime number generator using the treesort principle [A1]. *Communications of the ACM*, 12(10), 563. doi:10.1145/363235.363244.

Singleton, R.C. (1969b). Algorithm 357: An efficient prime number generator [A1]. *Communications of the ACM*, 12(10), 563–564. doi:10.1145/363235.363247.

Sorenson, J. (1990). An Introduction to Prime Number Sieves. Technical report, University of Wisconsin-Madison, Computer Sciences Department.

Sorenson, J. (1991). An Analysis of Two Prime Number Sieves. Technical report, University of Wisconsin-Madison, Computer Sciences Department.

Sorenson, J. (1998). Trading time for space in prime number sieves. Technical report, Springer-Verlag Springer e-books, Berlin, Heidelberg.

Walisch, K. (2023). Primesieve.

Wood, T.C. (1961). Algorithm 35: Sieve. *Communications of the ACM*, 4(3), 151. doi:10.1145/366199.366257.

[2] `https://credit.niso.org/`