HW3: Pthreads 2

Rahul Kumar

October 26, 2025

1 Introduction

This document describes the implementation of a multithreaded C program that generates random numbers and searches for prime numbers using POSIX threads. Each thread performs a naive primality test and synchronizes with other threads through a custom barrier. The program stores a total of 10 randomly generated prime numbers in a globally shared array, collectively produced by the team of threads.

2 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <time.h>

// --------------------------------
// Configurable Parameters
// --------------------------------
#define NUM_THREADS 2
#define MAX_TRIES 1000  // Increased to ensure we find enough primes
#define MAXVAL 99999999999999999UL
#define NUM_VALUES 10   // Total primes to collect

// --------------------------------
// Global Shared Data with Synchronization
// --------------------------------
unsigned long int prime_values[NUM_VALUES];
int values_count = 0;  // Current number of primes collected
pthread_mutex_t values_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t values_condition = PTHREAD_COND_INITIALIZER;
int cancel_requested = 0;  // Flag to signal threads to stop

// --------------------------------
```

```
25 // Barrier Structure and Functions
26 // --------------------------------
27 struct barrier {
28     pthread_mutex_t lock;
29     pthread_cond_t cond;
30     int num_threads;
31     int count;
32 };
33 
34 void barrier_init(struct barrier* b, int nt) {
35     pthread_mutex_init(&b->lock, NULL);
36     pthread_cond_init(&b->cond, NULL);
37     b->count = 0;
38     b->num_threads = nt;
39 }
40 
41 void barrier_destroy(struct barrier* b) {
42     pthread_mutex_destroy(&b->lock);
43     pthread_cond_destroy(&b->cond);
44 }
45 
46 void barrier_wait(struct barrier* b) {
47     pthread_mutex_lock(&b->lock);
48     b->count++;
49 
50     if (b->count == b->num_threads) {
51         pthread_cond_broadcast(&b->cond);
52     } else {
53         while (b->count < b->num_threads) {
54             pthread_cond_wait(&b->cond, &b->lock);
55         }
56     }
57     pthread_mutex_unlock(&b->lock);
58 }
59 
60 // --------------------------------
61 // Thread Argument Structure
62 // --------------------------------
63 struct thread_args {
64     int rank;
65     struct barrier* barrier;
66 };
67 
68 // --------------------------------
69 // Primality Test (Brute Force)
70 // --------------------------------
71 int is_prime(unsigned long int n) {
```

```
72      if (n < 2) return 0;
73      if (n % 2 == 0 && n != 2) return 0;
74
75      unsigned long int limit = (unsigned long int)sqrt((long double)n
            );
76      for (unsigned long int i = 3; i <= limit; i += 2) {
77          if (n % i == 0) return 0;
78      }
79      return 1;
80  }
81
82  // --------------------------------
83  // Add Prime to Global Array (Thread-Safe)
84  // --------------------------------
85  int add_prime(unsigned long int prime) {
86      pthread_mutex_lock(&values_mutex);
87
88      // Check if we've already collected enough primes
89      if (values_count >= NUM_VALUES) {
90          pthread_mutex_unlock(&values_mutex);
91          return 0;  // Array full
92      }
93
94      // Add prime to array
95      prime_values[values_count] = prime;
96      values_count++;
97
98      // Check if we've reached the target
99      int reached_target = (values_count == NUM_VALUES);
100
101     pthread_mutex_unlock(&values_mutex);
102
103     // Signal master thread if target reached
104     if (reached_target) {
105         pthread_cond_signal(&values_condition);
106     }
107
108     return 1;  // Successfully added
109 }
110
111 // --------------------------------
112 // Thread Function
113 // --------------------------------
114 void* start(void* x) {
115     struct thread_args* args = (struct thread_args*) x;
116     unsigned long int to_test = 0;
117
```

