ChatGPT

Ultra-Fast Prime Generator Design and Implementation

Overview of the Goal

The goal is to create an **ultra-fast prime number generator** that uses massive parallelism. We want to continuously generate large random numbers and test each for primality using the Miller–Rabin probabilistic test. The system should be structured as a high-throughput pipeline with distinct stages for random number generation and primality testing, efficiently utilizing all available CPU cores.

Key requirements: - **Massively parallel random generation:** Use multiple threads (and recursive thread spawning) to fill buffers with cryptographically strong random bytes (using `getrandom()` from `<sys/random.h>`). - **Concurrent primality testing:** Use a pool of threads to perform Miller–Rabin primality tests on the generated numbers in parallel (using a big-integer library like GMP for huge integers). - **Pipelined design:** While some threads generate new random numbers, others simultaneously test previously generated numbers for primality. This ensures no CPU core sits idle – as soon as a random candidate is ready, it is dispatched for testing. - **Optimized resource split:** Roughly 26.67% of CPU resources dedicated to generation, ~53.33% to primality tests, and the remainder as overhead/buffer management. On a machine with 160 virtual cores (e.g. 128 for main tasks + 32 reserved), this corresponds to about 42 generation threads, 85 tester threads, and ~32 cores headroom for system or buffering tasks. We will derive thread counts from the available cores to maintain this balance.

The ultimate objective is to maximize the throughput of prime number discovery – generating and testing as many random numbers as possible in a given time, producing a pipeline of prime numbers. In future iterations, a GPU or AI model might even assist by learning patterns from tested numbers (for example, to predict likely prime candidates), but in this implementation we will focus on the CPU pipeline and leave hooks for such enhancements.

Multi-Threaded Pipeline Approach

Random Number Generation Strategy

For generating random bytes, we use the *hexadecimal entropy tree* approach inspired by the original `hexentropy_v1.c`. In that method, a buffer is filled recursively by splitting the range and spawning two threads at each recursion level (creating an **exponential thread tree**). Each recursion level writes a “pivot” byte (for even-length segments, using a random bit to set it to either `0x00` or `0x07` as a recognizable pattern) and then splits into two sub-tasks.

However, to avoid excessive thread creation and to adapt to a continuous pipeline, we adjust this approach: - We **remove the fixed recursion depth limit** (originally `MAX_DEPTH = 4`) or make it dynamic based on available cores. This ensures we can fully exploit a large number of CPUs by continuing to spawn threads per recursion until segments are small enough or we’ve reached a sensible depth relative to core count. - We **tune the chunk size threshold** (originally `SMALL_CHUNK = 32` bytes) to avoid threading overhead on tiny segments. Very small segments are filled in-place without further recursion. - If generating *extremely large* numbers (thousands of bytes), the recursive splitting ensures no single thread becomes a bottleneck – multiple threads will fill different portions of the buffer concurrently. On the other hand, if we are generating many numbers concurrently, we might opt for a simpler generation (single `getrandom()` call per number) to avoid *over-subscribing* threads. In this design, we use **multiple parallel generation tasks** to saturate CPU cores, so each task can simply fill its buffer sequentially (one `getrandom` call) – the concurrency across tasks provides the parallelism. This simplifies the generation stage and avoids duplicating effort with both inter-number and intra-number threading. - Each generated random number is stored in a buffer (byte array). We treat the byte array as a big-endian representation of an integer (the first byte as the most significant, to preserve intuitive ordering when printing in hex).

Primality Testing Strategy

For each random number generated, we perform a **Miller–Rabin probabilistic primality test** (with a high number of rounds for reliability, e.g. 25–40 rounds). We leverage the GNU Multiple Precision (GMP) library to handle large integers efficiently: - We import the random byte buffer into a GMP `mpz_t` big integer. - We use `mpz_probab_prime_p(n, reps)` which does some trial divisions and then `reps` Miller–Rabin tests [1]. This function returns 0 for "definitely composite," 1 for "probably prime," and 2 if it's certain the number is prime (e.g., for small numbers or after extra checks) [2]. - A higher number of rounds (`reps`) reduces the probability of a composite being misidentified as prime (the error probability is < 4^(-reps) per Miller–Rabin theory [1]). We will use a default like 25 or 40 rounds to ensure a very high level of confidence in primality results. - If `mpz_probab_prime_p` indicates the number is prime (return value 1 or 2), we count it as a prime found. (In a real cryptographic prime generator, we might even perform additional tests or deterministic checks if needed, but Miller–Rabin with enough rounds is extremely reliable for large random numbers.)

Thread Allocation and Pipeline Coordination

We adopt a **producer-consumer model** with two thread pools: - **Producer threads (Random Generators):** These threads continuously generate random numbers and place them into a thread-safe queue/buffer for testing. We will allocate roughly 1/3 of available cores to generation. For example, on a 160-vCPU setup, ~42 threads would be dedicated to random generation. - **Consumer threads (Prime Testers):** These threads continuously take numbers from the queue and perform the primality test. About 2/3 of cores will be for testing (e.g. ~85 threads on a 160-core machine), reflecting that primality tests are more computationally intensive and should get more CPU share. - **Buffer and Orchestration:** The remaining small fraction of cores (if any) can handle buffer management or remain idle as headroom. We implement a fixed-size thread-safe queue to hold numbers awaiting testing. This prevents unbounded memory usage if generators are faster than testers. If the queue is full, generator threads will wait (back-pressure). This ensures we don't overwhelm memory. In a future iteration, if we wanted **no waiting** on generation (to truly use 100% of generation allotment at all times), we could integrate an offloading mechanism: for example, if the test queue is full and no tester thread is free, we might hand off the candidate to an AI model or GPU to predict primality – but that is beyond the scope of our current C implementation, so we will simply block until a tester is available.

We determine thread counts dynamically from the hardware: - We get the number of online processors (e.g., using `sysconf(_SC_NPROCESSORS_ONLN)`). - We then calculate counts for generation and testing threads according to the 4:8:3 ratio (Gen : Test : Buffer) that corresponds to 26.67% vs 53.33% (the remaining ~20% are buffer/overhead). For simplicity, we might round these or ensure at least 1 thread in each category. - If the machine has fewer cores (for example, during testing on a smaller machine), the ratio still applies. (On very small core counts, generation or testing might end up with 1 thread while others get 2, etc. – the program will still function with reduced parallelism.)

Memory Management and Data Flow

Each number is represented as a *byte buffer*. In our pipeline: 1. A generator thread allocates a buffer of the desired length (in bytes) for a new number (or uses a recycled buffer from a pool), fills it with random bytes via `getrandom()`, then enqueues it for testing. 2. A tester thread dequeues the ready buffer, imports it into a big integer (`mpz_t`), and runs the primality test. After testing: - If the number is prime, we increment a prime counter (and we could output or log the prime if needed – though printing every prime would slow down the system significantly, so for benchmarking we might skip immediate output and just count). - The tester thread then frees the buffer memory (returning it to the system or to a pool). 3. This cycle repeats until we have generated the requested number of candidates (if a target count is specified), or indefinitely if we are just measuring throughput.

To minimize overhead: - We can maintain a fixed-size pool of buffer slots to avoid constantly allocating and freeing memory. However, for simplicity, our initial implementation will allocate a new buffer for each number and free it after use. This is straightforward and safe, and modern memory allocators handle small allocations efficiently. If this becomes a bottleneck, we could switch to a custom buffer pool. - We avoid locking during heavy computation: the queue lock is held **only** to push or pop pointers (which is very quick). The expensive operations (random byte generation and primality testing) happen outside the lock. This maximizes parallelism and avoids holding up other threads. - We use condition variables to block threads when they can't proceed (e.g., generation waits if queue is full, testing waits if queue is empty). This way, CPU cycles aren't wasted on active waiting.

Benchmarking and Output

The program will measure the overall time taken to generate and test a given number of candidates. For example, we might attempt to generate and test 10,000,000,000 (10 billion) numbers and observe how many seconds that takes – this is a very large number likely used as a theoretical target. In practice, one would start with a smaller count to gauge performance, then extrapolate or gradually increase.

At the end of execution (or periodically), the program can output: - Total numbers generated and tested. - Total prime numbers found. - Total time taken (in seconds, possibly with high-resolution). - Throughput metrics (e.g. candidates per second, primes found per second, success probability of random candidates being prime, etc.).

For now, we will implement a final summary after generating the requested count: - E.g., "Generated X candidate numbers of length N bytes in Y.YY seconds. Primes found: Z."

This provides a clear benchmark result.

With the design explained, below is the **proposed code**. We break it into components for clarity, though it could be combined. You can compile this with **GCC and pthread & GMP libraries** (e.g., `gcc -O2 -pthread -lgmp main.c -o primegen_pipeline`). Make sure GMP is installed for linking. (OpenSSL's BN library could be used as well, but GMP offers a convenient high-level API for primality testing.)

Implementation Details and Code

Main Program (`main.c`)

This file sets up the pipeline, parses arguments, and launches the threads. It contains the shared queue and synchronization primitives.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#include <sys/random.h>
#include <sys/time.h>
#include <stdatomic.h>
#include <gmp.h>

// Default ratio parts for generation, testing, overhead
static const int RATIO_GEN = 4;
static const int RATIO_TEST = 8;
static const int RATIO_OVERHEAD = 3;

// Shared queue for buffers awaiting primality test
typedef struct {
    unsigned char **buffers;  // array of pointers to byte buffers
    size_t capacity;
    size_t front;
    size_t count;
    pthread_mutex_t mutex;
    pthread_cond_t not_empty;
    pthread_cond_t not_full;
} BufferQueue;

// Global queue instance
static BufferQueue queue;

// Global flags/counters
static atomic_ullong next_index;    // atomic counter for number generation
(to assign tasks)
static unsigned long long total_to_generate = 0;
static atomic_ullong primes_found;  // atomic counter for primes found

// Thread counts
static int num_gen_threads = 0;
static int num_test_threads = 0;

// Buffer parameters
static size_t number_length = 0;  // length of each number in bytes
```

```
// Initialize the queue
static void queue_init(BufferQueue *q, size_t capacity) {
    q->buffers = malloc(capacity * sizeof(unsigned char*));
    if (!q->buffers) {
        fprintf(stderr, "Queue buffer malloc failed\\n");
        exit(EXIT_FAILURE);
    }
    q->capacity = capacity;
    q->front = 0;
    q->count = 0;
    pthread_mutex_init(&q->mutex, NULL);
    pthread_cond_init(&q->not_empty, NULL);
    pthread_cond_init(&q->not_full, NULL);
}

// Destroy the queue
static void queue_destroy(BufferQueue *q) {
    free(q->buffers);
    pthread_mutex_destroy(&q->mutex);
    pthread_cond_destroy(&q->not_empty);
    pthread_cond_destroy(&q->not_full);
}

// Enqueue a buffer pointer (thread-safe)
static void queue_enqueue(BufferQueue *q, unsigned char *buffer) {
    pthread_mutex_lock(&q->mutex);
    // Wait until there is space
    while (q->count == q->capacity) {
        pthread_cond_wait(&q->not_full, &q->mutex);
    }
    // Compute insertion index (front + count mod capacity)
    size_t idx = (q->front + q->count) % q->capacity;
    q->buffers[idx] = buffer;
    q->count++;
    // Signal that queue is not empty
    pthread_cond_signal(&q->not_empty);
    pthread_mutex_unlock(&q->mutex);
}

// Dequeue a buffer pointer (thread-safe)
static unsigned char* queue_dequeue(BufferQueue *q) {
    pthread_mutex_lock(&q->mutex);
    // Wait until there is an item
    while (q->count == 0) {
        // If no more numbers will ever be produced (stop condition), break
out
        // (We handle this via sentinel NULL pointers, so normally not needed
to check here)
        pthread_cond_wait(&q->not_empty, &q->mutex);
    }
    // Remove item from front
```

```
    unsigned char *buffer = q->buffers[q->front];
    q->front = (q->front + 1) % q->capacity;
    q->count--;
    // Signal that queue has space
    pthread_cond_signal(&q->not_full);
    pthread_mutex_unlock(&q->mutex);
    return buffer;
}

// Generator thread function: produce random numbers and enqueue them
static void* generator_thread_func(void *arg) {
    (void)arg; // unused
    // Each thread uses its own buffer for each number to avoid contention
    // We allocate fresh buffers per number below for simplicity
    unsigned char *buffer;
    // Loop until we've generated all required numbers
    while (true) {
        // Atomically get the next index to generate
        unsigned long long idx = atomic_fetch_add(&next_index, 1);
        if (idx >= total_to_generate) {
            // All numbers assigned
            break;
        }
        // Allocate memory for the new number's bytes
        buffer = (unsigned char*)malloc(number_length);
        if (!buffer) {
            perror("malloc (number buffer)");
            break;
        }
        // Fill the buffer with cryptographically secure random bytes
        if (getrandom(buffer, number_length, 0) != (ssize_t)number_length) {
            perror("getrandom");
            free(buffer);
            // If getrandom fails, we abort the generation
            break;
        }
        // Enqueue the buffer for testing
        queue_enqueue(&queue, buffer);
    }
    return NULL;
}

// Tester thread function: consume numbers and test for primality
static void* tester_thread_func(void *arg) {
    (void)arg;
    // Initialize a big integer for reuse in this thread
    mpz_t n;
    mpz_init(n);
    // Continuously dequeue buffers and test them
    while (true) {
        unsigned char *buffer = queue_dequeue(&queue);
```

```c
        if (buffer == NULL) {
            // NULL buffer is a signal to terminate
            break;
        }
        // Import the bytes into a big integer (treat buffer as big-endian)
        mpz_import(n, number_length, 1, 1, 1, 0, buffer);
        // Perform Miller-Rabin primality test with a given number of rounds
        int result = mpz_probab_prime_p(n, 25);
        if (result >= 1) {
            // Probably prime (or definitely prime)
            atomic_fetch_add(&primes_found, 1);
            // (Optionally, we could output or log the prime here,
            // but that is omitted for performance.)
        }
        // Free the buffer memory
        free(buffer);
    }
    // Clear big integer
    mpz_clear(n);
    return NULL;
}

int main(int argc, char *argv[]) {
    // Parse command-line arguments
    // Usage: ./primegen [count] [length]
    // If only one argument, treat it as the byte length (for one number);
    // if two arguments, first is count of numbers, second is byte length.
    total_to_generate = 1;
    number_length = 64; // default 64 bytes (~512 bits)
    if (argc >= 2) {
        // If two arguments provided, use first as count, second as length
        char *endptr = NULL;
        unsigned long long maybeCount = strtoull(argv[1], &endptr, 10);
        if (argc >= 3) {
            if (maybeCount == 0) {
                fprintf(stderr, "Invalid count\\n");
                return EXIT_FAILURE;
            }
            total_to_generate = maybeCount;
            number_length = strtoul(argv[2], &endptr, 10);
        } else {
            // Only one arg: interpret as length (keep count = 1)
            number_length = strtoul(argv[1], &endptr, 10);
        }
        if (number_length == 0) {
            fprintf(stderr, "Invalid length\\n");
            return EXIT_FAILURE;
        }
    }
    // Determine thread counts based on available CPUs
    long nproc = sysconf(_SC_NPROCESSORS_ONLN);
```

```
    if (nproc < 1) nproc = 1;
    // Calculate total ratio parts
    int total_ratio = RATIO_GEN + RATIO_TEST + RATIO_OVERHEAD;
    // Assign threads according to ratio, ensuring at least 1 in each
category if possible
    num_gen_threads = (int)((RATIO_GEN * nproc) / total_ratio);
    num_test_threads = (int)((RATIO_TEST * nproc) / total_ratio);
    if (num_gen_threads < 1) num_gen_threads = 1;
    if (num_test_threads < 1) num_test_threads = 1;
    // Keep a few cores as overhead if possible (not creating threads for
those)
    // (We don't explicitly use overhead threads in this implementation; they
remain idle or for OS.)

    size_t queue_capacity = (size_t)(num_test_threads * 2);
    queue_init(&queue, queue_capacity);
    atomic_init(&next_index, 0);
    atomic_init(&primes_found, 0);

    // Allocate and launch generator threads
    pthread_t *gen_threads = malloc(num_gen_threads * sizeof(pthread_t));
    pthread_t *test_threads = malloc(num_test_threads * sizeof(pthread_t));
    if (!gen_threads || !test_threads) {
        fprintf(stderr, "Thread allocation failed\\n");
        return EXIT_FAILURE;
    }

    for (int i = 0; i < num_gen_threads; ++i) {
        if (pthread_create(&gen_threads[i], NULL, generator_thread_func,
NULL) != 0) {
            fprintf(stderr, "Error creating generator thread %d\\n", i);
            return EXIT_FAILURE;
        }
    }
    for (int j = 0; j < num_test_threads; ++j) {
        if (pthread_create(&test_threads[j], NULL, tester_thread_func,
NULL) != 0) {
            fprintf(stderr, "Error creating tester thread %d\\n", j);
            return EXIT_FAILURE;
        }
    }

    // Start timing
    struct timeval start, end;
    gettimeofday(&start, NULL);

    // Wait for all generator threads to finish producing
    for (int i = 0; i < num_gen_threads; ++i) {
        pthread_join(gen_threads[i], NULL);
    }
    // After generation is done, enqueue termination signals (NULL) for each
```

```
tester thread
    for (int k = 0; k < num_test_threads; ++k) {
        queue_enqueue(&queue, NULL);
    }
    // Wait for all tester threads to finish
    for (int j = 0; j < num_test_threads; ++j) {
        pthread_join(test_threads[j], NULL);
    }

    gettimeofday(&end, NULL);
    // Compute elapsed time in seconds
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec -
start.tv_usec) / 1000000.0;

    // Print benchmark results
    unsigned long long primes = atomic_load(&primes_found);
    printf("Generated %llu %zu-byte numbers in %.2f seconds. Primes found:
%llu.\\n",
           total_to_generate, number_length, elapsed, primes);

    // Cleanup
    free(gen_threads);
    free(test_threads);
    queue_destroy(&queue);
    return 0;
}
```

Explanation: In `main`, we parse the desired number of random numbers (`count`) and the length of each number in bytes. We determine how many threads to use for generation and testing based on the system's core count and the specified ratios (ensuring at least one thread each to avoid zero threads if core count is small). We initialize a `BufferQueue` structure to manage the pointer queue with a given capacity (we chose `2 * num_test_threads` as the queue length to buffer up to twice the number of concurrent tests). The queue is protected by a mutex and two condition variables (`not_empty` and `not_full`).

We create the generator and tester threads. The **generator threads** run `generator_thread_func` which loops until the global atomic `next_index` reaches `total_to_generate`. Each iteration, a generator thread allocates a buffer, fills it with `getrandom()`, and enqueues it. The `atomic_fetch_add` on `next_index` ensures that exactly `total_to_generate` numbers will be generated in total (threads share this counter and stop when the quota is reached). The **tester threads** run `tester_thread_func` which continuously dequeues buffers and checks for primality using GMP. We reuse a GMP `mpz_t` in each tester thread to avoid re-initializing it for every number (improving performance). The use of `mpz_import` interprets the bytes in big-endian order into the big integer. After testing, the buffer is freed.

To gracefully shut down, once all generator threads are done, the main thread enqueues a `NULL` pointer sentinel for each tester thread. When a tester thread dequeues a `NULL`, it breaks out of the loop and exits. This ensures all tester threads terminate once processing of real tasks is complete. We then join all threads and measure the elapsed time.

Usage Example

If you compile this program (let's call it `primegen_pipeline`), you can run it as:

```
./primegen_pipeline 10000 64
```

This would use the default thread distribution based on your cores to generate **10,000 random 512-bit numbers** and test each for primality. At the end, it will report how many primes were found and how long it took.

Potential Enhancements

- **Dynamic Depth for Generation Threads:** We simplified generation to one `getrandom` call per number for efficiency when generating many numbers concurrently. If instead we wanted to generate a single very large number as fast as possible, we could reintroduce the recursive splitting (as in the original hexentropy approach) to spawn threads within the generation of one number. For example, if only one number is being generated at a time, using the recursive thread splitting would utilize more cores for that single generation. In a pipeline scenario with many numbers, inter-number parallelism already keeps cores busy. We could make this adaptive: e.g., if `num_gen_threads` is much less than available cores (meaning each generation thread could itself use multiple cores), spawn threads inside generation for chunks beyond a certain size. This adds complexity and would require a thread pool or nested pthread usage, so it must be balanced against the overhead.
- **Memory Pooling:** As noted, the current implementation calls `malloc`/`free` for each number's byte buffer. This could be optimized by maintaining a pool of fixed-size buffers (since all numbers are the same length) and recycling them. This would avoid repeated allocation and improve cache reuse. Given that primality testing is relatively heavy, the impact of allocation might be small, but for generating billions of numbers it could become significant.
- **Miller–Rabin Round Tuning:** We used 25 rounds of Miller–Rabin by default. This is usually sufficient for cryptographic prime testing (it makes the probability of a false prime extremely low: < 4^(-25) ≈ 1e-15). We could allow this to be configurable or increased for even more certainty. GMP's implementation also does some trial division and possibly a strong Lucas test [1], so it's quite robust.
- **Output and Logging:** Currently, we only output a summary. If needed (for debugging or smaller runs), one could print each prime number found or log them to a file. That should be done cautiously, as I/O can severely bottleneck the pipeline if too frequent.
- **Integration with Learning/AI (Future):** The design leaves about 20% of CPU resources unallocated to threads, which could be used for ancillary tasks. In the future, one might run a lightweight thread that monitors the bit patterns of numbers tested and primes found, to feed a machine learning model (possibly running on GPU) that tries to predict which random candidates might be prime. Such a model could propose candidates (bypassing some random generation) or assist the primality test (e.g., a heuristic to skip Miller–Rabin for obviously composite patterns). Implementing this would require a significantly more complex system (and likely inter-process or GPU communication), but the pipeline structure here is a solid foundation for experimentation.

Conclusion

We have rewritten and optimized the original `hexentropy_v1.c` into a **multi-threaded prime generator pipeline**. The solution uses **recursive multi-threading for generation (conceptually)** and a **parallel primality-testing stage**, achieving a highly parallel flow of generating random numbers and identifying primes among them. By carefully balancing thread usage (roughly one part generation to two parts testing) and using thread-safe queues to connect stages, the design ensures maximal throughput. This implementation can generate an “infernal pipeline” of prime numbers, continuously pumping candidates through the Miller–Rabin test as fast as the hardware allows. The use of GMP for big integers and primality ensures correctness and efficiency for **very, very large numbers**, and the structure is in place for further enhancements such as GPU inference integration.

This completes the deep dive into the code. *Bonne chance* – we now have a blueprint for an ultra-fast prime number generator!

---

[1] [2] gmp_lib.mpz_probab_prime_p Method
https://machinecognitis.github.io/Math.Gmp.Native/html/52ce0428-7c09-f2b9-f517-d3d02521f365.htm