CPE 345: Operating Systems Laboratory

Lab06

POSIX Threads

Submitted by: Gianna Foti

Date of Experiment:_____17 February 2025_____

Report Deadline:_____19 February 2025_____

Lab Section: _____CPE345-03_____

Demonstration Deadline: _____19 February 2025_____

Introduction

In this lab, we learned about POSIX Threads (Pthreads) and parallel programming in the Linux environment. The objective was to explore the concepts of multithreading, thread synchronization, and workload distribution using the Pthreads API in C. We were tasked with coding two assignments: the first focused on serial integral computation, where we implemented a program to compute a definite integral using the rectangular decomposition method in a single-threaded manner. The execution time of this method was recorded to later compare against the parallel approach. The second assignment involved parallel integral computation, where we used multiple threads to divide the workload. Each thread computed a portion of the integral, and the final result was obtained by accumulating the partial results. By analyzing the execution time of both implementations, we observed the performance improvements achieved through parallelism and examined the impact of factors such as thread overhead and load balancing. Additionally, this lab introduced us to thread-local storage, mutex synchronization, and timing measurement using gettimeofday(). Through this hands-on experience, we gained a deeper understanding of concurrent execution, efficiency analysis, and the practical implementation of parallel computing using Pthreads.

Theory Topics

1. Difference between Threads and Processes
    a. A process is an independent execution unit with its own memory space, requiring inter-process communication (IPC) to share data. In contrast, a thread is a lightweight execution unit within a process that shares memory with other threads, allowing for efficient communication.
    b. Threads are faster to create and manage than processes but require synchronization mechanisms like mutexes and semaphores to avoid race conditions when accessing shared resources.
2. Rectangular Decomposition Method
    a. The rectangular decomposition method is a numerical integration technique used to approximate the definite integral of a function. The method divides the integration range into small subintervals (rectangles) and sums their areas to estimate the integral.
    b. In this method, the midpoint rule is often used, where the function is evaluated at the center of each rectangle for better accuracy.
3. Thread Local Storage
    a. Thread Local Storage (TLS) allows each thread to have its own independent copy of a global variable. Unlike normal global variables, which are shared across all threads, TLS ensures that each thread has a unique instance of a variable, preventing unintended modifications from other threads.
4. Mutex and Semaphore
    a. A mutex is a locking mechanism that allows only one thread to access a resource at a time. A thread must lock (pthread_mutex_lock()) before accessing a shared variable and unlock (pthread_mutex_unlock()) after finishing. If another thread tries to access the resource while it is locked, it will be forced to wait.

b. A semaphore is a counter-based synchronization tool that controls access to a shared resource by multiple threads. Unlike a mutex, a semaphore can allow more than one thread to access a resource at the same time (depending on the counter value). It supports operations like wait (sem_wait()) and signal (sem_post()) to decrease and increase the counter, respectively.

Part 1 (Serial):

Program Description: This program computes the numerical approximation of a definite integral using the rectangular decomposition method in a serial manner. It approximates the integral of $f(x) = 4 / (1 + x^2)$ over [0,1], estimating the value of π. The number of intervals is provided as a command-line argument, and a while loop is used to sum the function values at midpoints. Execution time is measured using gettimeofday(), and the computed integral along with the time taken is displayed.

Process:

1. First, I had to understand the problem, which involved approximating the integral of $f(x) = 4 / (1 + x^2)$ over the interval [0,1] using the rectangular decomposition method.
    a. I learned that this method divides the interval into small rectangles and sums their areas using the midpoint rule.
2. Then, I implemented the function f(x), which returns $4 / (1 + x^2)$, representing the function to be integrated.
    a. I wrote a separate function, compute_integral(), that uses a while loop to iterate through the given number of intervals, compute the midpoint of each rectangle, and sum up the function values.
3. After that, I worked on the main function, ensuring it correctly parsed the command-line argument for the number of intervals.

a. I added an input validation check to ensure the number of intervals was positive. Before calling compute_integral(), I started the timer, and after the computation, I stopped it and displayed both the computed integral value and execution time.

