The Segmented Sieve of Zakiya (SSoZ)

Jabari Zakiya © June 2014
jzakiya@gmail.com

Introduction

This paper describes an efficient and fast method to implement a Segmented Sieve of Zakiya (SSoZ). Its structure is based on two key design concepts: 1) a segment is a byte array consisting of B integral bytes of KB residues groups for a given Prime Generator (pg), and 2) the bits in each byte represent the residues (tracks) for the pg, and are processed the same within each byte. This structure creates very simple components that can be efficiently coded, that scale to any size prime generator for any size cpu that provides byte addressable memory. This allows the SSoZ to be memory independent of cpu size.

The SSoZ extends the Sieve of Zakiya (SoZ) to allow it to process very large numbers with a small memory footprint. It is also inherently realizable by parallel processing, utilizing multicore processors. This will allow the SSoZ to actualize the theoretically possible performance gains by using larger size Strictly Prime (SP) Generators

A Short Prime Generator Tutorial

All integers can be generated by an expression of form $\mathbf{I_n = mod*k+r}$, with r ε {0,1,2...mod-1}, k ≥ 0, which is a cyclic generator of modulo **mod** with residues **r**, with **mod** odd or even. For **mod** even, the expression can create a *prime generator* (**pg**) with properties useful for creating efficient prime sieves. To create a prime generator $\mathbf{P_n}$ eliminate the $\mathbf{r_i}$ which generate only nonprimes from an $\mathbf{I_n}$.

To illustrate, let's start with the integer generator I6 = 6*k+r, r ε {0,1,2,3,4,5}, k ≥ 0.
Identify from the integer list the residues **r** which only generate possible *prime candidates* (**pcs**).

I6 = 6*k+(0,1,2,3,4,5)	P6 = 6*k+(1,5)
r=0: 0, 6, 12, 18, 24, 30, 36, 42, 48...	r=0: 6*k+0 is divisible by 6. No primes.
r=1: 1, 7, 13, 19, 25, 31, 37, 43, 49...	r=1: 6*k+1 has no immediate factors. Has primes.
r=2: 2, 8, 14, 20, 26, 32, 38, 44, 50...	r=2: 6*k+2 = 2 x (3*k+1). No primes.
r=3: 3, 9, 15, 21, 27, 33, 39, 45, 51...	r=3: 6*k+3 = 3 x (2*k+1). No primes.
r=4: 4,10, 16, 22, 28, 34, 40, 46, 52...	r=4: 6*k+4 = 2 x (3*k+2). No primes.
r=5: 5,11, 17, 23, 29, 35, 41, 47, 53...	r=5: 6*k+5 has no immediate factors. Has primes.

So a pg's residues are coprime to its **mod**, i.e. their greatest common denominator is 1: gcd(**r**,**mod**) = 1.

For I6 only the residues 1 and 5 can generate prime numbers, i.e. r ε {1,5} for primes, which represent only 1/3 of the integer space that could be generated for mod=6. Thus, P6 = 6*k+(1,5) or (6*k±1) is a prime generator which generates all primes > 3, where here I designate the generator by its *modulus* (**mod**) value. However, because mod = 6 = 2*3 is the product of the first two primes, P6 is also what I call a **Strictly Prime** (**SP**) generator, designated as P3.

While any even **mod** will create a prime generator, we want to select values to create *"good"* ones. By *"good"* I mean 1) for a given number of residues (**rescnt**), the fraction (**rescnt/mod**) is the smallest possible, and 2) the generated primes are in order, i.e. except for a small number of consecutive *excluded primes* the generator skips over, all the other primes are generated with no missing ones.

Example of property 1.

P4 = 4*k+(1,3) and P6 = 6*k+(1,5) are prime generators with two residues (rescnt = 2). They both generate all the primes "in order" with P4 all primes > 2 and P6 all primes > 3. However P6 is "*better*" because its **rescnt/mod** ratio is 1/3 vs 1/2 for P4. This is the percentage of the number space needed to consider to find the primes up to some N.

Example of property 2.

P8 = 8*k+(1,3,5,7) is an in order generator as its residues have all the primes > 2 and < 8, but
P10 = 10*k+(1,3,7,9) is missing 5 as a residue, thus its excluded primes (2,5) are not in order.

Prime Generator Efficiency

The efficiency of a prime generator signifies the percentage of prime candidates (pcs) it needs to consider to find the primes up to an N, i.e. the ratio **(residues count/mod)**. The lower the percentage the smaller the number of pcs (number space) to consider, then the higher the generator efficiency.

Strictly Prime (SP) generators are the most efficient. Their moduli all have the form **mod = $\prod(p_i)$**, and their **residues count (rescnt) = $\prod(p_i\text{-}1)$** for primes 2,3,5... p_i. Table 1 shows the efficiency of the first few SP generators and a few selected in-order non-SP mod values.

Table 1.

P_n	P3	P5	P60	P120	P150	P180	P7	P11	P13	P17
modulus	6	30	60	120	150	180	210	2310	30030	510510
residues count (rescnt)	2	8	16	32	40	48	48	480	5760	92160
pc % of N	33.33	26.67	26.67	26.67	26.67	26.67	22.86	20.78	19.18	18.05
$ef(P_n)$ = ln(mod/res)	1.099	1.322	1.322	1.322	1.322	1.322	1.476	1.571	1.651	1.712

Distribution of Primes

For a given prime generator, the primes are uniformly distributed along each residue. Below is the tabulation of primes along the residues for P5 for the first 50 million primes it identifies.

Table 2.

Distribution of the 50 Million primes[4 – 50000003] = [7 – 982,451,809] for Prime Generator P5: Expected Mean (avg) = 50,000,000 / 8 = 6,250,000								
# primes	r=1	r=7	r=11	r=13	r=17	r=19	r=23	r=29
50M	6,249,269	6,250,543	6,250,224	6,249,930	6,250,078	6,249,248	6,250,326	6,250,382

This also means the number of nonprimes to be sieved are uniformly distributed along each residue too.

Modular Compliment Pairs

Residues come in **modular compliment (mc) pairs (mcp)**, e.g. for any mod: **mod = $r_i + r_j$**.
For every pg, each residue value has a modular compliment pair residue, and their sum equals **mod**.

Sieve of Zakiya

The SoZ [1][2] first computes the maximum number of prime candidates (pcs) up to N. To identify the nonprimes, each prime within a residues group (a column in Table 3) multiplies every member of that group. These rescnt products (first multiples) map distinctly to one residue track, within some residues group. From these values, along each residue track every primenth resgroup value is a prime multiple and is marked as nonprime, continuing until exceeding the resgroup value for N. In this manner for every prime ≤ sqrt(N) all the nonprimes up to resgroup of N are marked, leaving only the primes ≤ N.

SoZ Algorithm

To find all the primes ≤ N

1. choose a Prime Generator (PG), which has the form: Pn = mod*k+(1,r1..mod-1)
2. create list of prime candidates ≤ N of form: prime candidates = [mod*k+(r1..mod+1)], where k = 0,1.. represents a residues group of rescnt (# of residues) prime candidate values
3. set prime = first prime candidate (pc)
4. set primestep = prime*mod
5. perform sieve with prime value (can be done sequentially or in parallel):
    - multiply prime by each pc value of the residue group it belongs to
    - mark each of these product values in pc list as nonprime
    - from each product value, successively add primestep and mark values as nonprime until end of list
6. set prime to next unmarked pc value in list and repeat from 4 while prime ≤ sqrt(N)

Example:

Find the primes up to N = 541 (100th prime) using SP prime generator P5.

Table 3. Residues Table Illustration of Sieve of Zakiya for P5(541)

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
r1	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
r2	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
r3	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
r4	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
r5	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
r6	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
r7	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
r8	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

SoZ algorithm shown in Table 3.

1. choose prime generator P5 = 30*k+(1,7,11,13,17,19,23,29)

2. create prime candidates list of form: [30*k+(7,11,13,17,19,23,29,31)], k = 0,1,2..17

   prime candidates rows are: r1 = 30*k+7, r2 = 30*k+11, ... r7 = 30*k+29, r8 = 30*k+31
   where each column is a residues group of prime candidates for k = 0,1,2..17

3. set prime = 7 (first pc value)

4. set primestep = prime*mod = 7*30=210

5. perform sieve with prime = 7 (can be done sequentially or in parallel):

   multiply prime by each member of its residue group
   for prime = 7: 7*[7,11,13,17,19,23,29,31] => [49,77,91,119,133,161,203,217]
   starting at these values, successively add primestep=210 and mark values as nonprime
   along the residue track (row) they occur on, until end of list

6. repeat from 4 with successive unmarked candidates (11,13..23) as primes ≤ (sqrt(541)= 23)

Each color in the prime candidates list represents the nonprimes eliminated by the sieve primes, starting with 7 (blue), 11 (red), etc. For 7, [49,77,91,119,133,161,203,217] are the starting values in a row, then 210 (primestep) is successively added to mark additional nonprimes in these rows.

An easier equivalent way to do this is to mark each primenth column to the right of the starting location as nonprime for a given prime (color). Thus for 7 (blue), mark each 7th column to the right of the starting locations [49,77,91,119,133,161,203,217] as blue for those rows, then repeat for each successive prime, 11 (red), etc.

The primes are the 97 unmarked members in Table 3 plus the 3 primes p_j < (r1 = 7), i.e. 2,3,5.

To further highlight the power and efficiency of (especially SP) prime generators, for P5 only the base residues {7,11,13,17,19,23,29,31} are needed to sieve the primes up to 1368 = $37^2 - 1$. For P7 just the base residues (last is 211) can sieve all the primes up to 49728 = $223^2 - 1$.

The SoZ is structurally more efficient and faster than the classical Sieve of Eratosthenes because it only considers the reduced domain space associated with a given prime generator. The SoZ is also inherently parallelizable, thereby making a real implementation potentially even faster.

The SoZ also uses simple mathematical operations and minimal memory requirements. For ultimate memory efficiency, it can be implemented with a compressed memory model where each residue is represented as a bit in the prime candidates data array.

Coding the Sieve of Zakiya

The SoZ is a single algorithm that can accommodate any prime generator. For a given generator the variables mod holds its modulus and rescnt its residues count. The array residues[rescnt+1] holds the residues values plus the value (mod+1). The variable maxprms (or maxpcs) holds the number of pcs up to N, and the 1D boolean array prms[maxprms] represents each pc, which are all initialized to be prime.

We also need a way to determine the residue track (row position in Table 3) for each residue value. A hash (associative array) can be used to do this but the code uses the array pos[mod] because arrays are faster and available in more languages. This is initialized with the other variables before sieving.

To start, we initialize maxprms, prms[maxprms], pos[mod], find sqrt(N), and then perform the sieve.

The first multiples for each prime have form **– prime*(modk + ri) –** where **modk = mod*k**, and **k** is the resgroup index value for prime. We then translate this number into its residues table position where we start crossing off further multiples. There are two ways to code this; 1) we can work in the integer domain to identify the nonprimes, as shown in Table 3, or 2) we can work in the resgroup domain.

Working in the integer number domain we first compute **product = prime*(modk + ri)**. We then find its resgroup index, **ki = product/mod**, and its base residue value **rr = product % mod** and then use the **pos[]** translation array to get its restrack value **pos[rr]**. We now want to find its position in the residues table. Since the residues table is created as a 1D array representing all the residues groups residues ≤ N laid end-to-end, for some residue **r** in some resgroup **k** its residues table address is,
res_address = k*rescnt + pos[r]. Thus, the residues table address (location) for **product** is:
product_addr = ki*rescnt + pos[rr], and these constitute the **rescnt** first multiples for prime. Along **products** restracks we mark as nonprime every **prmstep = prime*rescnt** residue ≤ N.

Working in the group domain we identify the first multiples **– prime * (modk + ri) –** as follows:

$$\begin{gathered}
\mathrm{prime} * (\mathrm{modk} + \mathrm{ri}) \\
(\mathrm{modk} + \mathrm{r}) * (\mathrm{modk} + \mathrm{ri}) \\
\mathrm{modk} * \mathrm{modk} + \mathrm{modk} * (\mathrm{r} + \mathrm{ri}) + (\mathrm{r} * \mathrm{ri}) \\
\mathrm{modk} * (\mathrm{modk} + \mathrm{r} + \mathrm{ri}) + (\mathrm{r} * \mathrm{ri}) \\
(\mathrm{mod} * \mathrm{k}) * (\mathrm{prime} + \mathrm{ri}) + (\mathrm{r} * \mathrm{ri}) \\
\mathrm{mod} * [\underbrace{\mathrm{k} * (\mathrm{prime} + \mathrm{ri})}_{\mathrm{kk}}] + \underbrace{(\mathrm{r} * \mathrm{ri})}_{\mathrm{rr}}
\end{gathered}$$

These first multiples have form – **mod*kk + rr** – where **kk = k*(prime + ri)** and **rr = r*ri**. From **rr** we get its resgroup **kn = rr/mod** and residue track its on **pos[rr % mod]** for all the base residues (k=0) cross-products. Thus, for a prime in resgroup **k**, its **prime * (mod*k + ri)** products (first multiples) translate to the **(resgroup||restrack)** pair values **(k*(prime+ri) + kn||position[rr % mod])**. If we let the resgroup value be **ki = (k*(prime+ri) + kn)**, then the number of residue values to the first multiples (nonprimes) for each prime is **first_multiples = ki*rescnt + pos[rr % mod]**,

The advantage here is the size of the cross-products and other values stay small, and can be computed faster than the first method as N becomes large. Only **ki** and the residue address value approach the number size for N. This method is also more amenable for use in the SSoZ and parallel processing.

SoZ Prototype Code

Below is prototype code for implementing the SoZ using the parameters for the P5 prime generator. It shows both methods for determining the primes first multiples. This is a serial implementation of the SoZ, however the sieve can be performed in parallel to increase speed. Also for speed, the approximate number of primes ≤ N is first computed as the **primes[]** array length, which holds those primes values. The SoZ will be used in the SSoZ to find the primes and primes count ≤ sqrt(N).

Listing 1.

```
sozP5(N);
{
  mod = 30;
  rescnt = 8;
  residues = {1, 7, 11, 13, 17, 19, 23, 29, 31};

  val = N;                         // receive input value N

  num = val-1 | 1;                 // if value even number then subtract 1
  k=num/mod; modk = mod*k; r=1;    // kth residue group, base num value
  while (num >= modk+residues[r]) r++;  // find last pc position <= num
  maxprms = k*rescnt + r - 1;      // maximum number of prime candidates
  prms[] = new char(maxprms);      // array of prime candidates set as 'prime' (0)

  pos[mod];
  for (i=0; i < rescnt; i++) pos[residues[i]] = i-1;

  sqrtN = sqrt(num);
  modk=0; r=0; k=0;

  for (i=0; i < maxprms; i++){                 for (i=0; i < maxprms; i++){
    r++; if (r > rescnt) {r=1; modk += mod;}     r++; if (r > rescnt) {r=1; modk +=mod; k++}
    if (prms[i]) continue;                       if (prms[i]) continue;
    prime = modk + residues[r];                  res_r = res[r];
    if (prime > sqrtN) break;                    prime = modk + res_r;
    prmstep = prime*rescnt;                      if (prime > sqrtN) break;
    for (ri=1; ri <= rescnt; ri++){              prmstep = prime*rescnt;
      prod = prime*(modk+residues[ri]);          for (ri=1; ri <= rescnt; ri++){
      if (prod > num) break;                       prod = res_r * residues[ri];
      rr = prod % mod;                             rr = prod % mod;
      ki = prod / mod;                             ki = k*(prime+residues[ri]) + prod/mod;
      np = ki*rescnt + pos[rr];                    np = ki*rescnt + pos[rr];
      for (; np < maxprms; np += prmstep)          for (; np < maxprms; np += prmstep)
        prms[np]=true;                               prms[np]=true;
    }                                            }
    // the prms array now has identified all the positions for primes r1..N
    // now approximate primes array size; make greater than N/ln(N)
    max = (num/log(num) * 1.13)+8;  // compute approximate number of primes <= N
    primes[] = new [max];           // allocate mem at runtime
    pcnt = 0;                       // count of primes
    // extract prime numbers and count from prms into primes[] array
    modk=0; r=0;
    for (i=0; i < maxprms; i++){
      r++; if (r > rescnt) {r=1; modk += mod;}
      if (!prms[i]) primes[pcnt++] = modk + residues[r];
    }
  }
}
```

Segmented Sieve of Zakiya(SSoZ)

General methods to perform segmented sieves are presented in [4]. However, Daniel J. Bernstein (DJB) created a gem of a program **eratpeed.c** as part of his primegen code (circa 1999) found at his site [5]. This creates a segmented sieve using what I call (he didn't) the P5 generator as its basis. It is a very specialized implementation that was more of a proof of concept. I use this particular structure to form the basis for a generalized method that works with any SoZ prime generator for any size N, that is simpler, more efficient, scalable, faster, and ubiquitous to implement for virtually any cpu.

As with the SoZ, we conceptually create the residues table for an N for a prime generator, similar to Table 3. As N becomes larger, the number of resgroups (columns) in the table increases. Instead of serially sieving out the multiples for each prime ≤ sqrt(N), we slice the table into chunks of resgroups, and sieve out all the multiples of each prime from each slice, count the primes found (and optionally display or store them), then do the same with the next slice until the full residues table is processed.

SSoZ Algorithm

To find all the primes ≤ N

1. choose a Prime Generator (PG), which has the form: Pn = mod*k+(1,r1..mod-1)
2. set the maximum byte size B for the segment, usually the L1 or L2 cpu cache size
3. determine system parameters:
   - bprg, the number of bytes per residues group for the PG
   - KB, the number of residues groups of bprg size that fits into B bytes
   - B, reset the segment byte size to be just the KB * bprg bytes
   - seg[B], the segment array to have length of B bytes
   - Kmax, max number of residues groups for N
4. generate the r1..pn primes ≤ sqrt(N), and their pcnt number of primes
5. create and initialize next[rescnt * pcnt] array of first multiples resgroups for the primes
6. perform segmented sieve for all Kmax residues groups:
   - determine the byte size for the current segment
   - perform segmented sieve of all primes ≤ sqrt(N) on the segment
   - count the number of primes found in the segment
   - display and/or store the primes (optional)
   - repeat until Kmax residues groups processed
7. determine the total prime count and value of last prime ≤ N

The key computational components to perform are the generation of primes ≤ sqrt(N), the next[] array initialization, and the segmented sieve. I use the SoZ to generate the prime count and primes ≤ sqrt(N).

To generate the next[] array of first multiples resgroups for prime*(modk + ri) we'll perform this in the residues domain shown previously. next[] will be used to perform the segmented sieve, and will be updated for each prime for each residues track, to use with the next segment chunk. Tables 4-6 show visually how to extract the information from the residues table for P5 to initialize next[]. Later I show the generic method to computationally initialize next[] which works for any prime generator.

Table 4. Residues Tracks Table for P5(541).

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
rt0	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
rt1	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt2	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
rt3	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
rt4	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
rt5	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
rt6	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
rt7	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

Table 5. P5 base resgroup (k=0) cross-product values and (residue track (rt) || resgroup) pairs.

rt	res	7		11		13		17		19		23		29		31	
0	7	49		77		91		119		133		161		203		217	
		4	1	3	2	7	2	6	3	2	4	1	5	5	6	0	7
1	11	77		121		143		187		209		253		319		341	
		3	2	7	3	5	4	0	6	6	6	2	8	4	10	1	11
2	13	91		143		169		221		247		299		377		403	
		7	2	5	4	4	5	1	7	0	8	6	9	3	12	2	13
3	17	119		187		221		289		323		391		493		527	
		6	3	0	6	1	7	4	9	5	10	7	12	2	16	3	17
4	19	133		209		247		323		361		437		551		589	
		2	4	6	6	0	8	5	10	7	11	3	14	1	18	4	19
5	23	161		253		299		391		437		529		667		713	
		1	5	2	8	6	9	7	12	3	14	4	17	0	22	5	23
6	29	203		319		377		493		551		667		841		899	
		5	6	4	10	3	12	2	16	1	18	0	22	7	27	6	29
7	31	217		341		403		527		589		713		899		961	
		0	7	1	11	2	13	3	17	4	19	5	23	6	29	7	31

Table 5 shows the r1 * r2 cross-products translated into a k_row array, which lists the residue tracks they fall on, and a k_col array, which lists the residues groups they belong to.

```
           k_row[64]                         k_col[64]
     {4, 3, 7, 6, 2, 1, 5, 0,          {1, 2, 2, 3, 4, 5, 6, 7,
      3, 7, 5, 0, 6, 2, 4, 1,           2, 3, 4, 6, 6, 8,10,11,
      7, 5, 4, 1, 0, 6, 3, 2,           2, 4, 5, 7, 8, 9,12,13,
      6, 0, 1, 4, 5, 7, 2, 3,           3, 6, 7, 9,10,12,16,17,
      2, 6, 0, 5, 7, 3, 1, 4,           4, 6, 8,10,11,14,18,19,
      1, 2, 6, 7, 3, 4, 0, 5,           5, 8, 9,12,14,17,22,23,
      5, 4, 3, 2, 1, 0, 7, 6,           6,10,12,16,18,22,27,29,
      0, 1, 2, 3, 4, 5, 6, 7}           7,11,13,17,19,23,29,31}
```

Initializing next[] for P5
From the information in Table 5 we can code it to produce the first prime multiples shown in Table 4. We can visualize **next[]** having rescnt rows (residues tracks) of **pcnt** (number of primes) columns, i.e. **next[rescnt][pcnt]**. Every prime ≤ sqrt(N) represents a column in **next[]**. For each prime we store the resgroups values for the cross-products **prime*(modk +ri)** in the rows (residue tracks) they fall on.

The code prototype below shows an implementation starting from the first resgroup k=0, and provides a 1D and 2D version for the **next[]** and **k_row|col** arrays to illustrate both implementations.

Listing 2.

```
nextinit()
{
  char posn[30] = {
    0,7,0,0,0,0,0,0,0,0,0,1,0,2,0,0,0,3,0,4,0,0,0,5,0,0,0,0,0,6};

  char k_row[8][8] = {                         char k_row[64] = {
   {4, 3, 7, 6, 2, 1, 5, 0},                     4, 3, 7, 6, 2, 1, 5, 0,
   {3, 7, 5, 0, 6, 2, 4, 1},                     3, 7, 5, 0, 6, 2, 4, 1,
   {7, 5, 4, 1, 0, 6, 3, 2},                     7, 5, 4, 1, 0, 6, 3, 2,
   {6, 0, 1, 4, 5, 7, 2, 3},                     6, 0, 1, 4, 5, 7, 2, 3,
   {2, 6, 0, 5, 7, 3, 1, 4},                     2, 6, 0, 5, 7, 3, 1, 4,
   {1, 2, 6, 7, 3, 4, 0, 5},                     1, 2, 6, 7, 3, 4, 0, 5,
   {5, 4, 3, 2, 1, 0, 7, 6},                     5, 4, 3, 2, 1, 0, 7, 6,
   {0, 1, 2, 3, 4, 5, 6, 7}};                    0, 1, 2, 3, 4, 5, 6, 7};

  char k_col[8][8] = {                         char k_col[64] = {
   {1, 2, 2, 3, 4, 5, 6, 7},                     1, 2, 2, 3, 4, 5, 6, 7,
   {2, 3, 4, 6, 6, 8,10,11},                     2, 3, 4, 6, 6, 8,10,11,
   {2, 4, 5, 7, 8, 9,12,13},                     2, 4, 5, 7, 8, 9,12,13,
   {3, 6, 7, 9,10,12,16,17},                     3, 6, 7, 9,10,12,16,17,
   {4, 6, 8,10,11,14,18,19},                     4, 6, 8,10,11,14,18,19,
   {5, 8, 9,12,14,17,22,23},                     5, 8, 9,12,14,17,22,23,
   {6,10,12,16,18,22,27,29},                     6,10,12,16,18,22,27,29,
   {7,11,13,17,19,23,29,31}};                    7,11,13,17,19,23,29,31};

  for (j = 0; j < pcnt; ++j) {                 for (j = 0; j < pcnt; ++j) {
    prime = primes[j];                           prime = primes[j];
    k  = (prime-2)/mod;                          k  = (prime-2)/mod;
    row = posn[prime % mod];                     row = posn[prime % mod] * rescnt;
    for (i=0; i < rescnt; ++i)                   for (i=0; i < rescnt; ++i)
      next[k_row[row][i]][j] =                     next[k_row[row+i]*pcnt + j] =
        k*(prime+residues[i+1])+k_col[row][i];       k*(prime+residues[i+1])+k_col[row+i];
  }
}
```

For P5, mod=30 and rescnt=8. The array **posn[30]** (which could be a hash|associative array|dictionary, but arrays are standard and faster across languages) is used to translate a prime's residue value to its residue track value (0..rescnt-1). Because I order the resgroups from 7,11,...31 (r1...mod+1) I subtract 2 for **k = (prime-2)/mod** to make sure the last resgroup values (modk+1) are accounted for correctly.

This code creates a nice small/fast implementation for P5, but isn't applicable to other prime generators. Table 6 shows the first multiples resgroup values along each restrack for the first few primes in next[]. Following that I show a generic implementation for nextinit() (where next[rescnt*pcnt] is a 1D array) which works for any prime generator.

Table 6.   next[] array values for P5.

rt	res	List of resgroup values for the first prime multiples – prime*(modk+ri) – for the primes shown.																	
		7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73
0	7	7	6	8	6	8	22	22	7	75	64	70	64	104	104	75	203	182	192
1	11	5	11	7	7	18	5	18	11	65	83	67	67	65	96	83	185	215	187
2	13	4	8	13	16	4	8	16	13	60	72	87	92	72	92	87	176	196	221
3	17	2	2	12	17	14	14	12	17	50	50	84	95	86	84	95	158	158	216
4	19	1	10	5	9	19	17	10	19	45	80	61	73	93	80	99	149	210	177
5	23	6	4	4	10	10	23	6	23	72	58	58	76	107	72	107	198	172	172
6	29	3	6	9	3	6	9	29	29	57	66	75	57	75	119	119	171	186	201
7	31	2	3	2	12	11	12	27	31	52	55	52	82	82	115	123	162	167	162

Generic nextinit()

Listing 3.

```
nextinit()
{
  pos[mod]; for (i=0; i < rescnt; i++) {pos[residues[i]] = i-1;}; pos[1] = rescnt-1;

  for (j = 0; j < pcnt; ++j) {      // for the pcnt primes r1..sqrt(N)
    prime = primes[j];              // for each prime
    k = (prime-2)/mod;              // find the resgroup it's in
    r = (prime-2)%mod + 2;          // its base residue value
    for (i = 1; i <= rescnt; ++i) { // for each residue value
      prod = r * residues[i];       // create residues cross-product r*ri
      row  = pos[prod % mod]*pcnt;  // create residue track address
      next[row+j] = k*(prime+residues[i])+(prod-2)/mod; // resgroup for prime*(modk+ri)
    }
  }
}
```

The code body can be modified to produce the first regroup value starting at any Ki resgroup.

Listing 4.

```
  for (j = 0; j < pcnt; ++j) {      // for the pcnt primes r1..sqrt(N)
    prime = primes[j];              // for each prime
    k = (prime-2)/mod;              // find the resgroup it's in
    r = (prime-2)%mod + 2;          // its base residue value
    for (i = 1; i <= rescnt; ++i) { // for each residue value
      prod = r * residues[i];       // create residues cross-product r*ri
      row = pos[prod % mod]*pcnt;   // create residue track address
      ki = k*(prime+residues[i]) + (prod-2)/mod; // ki is resgroup for prime*(modk+ri)
      if (ki > Ki) ki -= Ki;        // if ki bigger than Ki reduce it by Ki
      else {                        // if not then translate value up to Ki
        ki = (Ki-ki)%prime;         // ki is now resgroups count <= Ki index
        if (ki != 0) ki = prime-ki; // if not 0 compute resgroups count past Ki
      }
      next[row+j] = ki;             // store first multiple resgroup starting at Ki
    }
  }
```

Segmented Sieve

For the primes ≤ sqrt(N), along each conceptualized residue track in the segment array, their multiples are sieved out by marking as nonprime each primenth resgroup value along each residue track starting at the resgroups values stored in next[] for each segment. Below are the P5 and generic functions.

Listing 5. segsieve() for P5

```
segsieve(Kn)
{                                      // for Kn number of resgroups in segment
  for (b = 0; b < Kn; ++b)             // for every byte in the segment
    seg[b] = 0;                        // set every byte bit to prime (0)
  for (r = 0; r < rescnt; ++r) {       // for each ith (of 8) residues for P5
    biti = 1 << r;                     // set the ith residue track bit mask
    row  = r*pcnt;                     // set address to ith row in next[]
    for (j = 0; j < pcnt; ++j) {       // for each prime <= sqrt(N) for restrack
      if (next[row+j] < Kn) {          // if 1st mult resgroup index <= seg size
        k = next[row+j];               // get its resgroup value
        prime = primes[j];             // get the prime
        for (; k < Kn; k += prime)     // while each primenth byte < segment size
          seg[k] |= biti;              // set ith residue in byte as nonprime
        next[row+j] = k - Kn;          // set 1st resgroup in next eligible segment
      }
      else next[row+j] -= Kn;          // do if 1st mult resgroup index > seg size
    }
  }                                    // count the primes in the segment
  for (s = 0; s < Kn; ++s)             // for the Kn resgroup bytes
    primecnt += pbits[seg[s] & 0xff];  // count the '0' bits as primes
}
```

Listing 6.   Generic segsieve()

```
segsieve(Kn)
{                                     // for Kn number of resgroups in segment
  for (b = 0; b < Kn*bprg; ++b)       // for every byte in the segment
    seg[b] = 0;                       // set every byte bit to prime (0)
  for (r = 0; r < rescnt; ++r) {      // for each residue track
    byti = r/8;                       // get its residues group byte[i]
    biti = 1 << r%8;                  // set the ith residue track bit mask
    row  = (byti*8 + r%8) * pcnt;     // set address to ith row|restrack in next[]
    for (j = 0; j < pcnt; ++j) {      // for each prime <= sqrt(N) for restrack
      if (next[row+j] < Kn) {         // if 1st mult resgroup index <= seg size
        k  = next[row+j];             // get its resgroup value
        ki = k*bprg + byti;           // convert it to a byte address in seg[]
        prime = prime[j];             // get prime, convert it to number of
        prmstep = prime * bprg;       // bytes to next primenth resgroup byte
        for (; k < Kn; k += prime) {  // for each primenth resgroup in segment
          seg[ki] |= biti;            // set ith residue in byte as nonprime
          ki += prmstep;              // set next primenth byte address in seg[]
        }
        next[row+j] = k - Kn;         // set 1st resgroup in next eligible segment
      }
      else next[row+j] -= Kn;         // do if 1st mult resgroup index > seg size
    }
  }                                   // count the primes in the segment
  for (s = 0; s < Kn*bprg; ++s)       // for the Kn resgroup bytes
    primecnt += pbits[seg[s] & 0xff]; // count the '0' bits as primes
                                      // subtract count of unused bits if rescnt not
  if (rescnt%8 != 0) primecnt -= Kn*(8-rescnt%8); // multiple of 8
}
```

Outputting and storing primes
To display (store) the segment primes, starting at resgroup index Ki for segment length Kn, we do:

Listing 7. Outputting primes for P5

```
printprms(Kn,Ki)
{                                              // starting at resgroup index Ki
  for (k = 0; k < Kn; ++k)                     // for Kn residues groups
    for (r = 0; r < 8; ++r)                    // for each residue bit in resgroup
      if (!(seg[k] & (1 << r)))                // if it's prime '0' show value
        print(" ",mod*(Ki+k) + residues[r+1]);
}
```

Listing 8. Generic Outputting primes (faster version)

```
printprms(Kn,Ki)
{
  modk = mod*Ki; b = 0;                        // starting at resgroup index Ki
  for (k = 0; k < Kn; ++k)                     // for Kn residues groups
    for (r = 0; r < rescnt; ++r)               // for each residue bit in resgroup
      if (!(seg[b + r/8] & (1 << r%8)))        // if restrack in seg[] byte[i] prime (0)
        print(" ",modk + residues[r+1]);       // print its value
    modk += mod; b += bprg;                    // set mod|start byte for next resgroup
}
```

To determine the primecount and lastprime ≤ N, after the primecount has been found for all the Kmax resgroups we have to determine how many of those primes are greater than N. We start with the last residue in the last resgroup in the last segment chunk and move backwards to the first marked prime. We then check if this prime value is ≤ N, and if so its the correct lastprime and the primecount belongs to it. If not, we decrement primecount and find the next smaller prime and check if its ≤ N. We continue doing this until we find the correct lastprime and primecount for N.

Listing 9. Generic lastprime()

```
lastprime(Kmax, Kn, num)
{
  lprime=0;                        // get last prime and primecnt <= val
  modk = mod*(Kmax-1);             // mod for last resgroup in last segment
  b = (Kn-1)*bprg;                 // num bytes to last resgroup in segment
  r = rescnt-1;                    // from last restrack in resgroup
  while (true) {                   // repeat until last prime determined
    if (!(seg[b + r/8] & 1 << r%8)) { // if restrack bit in byte[i] is prime
      lprime = modk+residues[r+1]; // determine the prime value
      if (lprime <= num) break;    // if <= num exit from loop with lprime
      primecnt--;                  // else decrement total primecnt
    }                              // decrement restrack, setup next resgroup
    r--; if (r < 0) {r=rescnt-1; modk -= mod; b -= bprg;} // if necessary
  }
}
```

Specifically for P5 we can do: **`b = Kn-1;`** and **`if (!(seg[b] & 1 << r))`** and **`b -= 1;`** (or **`b--;)`**

We now have all the components to construct the main() routine to perform the SSoZ.

SSoZ Main routine

For the SSoZ: 1) the segment byte size **B** consists of integral bytes of **KB** residues groups for a given prime generator, and 2) the byte bits represent the residues (tracks) for the prime generator. For an actual implementation first set **B** as the L1 or L2 cache size for the cpu being used and calculate:

```
  bprg = (rescnt-1)/8 + 1;       // number of bytes per resgroups
  KB = B/bprg;                   // number of resgroups per segment
  B  = KB*bprg;                  // number of bytes per segment
  seg[] = new char[B];           // create segment array of B bytes
```

These are global parameters, along with **mod**, **rescnt**, and **residues[rescnt+1]** for the prime generator.

Because P5 has 8 residues each resgroup is represented by one byte, i.e. **bprg = 1**, and thus **KB = B/1** with segment byte size **B = KB * 1**. Solely adjusting the size of **B** determines the segment's speed.

In the code, **next[]** is a 1D array of size **next[rescnt * pcnt]** elements and **Kn ≤ KB** is the current segment's byte size. The 1D segment array **seg[B]** of **B** bytes, can be visualized conceptually as a **seg[rescnt][KB]** 2D array. This applies to any generator. A byte array **pbits[256]** is used in segsieve() to translate **seg[B]** byte values into the number of primes bits (0s) in their binary representation.

Below is prototype code to perform the complete SSoZ.

Listing 10.

```
main()
{
  val = input;                    // value to find primes upto

  num = val-1 | 1;                // if val even subtract 1
  k=num/mod; modk = mod*k; r=1;   // num's resgroup and base num values
  while (num >= modk+residues[r]) r++; // find last pc position <= num
  maxpcs = k*rescnt + r-1;        // maximum number of prime candidates
  Kmax = (num-2)/mod + 1;         // number of resgroups to process

  sqrtN = sqrt(num);              // get integer sqrt of input value

  sozP7(sqrtN);                   // get pcnt and primes <= sqrt(n) using SoZ

  next[] = new[rescnt*pcnt];      // create the next[rescnt * pcnt] array
  nextinit();                     // load with first prime multiples resgroups

  primecnt = 4;                   // 2,3,5,7 the P7 excluded primes count
  Kn = KB;                        // set sieve resgroups size to segment size
  Ki = 0;                         // starting resgroup index for each segment

  for (; Ki < Kmax; Ki += KB) {   // for KB size resgroups slices up to Kmax
    if (Kmax-Ki < KB) Kn=Kmax-Ki; // determine resgroups size for last segment
    segsieve(Kn);                 // sieve primes for current segment
    printprms(Kn,Ki);             // print the primes for the segment (optional)
  }

  lastprime(Kmax,Kn,num);         // get last prime and primecnt <= val
}
```

Sieving over Ranges

To sieve over a range of values n1 to n2, first find the residues group each value is in, kn1 and kn2. We then must create the next[] array to be initialized starting at kn1, using the method shown in Listing 4, for the primes r1...pi ≤ sqrt(n2). To get the exact count of primes over the range, after doing segsieve() on the first segment, we have to find any primes in kn1 which are smaller than n1 and reduce the total primecount for the segment by that amount. This is done with similar code shown in Listing 9, but instead we start from the front of the segment and subtract the primes < n1 from the total found in the segment. We then proceed as normal for the resgroup count up thru kn2, and adjust at the end to find the exact count and last prime value for n2. To speed up the process, pick n1 to be the first value in a resgroup to eliminate the first segment primecount adjustment check.

This procedure will work using any prime generator for any size values for n1 and n2. The physical implementation requirements are, again, next[] must be able to hold the required numbers of resgroup values for the number size of kn2. However, for a given segment byte size B, sieving over different ranges of the same resgroup size should perform similarly, i.e. not be dependent on the n1|n2 values.

Twin Primes

Twin primes are pairs of primes that differ by only two, i.e. they are two consecutive odd numbers. Any prime generator can find them but some are more efficient at it. Efficient searches can be done using a set of reduced residues for prime generators with residues counts greater than 2.

For example, using the P5 residues sequence {7, 11, 13, 17, 19, 23, 29, 31} we see twin primes can only reside along the residues tracks for (11,13), (17,19) and (29,31). These six residues represent a number space of 6/30 = 1/5 = 0.20, or 20% of all integers. The twin primes occur when a pair of residues are prime within the same residues groups. A more efficient search implementation can be created by using a more efficient SP generator.

The SoZ residues sequence for P7 (48 residues, mod = 210) is shown below.

{11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,101,103,107,109,
113,121,127,131,137,139,143,149,151,157,163,167,169,173,179,181,187,191,193,197,199,209,211}

The residues pairs (11,13), (17,19), (29,31), (41,43), (59,61), (71,73), (101,103), (107,109), (137,139), (149,151), (167,169), (179,181), (191,193), (197,199) and (209,211) are the only ones to search along to find twin primes. These 30 residues (15 pairs) represent 30/210 = 1/7 = 0.142857, or 14.2857% of all integers to search through, about 25% (¼) more efficient than P5's 20%.

At the time of writing the largest known twin primes are $3756801695685 \cdot 2^{666669} \pm 1$ [6] which reside along the residues tracks P5(29,31) and P7(149,151).

Larger SP generators can search thru larger number spaces with increasing efficiency. A distributed network of search entities could be scaled to work at whatever efficiency it would be physically designed to accommodate, to create a custom twin prime generator.

Though not the focus of this paper, since all primes are uniformly distributed along a pg's residues, I would conjecture twin primes are uniformly distributed along a pg's twin pair residues as well.

Parallel SSoZ Implementations

The SSoZ (and SoZ) are inherently conducive to parallel implementations. There are various independent operations which can be performed in parallel within their subsections. This includes the SoZ prime generator, the nextinit() and segsieve() functions, and even parallel computation of segments in main(). An ultimate custom machine could perform the operations for each residues track in parallel, but I will show just a few coding examples here to give a taste of how to parallelize some subsections.

I will use the C++ OpenMP [15] compiler directives as a platform independent way to simply show parallel coding techniques. OpenMP allows you to compile the same source code with the compiler flag **-fopenmp** to produce parallel compiled code, or omit it and the code will compile as normal. This allows easy comparisons between versions.

As a start, the SoZ generator can be parallelized by adding just one OpenMP pragma line of code.

Listing 11.

```
// sieve to eliminate nonprimes from small primes prms array
for (i=0; i < maxprms; i++){
  r++; if (r > rescnt) {r=1; modk += mod; k++}
  if (prms[i]) continue;
  res_r = residues[r]
  prime = modk + res_r;
  if (prime > sqrtN) break;
  prmstep = prime*rescnt;

  #pragma omp parallel for private(product,nonprm)
  for (ri=1; ri < rescnt+1; ri++){
    product = res_r * residues[ri];
    nonprm  = rescnt*(k*(prime+residues[ri])+product/mod) + posn[product % mod];
    for (; nonprm < maxpcs; nonprm += prmstep) prms[nonprm]=true;
  }
}
```

Here I am performing the inner loop in parallel (which eliminates a prime's multiples). I do this, instead of the larger loop, because my system only has 4 threads to work with, and testing showed I got better results with this configuration.

Writing *"good"* and fast parallel code requires the consideration of many of variables. You really need to know what kind of hardware you will be running on, the language and compiler you intend to use, the number of threads|cores you have to use, the amount and type of memory, and potentially many other parameters that could affect the accuracy and performance of the code. This is assuming that the tasks you want to perform are amenable to parallel processing techniques.

In Listing 12, I show one way to split segsieve() into a parallel implementation that works using OpenMP. Here I perform parallel processing on a byte level within ressieve(). The ultimately faster method is to parallel process each residue track.

Listing 12.

```
void segsieve(uint Kn)
{
  uint j, r, bi, biti;
  uint k, ki, row, prime, prmstep;

  for (k = 0; k < Kn*bprg; ++k)
    seg[k] = 0;
  for (bi = 0; bi < bprg; ++bi) {
    for (r = 0; r < 8; ++r) {
      row = (bi*8 + r) * pcnt;
      biti = 1 << r;
      for (j = 0; j < pcnt; ++j) {
        if (next[row+j] < Kn) {
          k  = next[row+j];
          ki = k*bprg + bi;
          prime = primes[j];
          prmstep = prime * bprg;
          for (; k < Kn; k += prime) {
            seg[ki] |= biti;
            ki += prmstep;
          }
          next[row+j] = k - Kn;
        }
        else next[row+j] -= Kn;
      }
    }
  }
  // count the primes in the segment
  for (j = 0; j < Kn*bprg; ++j)
  primecnt += pbits[seg[j] & 0xff];
}
```

```
void ressieve(uint bi, uint Kn,)
{
  uint j, r, biti;
  uint k, ki, row, prime, prmstep;

  for (r = 0; r < 8; ++r) {
    row  = (bi*8 + r) * pcnt;
    biti = 1 << r;
    for (j = 0; j < pcnt; ++j) {
      if (next[row+j] < Kn)
        k  = next[row+j];
        ki = k*bprg + bi;
        prime = primes[j];
        prmstep = prime * bprg;
        for (; k < Kn; k += prime) {
          seg[ki] |= biti;
          ki += prmstep;
        }
        next[row+j] = k - Kn;
      }
      else next[row+j] -= Kn;
    }
  }
}


void segsieve(uint Kn)
{
  #pragma omp parallel
  {
  #pragma omp for
  for (uint b = 0; b < Kn*bprg; ++b)
    seg[b] = 0;

  #pragma omp for
  for (uint bi = 0; bi < bprg; ++bi)
    ressieve(bi,Kn);

  #pragma omp for reduction(+:primecnt)
  for (uint j = 0; j < Kn*bprg; ++j)
    primecnt += pbits[seg[j] & 0xff];
  }
}
```

Implementation Considerations
To achieve accurate results as N becomes large a few variables need to represent numbers of the size N. Following is a list of functions with variables that need special attention as N becomes large.

Global parameters:
The variable primecnt needs to generally represent values > 32-bits, or more, based on the size of N.
The variable lprime needs to represent values of size N.

In SoZ(n):
Because the sozp() finds the count and prime values ≤ sqrt(N) they can be represented as 32-bit values for all N up to 2**64 -1. Adjust appropriately for the largest size prime it needs to represent.

In nextinit():
The array next[] must store resgroups of Kmax size. The variable prime can be unsigned 32-bit value for N up to ~2**64.

In segsieve(Kn):
For speed, set k = next[row+j] as 32-bit value after checking if next[adr] < Kn, to prevent large uint to small uint conversion. The variable prime needs to represent the size of the largest prime ≤ sqrt(N).

In printprms(Kn,Ki):
The variables modk and Ki needs to represent values of size N.
If primes are to be stored, or displayed, they must represent values of size N.

In lastprime(Kn, Ki, num):
The variables Ki and num need to represent values of size N.
The variables modk and lprime need to represent values of size N.

In main():
The variables val, num, modk, Ki, and k must represent values of size N.
The variables Kmax and maxpcs grow slowly, but generally set to represent values the size of N.
The variable sqrtN needs to represent appropriate values up to the size of N.

The implementation considerations listed here are predicated on the fact that real software runs on hardware of fixed bit and memory size, and for compiled languages like C/C++, D, Fortran, etc it's necessary to declare variable types. This explicitly means for compiled languages you have to first decide on the range of values eligible to be processed before coding.

For interpreted untyped languages like Ruby, Python, etc, variable are automatically sized. Of course, these languages are generally slower than compiled languages, but are more flexible if you just want to see results without having to consider variable size details.

There are special libraries for languages, like the gmplib [7] for C/C++, to perform math with large, or even indefinite, sized numbers. These can be used where appropriate to achieve a user's end goals. One source that provides optimization techniques for writing C++ code can be found at [12]. For some interpreted languages, there are also techniques and libraries to work more efficiently and faster with large numbers.

Performance

Timing tests were performed on the following system:
Hardware: Lenovo V570 laptop, I5-2410M 2.3 GHz, 32K L1|256K L2 cache, 6 GB mem.
Software: Linux 3.14.4-pclos2.pae kernel, 4.7.2 g++ compiler, PCLinuxOS-2014 KDE 32-bit distro.
All program compiled with -O optimize flag: `$ g++ -O (-fopenmp) infilename -o outfilename`
For SSoZ, segment size B first set to 256K (256*1024 = 262,144) byte I5 L2 cache maximum.
(On *nix systems, the command **$ lscpu** will provide information on the cpu's hardware features.)

Table 7. All times in seconds.

ssozp5	ssozp7	ssozp7mp	N	Last Prime	Number of Primes
0.84	0.89	1.0	1,000,000,000	999,999,937	50,847,534
4.42	5.15	4.8	5,000,000,000	4,999,999,937	234,954,223
8.94	11.35	10.3	10,000,000,000	9,999,999,967	455,052,511
46.9	76.7	62.3	50,000,000,000	49,999,999,967	2,119,654,578
97.7	176.7	136.7	100,000,000,000	99,999,999,977	4,118,054,813
577.2	1299.2	895.5	500,000,000,000	499,999,999,979	19,308,136,142
1261.3	3070.3	2061.8	1,000,000,000,000	999,999,999,989	37,607,912,018

The table shows the times for single threaded versions for P5 and P7, and a parallel coded version for P7 using OpenMP. All versions accurately produce the number of primes, and the last prime, up to the values of N shown.

We see ssozp5 is fastest, performing at less than 1 second per billion numbers up to N = 100B. Both implementations of P7 are slower here, even though P7 is mathematically “better” than P5. However, the parallel version, ssozp7mp is significantly faster than ssozp7 as N increases.

Because P7 has 6 times more residues|bytes to process, initializing next[], and performing segsieve(), will take 6x more time than P5 when done serially. However, the serial P7 isn't 6x slower because it processes fewer residues groups than P5. We see the times for P7 significantly decrease when using parallel processing for some of the subsections. The times would further decrease, and surpass P5, with enough threads to perform a complete resgroup computation in one 'unit time'. A 'unit time' in this case would be the time needed to process a byte worth of residues in nextint() and segsieve(), which is what P5 does. The optimum system would be able to perform 'unit time' processing on a restrack level.

Thus, to get the benefits of using larger SP generators more hardware must be applied to them to perform them in parallel. This can be done readily using general purpose multi-core cpus, fast gpus, and soc systems. More customized systems can be built with networked computing elements, such as single board computers, and alterable hardware (field programmable arrays, etc) dedicated processing elements.

Finally, with newer software languages designed from scratch to perform parallel processing, and older languages incorporating parallel processing capabilities, it will become easier, and faster, to implement the SSoZ with larger SP generators.

Conclusion

Prime Generators in general, and Strictly Prime ones specifically, provide the basis for creating very efficient, fast, and scalable structures for finding prime numbers. By exploiting their mathematical properties, I show here how to implement small and simple components to form a segmented sieve architecture that can be implemented in software and/or hardware. The method can be easily modified to find primes over a range of integers, searching for Twin Primes, or finding other specific prime types. The method can also be used as an integer primality tester.

To fully achieve the possible performance gains from using larger Strictly Prime generators a system must be implemented using parallel processing. For a strictly software implementation this primarily amounts to how many independent processing units (cores|threads) can be used. For modern systems this could include using graphics processing units (gpu), and distributed networks, such as used for the Grand Internet Mersenne Prime Search (GIMPS) [11].

The greatest possible performance will be achieved through developing hardware assisted architectures which can be replicated to accommodate whatever sized generator desired. Because the components are so simple this is very easy to do using small single board computers (sbc) and/or alterable hardware (field programmable arrays, etc) or specifically designed parallel processors such as [13]. The ultimate hardware architecture would be a full custom design, with lots of fast memory and copious independent computing elements, to parallel process individual residue tracks within a system-on-chip (soc) design.

Hopefully the methodology shown here using prime generators will inspire others to investigate using them, in various forms and systems, to advance the field and means for fast methods for finding primes.

Acknowledgments
I want to express an enormous amount of gratitude and respect to the Open Software Software (OSS) community for providing the tools and knowledge which allowed me to easily, and cost effectively, pursue research in this area, write and test code, and even produce this paper. I particularly want to thank Yukihiro “Matz” Matsumoto, for creating the Ruby software language [14]. Ruby was my language of choice, providing me the tools, with idioms, that allowed me to easily write and test code, to check concepts and ideas, without having to think too hard about how to do it in software.

References

[1] Ultimate Prime Sieve – Sieve of Zakiya, Jabari Zakiya, 2008
http://www.scribd.com/doc/73384039/Ultimate-Prime-Sieve-Sieve-Of-Zakiya
[2] The Sieve of Zakiya, Jabari Zakiya, 2008
http://www.scribd.com/doc/73385696/The-Sieve-of-Zakiya
[3] Improved Primality Testing and Factorization in Ruby revised, Jabari Zakiya, June 2013
http://www.scribd.com/doc/150217723/Improved-Primality-Testing-and-Factorization-in-Ruby-revised
[4] An Introduction to Prime Number Sieves, Jonathan Sorensen, January 1990,
http://minds.wisconsin.edu/handle/1793/59248, TR909.pdf.
[5] primegen.c, Daniel J. Bernstein (DJB), http://cr.yp.to/primegen.html
[6] Twin Prime, https://en.wikipedia.org/wiki/Twin_prime
[7] The GNU Multiple Precision Arithmetic Library, https://gmplib.org/
[8] Tomás Oliveira e Silva, http://sweet.ua.pt/tos/software/prime_sieve.html
[9] Kim Walisch, http://primesieve.org/links.html,
[10] YAFU (Yet Another Factorization Utility), https://sites.google.com/site/bbuhrow/home
[11] Grand Internet Mersenne Prime Search, http://mersenne.org/
[12] Optimizing_cpp.pdf, Anger Fog, http://agner.org/optimize/#manuals
[13] Parallela Board, Adapteva, http://www.adapteva.com/, http://www.parallella.org/
[14] Ruby language, https://www.ruby-lang.org/en/
[15] OpenMP, http://openmp.org/wp/

Code Repository
All the code I have developed can be downloaded from this site:
www.4shared.com/dir/TcMrUvTB/sharing.html

I also welcome news and information on other implementations (code, benchmarks, systems) and uses of the SSoZ.

Code 1. C++ source code for ssozp5cpp.cpp.

```
/*
 This C++ source file will compile to an executable program to
 perform the Segmented Sieve of Zakiya (SSoZ) to find primes <= N.
 It is based on the P5 Strictly Prime (SP) Prime Generator.

 Prime Generators have the form:  mod*k + ri; ri -> {1,r1..mod-1}
 The residues ri are integers coprime to mod, i.e. gcd(ri,mod) = 1
 For P5, mod = 2*3*5 = 30 and the number of residues are
 rescnt = (2-1)(3-1)(5-1) = 8, which are {1,7,11,13,17,19,23,29}.

 Adjust segment byte length parameter B (usually L1|l2 cache size)
 for optimum operational performance for cpu being used.

 On Linux use -O compiler flag to compile for optimum speed:

 $ g++ -O ssozp5cpp.cpp -o ssozp5cpp

 Then run executable: $ ./ssozp5cpp <cr>, and enter value for N.
 As coded, input values cover the range: 7 - 2^64-1

 Related code, papers, and tutorials, are downloadable here:

 http://www.4shared.com/folder/TcMrUvTB/_online.html

 Use of this code is free subject to acknowledgment of copyright.
 Copyright (c) 2014 Jabari Zakiya -- jzakiya at gmail dot com
 Version Date: 2014/05/22

 This code is provided under the terms of the
 GNU General Public License Version 3, GPLv3, or greater.
 License copy/terms are here:  http://www.gnu.org/licenses/
*/



#include <cmath>
#include <vector>
#include <cstdlib>
#include <iostream>
#include <stdint.h>

using namespace std;

typedef uint64_t uint64;

char pbits[256] = {
 8,7,7,6,7,6,6,5,7,6,6,5,6,5,5,4,7,6,6,5,6,5,5,4,6,5,5,4,5,4,4,3
,7,6,6,5,6,5,5,4,6,5,5,4,5,4,4,3,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2
,7,6,6,5,6,5,5,4,6,5,5,4,5,4,4,3,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2
,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2,5,4,4,3,4,3,3,2,4,3,3,2,3,2,2,1
,7,6,6,5,6,5,5,4,6,5,5,4,5,4,4,3,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2
,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2,5,4,4,3,4,3,3,2,4,3,3,2,3,2,2,1
,6,5,5,4,5,4,4,3,5,4,4,3,4,3,3,2,5,4,4,3,4,3,3,2,4,3,3,2,3,2,2,1
,5,4,4,3,4,3,3,2,4,3,3,2,3,2,2,1,4,3,3,2,3,2,2,1,3,2,2,1,2,1,1,0
};

char residues[9] = {1, 7, 11, 13, 17, 19, 23, 29, 31};
```

```
// Global parameters
uint B;                // segment byte size
uint KB;               // segment resgroup size
uint mod = 30;         // prime generator mod value
uint rescnt = 8;       // number of residues for prime generator
uint pcnt;             // number of primes from r1..sqrt(N)
uint64 primecnt;       // number of primes <= N
uint64 *next;          // pointer to array of primes first multiples
uint *primes;          // pointer to array of primes r1..pi <= sqrt(N)
char *seg;             // pointer to seg[B] segment byte array

void sozP7(uint val)
{
  int md = 210;
  int rscnt = 48;
  int res[49] = {
     1, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67
  , 71, 73, 79, 83, 89, 97,101,103,107,109,113,121,127,131,137,139
  ,143,149,151,157,163,167,169,173,179,181,187,191,193,197,199,209,211};

  int posn[210];
  for (int i=0; i < rscnt; i++) posn[res[i]] = i-1;
  uint i, r, modk;

  uint num = val-1 | 1;             // if value even number then subtract 1
  uint k=num/md; modk = md*k; r=1;  // kth residue group, base num value
  while (num >= modk+res[r]) r++;   // find last pc position <= num
  uint maxprms = k*rscnt + r - 1;   // maximum number of prime candidates
  vector<char> prms(maxprms);       // prime candidates array, default all '0'|prime

  uint sqrtN = (uint) sqrt((double) num);
  modk=0; r=0; k=0;

  // sieve to eliminate nonprimes from small primes prms array
  for (i=0; i < maxprms; i++){
    r++; if (r > rscnt) {r=1; modk += md; k++;}
    if (prms[i]) continue;          // if prime candidate is a prime
    uint res_r = res[r];            // keep its residue value
    uint prime = modk + res_r;      // create its numerical value
    if (prime > sqrtN) break;       // if > sqrtN we're finished, exit loop
    uint prmstep = prime * rscnt;   // if not, create its primestep value
    for (int ri=1; ri < (rscnt+1); ri++){ // for each residue value ri
      uint prod = res_r * res[ri];  // compute the residues cross-product
      // compute prime's 1st multiple position for ri, then eliminate multiples
      uint nonprm = (k*(prime + res[ri]) + prod/md)*rscnt + posn[prod % md];
      for (; nonprm < maxprms; nonprm += prmstep) prms[nonprm]=true;
    }
  }
  // the prms array now has all the positions for primes r1..N
  // approximate primes array size; make greater than N/ln(N)
  uint max = (double) ((num/log( (double) num) * 1.13)+8);
  primes = new uint[max];           // allocate mem at runtime
  pcnt = 1;                         // account for r1 for P5
  primes[0] = 7;                    // r1 prime for P5
  // extract prime numbers and count from prms into primes array
  modk=0; r=0;
  for (i=0; i < maxprms; i++){
    r++; if (r > rscnt) {r=1; modk +=md;}
    if (!prms[i]) primes[pcnt++] = modk + res[r];
  }
}
```

```
void nextinit()
{
  char pos[30] = {
    0,7,0,0,0,0,0,0,0,0,0,1,0,2,0,0,0,3,0,4,0,0,0,5,0,0,0,0,0,6};

  char k_row[64] = {
    4, 3, 7, 6, 2, 1, 5, 0
  , 3, 7, 5, 0, 6, 2, 4, 1
  , 7, 5, 4, 1, 0, 6, 3, 2
  , 6, 0, 1, 4, 5, 7, 2, 3
  , 2, 6, 0, 5, 7, 3, 1, 4
  , 1, 2, 6, 7, 3, 4, 0, 5
  , 5, 4, 3, 2, 1, 0, 7, 6
  , 0, 1, 2, 3, 4, 5, 6, 7
  };

  char k_col[64] = {
    1, 2, 2, 3, 4, 5, 6, 7
  , 2, 3, 4, 6, 6, 8,10,11
  , 2, 4, 5, 7, 8, 9,12,13
  , 3, 6, 7, 9,10,12,16,17
  , 4, 6, 8,10,11,14,18,19
  , 5, 8, 9,12,14,17,22,23
  , 6,10,12,16,18,22,27,29
  , 7,11,13,17,19,23,29,31
  };

  // for each prime store resgroup on each restrack for prime*(modk+ri)
  for (uint j = 0; j < pcnt; ++j) {   // for the pcnt primes r1..sqrt(N)
    uint prime = primes[j];           // for each prime
    uint64 k  = (prime-2)/mod;        // find the resgroup it's in
    uint row = pos[prime % mod] * rescnt; // set row start for k_row|col tables
    for (uint r=0; r < rescnt; ++r)   // for each residue store 1st multiple resgroup
      next[k_row[row+r]*pcnt + j] = k*(prime + residues[r+1]) + k_col[row+r];
  }
}

void segsieve(uint Kn)
{                                     // for Kn resgroups in segment
  for (uint b = 0; b < Kn; ++b)       // for every byte in the segment
    seg[b] = 0;                       // set every byte bit to prime (0)
  for (uint r = 0; r < rescnt; ++r) { // for each ith (of 8) residues for P5
    uint biti = 1 << r;               // set the ith residue track bit mask
    uint row  = r*pcnt;               // set address to ith row in next[]
    for (uint j = 0; j < pcnt; ++j) { // for each prime <= sqrt(N) for restrack
      if (next[row+j] < Kn) {         // if 1st mult resgroup index <= seg size
        uint k = next[row+j];         // get its resgroup value
        uint prime = primes[j];       // get the prime
        for (; k < Kn; k += prime)    // for each primenth byte < segment size
          seg[k] |= biti;             // set ith residue in byte as nonprime
        next[row+j] = k – Kn;         // set 1st resgroup in next eligible segment
      }
      else next[row+j] -= Kn;         // do if 1st mult resgroup index > seg size
    }
  }
                                      // count the primes in the segment
  for (uint s = 0; s < Kn; ++s)       // for the Kn resgroup bytes
    primecnt += pbits[seg[s] & 0xff]; // count the '0' bits as primes
}
```

```
void printprms(uint Kn, uint64 Ki)
{
  // Extract and print the primes in each segment:
  // recommend piping output to less: ./ssozpxxx | less
  // can then use Home, End, Pgup, Pgdn keys to see primes
  for (uint k = 0; k < Kn; ++k)       // for Kn residues groups|bytes
    for (uint r = 0; r < 8; ++r)      // for each residue|bit in a byte
      if (!(seg[k] & (1 << r)))       // if it's '0' it's a prime
        cout << " " << mod*(Ki+k) + residues[r+1];
  cout << "\n";
}

main()
{
  cout << "Enter number value: ";
  uint64 val;                      // find primes <= val (7..2^64-1)
  cin >> val;

  B = 262144;                      // L2D_CACHE_SIZE 256*1024 bytes, I5 cpu
  KB = B;                          // number of segment resgroups
  seg = new char[B];               // create segment array of B bytesize

  cout << "segment has "<< B << " bytes and " << KB << " residues groups\n";

  int r;
  uint64 k, modk;

  uint64 num = val-1 | 1;          // if val even subtract 1
  k=num/mod; modk = mod*k; r=1;    // kth residue group, base num value
  while (num >= modk+residues[r]) r++; // find last pc position <= num
  uint64 maxpcs =k*rescnt + r-1;   // maximum number of prime candidates
  uint64 Kmax = (num-2)/mod + 1;   // maximum number of resgroups for val

  cout <<"prime candidates = "<< maxpcs <<"; resgroups = "<< Kmax << endl;

  uint sqrtN = (uint) sqrt((double) num);

  sozP7(sqrtN);                    // get pcnt and primes <= sqrt(nun)

  cout << "create next["<< rescnt << "x" << pcnt << "] array\n";

  next = new uint64[rescnt*pcnt];  // create the next[] array
  nextinit();                      // load with first nonprimes resgroups

  primecnt  = 3;                   // 2,3,5 the P5 excluded primes count
  uint   Kn = KB;                  // set sieve resgroups size to segment size
  uint64 Ki = 0;                   // starting resgroup index for each segment

  cout << "perform segmented SoZ\n";

  for (; Ki < Kmax; Ki += KB) {    // for KB size resgroup slices up to Kmax
    if (Kmax-Ki < KB) Kn=Kmax-Ki;  // set sieve resgroups size for last segment
    segsieve(Kn);                  // sieve primes for current segment
    //printprms(Kn,Ki);            // print primes for the segment (optional)
  }

  uint64 lprime=0;                 // get last prime and primecnt <= val
  modk = mod*(Kmax-1);             // mod for last resgroup in last segment
  uint b = Kn-1;                   // num bytes to last resgroup in segment
  r = rescnt-1;                    // from last restrack in resgroup
  while (true) {                   // repeat until last prime determined
```

```
      if (!(seg[b] & 1 << r)) {       // if restrack in byte[i] is prime
        lprime = modk+residues[r+1]; // determine the prime value
        if (lprime <= num) break;    // if <= num exit from loop with lprime
        primecnt--;                  // else reduce total primecnt
      }                              // reduce restrack, setup next resgroup if necessary
      r--; if (r < 0) {r=rescnt-1; modk -= mod; b--;}
    }

    cout << "last segment = " << Kn << " resgroups; segment iterations = " << Ki/KB << "\n";
    cout << "last prime (of " << primecnt << ") is " << lprime << endl;

    delete[] next; delete[] primes;
    return 0;
  }
```