

CP 331 Assignment 1: Finding Prime Gaps
Group 2

Hilts, Vaughan
hilt2740@mylaurier.ca, 120892740

Morouney, Robert
moro1422@mylaurier.ca, 069001422

Rusu, David
rusu0260@mylaurier.ca, 131920260

February 2nd 2016

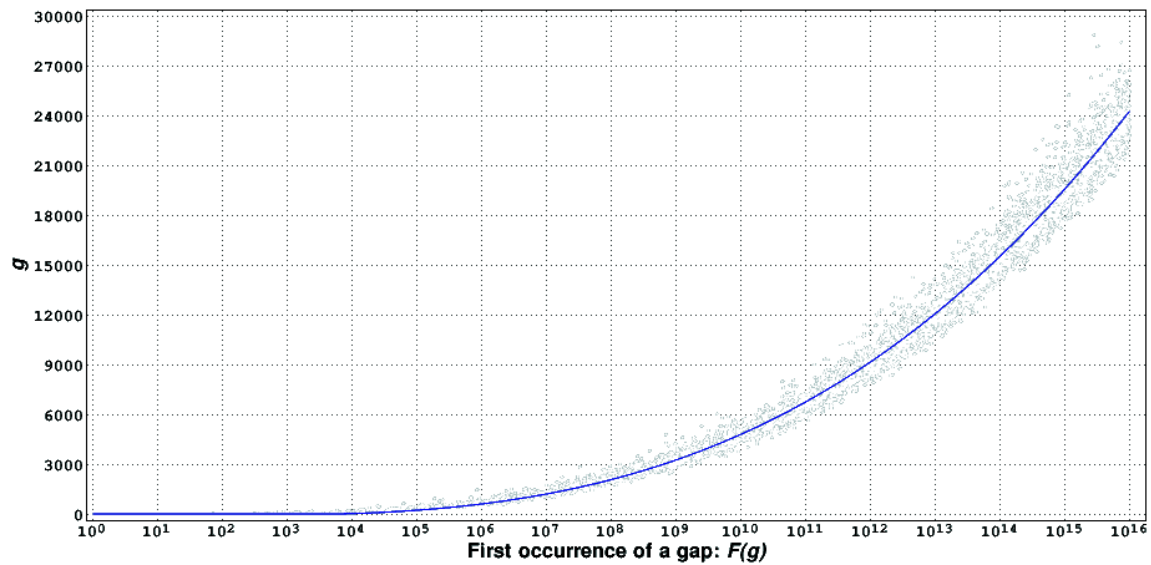
Contents

0.1	Abstract	1
0.2	Introduction	2
0.3	Trial Division	3
0.3.1	Generating the sieve	3
0.3.2	Reverse Sieve!	4
0.3.3	Results	5
0.4	Progressive Gap Improvement Method	6
0.4.1	Pseudo Code	6
0.4.2	Actual Code	7
0.4.3	Results	8
0.5	The Quest to Find the Maximal Gap Below 10^{13}	9
0.5.1	The Segmentation	10
0.5.2	The Sieve	13
0.5.3	The Results	14
0.6	Full Code	16
0.6.1	Trial Division	16
0.6.2	Interval Method	20
0.6.3	Segmented Sieve	23

0.1 Abstract

Upper Bound	Maximum Gap	Prime Preceeding	Prime After	Speed (s)	Cores
1e6	114	492113	492227	1.21	8
1e9	282	436273009	436273291	6.10	8
1e12	545	738832927927	738832928467	561.69	8
1e13	674	7177162611713	7177162612387	4027.23	128

In the following paper we explore 3 methods for finding maximal prime gaps below a given number. Our fastest method, a segmented sieve, is presented in the table above. All of our tests were run on SHARCNET's Orca cluster specifically targeting the Intel Xenon 2600 series CPU's within the cluster.



0.2 Introduction

Finding prime gaps proved to be far more difficult than first anticipated. Over the course of two weeks our group has experimented with various methods of finding prime numbers and then calculating the gaps between them. Initially we hypothesized that our biggest challenge would be work distribution, It wasn't. In reality we faced two different major challenges for this assignment:

1. Time complexity
2. Memory constraints

Because striking a balance between the two was so difficult we ended up with two distinct solutions which we will explore in this report. For each method there will be a brief overview of the algorithm followed up by statistical data from testing. Though code will be listed throughout the report it should be noted that all of the code can be found in its entirety in the final section. We will define N to be the upper bounds of the range which is searched for either primes or the gaps between them. We will also define the prime counting function $\pi(N)$ and the maximal gap beneath N to be $g(n)$

"Time is free, but it's priceless. You can't own it, but you can use it. You can't keep it, but you can spend it. Once you've lost it you can never get it back."

Harvey MacKay

0.3 Trial Division

The first algorithm uses a modified sieve technique which does not store any primes $> \sqrt{N}$ in memory except for the primes used to find the maximum gap. Once a gap is found to be larger than the current maximum, the maximum gap and its associated primes are modified with the larger numbers.

Algorithm steps to find max_gap less than N

1. Calculate \sqrt{N} since all numbers $\geq \sqrt{N}$ are composites of primes $< \sqrt{N}$
2. Find all primes under \sqrt{N} by looping through all odd $(2k + 1)$ numbers and checking each one with `mpz_probab_prime_p(...)`
3. Use the primes found as a basis for trial division for all numbers $> \sqrt{N}$. This is accomplished by looping through all of the odd integers $> \sqrt{N}$ and for each integer loop through all the primes checking if any are divisible.

With this method we have estimated the complexity at $O(\text{mpz_probab_prime_p} * \sqrt{N}) + O(N * \pi(\sqrt{N}))$ where $\pi(..)$ is the prime counting function. The thinking behind using this algorithm was that we could keep the memory used very low. We estimate it uses about $\pi(\sqrt{N}) * 64$ bits to store the list. This algorithm would also have the effect of keeping the work division simple since the same maximum complexity would be seen across all numbers $> \sqrt{N}$ since each number has to be checked against the same list.

0.3.1 Generating the sieve

The code we used to generate and store the primes under \sqrt{N} is:

```

1 uint64_t gen_sieve(int64_t N, uint64_t* primes) {
2     mpz_t mpz_num;
3     mpz_init(mpz_num);
4     unsigned int top, sq_top, totalPrimes = 0, n = 0;
5     top = sqrt(N) + 1;
6     sq_top = sqrt(top) + 1;
7     primes[n++] = 2;
8
9     unsigned char isPrime;
10    unsigned int i, j;
11
12    for (i = 3; i < sq_top; i += 2) {
13        mpz_set_ui(mpz_num, i);
14        isPrime = mpz_probab_prime_p(mpz_num, 15);
15        if (isPrime > 0) {
16            primes[n++] = i;
17        }
18    }
19
20    if (sq_top % 2 == 0) sq_top++; // Primes and even numbers don't get along
21
22    unsigned char notPrime = 0;
23
24    for (i = sq_top; i < top; i += 2) {

```

```

25     for (j = 0; j < n; j++) {
26         if (i % primes[j] == 0) {
27             notPrime = 1;
28             break;
29         }
30     }
31     if (notPrime == 0) {
32         primes[n++] = i;
33     } else
34         notPrime = 0;
35 }
36 return n;
37 }

```

Listing 1: Creating the Sieve

After the primes have been generated we can move to the next step which is the trial division.

0.3.2 Reverse Sieve!

In the code below you can see that we attempt to divide each odd number between \sqrt{N} and N with every prime, exiting on any successes.

```

1 int sieve_primes(uint64_t* primes, unsigned int num_primes,
2                 uint64_t start, uint64_t end, uint64_t*
3                 lprime_hprime_lgprime_hgprime_gap) {
4     uint64_t last_prime, cur_prime, first_prime, gap, max_gap, low_gap_prime,
5     high_gap_prime;
6
7     time_t startTime = time(NULL);
8
9     // Only care about odd sizes
10    if (start % 2 == 0) {
11        start++;
12    }
13
14    cur_prime = 0;
15    max_gap = 0;
16
17    printf("Start: %" PRIu64 "\n", start);
18    printf("End: %" PRIu64 "\n", end);
19
20    int notPrime = 0;
21    uint64_t i, j;
22    for (i = start; i < end; i += 2) {
23        for (j = 0; j < num_primes; j++) {
24            if ((i % primes[j]) == 0) {
25                notPrime = 1;
26                break;
27            }
28        }
29        if (notPrime == 0) {
30            last_prime = cur_prime;
31            cur_prime = i;
32            if (last_prime != 0) {

```

```

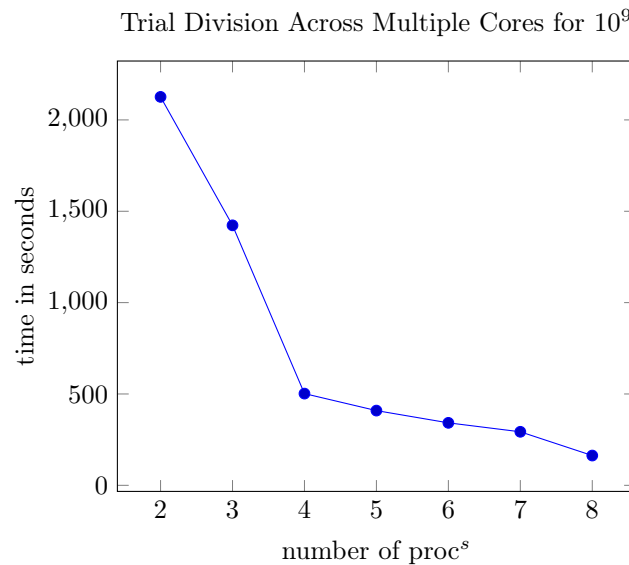
33 gap = cur_prime - last_prime;
34 if (gap > max_gap) {
35     max_gap = gap;
36     low_gap_prime = last_prime;
37     high_gap_prime = cur_prime;
38 }
39     } else {
40 first_prime = cur_prime;
41     }
42     } else {
43     notPrime = 0;
44 }
45 }
46
47 time_t finish = time(NULL) - startTime;
48 printf("Sieve finished in %ld s\n", finish);
49
50 lprime_hprime_lgprime_hgprime_gap[0] = first_prime;
51 lprime_hprime_lgprime_hgprime_gap[1] = cur_prime;
52 lprime_hprime_lgprime_hgprime_gap[2] = low_gap_prime;
53 lprime_hprime_lgprime_hgprime_gap[3] = high_gap_prime;
54 lprime_hprime_lgprime_hgprime_gap[4] = max_gap;
55 return max_gap;
56 }

```

Listing 2: Sieving the Primes

0.3.3 Results

The following graph shows the relationship between the total number of cores and the run time of the program. From the graph below it is clear that this method takes advantage of the multiple cores and benefits from the addition of more processors.



Despite this method's ability to gain speed from the addition of more cores it has very little going for it. While it is possible to find all of the primes $< 10^{12}$ with this method, it is not feasible due to the vast amount of time it would take.

“Our wretched species is so made that those who walk on the well-trodden path always throw stones at those who are showing a new road.”

Voltaire

0.4 Progressive Gap Improvement Method

While ultimately we did not use this method, we have included it due to its elegance. We understood that the current and best method for finding primes is with brute force, not believing everything we read on the internet we decided to test it out for ourselves. The algorithm works like this:

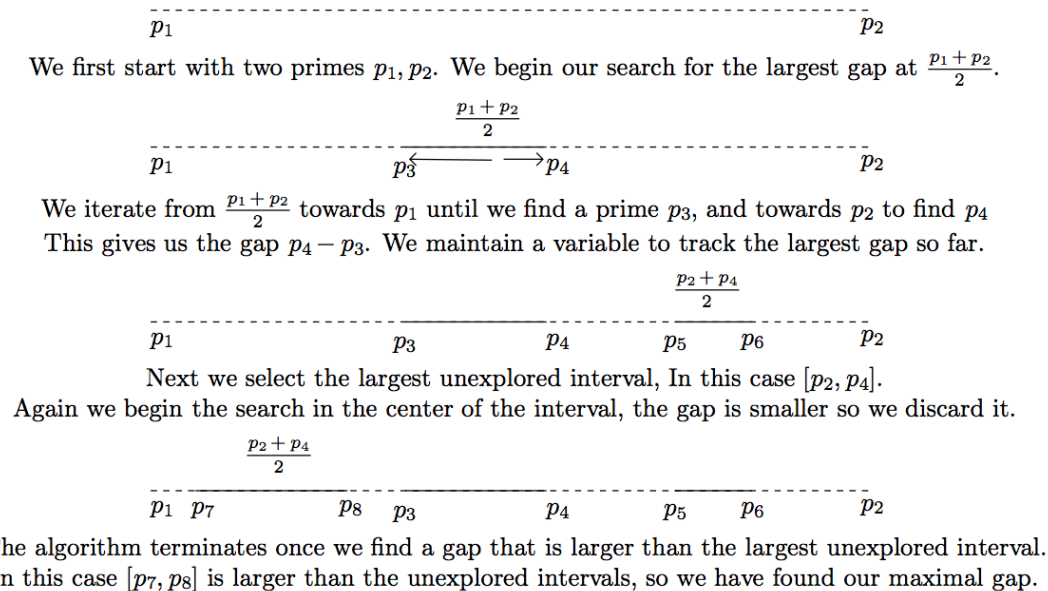


Figure 1. Intuition behind the Progressive Gap Improvement approach

0.4.1 Pseudo Code

```

1 PQ = PriorityQueue()
2 largest_gap = 0
3 while (True):
4     interval = PQ.pop()
5     if size(interval) < largest_gap:

```

```

6     return largest_gap
7     mid = size(interval)/2 + interval.start
8     gap_high_prime = mid;
9
10    while !isPrime(gap_high_prime):
11        gap_high_prime += 1
12
13    gap_low_prime = max(mid-1, interval.start)
14
15    while !isPrime(gap_low_prime):
16        gap_low_prime -= 1
17
18    largest_gap = max(largest_gap, gap_high_prime - gap_low_prime)
19    if gap_low_prime - interval.start > largest_gap:
20        PQ.push(Interval(interval.start, gap_low_prime))
21    if interval.end - gap_high_prime > largest_gap:
22        PQ.push(Interval(gap_high_prime, interval.end))

```

Listing 3: Pseudo Code

0.4.2 Actual Code

```

1 int search(uint64_t *primes, uint32_t n_primes, uint64_t start, uint64_t end) {
2     struct node *largest_interval = NULL;
3     insert_interval(&largest_interval, start, end);
4
5     uint64_t largest_gap = 0;
6     for (int k = 0;; k++) {
7         if (largest_interval == NULL) {
8             return largest_gap;
9         }
10        struct node *interval = pop(&largest_interval);
11        uint64_t interval_size = interval->end - interval->start;
12
13        if (largest_gap >= interval_size) {
14            return largest_gap;
15        }
16
17        uint64_t mid = (interval->end - interval->start) / 2 + interval->start; // we do
18        mid point calc this way to avoid overflow
19        uint64_t next_prime;
20        for (next_prime = mid; next_prime <= interval->end; next_prime++) {
21            if (is_prime(primes, n_primes, next_prime)) {
22                break;
23            }
24
25            uint64_t prev_prime;
26            for (prev_prime = max(mid-1, interval->start); prev_prime > interval->start;
27                prev_prime--) {
28                if (is_prime(primes, n_primes, prev_prime)) {
29                    break;
30                }
31
32                if (prev_prime - interval->start > largest_gap) {
33                    insert_interval(&largest_interval, interval->start, prev_prime);
34                }

```



```

35     if (interval->end - next_prime > largest_gap) {
36         insert_interval(&largest_interval, next_prime, interval->end);
37     }
38
39     uint64_t my_size = next_prime - prev_prime;
40     if (my_size > largest_gap) {
41         largest_gap = my_size;
42     }
43 }
44 }

```

Listing 4: Interval method

0.4.3 Results

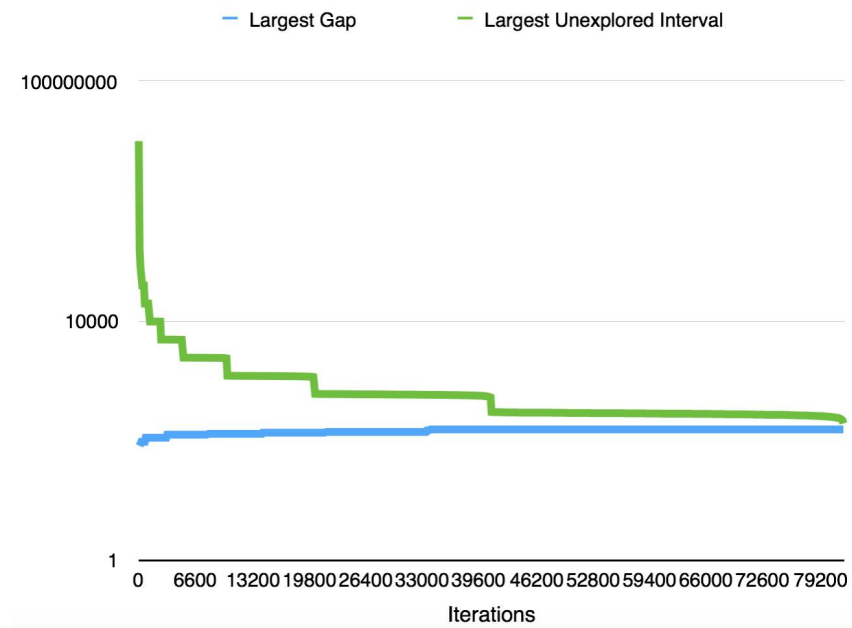
Best Case: The first prime gap we check spans $\frac{1}{2}$ of the entire interval.

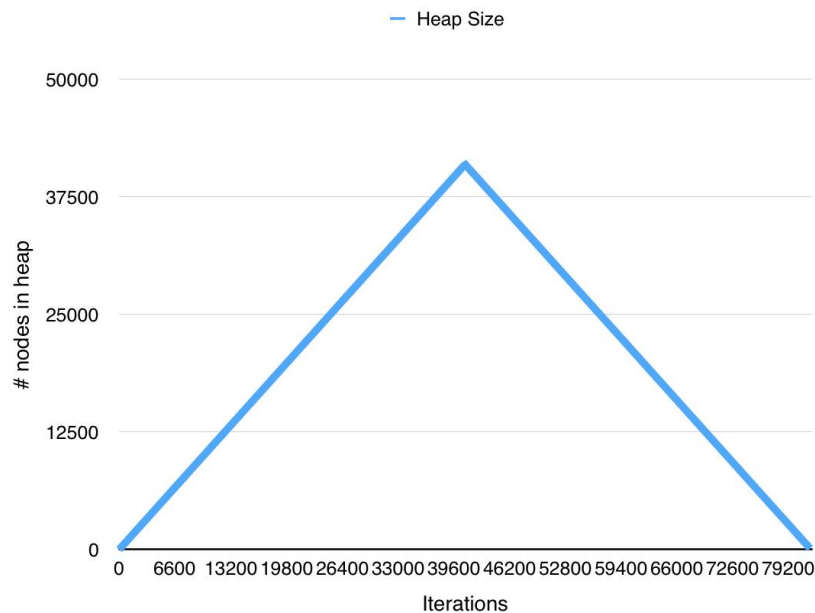
$$\left(\frac{n}{2}\right) = (n)$$

Worst Case: We find the gap of size 2 every time we split an interval: In this case every iteration results in the size of the unexplored interval to be decreased by 3. Assuming the Priority Queue is implemented as a heap, it costs us $O(\log n)$ to find the largest unexplored interval. This gives us a worst case runtime of

$$O(n \log n)$$

This is clearly worse than a linear scan. For completeness we include some graphs demonstrating the terrible rate of convergence.





“I am the bone of my sword, Steel is my
body and fire is my blood, I Have
withstood pain to generate many primes,
Yet, the Mersenne Twin I will never find,
So as I pray, Find me at least a trillion”

Type-Moon
Vaughan Hiltz

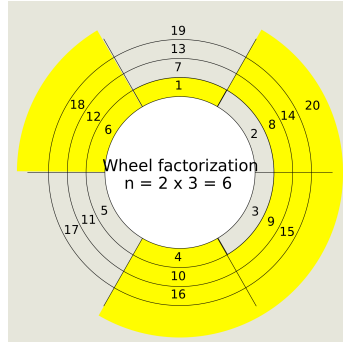
0.5 The Quest to Find the Maximal Gap Below 10^{13}

Upper Bound	Maximum Gap	Prime Preceding	Prime After	Speed (s)	Cores
1e6	114	492113	492227	1.21	8
1e9	282	436273009	436273291	6.10	8
1e12	545	738832927927	738832928467	561.69	8
1e13	674	7177162611713	7177162612387	4027.23	128

To find the maximal gap between all primes less than 1 trillion we used a segmented sieve of Eratosthenes with a mod 2 wheel where the segment size was chosen based on the available processors in the Orca SHARCNET cluster. The sieve of Eratosthenes is an extremely fast ancient algorithm for finding prime numbers. The algorithm find the primes under N by first finding all the primes less than \sqrt{N} and then iterates by each prime through a list of bits marking each one

as a composite and therefore not prime if it is a multiple of another prime in the list. Once the composites of primes $< \sqrt{N}$ are marked the remaining 0's represent all of the primes less than N . In our case we add a mod 2 wheel and segmentation. A mod 2 wheel uses wheel factorization to reduce the total number of potential primes in each segment. Wheel factoring works by using the first few primes to quickly mark many composite numbers to reduce the initial numbers to sieve by finding "relatively" prime numbers.

Figure 1: Wheel Factorization



In our case the mod 2 basically means we remove the even numbers. The implication is that we are then able store twice the numbers in the same cache space since we will cut the list in half through our mod 2 wheel. This allowed us to reach the speeds shown in the table above.

0.5.1 The Segmentation

Below is the code used by each process to assign its total range and then segment that range into manageable cache sized pieces. On lines 43-50, we take the size of the cache of the CPU we know we're going to target and allocate just enough space, minus a bit for local variables, to fit the range of numbers we want to sieve per processor iteration. This allows the processor to keep everything in its high-speed cache, ideally the L1 or L2 cache where possible. In the worst cases, the L3 but never the main memory, as it is many orders of magnitude slower.

```

1 int main(int argc, char** argv) {
2     char hostname[MPI_MAX_PROCESSOR_NAME];
3     int64_t N = 1 * pow(10, 12);
4
5     double start_time = MPI_Wtime();
6
7     // printf("Sieve, strike them down! N: %" PRId64 "\n", N);
8
9     uint16_t rc = MPI_Init(&argc, &argv);
10    if (rc != MPI_SUCCESS) {
11        printf("Error starting MPI program. Terminating...\n");
12        MPI_Abort(MPI_COMM_WORLD, rc);
13    }
14
15    int32_t numtasks;
16    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
17

```

```
18     uint64_t proc_gap_primes[numtasks * 3];
19
20     int32_t rank, len;
21     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
22     MPI_Get_processor_name(hostname, &len);
23
24
25     uint64_t root_n = sqrt(N);
26     uint64_t sieve[root_n];
27     uint64_t prime_prime_gap[5];
28
29
30     uint64_t block_size = (N - root_n + 1) / numtasks;
31
32
33     // integer cast safety
34     uint64_t cast_rank = (uint64_t) rank;
35
36     uint64_t start = (root_n) + (block_size * cast_rank) + 2;
37     uint64_t end = (root_n) + (block_size * (cast_rank + 1)); // extend out far
38     // enough
39
40     uint64_t num_primes = gen_sieve(N, sieve);
41
42     // By giving a CPU cache size ahead of time, we know how much of a range to
43     // allocate
44     // Since we only allocate for odds, we can afford to store twice the cache in
45     // here
46     uint64_t max_cache_bytes = 1200000;
47
48     // quick explanation: cache size is multiplied by 2 since we only store for odd
49     // numbers, so 2x
50     // the portion at the end takes off 32kB from the cache. Since other stuff OTHER
51     // than this array will
52     // be allocated, one can think of this as 32KB of "safe" space to play around
53     // with.
54     // It's slightly too much in many cases but it will prevent users from getting
55     // bitten by bugs if they modify the function and spill
56     // over into too much RAM.
57     uint64_t range_to_allocate = (max_cache_bytes * 2) - (1024 * 32);
58
59     uint64_t sub_blocks = 0, cur_block = 0, nprimes = 0, base = start, maximal_gap =
60     0, l_prime, r_prime;
61     sub_blocks = (end - start) / range_to_allocate;
62
63     for(cur_block = 0; cur_block < sub_blocks; cur_block++) {
64         uint64_t sub_block_start = start + (cur_block * range_to_allocate);
65         uint64_t sub_block_end = start + min((cur_block + 1) * range_to_allocate, end)
66         ;
67         sieve_primes(sieve, num_primes, sub_block_start, sub_block_end,
68         prime_prime_gap);
69
70         if(prime_prime_gap[2] > maximal_gap) {
71             maximal_gap = prime_prime_gap[2];
72             l_prime = prime_prime_gap[3];
73             r_prime = prime_prime_gap[4];
74         }
75     }
```

```

66
67     prime_prime_gap[2] = maximal_gap;
68     prime_prime_gap[3] = l_prime;
69     prime_prime_gap[4] = r_prime;
70
71     uint32_t dest = 0, tag = 0;
72
73     if (rank > 0) {
74         MPI_Send(prime_prime_gap, 5, MPI_LONG, dest, tag, MPI_COMM_WORLD);
75     }
76
77     if (rank == 0) {
78         uint64_t max_gap = 0, left_prime, right_prime;
79         uint32_t source = 0;
80         printf("Processor 0 awaiting commands...\n");
81         MPI_Status status;
82         int m = 0;
83
84         proc_gap_primes[m++] = prime_prime_gap[0];
85         proc_gap_primes[m++] = prime_prime_gap[1];
86
87         max_gap = prime_prime_gap[2];
88         left_prime = prime_prime_gap[3];
89         right_prime = prime_prime_gap[4];
90
91         for (source = 1; source < numtasks; source++) {
92             MPI_Recv(prime_prime_gap, 5, MPI_LONG, source, MPI_ANY_TAG,
93                     MPI_COMM_WORLD,
94                     &status);
95             printf("Processor %d is now reporting in w/ gap %lu\n", source,
96                    prime_prime_gap[2]);
97             proc_gap_primes[m++] = prime_prime_gap[0];
98             proc_gap_primes[m++] = prime_prime_gap[1];
99             uint64_t proc_gap = prime_prime_gap[2];
100             if (proc_gap > max_gap) {
101                 max_gap = proc_gap;
102                 left_prime = prime_prime_gap[3];
103                 right_prime = prime_prime_gap[4];
104             }
105
106             int32_t gap = 0;
107             uint32_t i = 0;
108
109             for (i = 0; i < numtasks - 1; i++) {
110                 // First gap of the next processor in line - last prime of current processor
111                 uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
112                 uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
113                 gap = next_proc_first_prime - cur_proc_last_prime;
114
115                 if (gap > max_gap) {
116                     max_gap = gap;
117                     left_prime = cur_proc_last_prime;
118                     right_prime = next_proc_first_prime;
119                 }
120
121                 printf("The maximum gap between primes below N = %ld is %ld. Interval: [%lu,

```

```

122     %lu]\n\n", (long long) N, (long long) max_gap, left_prime, right_prime);
123     double end_time = MPI_Wtime();
124     printf("Total Seconds: %lf\n", end_time - start_time);
125     }
126     MPI_Finalize();
127     return 0;
128 }

```

Listing 5: Segmentation

0.5.2 The Sieve

```

1  int sieve_primes(uint64_t* primes, unsigned int num_primes,
2                  uint64_t start, uint64_t end, uint64_t* prime_prime_gap_l_h) {
3
4      time_t startTime = time(NULL);
5
6      // Only care about odd sizes; make sure we start on one
7      if (start % 2 == 0) {
8          start++;
9      }
10
11     // 0 = prime
12     // 1 = not prime
13     // This may seem a little counter-intuitive, but calloc is so fast compared to
14     // writing
15     // zeroes that we are forced to make a readability trade-off here to achieve
16     // that
17     // There is a *POTENTIAL* speed gain here by allocating on the stack instead
18     char* notPrime = (char*) calloc(1 + (end - start)/2, sizeof(char));
19
20     int64_t j, k = 0;
21     int64_t total_count = 0;
22     int64_t begin = start;
23
24     for(j = 0; j < num_primes; j++) {
25         uint64_t prime = primes[j];
26
27         if(prime*prime >= begin) {
28             for(k = prime * prime; k <= end; k += 2*prime) {
29                 int64_t shouldDestroy = floor((k-start)/2);
30                 if(shouldDestroy >= 0) {
31                     notPrime[shouldDestroy] = 1;
32                 }
33             }
34         }
35         else {
36             int64_t l = floor((begin - prime*prime)/(2*prime));
37             for(k = (prime*prime) + (2*l*prime); k <= end; k += 2*prime) {
38                 int64_t shouldDestroy = floor((k - begin)/2);
39                 if(shouldDestroy >= 0) {
40                     notPrime[shouldDestroy] = 1;
41                 }
42             }
43         }
44     }
45 }

```

```

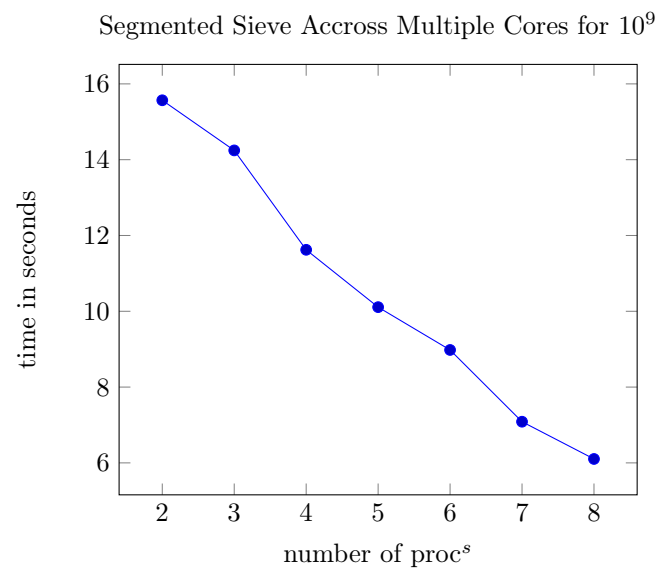
42
43 } // end prime time loop
44
45
46 uint64_t counter = 0;
47 uint64_t prior_prime = 0, fprime = 0, who_prime, mgap, new_gap = 0, l_prime,
r_prime;
48 for(counter = 0; counter < (end - start)/2+1; counter++) {
49     if(notPrime[counter] == 0) {
50         who_prime = (start + counter*2);
51         if(prior_prime != 0) {
52             mgap = who_prime - prior_prime;
53             if(mgap > new_gap) {
54                 new_gap = mgap;
55                 l_prime = prior_prime;
56                 r_prime = who_prime;
57             }
58         }
59         else {
60             fprime = who_prime;
61         }
62         prior_prime = who_prime;
63         total_count++;
64     }
65 }
66
67
68 // end sieve, stop the clock
69 time_t finish = time(NULL) - startTime;
70
71 prime_prime_gap_l_h[0] = fprime << 48; // shift out, allow for truncation; see.
seg. e.
72 prime_prime_gap_l_h[1] = who_prime << 48;
73 prime_prime_gap_l_h[2] = new_gap;
74 prime_prime_gap_l_h[3] = l_prime;
75 prime_prime_gap_l_h[4] = r_prime;
76
77 // free memory; signal to CPU cache that it can be evicted
78 free(notPrime);
79
80 return total_count;
81 }

```

Listing 6: Segmentation

0.5.3 The Results

Below is the graph of the result of finding all the primes less than 10^9 . The main results of the experiment were presented in the table at the beginning of the section but this graph is useful when comparing the speed and scalability of this algorithm to that of the trial division method.




```

26     MPI_Abort(MPI_COMM_WORLD, rc);
27 }
28
29 int32_t numtasks;
30 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
31
32 // XXX: This is * 3; but i'm pretty sure it can be * 2
33 // can anyone confirm?
34 uint64_t proc_gap_primes[numtasks * 3];
35
36 int32_t rank, len;
37 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
38 MPI_Get_processor_name(hostname, &len);
39
40 uint64_t root_n = sqrt(N);
41 uint64_t lprime_hprime_lgprime_hgprime_gap[5];
42
43 uint64_t block_size = (N - root_n + 1) / numtasks;
44
45 // integer cast safety
46 uint64_t cast_rank = (uint64_t) rank;
47
48 uint64_t start = root_n + (block_size * cast_rank);
49 uint64_t end = root_n + (block_size * (cast_rank + 1)); // extend out far enough
50
51 uint64_t sieve[root_n];
52 uint64_t num_primes = gen_sieve(N, sieve);
53 sieve_primes(sieve, num_primes, start, end, lprime_hprime_lgprime_hgprime_gap);
54
55 uint32_t dest = 0;
56 uint32_t tag = 0;
57
58 // Only send if you're not core 0
59 if (rank != 0) {
60     MPI_Send(lprime_hprime_lgprime_hgprime_gap, 5, MPI_LONG, dest, tag,
61 MPI_COMM_WORLD);
62 }
63
64 if (rank == 0) {
65 // TODO: Not sure if we really need this but MPI wants it to be outputted to, so
66 // for now, we should keep it around
67 MPI_Status status;
68
69 // Simple counter; keeps track of the current state of the prime blocks
70 int m = 0;
71
72 proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[0];
73 proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[1];
74
75 // Grab the gap, compare it against
76
77 uint64_t max_gap = lprime_hprime_lgprime_hgprime_gap[4];
78 uint64_t low_gap_prime = lprime_hprime_lgprime_hgprime_gap[2];
79 uint64_t high_gap_prime = lprime_hprime_lgprime_hgprime_gap[3];
80
81 uint32_t source = 0;
82 for (source = 1; source < numtasks; source++) {

```

```

83     MPI_Recv(&prime_hprime_lgprime_hgprime_gap, 5, MPI_LONG, source,
MPI_ANY_TAG, MPI_COMM_WORLD,
84             &status);
85     printf("Processor %d is now reporting in...", source);
86     proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[0];
87     proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[1];
88
89     uint64_t proc_gap = lprime_hprime_lgprime_hgprime_gap[4];
90     if (proc_gap > max_gap) {
91         max_gap = proc_gap;
92         low_gap_prime = lprime_hprime_lgprime_hgprime_gap[2];
93         high_gap_prime = lprime_hprime_lgprime_hgprime_gap[3];
94     }
95 }
96
97 uint32_t i = 0;
98 for (i = 0; i < numtasks - 1; i++) {
99     // First gap of the next processor in line - last prime of current processor
100    uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
101    uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
102    uint32_t gap = next_proc_first_prime - cur_proc_last_prime;
103    if (gap > max_gap) {
104        max_gap = gap;
105        low_gap_prime = cur_proc_last_prime;
106        high_gap_prime = next_proc_first_prime;
107    }
108    }
109    printf("The Maximum Gap Between Primes Below N = %" PRIu64 " is %" PRIu64 "\n",
N, max_gap);
110 }
111 MPI_Finalize();
112 return 0;
113 }
114
115 uint64_t gen_sieve(int64_t N, uint64_t* primes) {
116     mpz_t mpz_num;
117     mpz_init(mpz_num);
118     unsigned int top, sq_top, totalPrimes = 0, n = 0;
119
120     // Hopefully we don't overflow primes... :)
121
122     top = sqrt(N) + 1;
123     sq_top = sqrt(top) + 1;
124     primes[n++] = 2;
125
126     unsigned char isPrime;
127     unsigned int i, j;
128
129     for (i = 3; i < sq_top; i += 2) {
130         mpz_set_ui(mpz_num, i);
131         isPrime = mpz_probab_prime_p(mpz_num, 15);
132         if (isPrime > 0) {
133             primes[n++] = i;
134         }
135     }
136
137     if (sq_top % 2 == 0) sq_top++; // Primes and even numbers don't get along
138

```

```
139     unsigned char notPrime = 0;
140
141     for (i = sq_top; i < top; i += 2) {
142         for (j = 0; j < n; j++) {
143             if (i % primes[j] == 0) {
144                 notPrime = 1;
145                 break;
146             }
147         }
148         if (notPrime == 0) {
149             primes[n++] = i;
150         } else
151             notPrime = 0;
152     }
153     return n;
154 }
155
156 int sieve_primes(uint64_t* primes, unsigned int num_primes,
157                 uint64_t start, uint64_t end, uint64_t*
158                 lprime_hprime_lgprime_hgprime_gap) {
159     uint64_t last_prime, cur_prime, first_prime, gap, max_gap, low_gap_prime,
160     high_gap_prime;
161
162     time_t startTime = time(NULL);
163
164     // Only care about odd sizes
165     if (start % 2 == 0) {
166         start++;
167     }
168
169     cur_prime = 0;
170     max_gap = 0;
171
172     printf("Start: %" PRIu64 "\n", start);
173     printf("End: %" PRIu64 "\n", end);
174
175     int notPrime = 0;
176     uint64_t i, j;
177     for (i = start; i < end; i += 2) {
178         for (j = 0; j < num_primes; j++) {
179             if ((i % primes[j]) == 0) {
180                 notPrime = 1;
181                 break;
182             }
183         }
184         if (notPrime == 0) {
185             // printf("Discovered prime: %d\n", i);
186             last_prime = cur_prime;
187             cur_prime = i;
188
189             if (last_prime != 0) {
190                 gap = cur_prime - last_prime;
191                 if (gap > max_gap) {
192                     max_gap = gap;
193                     low_gap_prime = last_prime;
194                     high_gap_prime = cur_prime;
195                 }
196             } else {
```

```

195 // Must be the first prime, so we'll assign it to the first slot
196 first_prime = cur_prime;
197 }
198 } else {
199     notPrime = 0;
200 }
201 }
202
203 time_t finish = time(NULL) - startTime;
204 printf("Sieve finished in %ld s\n", finish);
205
206 // We're baically assuming here that we always find at least 2 primes in the range
207 ...
208 printf("low: %" PRIu64 " high: %" PRIu64 " gap: [%%" PRIu64 ", %" PRIu64 "] size: %"
        PRIu64 "\n", first_prime, cur_prime, low_gap_prime, high_gap_prime, max_gap);
209 lprime_hprime_lgprime_hgprime_gap[0] = first_prime;
210 lprime_hprime_lgprime_hgprime_gap[1] = cur_prime;
211 lprime_hprime_lgprime_hgprime_gap[2] = low_gap_prime;
212 lprime_hprime_lgprime_hgprime_gap[3] = high_gap_prime;
213 lprime_hprime_lgprime_hgprime_gap[4] = max_gap;
214 return max_gap;
215 }

```

Listing 7: Progressive Gap Method

0.6.2 Interval Method

```

1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <time.h>
6 #include <stdint.h>
7 #include <inttypes.h>
8 #include "heap.c"
9
10 #define MAX_PRIMES 100000
11 #define max(x,y) (((x) > (y)) ? (x) : (y))
12
13 int sieve_primes(uint64_t *, uint32_t);
14 int search(uint64_t *, uint32_t, uint64_t, uint64_t);
15
16 int main(int argc, char** argv) {
17     uint64_t primes[1000000];
18     time_t start_t = time(NULL);
19     uint32_t n_primes = sieve_primes(primes, 1000000);
20     time_t sieve_t = time(NULL) - start_t;
21     time_t start2_t = time(NULL);
22     uint32_t gap = search(primes, n_primes, 2, 10000003);
23     time_t gap_t = time(NULL) - start2_t;
24     printf("gap %d\n", gap);
25     return 0;
26 }
27
28 int is_prime(uint64_t *primes, uint32_t n_primes, uint64_t p) {
29     for (uint32_t i = 0; i < ceil(sqrtl(p)); i++) {

```

```

30     if (p != primes[i] && p % primes[i] == 0) {
31         return 0;
32     }
33 }
34 return 1;
35 }
36
37 int sieve_primes(uint64_t *primes, uint32_t size) {
38     primes[0] = 2;
39     uint32_t n = 1;
40
41     for (uint32_t i = 3; i < size; i += 2) {
42         uint8_t is_prime = 1;
43         for (uint32_t j = 0; j < n; j++) {
44             if (primes[j] >= sqrt(i) + 1) {
45                 break;
46             }
47             if (i % primes[j] == 0) {
48                 is_prime = 0;
49                 break;
50             }
51         }
52         if (is_prime) {
53             primes[n] = i;
54             n++;
55         }
56     }
57     return n;
58 }
59
60 int search(uint64_t *primes, uint32_t n_primes, uint64_t start, uint64_t end) {
61     struct node *largest_interval = NULL;
62     insert_interval(&largest_interval, start, end);
63
64     uint64_t largest_gap = 0;
65     for (int k = 0; k < n; k++) {
66         if (largest_interval == NULL) {
67             return largest_gap;
68         }
69         struct node *interval = pop(&largest_interval);
70         uint64_t interval_size = interval->end - interval->start;
71
72         if (largest_gap >= interval_size) {
73             return largest_gap;
74         }
75
76         uint64_t mid = (interval->end + interval->start) / 2; // we do
77         // mid point calc this way to avoid overflow
78         uint64_t next_prime;
79         for (next_prime = mid; next_prime <= interval->end; next_prime++) {
80             if (is_prime(primes, n_primes, next_prime)) {
81                 break;
82             }
83         }
84
85         uint64_t prev_prime;
86         for (prev_prime = max(mid-1, interval->start); prev_prime > interval->start;
87             prev_prime--) {

```

```
86     if (is_prime(primes, n_primes, prev_prime)) {
87     break;
88     }
89     }
90
91     if (prev_prime - interval->start > largest_gap) {
92         insert_interval(&largest_interval, interval->start, prev_prime);
93     }
94     if (interval->end - next_prime > largest_gap) {
95         insert_interval(&largest_interval, next_prime, interval->end);
96     }
97
98     uint64_t my_size = next_prime - prev_prime;
99     if (my_size > largest_gap) {
100         largest_gap = my_size;
101     }
102 }
103 }
```

0.6.3 Segmented Sieve

```

1  #include <gmp.h>
2  #include <mpi.h>
3  #include <math.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <time.h>
8  #include <stdint.h>
9  #include <inttypes.h>
10
11 #define MAX_PRIMES 100000
12 #define min(x,y) (((x) < (y)) ? (x) : (y))
13
14 uint64_t gen_sieve(int64_t N, uint64_t* primes);
15 int sieve_primes(uint64_t* primes, unsigned int num, uint64_t start,
16                 uint64_t end, uint64_t* prime_prime_gap);
17
18 int main(int argc, char** argv) {
19     char hostname[MPI_MAX_PROCESSOR_NAME];
20     int64_t N = 1 * pow(10, 12);
21
22     double start_time = MPI_Wtime();
23
24     // printf("Sieve, strike them down! N: %" PRId64 "\n", N);
25
26     uint16_t rc = MPI_Init(&argc, &argv);
27     if (rc != MPI_SUCCESS) {
28         printf("Error starting MPI program. Terminating...\n");
29         MPI_Abort(MPI_COMM_WORLD, rc);
30     }
31
32     int32_t numtasks;
33     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
34
35     uint64_t proc_gap_primes[numtasks * 3];
36
37     int32_t rank, len;
38     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
39     MPI_Get_processor_name(hostname, &len);
40
41
42     uint64_t root_n = sqrt(N);
43     uint64_t sieve[root_n];
44     uint64_t prime_prime_gap[5];
45
46
47     uint64_t block_size = (N - root_n + 1) / numtasks;
48
49
50     // integer cast safety
51     uint64_t cast_rank = (uint64_t) rank;
52
53
54     uint64_t start = (root_n) + (block_size * cast_rank) + 2;
55     uint64_t end = (root_n) + (block_size * (cast_rank + 1)); // extend out far
56     enough

```



```

56     uint64_t num_primes = gen_sieve(N, sieve);
57
58     // By giving a CPU cache size ahead of time, we know how much of a range to
59     // allocate
60     // Since we only allocate for odds, we can afford to store twice the cache in
61     // here
62     uint64_t max_cache_bytes = 1200000;
63
64     // quick explanation: cache size is multiplied by 2 since we only store for odd
65     // numbers, so 2x
66     // the portion at the end takes off 32kB from the cache. Since other stuff OTHER
67     // than this array will
68     // be allocated, one can think of this as 32KB of "safe" space to play around
69     // with.
70     // It's slightly too much in many cases but it will prevent users from getting
71     // bitten by bugs if they modify the function and spill
72     // over into too much RAM.
73     uint64_t range_to_allocate = (max_cache_bytes * 2) - (1024 * 32);
74
75     uint64_t sub_blocks = 0, cur_block = 0, nprimes = 0, base = start, maximal_gap =
76     0, l_prime, r_prime;
77     sub_blocks = (end-start)/range_to_allocate;
78
79     for (cur_block = 0; cur_block < sub_blocks; cur_block++) {
80         uint64_t sub_block_start = start + (cur_block * range_to_allocate);
81         uint64_t sub_block_end = start + min((cur_block + 1) * range_to_allocate, end)
82         ;
83         sieve_primes(sieve, num_primes, sub_block_start, sub_block_end,
84         prime_prime_gap);
85
86         if (prime_prime_gap[2] > maximal_gap) {
87             maximal_gap = prime_prime_gap[2];
88             l_prime = prime_prime_gap[3];
89             r_prime = prime_prime_gap[4];
90         }
91     }
92
93     prime_prime_gap[2] = maximal_gap;
94     prime_prime_gap[3] = l_prime;
95     prime_prime_gap[4] = r_prime;
96
97     uint32_t dest = 0, tag = 0;
98
99     if (rank > 0) {
100         MPI_Send(prime_prime_gap, 5, MPI_LONG, dest, tag, MPI_COMM_WORLD);
101     }
102
103     if (rank == 0) {
104         uint64_t max_gap = 0, left_prime, right_prime;
105         uint32_t source = 0;
106         printf("Processor 0 awaiting commands...\n");
107         MPI_Status status;
108         int m = 0;
109
110         proc_gap_primes[m++] = prime_prime_gap[0];
111         proc_gap_primes[m++] = prime_prime_gap[1];
112
113         max_gap = prime_prime_gap[2];

```

```

105 left_prime = prime_prime_gap[3];
106 right_prime = prime_prime_gap[4];
107
108     for (source = 1; source < numtasks; source++) {
109         MPI_Recv(prime_prime_gap, 5, MPI_LONG, source, MPI_ANY_TAG,
110 MPI_COMM_WORLD,
111                 &status);
112         printf("Processor %d is now reporting in w/ gap %lu\n", source,
113 prime_prime_gap[2]);
114         proc_gap_primes[m++] = prime_prime_gap[0];
115         proc_gap_primes[m++] = prime_prime_gap[1];
116         uint64_t proc_gap = prime_prime_gap[2];
117         if (proc_gap > max_gap) {
118             max_gap = proc_gap;
119             left_prime = prime_prime_gap[3];
120             right_prime = prime_prime_gap[4];
121         }
122     }
123     int32_t gap = 0;
124     uint32_t i = 0;
125
126     for (i = 0; i < numtasks - 1; i++) {
127         // First gap of the next processor in line - last prime of current processor
128         uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
129         uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
130         gap = next_proc_first_prime - cur_proc_last_prime;
131
132         if (gap > max_gap) {
133             max_gap = gap;
134             left_prime = cur_proc_last_prime;
135             right_prime = next_proc_first_prime;
136         }
137
138         printf("The maximum gap between primes below N = %ld is %ld. Interval: [%lu,
139 %lu]\n\n", (long long) N, (long long) max_gap, left_prime, right_prime);
140     double end_time = MPI_Wtime();
141     printf("Total Seconds: %lf\n", end_time - start_time);
142     }
143     MPI_Finalize();
144     return 0;
145 }
146
147 int cmp(const void *a, const void *b) { return (*(int *)a - *(int *)b); }
148
149 uint64_t gen_sieve(int64_t N, uint64_t* primes) {
150     mpz_t mpz_num;
151     mpz_init(mpz_num);
152     unsigned int top, sq_top, totalPrimes = 0, n = 0;
153
154     top = sqrt(N) + 1;
155
156     sq_top = sqrt(top) + 1;
157
158     // yes, this is on purpose; it's done to illustrate
159     // that since we're using a mod-2 wheel, that we should NOT
160     // include this as it'll cause redundant calculations.

```

```
160 // Do not uncomment.
161
162 // primes[n++] = 2;
163
164 unsigned char isPrime;
165 unsigned int i, j;
166
167 for (i = 3; i < sq_top; i += 2) {
168     mpz_set_ui(mpz_num, i);
169     isPrime = mpz_probab_prime_p(mpz_num, 15);
170     if (isPrime > 0) {
171         primes[n++] = i;
172     }
173 }
174
175 if (sq_top % 2 == 0)
176     sq_top++; // Primes and even numbers don't get along
177
178 unsigned char notPrime = 0;
179
180 for (i = sq_top; i < top; i += 2) {
181     for (j = 0; j < n; j++) {
182         if (i % primes[j] == 0) {
183             notPrime = 1;
184             break;
185         }
186     }
187     if (notPrime == 0) {
188         primes[n++] = i;
189     } else
190         notPrime = 0;
191 }
192
193 return n;
194 }
195
196 int sieve_primes(uint64_t* primes, unsigned int num_primes,
197                 uint64_t start, uint64_t end, uint64_t* prime_prime_gap_l_h) {
198
199     time_t startTime = time(NULL);
200
201     // Only care about odd sizes; make sure we start on one
202     if (start % 2 == 0) {
203         start++;
204     }
205
206     // printf("Starting point for segment: %" PRIu64 "\n", start);
207     // printf("Ending point for segment: : %" PRIu64 "\n", end);
208
209     // I am the bone of my sword
210     // Steel is my body and fire is my blood
211     // Have withstood pain to generate many primes
212     // Yet, the Mersene Twin I will never find
213     // So as I pray, find me at least a trillion
214
215     // 0 = prime
216     // 1 = not prime
```

```

217 // This may seem a little counter-intuitive, but calloc is so fast compared to
    writing
218 // zeroes that we are forced to make a readability trade-off here to achieve
    that
219 // There is a *POTENTIAL* speed gain here by allocating on the stack instead
    char* notPrime = (char*) calloc(1 + (end - start)/2 , sizeof(char));
220
221
222 int64_t j, k = 0;
223 int64_t total_count = 0;
224 int64_t begin = start;
225
226 for(j = 0; j < num_primes; j++) {
227     uint64_t prime = primes[j];
228
229     if(prime*prime >= begin) {
230 for(k = prime * prime; k <= end; k += 2*prime) {
231     int64_t shouldDestroy = floor((k-start)/2);
232     if(shouldDestroy >= 0) {
233         notPrime[shouldDestroy] = 1;
234     }
235     }
236     }
237     else {
238         int64_t l = floor((begin - prime*prime)/(2*prime));
239         for(k = (prime*prime) + (2*l*prime); k <= end; k += 2*prime) {
240 int64_t shouldDestroy = floor((k - begin)/2);
241         if(shouldDestroy >= 0) {
242             notPrime[shouldDestroy] = 1;
243         }
244         }
245     }
246 } // end prime time loop
247
248
249
250 uint64_t counter = 0;
251 uint64_t prior_prime = 0, fprime = 0, who_prime, mgap, new_gap = 0, l_prime,
    r_prime;
252 for(counter = 0; counter < (end - start)/2+1; counter++) {
253     if(notPrime[counter] == 0) {
254 who_prime = (start + counter*2);
255     if(prior_prime != 0) {
256         mgap = who_prime - prior_prime;
257         if(mgap > new_gap) {
258             new_gap = mgap;
259             l_prime = prior_prime;
260             r_prime = who_prime;
261         }
262     }
263     else {
264         fprime = who_prime;
265     }
266     prior_prime = who_prime;
267     total_count++;
268 }
269 }
270
271

```

```
272 // end sieve, stop the clock
273 time_t finish = time(NULL) - startTime;
274 // printf("Sieve finished in %lds.. gap: %lu. F: %lu P: %lu\n", finish, new_gap,
    fprime, who_prime);
275
276 prime_prime_gap_l_h[0] = fprime << 48; // shift out, allow for truncation; see.
    seg. e.
277 prime_prime_gap_l_h[1] = who_prime << 48;
278 prime_prime_gap_l_h[2] = new_gap;
279 prime_prime_gap_l_h[3] = l_prime;
280 prime_prime_gap_l_h[4] = r_prime;
281
282 // free memory; signal to CPU cache that it can be evicted
283 free(notPrime);
284
285 return total_count;
286 }
```

Listing 8: Interval method