Program Output:

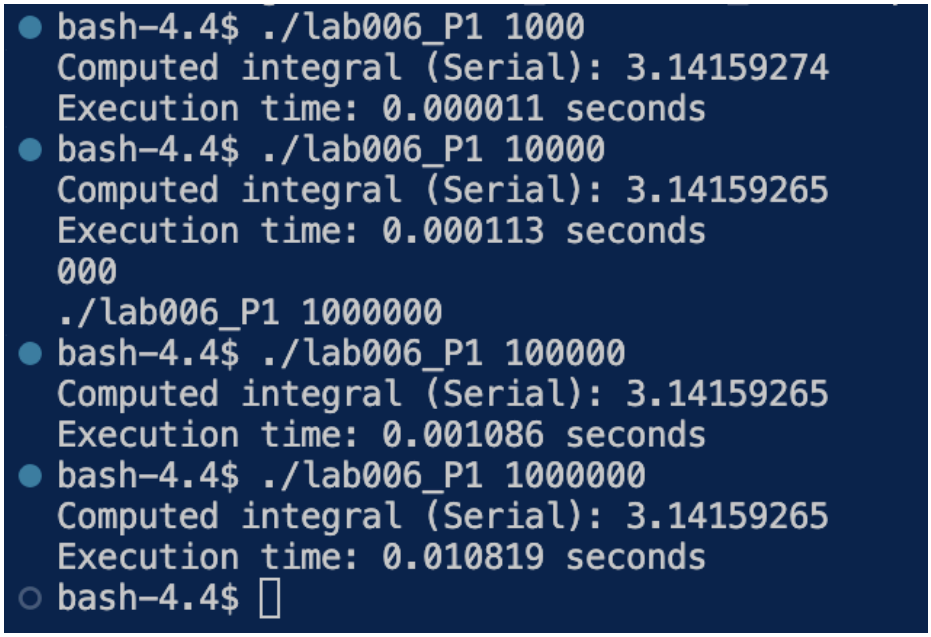

Figure 1: Program 1 at Provided Times

Part 2 (Threads):

Program Description: This program computes the numerical approximation of a definite integral using the rectangular decomposition method with POSIX threads (Pthreads) for parallelization. It estimates the integral of $f(x) = 4 / (1 + x^2)$ over the interval [0,1], which approximates π. The user specifies the number of intervals and number of threads via command-line arguments. The workload is evenly divided among the threads, each computing a portion of the integral independently. The results from all threads are accumulated to obtain the final approximation. Execution time is measured using gettimeofday() to analyze performance improvements achieved through parallelization.

Process:

1. First, I needed to understand how to parallelize numerical integration using POSIX threads (Pthreads).
    a. The goal was to approximate the integral of $f(x) = 4 / (1 + x^2)$ over [0,1] using the rectangular decomposition method while distributing the workload among multiple threads.
2. To manage parallel execution, I created a ThreadData structure to store each thread's assigned range of intervals, the width of each rectangle, and a partial sum for storing the thread's computed integral value.
3. I then implemented the compute_integral_parallel() function, which each thread executes.
    a. This function loops over its assigned interval range, computes the function values at midpoints, and accumulates the partial sum.
    b. The result is stored in the thread's ThreadData structure before the thread exits.
4. In the main function, I ensured proper handling of command-line arguments using strtol() for safe integer conversion.
    a. The input values were validated to ensure that the number of intervals is positive and that the number of threads does not exceed the number of intervals.
5. I then created an array of thread identifiers (pthread_t) and ThreadData structures to store thread-specific computations.
    a. The integral computation was divided into equal-sized chunks, and pthread_create() was used to spawn threads.
    b. After execution, pthread_join() was used to wait for all threads to complete, and their results were accumulated into the final sum.
6. Finally, after stopping the timer, the computed integral value and execution time were printed, allowing for performance analysis of parallel execution compared to the serial approach.

