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`

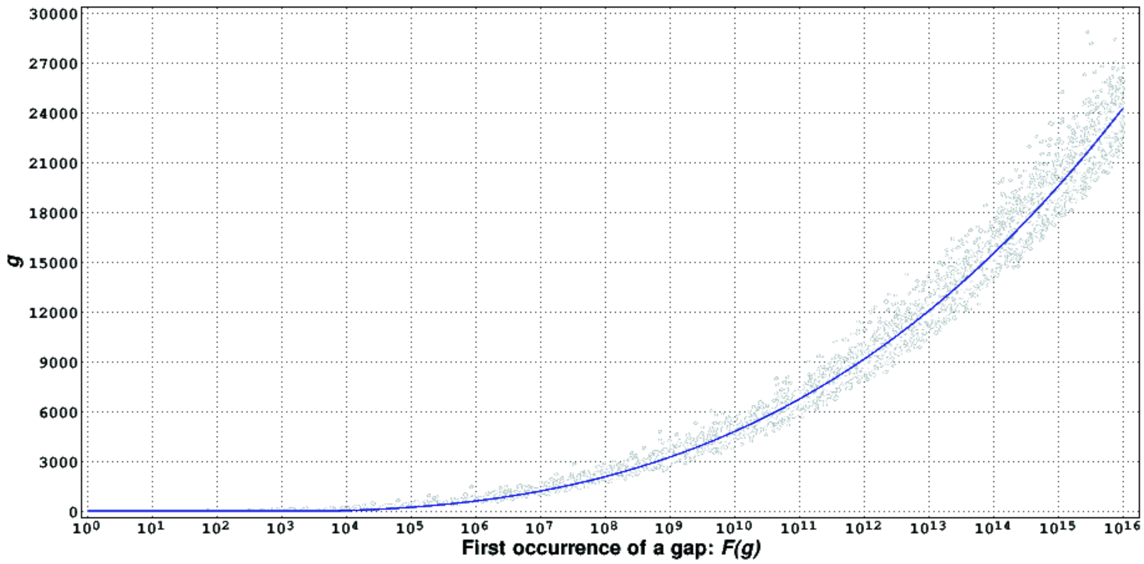
February 2^{nd} 2016

Contents

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 $>= \sqrt{N}$ 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 $O(mpz\_probab\_prime\_p * \sqrt{N}) + O(N * \pi(\sqrt{N}))$ 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:

```c
uint64_t gen_sieve(int64_t N, uint64_t* primes) {
    mpz_t mpz_num;
    mpz_init(mpz_num);
    unsigned int top, sq_top, totalPrimes = 0, n = 0;
    top = sqrt(N) + 1;
    sq_top = sqrt(top) + 1;
    primes[n++] = 2;

    unsigned char isPrime;
    unsigned int i, j;

    for (i = 3; i < sq_top; i += 2) {
        mpz_set_ui(mpz_num, i);
        isPrime = mpz_probab_prime_p(mpz_num, 15);
        if (isPrime > 0) {
            primes[n++] = i;
        }
    }

    if (sq_top % 2 == 0) sq_top++; // Primes and even numbers don't get along

    unsigned char notPrime = 0;

    for (i = sq_top; i < top; i += 2) {
```

```
        for (j = 0; j < n; j++) {
            if (i % primes[j] == 0) {
                notPrime = 1;
                break;
            }
        }
        if (notPrime == 0) {
            primes[n++] = i;
        } else
            notPrime = 0;
    }
    return n;
}
```

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.

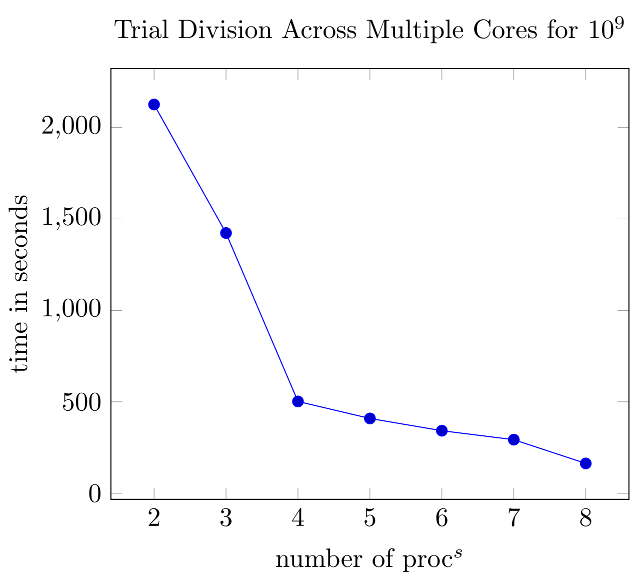
```
int sieve_primes(uint64_t* primes, unsigned int num_primes,
                 uint64_t start, uint64_t end, uint64_t* lprime_hprime_lgprime_hgprime_gap) {
  uint64_t last_prime, cur_prime, first_prime, gap, max_gap, low_gap_prime, high_gap_prime;

  time_t startTime = time(NULL);

  // Only care about odd sizes
  if (start % 2 == 0) {
    start++;
  }

  cur_prime = 0;
  max_gap = 0;

  printf("Start: %" PRId64 "\n", start);
  printf("End: %" PRId64 "\n", end);

  int notPrime = 0;
  uint64_t i, j;
  for (i = start; i < end; i += 2) {
    for (j = 0; j < num_primes; j++) {

      if ((i % primes[j]) == 0) {
  notPrime = 1;
  break;
      }
    }
    if (notPrime == 0) {
      last_prime = cur_prime;
      cur_prime = i;

      if (last_prime != 0) {
```

```
  gap = cur_prime - last_prime;
  if (gap > max_gap) {
    max_gap = gap;
    low_gap_prime = last_prime;
    high_gap_prime = cur_prime;
  }
      } else {
  first_prime = cur_prime;
      }
    } else {
      notPrime = 0;
    }
  }

  time_t finish = time(NULL) - startTime;
  printf("Sieve finished in %ld s\n", finish);

  lprime_hprime_lgprime_hgprime_gap[0] = first_prime;
  lprime_hprime_lgprime_hgprime_gap[1] = cur_prime;
  lprime_hprime_lgprime_hgprime_gap[2] = low_gap_prime;
  lprime_hprime_lgprime_hgprime_gap[3] = high_gap_prime;
  lprime_hprime_lgprime_hgprime_gap[4] = max_gap;
  return max_gap;
}
```

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:

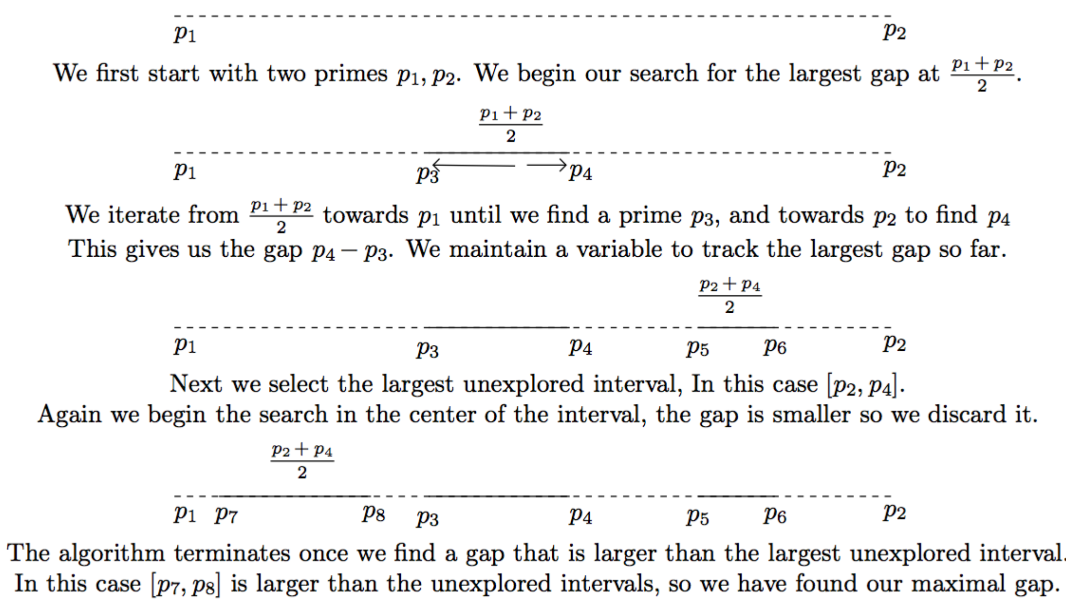


Figure 1. Intuition behind the Progressive Gap Improvement approach

0.4.1 Pseudo Code

```
PQ = PriorityQueue()
largest_gap = 0
while (True):
  interval = PQ.pop()
  if size(interval) < largest_gap:
```

```
    return largest_gap
  mid = size(interval)/2 + interval.start
  gap_high_prime = mid;

  while !isPrime(gap_high_prime):
    gap_high_prime += 1

  gap_low_prime = max(mid-1, interaval.start)

  while !isPrime(gap_low_prime):
    gap_low_prime -= 1

  largest_gap = max(largest_gap, gap_high_prime - gap_low_prime)
  if gap_low_prime - interval.start > largest_gap:
    PQ.push(Interval(interval.start, gap_low_prime))
  if interval.end - gap_high_prime > largest_gap:
    PQ.push(Interval(gap_high_prime, interval.end))
```

Listing 3: Pseudo Code

0.4.2 Actual Code

```
int search(uint64_t *primes, uint32_t n_primes, uint64_t start, uint64_t end) {
  struct node *largest_interval = NULL;
  insert_interval(&largest_interval, start, end);

  uint64_t largest_gap = 0;
  for (int k = 0;; k++) {
    if (largest_interval == NULL) {
      return largest_gap;
    }
    struct node *interval = pop(&largest_interval);
    uint64_t interval_size = interval->end - interval->start;

    if (largest_gap >= interval_size) {
      return largest_gap;
    }

    uint64_t mid = (interval->end - interval->start) / 2 + interval->start; // we do
     mid point calc this way to avoid overflow
    uint64_t next_prime;
    for (next_prime = mid; next_prime <= interval->end; next_prime++) {
      if (is_prime(primes, n_primes, next_prime)) {
  break;
      }
    }

    uint64_t prev_prime;
    for (prev_prime = max(mid-1, interval->start); prev_prime > interval->start;
    prev_prime--) {
      if (is_prime(primes, n_primes, prev_prime)) {
  break;
      }
    }

    if (prev_prime - interval->start > largest_gap) {
      insert_interval(&largest_interval, interval->start, prev_prime);
    }
```

```
    if (interval->end - next_prime > largest_gap) {
      insert_interval(&largest_interval, next_prime, interval->end);
    }

    uint64_t my_size = next_prime - prev_prime;
    if (my_size > largest_gap) {
      largest_gap = my_size;
    }
  }
}
```

Listing 4: Interval method

0.4.3 Results

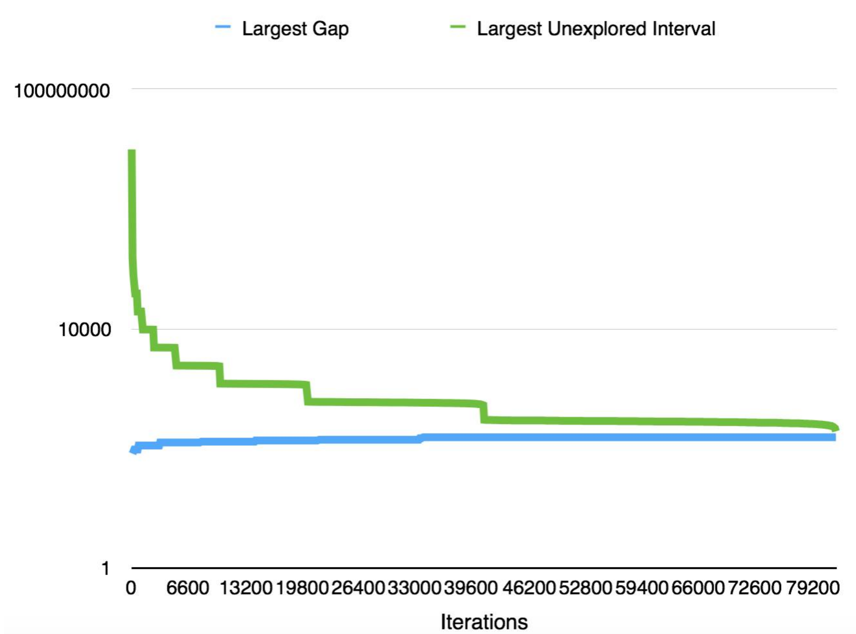
Best Case: The first prime gap we check spans $\frac{1}{2}$ of the entire interval.

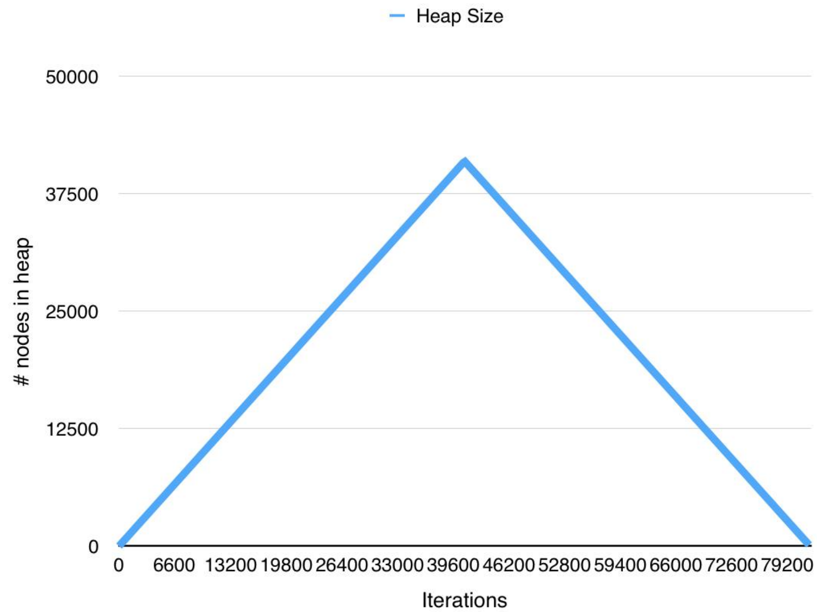
$$(\frac{n}{2}) = (n)$$

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"

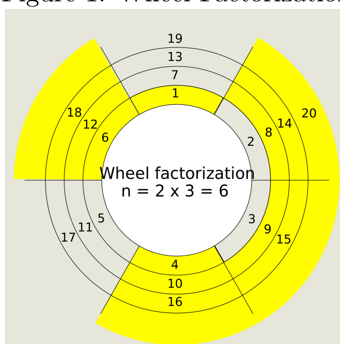
Type-Moon
Vaughan Hilts

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

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

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 
```

```
    uint64_t proc_gap_primes[numtasks * 3];

    int32_t rank, len;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);


    uint64_t root_n = sqrt(N);
    uint64_t sieve[root_n];
    uint64_t prime_prime_gap[5];


    uint64_t block_size = (N - root_n + 1) / numtasks;


    // integer cast safety
    uint64_t cast_rank = (uint64_t) rank;

    uint64_t start = (root_n) + (block_size * cast_rank) + 2;
    uint64_t end = (root_n) + (block_size * (cast_rank + 1)); // extend out far
    enough

    uint64_t num_primes = gen_sieve(N, sieve);

    // By giving a CPU cache size ahead of time, we know how much of a range to
    allocate
    // Since we only allocate for odds, we can afford to store twice the cache in
    here
    uint64_t max_cache_bytes = 1200000;

    // quick explanation: cache size is multiplied by 2 since we only store for odd
    numbers, so 2x
    // the portion at the end takes off 32kB from the cache. Since other stuff OTHER
     than this array will
    // be allocated, one can think of  this as 32KB of "safe' space to play around
    with.
    // It's slightly too much in many cases but it will prevent users from getting
    bitten by bugs if they modify the function and spill
    // over into too much RAM.
    uint64_t range_to_allocate = (max_cache_bytes * 2) - (1024 * 32);

    uint64_t sub_blocks = 0, cur_block = 0, nprimes = 0, base = start, maximal_gap =
     0, l_prime, r_prime;
    sub_blocks = (end-start)/range_to_allocate;

    for(cur_block = 0; cur_block < sub_blocks; cur_block++) {
      uint64_t sub_block_start = start + (cur_block * range_to_allocate);
      uint64_t sub_block_end = start + min((cur_block + 1) * range_to_allocate, end)
    ;
      sieve_primes(sieve, num_primes, sub_block_start, sub_block_end,
    prime_prime_gap);

      if(prime_prime_gap[2] > maximal_gap) {
  maximal_gap = prime_prime_gap[2];
  l_prime = prime_prime_gap[3];
  r_prime = prime_prime_gap[4];
      }
    }
```

```
    prime_prime_gap[2] = maximal_gap;
    prime_prime_gap[3] = l_prime;
    prime_prime_gap[4] = r_prime;

    uint32_t dest = 0, tag = 0;

    if (rank > 0) {
      MPI_Send(prime_prime_gap, 5, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    }

    if (rank == 0) {
        uint64_t max_gap = 0, left_prime, right_prime;
  uint32_t source = 0;
  printf("Processor 0 awaiting commands...\n");
  MPI_Status status;
  int m = 0;

  proc_gap_primes[m++] = prime_prime_gap[0];
  proc_gap_primes[m++] = prime_prime_gap[1];

  max_gap = prime_prime_gap[2];
  left_prime = prime_prime_gap[3];
  right_prime = prime_prime_gap[4];

        for (source = 1; source < numtasks; source++) {
            MPI_Recv(prime_prime_gap, 5, MPI_LONG, source, MPI_ANY_TAG, MPI_COMM_WORLD,
                     &status);
      printf("Processor %d is now reporting in w/ gap %lu\n", source, prime_prime_gap[2]);
            proc_gap_primes[m++] = prime_prime_gap[0];
            proc_gap_primes[m++] = prime_prime_gap[1];
      uint64_t proc_gap = prime_prime_gap[2];
      if(proc_gap > max_gap) {
        max_gap = proc_gap;
        left_prime = prime_prime_gap[3];
        right_prime = prime_prime_gap[4];
      }
        }

        int32_t gap = 0;
  uint32_t i = 0;

        for (i = 0; i < numtasks - 1; i ++) {
    // First gap of the next processor in line - last prime of current processor
    uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
    uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
    gap = next_proc_first_prime - cur_proc_last_prime;

    if(gap > max_gap) {
      max_gap = gap;
      left_prime = cur_proc_last_prime;
      right_prime = next_proc_first_prime;
    }
        }

        printf("The maximum gap between primes below N = %ld is %ld. Interval: [%lu,
```

```
    %lu]\n\n", (long long) N, (long long) max_gap, left_prime, right_prime);
  double end_time = MPI_Wtime();
  printf("Total Seconds: %lf\n", end_time - start_time);
    }
    MPI_Finalize();
    return 0;
}

```

Listing 5: Segmentation

0.5.2 The Sieve

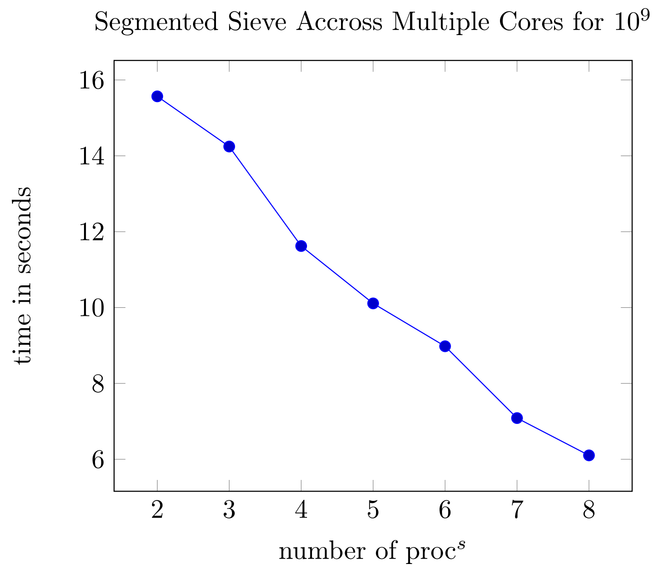
```
  int sieve_primes(uint64_t* primes, unsigned int num_primes,
                 uint64_t start, uint64_t end, uint64_t* prime_prime_gap_l_h) {

  time_t startTime = time(NULL);

  // Only care about odd sizes; make sure we start on one
    if (start % 2 == 0) {
        start++;
    }

    // 0 = prime
    // 1 = not prime
    // This may seem a little counter-intuitive, but calloc is so fast compared to writing
    // zeroes that we are forced to make a readability trade-off here to achieve that
    // There is a *POTENTIAL* speed gain here by allocating on the stack instead
    char* notPrime = (char*) calloc(1 + (end - start)/2 , sizeof(char));

    int64_t j, k = 0;
    int64_t total_count = 0;
    int64_t begin = start;

    for(j = 0; j < num_primes; j++) {
      uint64_t prime = primes[j];

      if(prime*prime >= begin) {
  for(k = prime * prime; k <= end; k += 2*prime) {
    int64_t shouldDestroy = floor((k-start)/2);
    if(shouldDestroy >= 0) {
      notPrime[shouldDestroy] = 1;
    }
        }
      }
    else {
      int64_t l =  floor( (begin - prime*prime)/(2*prime));
      for(k = (prime*prime) + (2*l*prime); k <= end; k += 2*prime) {
  int64_t shouldDestroy = floor((k - begin)/2);
  if(shouldDestroy >= 0) {
    notPrime[shouldDestroy] = 1;
  }
      }
    }
```

```
  } // end prime time loop


    uint64_t counter = 0;
    uint64_t prior_prime = 0, fprime = 0, who_prime,  mgap, new_gap = 0, l_prime,
    r_prime;
    for(counter = 0; counter < (end - start)/2+1; counter++) {
      if(notPrime[counter] == 0) {
  who_prime = (start + counter*2);
  if(prior_prime != 0) {
    mgap = who_prime - prior_prime;
    if(mgap > new_gap) {
      new_gap = mgap;
      l_prime = prior_prime;
      r_prime = who_prime;
    }
  }
  else {
    fprime = who_prime;
  }
  prior_prime = who_prime;
  total_count++;
      }
    }


    // end sieve, stop the clock
    time_t finish = time(NULL) - startTime;

    prime_prime_gap_l_h[0] = fprime << 48; // shift out, allow for truncuation; see.
     seg. e.
    prime_prime_gap_l_h[1] = who_prime << 48;
    prime_prime_gap_l_h[2] = new_gap;
    prime_prime_gap_l_h[3] = l_prime;
    prime_prime_gap_l_h[4] = r_prime;

    // free memory; signal to CPU cache that it can be evicted
    free(notPrime);

    return total_count;
}
```

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.


Segmented Sieve Accross Multiple Cores for 10^9
time in seconds
16
14
12
10
8
6
2
3
4
5
6
7
8
number of proc^s

0.6 Full Code

0.6.1 Trial Division

```
/*****************************************************************************
 * (            (             (    (         (      (   (   (       *        (       *
 * )\ )    (    )\ )    (     )\ ) )\ )      )\ )   )\ ))\ ))\ ) (   `       )\ )    *
 * (()/(   )\   (()/(   )\    (()/((()/( (   (()/(  (()/((()/((()/( )\))(  (  (()/(   *
 * /(_))((((_)(  /(_))((((_)(  /(_))/(_)))\  /(_))  /(_))(_))(_))|(_)()\ )\  /(_))   *
 * (_))  )\ _ )\(_))  )\ _ )\(_)) (_)) ((_)(_))   (_))(_))(_)) (_()((_|(_)(_))   *
 * | _ \ (_)_\(_) _ \ (_)_\(_) |  | |  | __| |    | _ \ _ \_ _||  \/  | __/ __|  *
 * |  _/  / _ \ |   /  / _ \ | |__| |__| _|| |__  |  _/   /| | | |\/| | _|\__ \  *
 * |_|   /_/ \_\|_|_\ /_/ \_\|____|____|___|____| |_| |_|_\___||_|  |_|___|___/  *
 *----------------------------------------------------------------------------*
 *          AUTHORS:                                                          *
 *                  > Robert Morouney     (moro1422@mylaurier.ca)             *
 *                  > Vaughan Hilts       (hilt2740@mylaurier.ca)             *
 *                  > David Rusu          (rusu0260@mylaurier.ca)             *
 *----------------------------------------------------------------------------*
 *          DESCRIPTION:                                                      *
 *                  Program finds all the prime numbers up to 10^9            *
 *                  then calculates the gaps between them to find the         *
 *                  largest gap and output the result.                        *
 *                                                                            *
 *                                                                            *
 *                                                                            *
 *                  Written for CP331 @ Wilfrid Laurier, Winter 2016          *
 *****************************************************************************/
```

```
#include <gmp.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdint.h>
#include <inttypes.h>

#define MAX_PRIMES 100000
#define max(x,y) (((x) > (y)) ? (x) : (y))

int cmp(const void *a, const void *b);
uint64_t gen_sieve(int64_t N, uint64_t* primes);
int sieve_primes(uint64_t* primes, unsigned int num, uint64_t start,
                 uint64_t end, uint64_t* prime_prime_gap);

int main(int argc, char** argv) {
    char hostname[MPI_MAX_PROCESSOR_NAME];
    int64_t N = 1 * pow(10, 9);

    uint16_t rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating...\n");
```

```
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    int32_t numtasks;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    // XXX: This is * 3; but i'm pretty sure it can be * 2
    // can anyone confirm?
    uint64_t proc_gap_primes[numtasks * 3];

    int32_t rank, len;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);

    uint64_t root_n = sqrt(N);
    uint64_t lprime_hprime_lgprime_hgprime_gap[5];

    uint64_t block_size = (N - root_n + 1) / numtasks;

    // integer cast safety
    uint64_t cast_rank = (uint64_t) rank;

    uint64_t start = root_n + (block_size * cast_rank);
    uint64_t end = root_n + (block_size * (cast_rank + 1)); // extend out far enough

    uint64_t sieve[root_n];
    uint64_t num_primes = gen_sieve(N, sieve);
    sieve_primes(sieve, num_primes, start, end, lprime_hprime_lgprime_hgprime_gap);

    uint32_t dest = 0;
    uint32_t tag = 0;

    // Only send if you're not core 0
    if (rank != 0) {
      MPI_Send(lprime_hprime_lgprime_hgprime_gap, 5, MPI_LONG, dest, tag,
   MPI_COMM_WORLD);
    }

    if (rank == 0) {

  // TODO: Not sure if we really need this but MPI wants it to be outputted to, so
  // for now, we should keep it around
  MPI_Status status;

  // Simple counter; keeps track of the current state of the prime blocks
  int m = 0;

  proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[0];
  proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[1];

  // Grab the gap, compare it against

        uint64_t max_gap = lprime_hprime_lgprime_hgprime_gap[4];
  uint64_t low_gap_prime = lprime_hprime_lgprime_hgprime_gap[2];
  uint64_t high_gap_prime = lprime_hprime_lgprime_hgprime_gap[3];

  uint32_t source = 0;
        for (source = 1; source < numtasks; source++) {
```

```
            MPI_Recv(lprime_hprime_lgprime_hgprime_gap, 5, MPI_LONG, source,
    MPI_ANY_TAG, MPI_COMM_WORLD,
                        &status);
      printf("Processor %d is now reporting in...", source);
            proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[0];
            proc_gap_primes[m++] = lprime_hprime_lgprime_hgprime_gap[1];

      uint64_t proc_gap = lprime_hprime_lgprime_hgprime_gap[4];
      if (proc_gap > max_gap) {
        max_gap = proc_gap;
        low_gap_prime = lprime_hprime_lgprime_hgprime_gap[2];
        high_gap_prime = lprime_hprime_lgprime_hgprime_gap[3];
      }
        }

  uint32_t i = 0;
        for (i = 0; i < numtasks - 1; i ++) {
    // First gap of the next processor in line - last prime of current processor
    uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
    uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
    uint32_t gap = next_proc_first_prime - cur_proc_last_prime;
    if (gap > max_gap) {
      max_gap = gap;
      low_gap_prime = cur_proc_last_prime;
      high_gap_prime = next_proc_first_prime;
    }
        }
        printf("The Maximum Gap Between Primes Below N = %" PRIu64 "is %" PRIu64 "\n
    ", N, max_gap);
    }
    MPI_Finalize();
    return 0;
}

uint64_t gen_sieve(int64_t N, uint64_t* primes) {
    mpz_t mpz_num;
    mpz_init(mpz_num);
    unsigned int top, sq_top, totalPrimes = 0, n = 0;

    // Hopefully we don't overflow primes... :)

    top = sqrt(N) + 1;
    sq_top = sqrt(top) + 1;
    primes[n++] = 2;

    unsigned char isPrime;
    unsigned int i, j;

    for (i = 3; i < sq_top; i += 2) {
        mpz_set_ui(mpz_num, i);
        isPrime = mpz_probab_prime_p(mpz_num, 15);
        if (isPrime > 0) {
            primes[n++] = i;
        }
    }

    if (sq_top % 2 == 0) sq_top++; // Primes and even numbers don't get along

```

```
    unsigned char notPrime = 0;

    for (i = sq_top; i < top; i += 2) {
        for (j = 0; j < n; j++) {
            if (i % primes[j] == 0) {
                notPrime = 1;
                break;
            }
        }
        if (notPrime == 0) {
            primes[n++] = i;
        } else
            notPrime = 0;
    }
    return n;
}

int sieve_primes(uint64_t* primes, unsigned int num_primes,
                 uint64_t start, uint64_t end, uint64_t* lprime_hprime_lgprime_hgprime_gap) {
  uint64_t last_prime, cur_prime, first_prime, gap, max_gap, low_gap_prime, high_gap_prime;

  time_t startTime = time(NULL);

  // Only care about odd sizes
  if (start % 2 == 0) {
    start++;
  }

  cur_prime = 0;
  max_gap = 0;

  printf("Start: %" PRId64 "\n", start);
  printf("End: %" PRId64 "\n", end);

  int notPrime = 0;
  uint64_t i, j;
  for (i = start; i < end; i += 2) {
    for (j = 0; j < num_primes; j++) {
      if ((i % primes[j]) == 0) {
  notPrime = 1;
  break;
      }
    }
    if (notPrime == 0) {
      // printf("Discovered prime: %d\n", i);
      last_prime = cur_prime;
      cur_prime = i;

      if (last_prime != 0) {
  gap = cur_prime - last_prime;
  if (gap > max_gap) {
    max_gap = gap;
    low_gap_prime = last_prime;
    high_gap_prime = cur_prime;
  }
      } else {
```

```
  // Must be the first prime, so we'll assign it to the first slot
  first_prime = cur_prime;
      }
    } else {
      notPrime = 0;
    }
  }

  time_t finish = time(NULL) - startTime;
  printf("Sieve finished in %ld s\n", finish);

  // We're baically assuming here that we always find at least 2 primes in the range ...
  printf("low: %" PRIu64 " high: %" PRIu64 " gap: [%" PRIu64 ", %" PRIu64"] size: %" PRIu64 "\n", first_prime, cur_prime, low_gap_prime, high_gap_prime, max_gap);
  lprime_hprime_lgprime_hgprime_gap[0] = first_prime;
  lprime_hprime_lgprime_hgprime_gap[1] = cur_prime;
  lprime_hprime_lgprime_hgprime_gap[2] = low_gap_prime;
  lprime_hprime_lgprime_hgprime_gap[3] = high_gap_prime;
  lprime_hprime_lgprime_hgprime_gap[4] = max_gap;
  return max_gap;
}
```

Listing 7: Progressive Gap Method

0.6.2 Interval Method

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdint.h>
#include <inttypes.h>
#include "heap.c"

#define MAX_PRIMES 100000
#define max(x,y) (((x) > (y)) ? (x) : (y))

int sieve_primes(uint64_t *, uint32_t);
int search(uint64_t *, uint32_t, uint64_t, uint64_t);

int main(int argc, char** argv) {
  uint64_t primes[1000000];
  time_t start_t = time(NULL);
  uint32_t n_primes = sieve_primes(primes, 1000000);
  time_t sieve_t = time(NULL) - start_t;
  time_t start2_t = time(NULL);
  uint32_t gap = search(primes, n_primes, 2, 10000003);
  time_t gap_t = time(NULL) - start2_t;
  printf("gap %d\n", gap);
  return 0;
}

int is_prime(uint64_t *primes, uint32_t n_primes, uint64_t p) {
  for (uint32_t i = 0; i < ceil(sqrtl(p)); i++) {
```

```
    if (p != primes[i] && p % primes[i] == 0) {
      return 0;
    }
  }
  return 1;
}

int sieve_primes(uint64_t *primes, uint32_t size) {
  primes[0] = 2;
  uint32_t n = 1;

  for (uint32_t i = 3; i < size; i += 2) {
    uint8_t is_prime = 1;
    for (uint32_t j = 0; j < n; j++) {
      if (primes[j] >= sqrt(i) + 1) {
  break;
      }
      if (i % primes[j] == 0) {
  is_prime = 0;
  break;
      }
    }
    if (is_prime) {
      primes[n] = i;
      n++;
    }
  }
  return n;
}

int search(uint64_t *primes, uint32_t n_primes, uint64_t start, uint64_t end) {
  struct node *largest_interval = NULL;
  insert_interval(&largest_interval, start, end);

  uint64_t largest_gap = 0;
  for (int k = 0;; k++) {
    if (largest_interval == NULL) {
      return largest_gap;
    }
    struct node *interval = pop(&largest_interval);
    uint64_t interval_size = interval->end - interval->start;

    if (largest_gap >= interval_size) {
      return largest_gap;
    }

    uint64_t mid = (interval->end - interval->start) / 2 + interval->start; // we do
     mid point calc this way to avoid overflow
    uint64_t next_prime;
    for (next_prime = mid; next_prime <= interval->end; next_prime++) {
      if (is_prime(primes, n_primes, next_prime)) {
  break;
      }
    }

    uint64_t prev_prime;
    for (prev_prime = max(mid-1, interval->start); prev_prime > interval->start;
    prev_prime--) {
```

```
      if (is_prime(primes, n_primes, prev_prime)) {
  break;
      }
    }

    if (prev_prime - interval->start > largest_gap) {
      insert_interval(&largest_interval, interval->start, prev_prime);
    }
    if (interval->end - next_prime > largest_gap) {
      insert_interval(&largest_interval, next_prime, interval->end);
    }

    uint64_t my_size = next_prime - prev_prime;
    if (my_size > largest_gap) {
      largest_gap = my_size;
    }
  }
}
```

0.6.3 Segmented Sieve

```c
#include <gmp.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdint.h>
#include <inttypes.h>

#define MAX_PRIMES 100000
#define min(x,y) (((x) < (y)) ? (x) : (y))

uint64_t gen_sieve(int64_t N, uint64_t* primes);
int sieve_primes(uint64_t* primes, unsigned int num, uint64_t start,
                 uint64_t end, uint64_t* prime_prime_gap);

int main(int argc, char** argv) {
    char hostname[MPI_MAX_PROCESSOR_NAME];
    int64_t N = 1 * pow(10, 12);

    double start_time = MPI_Wtime();

    // printf("Sieve, strike them down! N: %" PRId64 "\n", N);

    uint16_t rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating...\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    int32_t numtasks;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    uint64_t proc_gap_primes[numtasks * 3];

    int32_t rank, len;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);


    uint64_t root_n = sqrt(N);
    uint64_t sieve[root_n];
    uint64_t prime_prime_gap[5];


    uint64_t block_size = (N - root_n + 1) / numtasks;


    // integer cast safety
    uint64_t cast_rank = (uint64_t) rank;

    uint64_t start = (root_n) + (block_size * cast_rank) + 2;
    uint64_t end = (root_n) + (block_size * (cast_rank + 1)); // extend out far enough

```

```
    uint64_t num_primes = gen_sieve(N, sieve);

    // By giving a CPU cache size ahead of time, we know how much of a range to allocate
    // Since we only allocate for odds, we can afford to store twice the cache in here
    uint64_t max_cache_bytes = 1200000;

    // quick explanation: cache size is multiplied by 2 since we only store for odd numbers, so 2x
    // the portion at the end takes off 32kB from the cache. Since other stuff OTHER than this array will
    // be allocated, one can think of  this as 32KB of "safe' space to play around with.
    // It's slightly too much in many cases but it will prevent users from getting bitten by bugs if they modify the function and spill
    // over into too much RAM.
    uint64_t range_to_allocate = (max_cache_bytes * 2) - (1024 * 32);

    uint64_t sub_blocks = 0, cur_block = 0, nprimes = 0, base = start, maximal_gap = 0, l_prime, r_prime;
    sub_blocks = (end-start)/range_to_allocate;

    for(cur_block = 0; cur_block < sub_blocks; cur_block++) {
      uint64_t sub_block_start = start + (cur_block * range_to_allocate);
      uint64_t sub_block_end = start + min((cur_block + 1) * range_to_allocate, end);
      sieve_primes(sieve, num_primes, sub_block_start, sub_block_end, prime_prime_gap);

      if(prime_prime_gap[2] > maximal_gap) {
  maximal_gap = prime_prime_gap[2];
  l_prime = prime_prime_gap[3];
  r_prime = prime_prime_gap[4];
      }
    }

    prime_prime_gap[2] = maximal_gap;
    prime_prime_gap[3] = l_prime;
    prime_prime_gap[4] = r_prime;

    uint32_t dest = 0, tag = 0;

    if (rank > 0) {
      MPI_Send(prime_prime_gap, 5, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    }

    if (rank == 0) {
        uint64_t max_gap = 0, left_prime, right_prime;
  uint32_t source = 0;
  printf("Processor 0 awaiting commands...\n");
  MPI_Status status;
  int m = 0;

  proc_gap_primes[m++] = prime_prime_gap[0];
  proc_gap_primes[m++] = prime_prime_gap[1];

  max_gap = prime_prime_gap[2];
```

```
  left_prime = prime_prime_gap[3];
  right_prime = prime_prime_gap[4];

        for (source = 1; source < numtasks; source++) {
            MPI_Recv(prime_prime_gap, 5, MPI_LONG, source, MPI_ANY_TAG,
    MPI_COMM_WORLD,
                     &status);
      printf("Processor %d is now reporting in w/ gap %lu\n", source,
    prime_prime_gap[2]);
            proc_gap_primes[m++] = prime_prime_gap[0];
            proc_gap_primes[m++] = prime_prime_gap[1];
      uint64_t proc_gap = prime_prime_gap[2];
      if(proc_gap > max_gap) {
        max_gap = proc_gap;
        left_prime = prime_prime_gap[3];
        right_prime = prime_prime_gap[4];
      }
        }

        int32_t gap = 0;
  uint32_t i = 0;

        for (i = 0; i < numtasks - 1; i ++) {
    // First gap of the next processor in line - last prime of current processor
    uint64_t next_proc_first_prime = proc_gap_primes[(2*(i+1))];
    uint64_t cur_proc_last_prime = proc_gap_primes[(2*i)+1];
    gap = next_proc_first_prime - cur_proc_last_prime;

    if(gap > max_gap) {
      max_gap = gap;
      left_prime = cur_proc_last_prime;
      right_prime = next_proc_first_prime;
    }
        }

        printf("The maximum gap between primes below N = %ld is %ld. Interval: [%lu,
     %lu]\n\n", (long long) N, (long long) max_gap, left_prime, right_prime);
  double end_time = MPI_Wtime();
  printf("Total Seconds: %lf\n", end_time - start_time);
    }
    MPI_Finalize();
    return 0;
}

int cmp(const void *a, const void *b) { return (*(int *)a - *(int *)b); }

uint64_t gen_sieve(int64_t N, uint64_t* primes) {
    mpz_t mpz_num;
    mpz_init(mpz_num);
    unsigned int top, sq_top, totalPrimes = 0, n = 0;

    top = sqrt(N) + 1;

    sq_top = sqrt(top) + 1;

    // yes, this is on purpose; it's done to illustrate
    // that since we're using a mod-2 wheel, that we should NOT
    // include this as it'll cause redundant calculations.
```

```
    // Do not uncomment.

    // primes[n++] = 2;

    unsigned char isPrime;
    unsigned int i, j;

    for (i = 3; i < sq_top; i += 2) {
        mpz_set_ui(mpz_num, i);
        isPrime = mpz_probab_prime_p(mpz_num, 15);
        if (isPrime > 0) {
            primes[n++] = i;
        }
    }

    if (sq_top % 2 == 0)
        sq_top++; // Primes and even numbers don't get along

    unsigned char notPrime = 0;

    for (i = sq_top; i < top; i += 2) {
        for (j = 0; j < n; j++) {
            if (i % primes[j] == 0) {
                notPrime = 1;
                break;
            }
        }
        if (notPrime == 0) {
            primes[n++] = i;
        } else
            notPrime = 0;
    }

    return n;
}

int sieve_primes(uint64_t* primes, unsigned int num_primes,
                 uint64_t start, uint64_t end, uint64_t* prime_prime_gap_l_h) {

  time_t startTime = time(NULL);

  // Only care about odd sizes; make sure we start on one
    if (start % 2 == 0) {
        start++;
    }

    // printf("Starting point for segment:  %" PRId64 "\n", start);
    // printf("Ending point for segment: : %" PRId64 "\n", end);

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

    // 0 = prime
    // 1 = not prime
```

```
    // This may seem a little counter-intuitive, but calloc is so fast compared to
    writing
    // zeroes that we are forced to make a readability trade-off here to achieve
    that
    // There is a *POTENTIAL* speed gain here by allocating on the stack instead
    char* notPrime = (char*) calloc(1 + (end - start)/2 , sizeof(char));

    int64_t j, k = 0;
    int64_t total_count = 0;
    int64_t begin = start;

    for(j = 0; j < num_primes; j++) {
      uint64_t prime = primes[j];

      if(prime*prime >= begin) {
  for(k = prime * prime; k <= end; k += 2*prime) {
    int64_t shouldDestroy = floor((k-start)/2);
    if(shouldDestroy >= 0) {
      notPrime[shouldDestroy] = 1;
    }
        }
      }
    else {
      int64_t l =  floor( (begin - prime*prime)/(2*prime));
      for(k = (prime*prime) + (2*l*prime); k <= end; k += 2*prime) {
  int64_t shouldDestroy = floor((k - begin)/2);
  if(shouldDestroy >= 0) {
    notPrime[shouldDestroy] = 1;
  }
      }
    }

   } // end prime time loop


    uint64_t counter = 0;
    uint64_t prior_prime = 0, fprime = 0, who_prime,  mgap, new_gap = 0, l_prime,
    r_prime;
    for(counter = 0; counter < (end - start)/2+1; counter++) {
      if(notPrime[counter] == 0) {
  who_prime = (start + counter*2);
  if(prior_prime != 0) {
    mgap = who_prime - prior_prime;
    if(mgap > new_gap) {
      new_gap = mgap;
      l_prime = prior_prime;
      r_prime = who_prime;
    }
  }
  else {
    fprime = who_prime;
  }
  prior_prime = who_prime;
  total_count++;
      }
    }
```

```c
    // end sieve, stop the clock
    time_t finish = time(NULL) - startTime;
    // printf("Sieve finished in %lds.. gap: %lu. F: %lu P: %lu\n", finish, new_gap,
     fprime, who_prime);

    prime_prime_gap_l_h[0] = fprime << 48; // shift out, allow for truncuation; see.
     seg. e.
    prime_prime_gap_l_h[1] = who_prime << 48;
    prime_prime_gap_l_h[2] = new_gap;
    prime_prime_gap_l_h[3] = l_prime;
    prime_prime_gap_l_h[4] = r_prime;

    // free memory; signal to CPU cache that it can be evicted
    free(notPrime);

    return total_count;
}
```

Listing 8: Interval method