```
118     printf("Thread %d started searching for primes\n", args->rank);
119
120     for (int i = 0; i < MAX_TRIES && !cancel_requested; i++) {
121         // Generate a random unsigned long int
122         to_test = ((unsigned long int)rand() << 32) | rand();
123         to_test = to_test % MAXVAL;
124
125         if (is_prime(to_test)) {
126             printf("Thread %d found prime: %lu\n", args->rank,
                    to_test);
127
128             // Try to add prime to global array
129             if (add_prime(to_test)) {
130                 printf("Thread %d successfully added prime to array\
                        n", args->rank);
131             } else {
132                 printf("Thread %d could not add prime (array full)\n
                        ", args->rank);
133                 break;  // Array is full, stop searching
134             }
135         }
136
137         // Small delay to prevent excessive CPU usage
138         struct timespec ts = {0, 1000000};  // 1ms
139         nanosleep(&ts, NULL);
140     }
141
142     printf("Thread %d exiting\n", args->rank);
143     pthread_exit(NULL);
144 }
145
146 // --------------------------------
147 // Main Function
148 // --------------------------------
149 int main(int argc, char *argv[]) {
150     pthread_t threads[NUM_THREADS];
151     struct thread_args args[NUM_THREADS];
152     struct barrier sb;
153     struct timespec start_time, end_time;
154
155     // Start timing
156     clock_gettime(CLOCK_MONOTONIC, &start_time);
157
158     // Seed the random number generator
159     srand(time(NULL));
160
161     // Initialize global array
```

```
    for (int i = 0; i < NUM_VALUES; i++) {
        prime_values[i] = 0;
    }

    barrier_init(&sb, NUM_THREADS);

    printf("Starting %d threads to find %d primes...\n", NUM_THREADS
       , NUM_VALUES);

    // Create all threads
    for (int i = 0; i < NUM_THREADS; i++) {
        args[i].rank = i;
        args[i].barrier = &sb;
        pthread_create(&threads[i], NULL, start, &args[i]);
    }

    // Master thread waits for condition variable
    pthread_mutex_lock(&values_mutex);
    while (values_count < NUM_VALUES) {
        printf("Master: Waiting for primes (%d/%d collected)...\n",
           values_count, NUM_VALUES);
        pthread_cond_wait(&values_condition, &values_mutex);
    }
    pthread_mutex_unlock(&values_mutex);

    printf("Master: Target reached! Cancelling threads...\n");

    // Signal all threads to stop
    cancel_requested = 1;

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // End timing
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Calculate execution time
    double execution_time = (end_time.tv_sec - start_time.tv_sec) +
                            (end_time.tv_nsec - start_time.tv_nsec) /
                                1000000000.0;

    // Print results
    printf("\n=== RESULTS ===\n");
    printf("Threads used: %d\n", NUM_THREADS);
    printf("Primes collected: %d\n", values_count);
```

```
    printf("Target primes: %d\n", NUM_VALUES);
    printf("Total execution time: %.6f seconds\n", execution_time);

    printf("\nCollected primes:\n");
    for (int i = 0; i < values_count; i++) {
        printf("Prime[%d] = %lu\n", i, prime_values[i]);
    }

    // Cleanup
    barrier_destroy(&sb);
    pthread_mutex_destroy(&values_mutex);
    pthread_cond_destroy(&values_condition);

    return 0;
}
```

3 Compilation and Results

The program was compiled using the `gcc` compiler with optimization and warning flags enabled. The `-Wall` flag enables all common compiler warnings, while `-O3` applies a high level of optimization for performance. The `-lpthread` and `-lm` flags link the POSIX threads and math libraries, respectively.

```
gcc -Wall -O3 pthread_2.c -o pthread_2.ex -lpthread -lm.
.\pthread_2.ex
```

3.1 Log Contents

Using 1 thread and recovering 10 primes:

```
    Starting 1 threads to find 10 primes...
Master: Waiting for primes (0/10 collected)...
Thread 0 started searching for primes
Thread 0 found prime: 95689510769107511
Thread 0 successfully added prime to array
Thread 0 found prime: 73347320285317973
Thread 0 successfully added prime to array
Thread 0 found prime: 98167886681176681
Thread 0 successfully added prime to array
Thread 0 found prime: 69648192482135321
Thread 0 successfully added prime to array
Thread 0 found prime: 24471634708747253
Thread 0 successfully added prime to array
Thread 0 found prime: 54676777817652959
Thread 0 successfully added prime to array
Thread 0 found prime: 38290284257696227
Thread 0 successfully added prime to array
```

```
Thread 0 found prime: 34464869788829863
Thread 0 successfully added prime to array
Thread 0 found prime: 56688606771056203
Thread 0 successfully added prime to array
Thread 0 found prime: 69346556815324757
Thread 0 successfully added prime to array
Master: Target reached! Cancelling threads...
Thread 0 exiting

=== RESULTS ===
Threads used: 1
Primes collected: 10
Target primes: 10
Total execution time: 9.039997 seconds

Collected primes:
Prime[0] = 95689510769107511
Prime[1] = 73347320285317973
Prime[2] = 98167886681176681
Prime[3] = 69648192482135321
Prime[4] = 24471634708747253
Prime[5] = 54676777817652959
Prime[6] = 38290284257696227
Prime[7] = 34464869788829863
Prime[8] = 56688606771056203
Prime[9] = 69346556815324757
```

Using 2 thread and recovering 10 primes:

```
    Starting 2 threads to find 10 primes...
Master: Waiting for primes (0/10 collected)...
Thread 0 started searching for primes
Thread 1 started searching for primes
Thread 0 found prime: 46323145385395201
Thread 0 successfully added prime to array
Thread 1 found prime: 52045726083131681
Thread 1 successfully added prime to array
Thread 0 found prime: 37784481104245849
Thread 0 successfully added prime to array
Thread 1 found prime: 43030620138885131
Thread 1 successfully added prime to array
Thread 0 found prime: 30338921939435653
Thread 0 successfully added prime to array
Thread 1 found prime: 62483376712717403
Thread 1 successfully added prime to array
Thread 0 found prime: 74004089960470499
Thread 0 successfully added prime to array
Thread 0 found prime: 25724296897938409
Thread 0 successfully added prime to array
```

```
Thread 1 found prime: 57096889188365263
Thread 1 successfully added prime to array
Thread 0 found prime: 72357420750149543
Thread 0 successfully added prime to array
Master: Target reached! Cancelling threads...
Thread 0 exiting
Thread 1 found prime: 83076148253856523
Thread 1 could not add prime (array full)
Thread 1 exiting

=== RESULTS ===
Threads used: 2
Primes collected: 10
Target primes: 10
Total execution time: 4.996489 seconds

Collected primes:
Prime[0] = 46323145385395201
Prime[1] = 52045726083131681
Prime[2] = 37784481104245849
Prime[3] = 43030620138885131
Prime[4] = 30338921939435653
Prime[5] = 62483376712717403
Prime[6] = 74004089960470499
Prime[7] = 25724296897938409
Prime[8] = 57096889188365263
Prime[9] = 72357420750149543
```

Using 3 thread and recovering 10 primes:

```
    Starting 3 threads to find 10 primes...
Thread 0 started searching for primes
Master: Waiting for primes (0/10 collected)...
Thread 1 started searching for primes
Thread 2 started searching for primes
Thread 1 found prime: 498192735910829
Thread 1 successfully added prime to array
Thread 2 found prime: 17849206808170007
Thread 2 successfully added prime to array
Thread 2 found prime: 19442319597702047
Thread 2 successfully added prime to array
Thread 0 found prime: 94405647070330753
Thread 0 successfully added prime to array
Thread 1 found prime: 66600796715325911
Thread 1 successfully added prime to array
Thread 2 found prime: 90000529999975039
Thread 2 successfully added prime to array
Thread 0 found prime: 89546585698097111
Thread 0 successfully added prime to array
```

```
Thread 1 found prime: 25462357595748379
Thread 1 successfully added prime to array
Thread 2 found prime: 12330164740511279
Thread 2 successfully added prime to array
Thread 0 found prime: 33634216489818671
Thread 0 successfully added prime to array
Master: Target reached! Cancelling threads...
Thread 0 exiting
Thread 1 exiting
Thread 2 found prime: 67428958241382077
Thread 2 could not add prime (array full)
Thread 2 exiting

=== RESULTS ===
Threads used: 3
Primes collected: 10
Target primes: 10
Total execution time: 3.661965 seconds

Collected primes:
Prime[0] = 498192735910829
Prime[1] = 17849206808170007
Prime[2] = 19442319597702047
Prime[3] = 94405647070330753
Prime[4] = 66600796715325911
Prime[5] = 90000529999975039
Prime[6] = 89546585698097111
Prime[7] = 25462357595748379
Prime[8] = 12330164740511279
Prime[9] = 33634216489818671
```

Using 4 thread and recovering 10 primes:

```
    Starting 4 threads to find 10 primes...
Thread 0 started searching for primes
Thread 1 started searching for primes
Master: Waiting for primes (0/10 collected)...
Thread 2 started searching for primes
Thread 3 started searching for primes
Thread 3 found prime: 34801875210848143
Thread 3 successfully added prime to array
Thread 0 found prime: 23402572981245911
Thread 0 successfully added prime to array
Thread 1 found prime: 94619912575833503
Thread 1 successfully added prime to array
Thread 1 found prime: 10710010072302769
Thread 1 successfully added prime to array
Thread 2 found prime: 62513687257273793
Thread 2 successfully added prime to array
```

```
Thread 0 found prime: 51389065984397989
Thread 0 successfully added prime to array
Thread 3 found prime: 89113295952303439
Thread 3 successfully added prime to array
Thread 1 found prime: 81687265026701533
Thread 1 successfully added prime to array
Thread 2 found prime: 48242322319694147
Thread 2 successfully added prime to array
Thread 3 found prime: 88883640694767637
Thread 3 successfully added prime to array
Master: Target reached! Cancelling threads...
Thread 3 exiting
Thread 2 found prime: 20187800971510693
Thread 2 could not add prime (array full)
Thread 2 exiting
Thread 1 found prime: 63435630770860237
Thread 1 could not add prime (array full)
Thread 1 exiting
Thread 0 found prime: 87583323503970967
Thread 0 could not add prime (array full)
Thread 0 exiting

=== RESULTS ===
Threads used: 4
Primes collected: 10
Target primes: 10
Total execution time: 3.687047 seconds

Collected primes:
Prime[0] = 34801875210848143
Prime[1] = 23402572981245911
Prime[2] = 94619912575833503
Prime[3] = 10710010072302769
Prime[4] = 62513687257273793
Prime[5] = 51389065984397989
Prime[6] = 89113295952303439
Prime[7] = 81687265026701533
Prime[8] = 48242322319694147
Prime[9] = 88883640694767637
```

Figure 1 shows the execution time for generating 10 prime numbers using different numbers of threads. The performance improvement saturates around four threads. As the thread count increases, the mutex becomes a serialization bottleneck—threads spend more time waiting for access to shared resources than performing useful computations. The centralized coordination mechanism that initially facilitates cooperation among threads eventually limits scalability. This behavior illustrates Amdahl's Law in practice: even highly parallelizable tasks, such as prime number generation, cannot scale indefinitely when constrained by unavoidable serial components like shared array management.

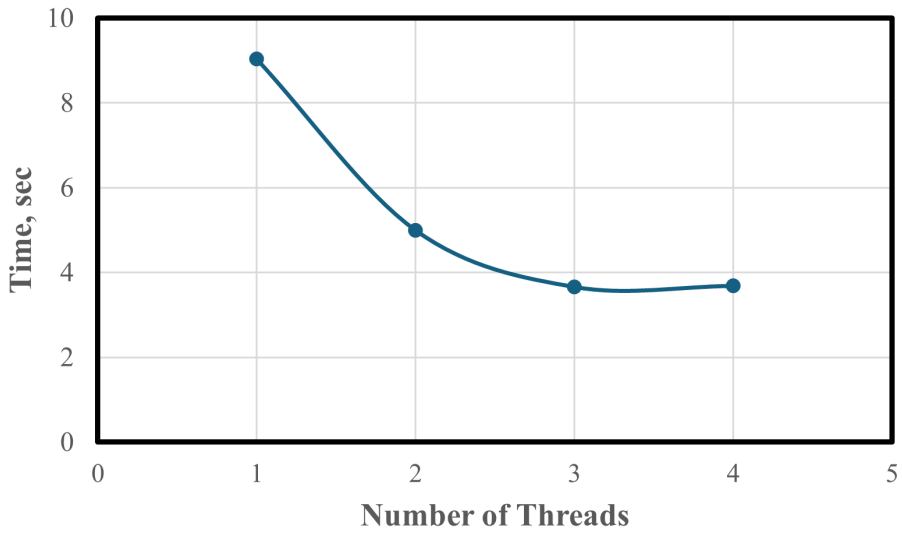

Figure 1: Code execution time vs Number of threads