Program Output:

```
bash-4.4$ ./lab006_P2 1000 2
Computed integral (Parallel with 2 threads): 3.14159274
Execution time: 0.000247 seconds
bash-4.4$ ./lab006_P2 10000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000224 seconds
 100000 2
./lab006_P2 1000000 2
bash-4.4$ ./lab006_P2 100000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000473 seconds
bash-4.4$ ./lab006_P2 1000000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.003027 seconds
bash-4.4$
```

```
bash-4.4$ gcc -o lab006_P2 lab006_P2.c -lpthread -lm
bash-4.4$ ./lab006_P1 100000    # Serial execution
Computed integral (Serial): 3.14159265
Execution time: 0.001081 seconds
bash-4.4$ ./lab006_P2 100000 1  # Parallel execution with 1 thread
Computed integral (Parallel with 1 threads): 3.14159265
Execution time: 0.000861 seconds
bash-4.4$ ./lab006_P2 100000 2  # Parallel execution with 2 threads
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000522 seconds
bash-4.4$ ./lab006_P2 100000 4  # Parallel execution with 4 threads
Computed integral (Parallel with 4 threads): 3.14159265
Execution time: 0.000380 seconds
bash-4.4$ ./lab006_P2 100000 8  # Parallel execution with 8 threads
Computed integral (Parallel with 8 threads): 3.14159265
Execution time: 0.000392 seconds
bash-4.4$ ./lab006_P2 100000 16 # Parallel execution with 16 threads
Computed integral (Parallel with 16 threads): 3.14159265
Execution time: 0.000634 seconds
```

Figure 2: Program 2 with Varying Threads

Observations:

1. Serial vs Parallel (2 Threads) Performance

```
bash-4.4$ ./lab006_P1 1000
Computed integral (Serial): 3.14159274
Execution time: 0.000011 seconds
bash-4.4$ ./lab006_P1 10000
Computed integral (Serial): 3.14159265
Execution time: 0.000113 seconds
000
./lab006_P1 1000000
bash-4.4$ ./lab006_P1 100000
Computed integral (Serial): 3.14159265
Execution time: 0.001086 seconds
bash-4.4$ ./lab006_P1 1000000
Computed integral (Serial): 3.14159265
Execution time: 0.010819 seconds
bash-4.4$
```

```
bash-4.4$ ./lab006_P2 1000 2
Computed integral (Parallel with 2 threads): 3.14159274
Execution time: 0.000247 seconds
bash-4.4$ ./lab006_P2 10000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000224 seconds
 100000 2
./lab006_P2 1000000 2
bash-4.4$ ./lab006_P2 100000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000473 seconds
bash-4.4$ ./lab006_P2 1000000 2
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.003027 seconds
bash-4.4$
```

Figure 3: Serial vs the Parallel with 2 pthreads for intervals 1,000, 10,000, 100,000, and 1,000,000.

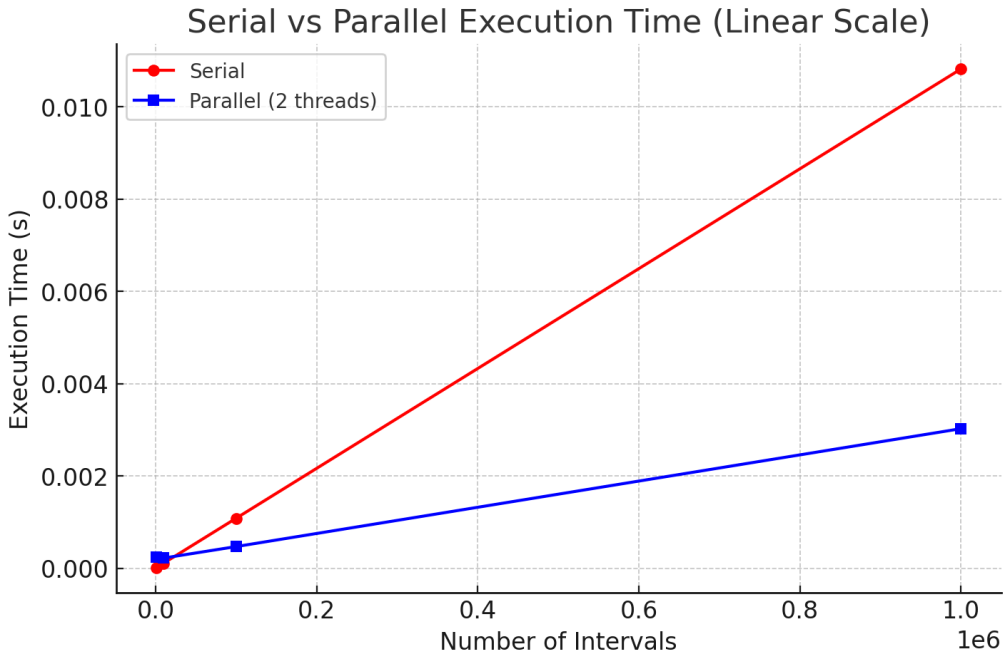

Figure 4: Graph Comparing Serial vs. Parallel

a. The comparison between the serial implementation and the parallel implementation using 2 threads was conducted for 1,000, 10,000, 100,000, and

1,000,000 intervals. The results indicate a significant difference in execution time between the two models.

i. For smaller interval sizes (1,000 and 10,000 intervals), the execution time for both serial and parallel implementations was relatively small, with little noticeable improvement from using 2 threads. This is because the overhead of creating and managing threads can outweigh the performance benefits for small computations.

ii. As the number of intervals increased (100,000 and 1,000,000 intervals), the parallel implementation began to show clear performance gains. The 2-thread parallel model executed faster than the serial model, demonstrating the advantages of distributing computational workload across multiple threads.

iii. However, the performance improvement was not exactly 2×, as expected from perfect parallelism. This is due to thread creation overhead, memory access contention, and synchronization costs. Despite this, the parallel method still reduced execution time significantly compared to the serial approach.

iv. At 1,000,000 intervals, the difference in execution time was most pronounced, with the parallel implementation being considerably faster than the serial approach. This highlights that parallel computing is most beneficial for large-scale computations, where the workload distribution justifies the threading overhead.

2. Threads

```
bash-4.4$ gcc -o lab006_P2 lab006_P2.c -lpthread -lm
bash-4.4$ ./lab006_P1 100000    # Serial execution
Computed integral (Serial): 3.14159265
Execution time: 0.001081 seconds
bash-4.4$ ./lab006_P2 100000 1  # Parallel execution with 1 thread
Computed integral (Parallel with 1 threads): 3.14159265
Execution time: 0.000861 seconds
bash-4.4$ ./lab006_P2 100000 2  # Parallel execution with 2 threads
Computed integral (Parallel with 2 threads): 3.14159265
Execution time: 0.000522 seconds
bash-4.4$ ./lab006_P2 100000 4  # Parallel execution with 4 threads
Computed integral (Parallel with 4 threads): 3.14159265
Execution time: 0.000380 seconds
bash-4.4$ ./lab006_P2 100000 8  # Parallel execution with 8 threads
Computed integral (Parallel with 8 threads): 3.14159265
Execution time: 0.000392 seconds
bash-4.4$ ./lab006_P2 100000 16 # Parallel execution with 16 threads
Computed integral (Parallel with 16 threads): 3.14159265
Execution time: 0.000634 seconds
```

Figure 5: 1, 2, 4, 8, and 16 threads for 100,000 intervals

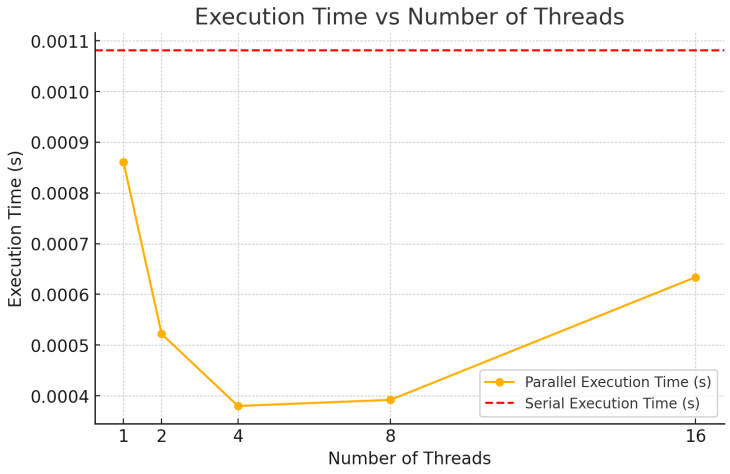

Figure 6: Comparison of Threads vs Execution Time

a. The execution time for computing the integral with 100,000 intervals was analyzed for both the serial implementation and the parallel implementation with 1, 2, 4, 8, and 16 threads. The results reveal interesting trends in performance improvements and the effects of increasing thread count.

b. Performance Improvements with More Threads
    i. The serial execution took 0.001081 seconds to compute the integral.
    ii. Running the parallel implementation with 1 thread resulted in a slight performance gain at 0.000861 seconds, indicating minimal threading overhead.
    iii. Using 2 threads nearly halved the execution time to 0.000522 seconds, demonstrating effective workload distribution.
c. Diminishing Returns as Thread Count Increases
    i. When increasing to 4 threads, the execution time further reduced to 0.000380 seconds, showing continued improvement.
    ii. However, at 8 threads, the execution time was 0.000392 seconds, slightly increasing compared to the 4-thread case.
    iii. With 16 threads, the execution time increased to 0.000634 seconds, showing a performance drop rather than an improvement.
d. Observations & Explanation
    i. Up to 4 threads, execution time consistently decreased due to effective parallelism.
    ii. At 8 threads, the improvement plateaued, likely due to increased overhead from thread management and synchronization costs.
    iii. At 16 threads, execution time worsened, likely due to context-switching overhead and resource contention, meaning the CPU had to manage too many threads for the given workload.

Conclusion

In this lab, I explored POSIX Threads (Pthreads) and parallel computing, gaining hands-on experience with multithreading, synchronization, and performance evaluation. I implemented two versions of numerical integration using the rectangular decomposition method: a serial version and a parallel version that distributed the computation across multiple threads. By comparing execution times, I observed how multithreading can improve efficiency, though the benefits depend on factors such as thread overhead and workload distribution.Additionally, I learned about thread-local storage (TLS), mutexes, and semaphores, which help manage shared resources in concurrent programs. Measuring execution time using gettimeofday() allowed me to analyze the impact of threading on performance.I did not have many struggles with this lab. The thread assignment was a little harder than the serial one, but it was still not too bad.

Appendix

```
// Author: Gianna Foti
// Teacher: Trevor H McClain
// Course: Operating Systems Laboratory - CPE 435-03
// Date: 12 February 2025
// Description: This program computes the numerical approximation of a definite integral
// using the rectangular decomposition method in a serial manner. It approximates the
// integral of the function f(x) = 4 / (1 + x^2) over the interval [0,1], which
// estimates the value of π. The number of intervals is provided as a command-line
// argument, and the execution time is measured to analyze performance.


#include <stdio.h>      // Standard I/O functions
#include <stdlib.h>     // Standard library functions (e.g., atoi)
#include <math.h>       // Math functions (e.g., sqrt, pow)
#include <sys/time.h>   // Timing functions for performance measurement

// Timer utility functions for measuring execution time
struct timeval tv1, tv2;  // Structs for storing start and end times
#define TIMER_CLEAR (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0) // Reset timer values
#define TIMER_START gettimeofday(&tv1, NULL) // Start the timer
#define TIMER_STOP gettimeofday(&tv2, NULL)  // Stop the timer
#define TIMER_ELAPSED (double)(tv2.tv_usec - tv1.tv_usec) / 1000000.0 + (tv2.tv_sec - tv1.tv_sec) // Compute elapsed
time

// Function to compute the function being integrated
// The function used here is f(x) = 4 / (1 + x^2), which approximates π when integrated from 0 to 1
double f(double x) {
   return 4.0 / (1.0 + x * x);
}


// Function to compute the integral using rectangular decomposition (serial version)
double compute_integral(int num_intervals) {
   double width = 1.0 / num_intervals; // Width of each rectangle
   double sum = 0.0; // Variable to accumulate function values

   int i = 0; // Initialize loop counter

   // Use a while loop to iterate through each interval
   while (i < num_intervals) {
      double x = (i + 0.5) * width; // