Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 205–223

Cache optimized linear sieve

Antal JÁRAI
Eötvös Loránd University
email: ajarai@moon.inf.elte.hu

Emil VATAI
Eötvös Lorńd University
email: emil.vatai@gmail.com

Abstract. Sieving is essential in different number theoretical algorithms. Sieving with large primes violates locality of memory access, thus degrading performance. Our suggestion on how to tackle this problem is to use cyclic data structures in combination with in-place bucket-sort.

We present our results on the implementation of the sieve of Eratosthenes, using these ideas, which show that this approach is more robust and less affected by slow memory.

1 Introduction

In this paper we present the results obtained by implementing the sieve of Eratosthenes [1] using the methods described at the 8th Joint Conference on Mathematics and Computer Science in Komárno [3]. In the first section, the problem which is to be solved by the algorithm and some basic ideas about implementation and representation are presented. In Section 2, the methods to speed up the execution are discussed. In Section 3 the numerical data of the measurement of runtimes is provided on two different platforms in comparison with the data found at [4].

Given an array (of certain size) and a set P of (p, q) pairs, where p is (usually) a prime and $0 \leq q < p$ is the offset (integer) associated with p. A sieving algorithm for each $(p, q) \in P$ pair performs an action on every element of the array with a valid index $i = q + mp$ (for $m \geq 0$ integers).

Sieving with small primes can be considered a simple and efficient algorithm: start at q and perform the action for sieving, then increase q by p and repeat.

Computing Classification System 1998: F.2.1, E.1
Mathematics Subject Classification 2010: 11-04, 11Y11, 68W99
Key words and phrases: sieve, cache memory, number theory, primality

But when sieving with large primes, larger than the cache, memory hierarchy comes into play. With these large primes, sieving is not sequential (i. e. q skips great portions of memory), thus access to the sieve array is not sequential and this causes the program to spend most of its time waiting to access memory, because of cache misses.

1.1 Sieve of Eratosthenes

The sieve of Eratosthenes is the oldest algorithm for generating consecutive primes. It can be used for generating primes "by hand" but it is also the simplest and most efficient way to generate consecutive primes of high vicinity using computers. The algorithm is quite simple and well-known. Starting with the number 2, declare it as prime and mark every even number as a composite number. The first number, which is not marked is 3. It is declared as the next prime, so all the numbers divisible by 3 (that is every third number) is sieved out (marked), and so on.

1.2 Basic ideas about implementation

The program finds all primes in an interval $[u, v] \subset \mathbb{N}$ represented in a bit table. The program addresses the issue of memory locality by sieving the $[u, v]$ in subintervals of predefined size, called segments, which can fit in the cache, thus every segment of the sieve table needs to pass through the cache only once. For simplicity and efficiency the size of segment is presumed to be the power of two.

Because every even number, except 2, is a composite, a trivial improvement is to represent only the odd numbers, and cutting the size of the task at hand in half.

1.2.1 Input and output

The program takes three input parameters: the base 2 logarithm of the segment size denoted by l,[1] and instead of the explicit interval boundaries u and v, the "index" of the first segment denoted by f, and the number of segments to be sieved denoted by n, is given. This means, the numbers between $u = f2^{l+1} + 1$ and $v = (f + n)2^{l+1}$ are sieved. For simpler comparison with results from [4],

[1] cache size $\approx$ segment size $= 2^l$ bits $= 2^{l-3}$ bytes $= 2^{l+1}$ numbers represented, because only odd numbers are represented in the bit table.

the exponent of the approximate midpoint of the interval can also be given as an input parameter instead of f[2].

As for the output, the program stores the finished bit table in a file in the `/tmp` directory, with the parameters written in the filename.

1.3 Sieve table and segments

Definition 1 (Sieve table, Segments) *The sieve table* S *(for the above given parameters) is an array of* $n2^l$ *bits. For* $0 \le j < n2^l$*, the bit* S_j *represents the odd number* $2(f2^l + j) + 1 \in [u, v] = [f2^{l+1}, (f + n)2^{l+1}]$*.* S_j *is initialized to* 0 *(which indicates that* $2(f2^l + j) + 1$ *is prime);* S_j *is set to* 1*, if* $2(f2^l + j) + 1$ *is sieved out i.e. it is composite.*

The t*-th segment, denoted by* $S^{(t)}$ *is subtable of the* t*-th* 2^l *bits of the sieve table* S*, i.e.* $S_q^{(t)} = S_{t2^l+q}$ *for* $0 \le q < 2^l$ *and* $0 \le t < n$*.*

After p sieves at S_j, marking $2(f2^l + j) + 1$ as a composite, the next odd composite divisible by p is the $2(f2^l + j) + 1 + 2p = 2(f2^l + j + p) + 1$, so the index j has to be incremented only by p. That is, not representing the even numbers doesn't change the sieving algorithm, except the calculation of the offsets (described in Lemma 3).

1.4 Initialization phase

For every prime p, the first composite number not marked by smaller primes, will be p^2, i.e. sieving with p can start from p^2. To sieve out the primes in the $[u, v]$ interval, only the primes $p \le \sqrt{v}$ are needed. Finding these primes and calculating the q offsets, so that $q \ge 0$ is the smallest integer satisfying $p \mid 2(fs^s + q) + 1$ is the initialization phase. Presumably $\sqrt{v}$ is small ($\sqrt{v} < u$), and finding primes $p < \sqrt{v}$ (and calculating offsets) can be done quickly.

Definition 2 *The set of primes, found during the initialization of the sieve is called the base.* $P = \{p \text{ prime} : 2 < p \le \sqrt{v}\}$

2 Addressing memory locality

Because the larger the prime, the more it violates locality of memory access when sieving, the basic idea is to treat primes of different sizes in a different

[2]Of course, this just relieves the user from the tedious task of calculating f by hand for the given exponent of the midpoint of the interval, but internally the flow of the program was same as if f is given.

way, and process the sieve table by segments in a linear fashion, loading each segment in the cache only once and sieving out all the composites in it.

2.1 Medium primes

The primes $p < 2^l$ are *medium primes*. Segment-wise sieving with medium primes is simple: (p, q) prime-offset pairs with $p \leq 2^l$ and $0 \leq q < p$ are stored. Each prime marks at least one bit in each segment. For each prime p, starting from q, every p-th bit has to be set, by sieving at the offset q and then incrementing it to $q \leftarrow q + p$ while $q < 2^l$. Now q would sieve in the next segment, so the offset is replaced with $q - 2^l$. The first offset for a prime p and the given parameters f and l can be found using the following Lemma.

Lemma 3 *For each odd prime p and positive integers l and f, there is a unique offset $0 \leq q < p$ satisfying:*

$$p \mid 2(f2^l + q) + 1 \tag{1}$$

Proof. Rearranging (1) gives $f2^{l+1} + 2q + 1 = mp$ for some m. The integer m has to be odd, because the left hand side and p are odd. The equation can further be rearranged to a form, which yields a coefficient and something similar to a reminder:

$$f2^{l+1} = (m-1)p + (p - (2q+1)).$$

The last term is even, so if the remainder $r = f2^{l+1} \mod p$ is even, then $q = (p-r-1)/2$ satisfies $0 \leq q < p$ and (1). If r is odd, then $q = (2p-r-1)/2$ satisfies $0 \leq q < p$ and (1). Because r is unique, q is also unique. □

2.2 Large primes

The primes $p > 2^l$ are *large primes*. These primes "skip" segments i.e. if a bit is marked in one segment by the large prime p, (usually) no bit is marked by the prime p in the adjacent segment. The efficient administration of large primes is based on the following observation:

Lemma 4 *If the prime p, which satisfies the condition $k2^l \leq p < (k+1)2^l$ (for some integer $k \geq 0$), marks a bit in $S^{(t)}$, then the next segment where the sieve marks a bit (with p) is the segment $S^{(t')}$ for $t' = t + k$ or $t' = t + k + 1$.*

Proof. If the prime p marks a bit in $S^{(t)}$ then it is the $S^{(t)}_q$ bit, for some offset $0 \le q < 2^l$. The S_{t2^l+q} bit is marked first, then the S_{t2^l+q+p} bit, so the index of the next segment is $t' = \lfloor (t2^l + q + p)/2^l \rfloor$, thus

$$t + k = \frac{t2^l + 0 + k2^l}{2^l} \le \underbrace{\left\lfloor \frac{t2^l + q + p}{2^l} \right\rfloor}_{=t'} < \frac{t2^l + 2^l + (k+1)2^l}{2^l} = t + k + 2.$$

□

2.3 Circles and buckets

The goal is, always to have the right primes available for sieving at the right time. This is done by grouping primes of the same magnitude together in so called circles, and within these circles grouping them together by magnitude of their offsets in so called buckets.

Definition 5 (Circles and Buckets) *A circle (of order k, in the t-th state) denoted by $C^{k,t}$ is sequence of $k+1$ buckets $B^{k,t}_d$ (where $0 \le d \le k$). Each bucket contains exactly those (p, q) prime-offset pairs, which have the following properties:*

$$k2^l < p < (k+1)2^l \tag{2}$$

$$0 \le q < \max\{p, 2^l\} \tag{3}$$

$$p \mid 2\big((f + t + d - b + k + 1)2^l + q\big) + 1 \qquad \text{if } 0 \le d < b \tag{4}$$

$$p \mid 2\big((f + t + d - b)2^l + q\big) + 1 \qquad \text{if } b \le d \le k \tag{5}$$

where $b = t \mod (k+1)$ is the index of the current bucket.

$(p, q) \in C^{k,t}$ means that there is an index $0 \le d \le k$ for which $(p, q) \in B^{k,t}_d$ and $p \in C^{k,t}$ means that there is an offset $0 \le q < 2^l$ for which $(p, q) \in C^{k,t}$.

As the state t is incremented b changes from 0 to k cyclically. This can be imagined as a circle turning through $k+1$ positions, justifying its name. Also, each bucket contains all the right primes with all the right offsets, that is when it becomes the current bucket, it will contain exactly those prime-offsets which are needed for sieving the current segment.

Circles and buckets can be defined for arbitrary $k, t \in \mathbb{N}$, but only $0 \le k \le \lfloor \max P/2^l \rfloor = K$ and $0 \le t < n$ are needed.

Theorem 6 *For each* $p \in P$ *there is a unique* $0 \le k \le K$ *and for each state* t*, a unique* $0 \le d \le k$ *and offset* q *such that* $(p, q) \in B_d^{k,t}$.

Proof. For every p prime, dividing (2) with 2^l gives $k = \lfloor p/2^l \rfloor$, and if $p \in P$, then $p \le \max P$, so $\lfloor p/2^l \rfloor \le \lfloor \max P/2^l \rfloor = K$, therefore $0 \le k \le K$. For each p, (2) is true independent of the state t.

For each t, a unique offset q satisfying (3) and a unique index $0 \le d \le k$ satisfying (4) and (5) has to be found . It should be noted that the precondition of (4) and (5) are mutually exclusive, that is an index d satisfies only one of the two preconditions, and only that one has to be proven.

Medium primes, that is $p < 2^l$ is the special case of $k = 0$. $C^{0,t}$ has only one bucket with the index $d = b = 0$, so the precondition of (4) is always false. (3) is equivalent to $0 \le q < p$ since $p < 2^l$. (5) is equivalent to $p \mid 2\big((f+t)2^l+q\big)+1$ because $0 \le b \le d$, that is $b = 0 = d$. Lemma 3 for the prime p and integers l and $t + f$ shows that there is an integer q which satisfies (3) and (5).

For $p > 2^l$, the proof is by induction. If $t = 0$ is fixed, then $b = 0$ is the current bucket's index. The precondition of (4) is false and (5) is equivalent to $p \mid 2\big((f + d)2^l + q\big) + 1$. Lemma 3 for the prime p and integers l and f gives a q' which satisfies $p \mid 2(f2^l + q') + 1$ and $0 \le q' < p$. Dividing q' by 2^l gives $q' = d2^l + q$, where d and q are unique and satisfy (3) and (5).

If the statement holds for $t \ge 0$, then there is an index $0 \le d \le k$ and an offset $0 \le q < 2^l$ such that $(p, q) \in B_d^{k,t}$. The current bucket is $b = t \mod (k + 1)$, and the statement will be proven for the next state $t' = t + 1$ with $b' = (b + 1) \mod (k + 1)$ as the index of the "next" current bucket.

The first case is $d \ne b$. It can be shown that incrementing the state, the prime remains in the same bucket with the same offset, i.e. $(p, q) \in B_d^{k,t'}$. If $b < k$, then $b' = b + 1 \le k$ holds, that is $t' - b' = (t + 1) - (b + 1) = t - b$ so (4) and (5) remain the same, except for the preconditions. But since $d \ne b$, $0 \le d < b < b'$ or $b < b' \le d$ will still remain true. If $b = k$, then $0 \le d < b = k$ and $b' = 0 = b - k$, so the precondition of (4) is false, and (5) becomes:

$$p \mid \big((f + (t + 1) + d - (-k))2^l + q\big) + 1 = \big((f + t + d + k + 1)2^l + q\big) + 1$$

for $0 = b' \le d$. Since (4) was true for d and t, now (5) is true for d and $t + 1$.

The second case is when $d = b$, and it can be shown that incrementing the state the prime remains in the same bucket or goes into the previous one (modulo $(k + 1)$) with a different offset. If $d = b$, d satisfies the precondition of (5), that is $p \mid 2\big((f+t)2^l+q\big)+1$ is true. The next odd number divisible by p can be obtained by incrementing the offset by p. As seen in Lemma 4 $q + p$

can be written as $q + p = k'2^l + q'$, where $k' = k$ or $k+1$ and $0 \le q' < 2^l$. With incrementing the offset by p for t (5) gives:

$$p \mid 2\big((f + t + k')2^l + q'\big) + 1. \tag{6}$$

Let d' be $d + (k' - 1) \mod (k+1)$, that is $d' \equiv d \pmod{k+1}$ or $d' \equiv d - 1 \pmod{k+1}$. The precondition of (5) for the next state t' is true if $b = k$ (then $b' = 0$ and $d' = k$ or $k-1$) or if $b = 0$ and $k' = k$ (then $b' = 1$ and $d' = k$). If these values are plugged in (5) for t', i.e. $p \mid \big((f + t' + d' - b')2^l + q'\big) + 1$, equation (6) is obtained, which is true. The preconditions of (4) are satisfied for every other case, that is, when $0 < d = b < k$ (then $0 \le d' < b' \le k$) or $b = 0$ and $k' = k + 1$ (then $b' = 1$ and $d' = 0$). Again, by plugging these values in (4) for t', that is $p \mid \big((f + t' + d' - b' + k + 1)2^l + q'\big) + 1$ equation (6) is obtained, which is also true. □

As a consequence, if it doesn't cause any confusion, the state may be omitted from the notation, because each prime with its offset is maintained only for the current state. As the program iterates through states, the primes may "move" between buckets, and offsets usually change.

The following Corollary shows, that sieving with circles and buckets sieves out all composites marked by large primes (sieving with medium primes is more or less trivial).

Corollary 7 *For each* $p > 2^l$ *and odd* $i' \in [u, v]$ *satisfying* $p \mid i'$ *there exists a unique state* t *and an offset* q*, so that* (p, q) *is in the current bucket of the circle to which* p *belongs to.*

Proof. Let i' be represented by S_i for $0 \le i < n2^l$, that is $i' = 2(f2^l+i)+1$. The statement is true for $t = \lfloor i/2^l \rfloor$, because then $i = t2^l + q$, $b = t \mod (k+1)$, and substituting d with b in (5), the equation $p \mid 2\big((f + t)2^l + q\big) + 1 = 2(f2^l + i) + 1$ is obtained. □

The proof of Theorem 6 could have been simpler, but the proof by induction gives some insight on how the circles and buckets work and behave, giving some idea about how to implement them. This behavior is explicitly stated in the following Corollary.

Corollary 8 *For each state* t*, each order* k*,* $b = t \mod (k+1)$ *and* $b' = (t + k) \mod (k+1)$*, if* $(p, q) \in B_b^{k,t+1}$ *then* $(p, q') \in B_b^{k,t}$ *for some offset* q'*, and* $B_{b'}^{k,t} \subset B_{b'}^{k,t+1}$ *and for every* $b \ne d \ne b'$ *and* $d \ne d'$ $B_d^{k,t} = B_d^{k,t+1}$.

The first statement says that with respect *only* to primes $B_b^{k,t+1} \subset B_b^{k,t}$.

Proof. The index b' refers to the current bucket in the previous state and as seen in the remarks in the proof of Theorem 6, iterating from state t to $t+1$ leaves the buckets with indexes $d \neq b$ and $d \neq b'$ untouched, and some primes with new offsets are left in the current bucket while others are put in the previous one. □

2.4 Modus operandi

The goal is to perform a segment-wise sieve:

Medium primes belonging to C^0 are a special case, and they sieve at least once in a segment. The t-th state of C^0 contains all medium primes with the smallest offsets for sieving in the $S^{(t)}$ segment. For a prime in C^0, after sieving with it the offset is replaced with the smallest offset for sieving the next segment $S^{(t+1)}$. After sieving with all medium primes, C^0 is in the $t+1$-th state. This is implemented in a single loop, iterating through all medium primes.

The circle C^k ($k > 0$), in the t-the state, for primes between $k2^l$ and $(k+1)2^l$, has the prime-offset pairs, needed for sieving $S^{(t)}$ in the current bucket. After sieving with all these primes, the circle is in it's next state, with offsets replaced, and some primes moved to the previous bucket, ready for sieving $S^{(t+1)}$. Sieving large primes is implemented via two embedded loops, the outer iterating through circles by their order, covering all primes, and the inner loop iterating through the primes of the current bucket of the current circle.

The above two procedures are called in a loop for segment $S^{(t)}$, iterating from $t = 0$ to $n-1$. Corollary 7 shows, that the primes for sieving the t-th segment are in the current buckets of circles in t-th state, so this procedure performs the sieve correctly.

2.5 Implementation

For sequential access, all prime-offset pairs, buckets and circles are stored as linear arrays: the array of prime-offset pairs is denoted by $(\hat{p}_i, \hat{q}_i)$, the array of buckets denoted by $\hat{b}_i$ and the array of circles denoted by $\hat{c}_i$ ($i \in \mathbb{N}$).

2.5.1 Array of circles

$\hat{c}_k$ is the data structure (C `struct`) implementing the circle C^k. It is responsible for most of the administration of the associated primes and buckets. Of course memory to store $K+1$ circles is allocated.

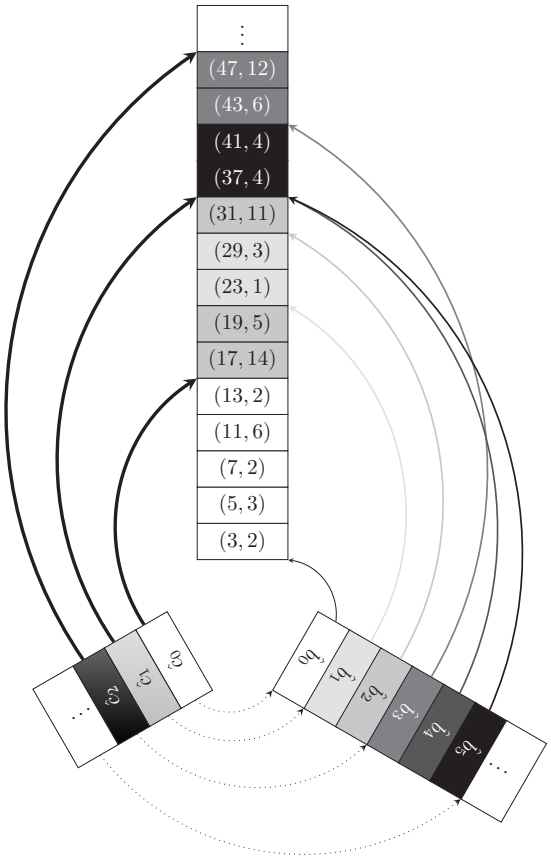

Figure 1: Array of circles, buckets and primes

In the implementation, primes of one circle are a continuous part of the array of primes. It was convenient to store the end-pointers of circles, i.e. a pointer to the prime after the last prime in the circle. So the medium primes are all primes before the up to but not including the prime at the end-pointer of $\hat{c}_0$, and all primes in C^k are the primes from the end-pointer of $\hat{c}_{k-1}$ up to but not including the prime pointed to by the end-pointer of $\hat{c}_k$. Since the primes are generated in an ascending order, these end-pointers can be determined easily, and they don't change during the execution of the program.

The circle $\hat{c}_k$ maintains the index of the current bucket. It is incremented by one (modulo $k+1$), that is $b \leftarrow b+1$, if $b < k$ or $b \leftarrow 0$ if $b = k$ assignment is performed after sieving with primes from the circle. Circle need to maintain

the index of the broken bucket explained in section 2.5.4.

The circle $\hat{c}_k$ could also maintain a pointer, to the first bucket B_0^k represented by $\hat{b}_{k(k+1)/2}$ in the buckets array, but it is not necessary because it can be calculated from k. There is a more efficient solution if the circles are processed with ascending orders: The starting bucket of $\hat{c}_1$ is $\hat{b}_1$ and this is stored as a temporary pointer. For every circle $\hat{c}_{k+1}$ the starting bucket is at $k+1$ buckets after the first bucket of the previous circle $\hat{c}_k$, so when finished with circle $\hat{c}_k$, this pointer has to be increased by $k+1$.

In Figure 1 the value l is 4 so the cache size is 16. C^0 has a white background as well as the bucket and primes associated with it. C^1 is light gray with black text, with two different shades for the two buckets and primes in them, and in a similar way C^2 is black with white text and slightly lighter shades of gray for the buckets and primes. The end-pointers are drawn as thick arrows, indicating that they don't move during the execution. The dotted lines are the calculable pointers to the first buckets.

2.5.2 Array of buckets

All buckets are stored consecutively in one array, i.e. the bucket B_d^k of circle C^k for $0 \le d \le k$ is represented by $\hat{b}_{k(k+1)/2+d}$. There are $K+1$ circles, with $k+1$ buckets for each $0 \le k \le K$, so memory for storing $(K+1)(K+2)/2$ buckets needs to be allocated.

The value of each $\hat{b}_d$ is the index (`uint32_t`) of the first prime-offset pair which belongs to the bucket $\hat{b}_d$. Primes that belong to one bucket are also in a continuous part of the primes array, so the bucket B_d^k contains the primes $\hat{p}_i$ (and the associated offsets $\hat{q}_i$) for $\hat{b}_{d'} \le i < \hat{b}_{d''}$ where $d' = k(k+1)/2+d$ and $d'' = k(k+1)/2+(d+1) \mod (k+1)$. The broken bucket is an exception to this. Empty buckets are represented with entries in the buckets array having the same value, i.e. B_d^k is empty if $\hat{b}_{d'} = \hat{b}_{d''}$ for $d' = k(k+1)/2+d$ and $d'' = k(k+1)/2+(d+1) \mod (k+1)$, e.g. $\hat{b}_4$ is empty in Figure 1.

Buckets are set during the initialization, but change constantly during sieving. First, using Lemma 3 an offset $0 \le q' < p$ is found for each p prime. All prime-offset pairs of the circle C^k are sorted in ascending offsets. Similarly, as primes are collected in circles, within one circles the offsets are collected in buckets. B_d^k contains all prime-offset pairs, so that $d2^l \le q' < (d+1)2^l$, but instead q', $0 \le q = q' - d2^l < 2^l$ is stored.

For each circle C^k, for each pair $(p,q) \in B_b^k$, the bit with index q in the current segment is set. After that, the new offset $q' = q+p$ is calculated, which is, because of Lemma 4, either $k2^l \le q' < (k+1)2^l$ or $(k+1)2^l \le q' < (k+2)2^l$.

In the former case $q' - k2^l$ is stored in the previous bucket (modulo $k+1$), or in the latter case $q' - (k+1)2^l$ is stored in the current bucket, as described in Corollary 8.

This can be implemented efficiently by keeping copies of a prime from the beginning and another prime from the end of the bucket. That way, in the first case (k segments were skipped), the prime-offset pair in the beginning of the bucket is overwritten and the value in the buckets array indicating the beginning of the bucket is incremented, thus putting the replaced, new prime-offset pair, in the previous bucket. In the second case ($k+1$ segments were skipped), the prime-offset pair in the end of the bucket is overwritten with the new prime-offset pair and it stays in the current bucket. After replacing a pair in one of the ends (the beginning or the end) of the bucket, the next prime is read from that end of the bucket, that is from the next entry, closer to the center of the bucket.

2.5.3 Array of primes

The array of primes contains all primes (medium and large), with the appropriate offsets, needed for sieving. The prime-offset pairs are stored as two 32 bit unsigned integers (two `uint32_t`s in a `struct`). Enough memory to store about $\frac{\sqrt{v}}{\log\sqrt{v}}$ pairs is allocated.

The array of primes is filled during the initialization phase. The values of the offsets q change after finishing a segment. Sometimes pairs from the end and the beginning of a bucket are swapped (as explained earlier), but this is all done in-place, that is, the array itself does not need to be modified or copied, just the values stored. All primes that belong to one circle as well as those that belong to one bucket (except the broken bucket) are stored in a coherent and continuous region of memory.

2.5.4 Broken bucket

For the circle C^k, the index of the broken bucket is $r = \max\{d : \hat{b}_{k(k+1)/2+d} = M\}$, where $M = \max\{\hat{b}_d : k(k+1)/2 \leq d < (k+1)(k+2)/2\}$. The primes which belong to this bucket, are the ones from the index $\hat{b}_{k(k+1)/2+r}$ and up to but not including the prime at the end-pointer of $\hat{c}_k$ and the primes from the end-pointer of $\hat{c}_{k-1}$ up to but not including the prime with the index $\hat{b}_{k(k+1)/2+r'}$ where $r' = (r+1) \mod (k+1)$, e.g. $\hat{b}_2$ in Figure 1. This idea also justifies the name circles, because logically the next prime after the end-pointer of a circle is the first prime of the circle. When the broken bucket is

not actually broken, the value of $\hat{b}_{k(k+1)/2+r'}$ is set to the index of the prime at the end-pointer of $\hat{c}_{k-1}$, e.g. $\hat{b}_3$ in Figure 1.

Every circle has a broken bucket and this has to be stored as a variable for each circle. The fact, that this can not be omitted is not trivial, but if all primes of a circle are one bucket, then all other buckets in that circle are empty. Because empty buckets are represented by having the same value as the following bucket, all buckets in that circle, that is all entries of $\hat{b}_d$ which represent the buckets of that circle, will have the same value. In this situation the program can't decide which buckets are empty and which one contains all the primes.

The broken bucket also moves around. When sieving with a bucket, its lower boundary is incremented. If sieving with the broken bucket, when the beginning of the bucket moves past the end of the circle (and jumps to the beginning), the previous bucket (modulo $k+1$) becomes the new broken bucket.

3 Speeding up the algorithm

The roughly described implementation of sieving can be further refined to gain valuable performance boosts.

3.1 Small primes

Sieving with primes $p < 64$ can be sped up by not marking individual bits, but rather applying bit masks. The subset of medium primes below 64 are called *small* primes.

The *AMD64*[3] architecture processors with *SSE2* extension, have sixteen 64-bit general purpose R registers, and sixteen 128-bit XMM registers. For sieving with small primes, the generated 64bit wide bit masks are loaded in these registers and `or`-ed together, to form the sieve table with small primes applied to it. The masks are then `shift`-ed, to be applied to the next 64 bits for R registers and 128 bits for XMM registers.

This is of course done in parallel, sieving by 128 bits at a time. The XMM registers first 64 bits are loaded from the memory at the beginning of sieving of a segment, and the last 64 bits are `shift`-ed (just like the R registers are shifted "mod 64"). There is two times as much sieving with the R registers than with the XMM registers.

With the first four primes "merged" into two, all the small primes can fit

[3] AMD™ is a trademark of Advanced Micro Devices, Inc.

in the R and XMM registers, so the only memory access is sequential and done once when starting and once when finished sieving. The primes 3 and 11 are merged into 33, that is, the masks of 3 and 11 are combined at the initialization, and the shifting needs to be done as if 33 was the prime for sieving, because the pattern repeats after 33 bits. 5 and 7 are merged into 35 and treated similarly.

3.2 Medium primes

As described earlier, for (p, q) pairs with medium primes, sieving starts from q by increasing it by p after sieving, until $q \geq 2^l$. Then the sieving is finished for that segment, and the sieving of the next segment starts from $q' = q - 2^l$. There are two methods in which this algorithm can be sped up.

3.2.1 Wheel sieve

In the special case of the sieve of Eratosthenes, the "wheel" algorithm (described in [5]) can be used to speed up the program. In some sense, it is an extension of the idea of not sieving with number 2.

Let W be the set of the first few primes and $w = \prod_{p \in W} p$. Sieving with the primes from W, sieves out a major part of the sieve table, and these bits can be skipped. Basically, when sieving with a prime $p \notin W$, the number i needs to be sieved (marked) by p, only if it is relative prime to w, that is, if i is in the reduced residue system modulo w denoted by W' (if $i \notin W'$ some prime from W will mark it).

Let $w_0 < \cdots < w_{\varphi(w)-1}$ be the elements of W', and Δ_s the number of bits that should be skipped, after sieving the bit with index congruent to w_s, that is $\Delta_i = (w + w_{(i+1) \bmod \varphi(w)} - w_i) \bmod w$. When i is sieved out by $p \notin W$, instead of sieving $i+p$ next, the program can skip to $i+\Delta_s p$ if $i \equiv w_s \pmod{w}$.

In the implementation, $W = \{2, 3, 5\}$, but 2 is "built in" the representation and this complicates thing a little bit: $w = 15$, $\varphi(w) = 8$ and $w_0 = 0$, $w_1 = 3$, $w_2 = 5$, $w_3 = 6$, $w_4 = 8$, $w_5 = 9$, $w_6 = 11$, $w_7 = 14$ are used (instead of $w = 30$ and 1, 7, 11, 13, 17, 19, 23, 29 for w_s). For each prime the offset q is initialized to the value $q' + mp$, where q' the offset found using Lemma 3 and m is the smallest non-negative integer, so that $f2^l + q \equiv w_s \pmod{w}$ for some $0 \leq s \leq 7$.

Let p^{-1} be the inverse of p modulo 15, and x a non-negative integer so that:

$$f2^l + q + xp \equiv 7 \pmod{15}. \tag{7}$$

Note that the residue class represented by 7 is 15, and that is the class divisible both by 3 and 5, and it is in a sense the "beginning" of the pattern generated by the primes in W when sieving. (7) states that after x times sieving (regularly) with p, the offset is at the "beginning" of the pattern, that is, in the residue class represented by 7, so $y = 7 - x$ is the residue class in which q actually is. $x \equiv (7 - (f2^l + q))p^{-1} \pmod{15}$ can be obtained from (7) by multiplying it with p^{-1}.

There is an index $0 \leq s \leq 7$, so that $w_s = y$. The index s, indicating where in the pattern is the offset q, is stored beside each (p, q) pair. Before sieving with p, the array $\Delta_0 p, \ldots, \Delta_7 p$ is generated in memory, and a pointer is set to $\Delta_s p$. After marking a bit, the offset is incremented by the values found at that pointer, and the pointer is incremented modulo 8, which can be implemented very efficiently with a logical `and` operation and a bit mask. Also all prime-offset pairs are stored on 64 bits and medium primes are $p < 2^l$ (where l is never more than 30), so at least 4 bits are not used where the index $0 \leq s \leq 7$ can fit.

3.2.2 Branch misses

Another speed boost can be obtained by treating the *larger medium primes* (near to 2^l) differently. This idea is somewhat similar to the one used with circles, because it is based on the observation that, if $\frac{2^l}{(k+1)} < p < \frac{2^l}{k}$, then p sieves k or $k+1$ times in one segment ($0 < k \in \mathbb{N}$). There is a different procedure g_k, for each of the first few values of k (e.g. $0 < k < 16$). g_k iterates the offset $k+1$ time, with the last iteration implemented using conditional move (`cmov`) operations. So, for each k, primes $\frac{2^l}{(k+1)} < p < \frac{2^l}{k}$ are collected in a different array, and the procedure g_k is invoked for each prime in that array. Having fixed number of iterations with a conditional move is faster then a branch miss, because the CPUs instruction stream is not interrupted.

3.3 Large primes

The sieving with large primes is roughly described above. Sieving with one prime, putting it back, with the new offset, and modifying the bucket boundary can be accomplished with only about 15 assembly instructions using conditional moves (`cmov`). This is very efficient, but other techniques can also be applied to reduce execution time.

3.3.1 Interleaved processing

Because the order in which the primes are processed doesn't matter, the memory latency can be hidden by processing primes from both ends of the bucket. As described earlier, for each bucket, two prime-offset pairs are loaded from the beginning and end of the current bucket and one of them is processed. To hide memory latency, the next prime is loaded into place of the processed prime *while* the other one is being processed. "Processing a prime" covers the following steps: marking the bit at the offset q; determining if $q+p$ skips k or $k+1$ segments; calculating the new offset $q' \leftarrow q+p-k2^l$, replacing the pair at the beginning of the bucket and incrementing the bucket's lower boundary, for the former case; or in the latter case, decrementing the pointer indicating the finished primes at the top of the bucket, after replacing the pair at the end of the bucket with the offset $q' \leftarrow q+p-(k+1)2^l$. Processing of one prime is about 15-18 assembly instructions, which is approximately 5-6 clock cycles on today's processors, about the same time needed for the other prime to be loaded in the registers.

3.3.2 Broken bucket and loop unrolling

The well-known technique of loop unrolling can efficiently be used for processing primes-offset pairs. The core of the loop described above, which processes two primes terminates when the difference between the pointer from the beginning and end of the bucket becomes zero. With right `shift` and a logical `and` instructions, the quotient a and remainder r of this difference when divided by 2^h can be obtained (e.g. $h = 4$ or 5). Then the loop core can be executed a times in batches of 2^h runs, and afterward r times, thus reducing the time spent on checking if the difference is zero.

The loop unrolling of the broken bucket is a bit trickier, but manageable. Let δ_1 denote the difference between the beginning of the bucket and the end of the circle, and δ_2 the difference between the beginning of the circle and the end of the bucket. The difference used for unrolling, as described above, would be $\delta_1 + \delta_2$ but the when modifying the pointers after processing a prime, it would have to be checked, if it moves past the beginning or end of the circle (to jump to the other side). Instead, the unrolling is applied to $\min\{\delta_1, \delta_2\}$. Since it can't be predicted if the beginning or end pointer is going to be modified, the values of δ_1 and δ_2, the maximum, quotient a and reminder r have to be reevaluated after each batch, until one of the pointers "jump" to the other side. Then the bucket will no longer be broken, so the simpler unrolling described above can be applied.

4 Results

The program was run on (a single core of) two computers referred to by their names *lime* and *complab07*. The goal was to supersede the implementation found in the speed comparison chart of [4], but the results can not be compared directly, because of the differences in hardware. Our implementation, running on *lime* would come in 7th and *complab07* the 16th in the speed comparison chart, but with significantly slower memory.

e	lime	cl07	[a0F80]	[a0FF0]	[i06E8]	[a0662]
12.0	1.45	2.09	0.57	0.68	1.28	1.07
12.3	1.45	2.10	0.64	0.75	1.37	1.17
12.7	1.45	2.27	0.74	0.85	1.48	1.29
13.0	1.45	2.28	0.80	0.92	1.57	1.38
13.3	1.46	2.38	0.86	0.99	1.66	1.47
13.7	1.45	2.46	0.95	1.08	1.76	1.59
14.0	1.46	2.5	1.01	1.14	1.85	1.67
14.3	1.45	2.58	1.08	1.21	1.94	1.76
14.7	1.68	2.70	1.16	1.29	2.04	1.87
15.0	1.63	2.80	1.22	1.36	2.11	1.96
15.3	1.71	2.86	1.28	1.42	2.19	2.06
15.7	1.83	2.95	1.37	1.50	2.29	2.20
16.0	1.89	3.04	1.42	1.56	2.37	2.32
16.3	1.95	3.13	1.49	1.63	2.45	2.47
16.7	2.03	3.25	1.58	1.75	2.57	2.72
17.0	2.08	3.34	1.64	1.86	2.67	2.93
17.3	2.15	3.45	1.72	2.02	2.79	3.23
17.7	2.22	3.60	1.84	2.21	2.96	3.66
18.0	2.29	3.75	1.99	2.39	3.13	4.03
18.3	2.33	0	2.26	2.61	3.31	4.52

Table 1: Execution times in seconds for intervals of $10^9 \approx 2^{30}$ with the midpoint at 10^e

Compared to the 666MHz DDR2 memory of *lime*, the first five or so computers from the speed comparison chart have memory speeds of 800MHz and above, and with a better memory our implementation could probably compete

lime	2000MHz Intel Core2 Duo (E8200) model 23, stepping 6, DDR2 666MHz
cl07	1595MHz AMD Athlon64 3500+, model 47, stepping 2, DDR 200MHz
a0F80	2600MHz 6-Core AMD Opteron (Istanbul), model 8, stepping 0, DDR2
a0FF0	2210MHz Athlon64 (Winchester), model 15, stepping 0, DDR 333
i06E8	1830MHz T2400 (Core Duo), model 14, stepping 8, DDR2 533
a0662	1669MHz Athlon (Palomino), model 6, stepping 2, DDR 333

Table 2: The CPU and memory configurations of the computers used for measurements

better. But the real improvement can be seen, when running on older hardware, like *complab07*. With the slow memory of 200MHz, the plot in Figure 2, is much flatter and closer to the theoretical speed of $n \log \log n$ than for example the similar *i0662* with a faster 333MHz RAM.

It should also be noted, that the major part of execution is spent on sieving with medium primes and more optimization is desired out of that part of the algorithm. We also had some unexpected difficulties optimizing assembly code for the Intel processors, due to confusing documentation and slow execution of the `bts` (bit test set) instruction.

For *lime*, *complab07* and some computers from [4], Table 1 shows the time needed to sieve out an interval (represented by $2^{30} \approx 10^9$ bits, with its midpoint at 10^e). This data is plotted out in Figure 2: values of e are represented on the horizontal, execution times in seconds on the vertical axis.

5 Future work

The program was originally written for verifying the Goldbach conjecture, but only the sieve for generating the table of primes was finished and measured because that takes up the majority of the work for the verification. The completion of the verification application would be desirable. Also the current implementation supports sieving with primes only up to 32 bits, on current architectures, the implementation of sieving with primes up to 64 bits would not be a problem.

Most of the techniques described here (except the wheel algorithm), especially the use of circles and buckets can be applied for a wider range of sieving algorithms. For example, in [2] a similar attempt is made to exploit

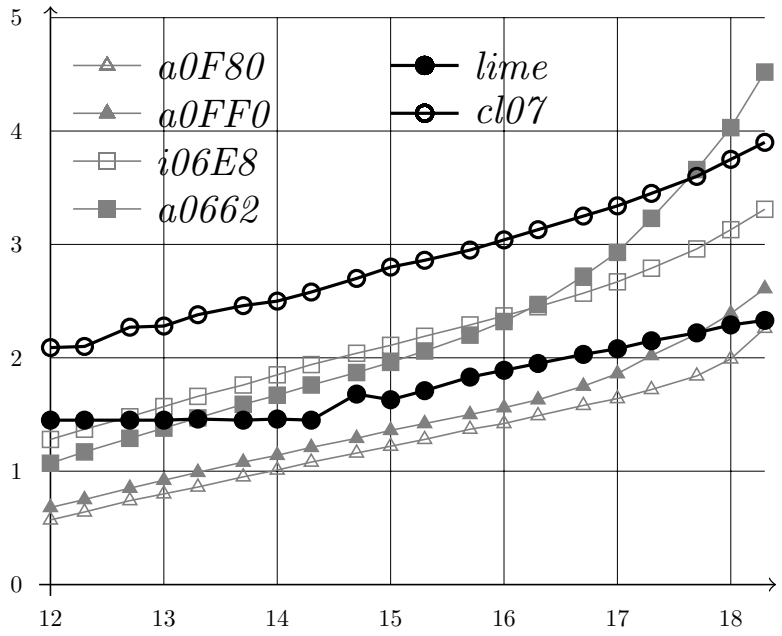

Figure 2: Speed comparison chart

the cache hierarchy, but the behavior of the large primes is more predictable with our method, and even an implementation for processors not designed for sieving algorithms is possible. Therefore the multiple polynomial quadratic sieve, on the Cell Broadband Engine Architecture[4], with 128K byte cache (i.e. Local Store) controlled by the user via DMA, can be implemented efficiently. Further performance can be gained by combining buckets and circles with parallel processing: sieving with different polynomials on different processors for MPQS-like algorithms, and a segment-wise pipeline-like processing for algorithms similar to the sieve of Eratosthenes.

Acknowledgements

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003).

[4]Cell Broadband Engine^{TM} is a trademark of Sony Computer Entertainment Incorporated

References

[1] D. M. Bressoud, *Factorization and Primality Testing*, Springer-Verlag, New York, 1989. ⇒205

[2] J. Franke, T. Keinjung, *Continued fractions and lattice sieving*, Proceeding SHARCS 2005, http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf. ⇒221

[3] A. Járai, E. Vatai, Cache optimized sieve, *8th Joint Conf. on Math and Comput. Sci. MaCS 2010, Selected papers,* Komárno, Slovakia, July 14–17, 2010, Novadat, 2011, pp. 249–256. ⇒205

[4] T. Oliveira e Silva, Goldbach conjecture verification, 2011, http://www.ieeta.pt/~tos/goldbach.html. ⇒205, 206, 220, 221

[5] P. Pritchard, Explaining the wheel sieve, *Acta Inform.*, **17,** 4 (1982) 477–485. ⇒217

Received: October 2, 2011 • Revised: November 8, 2011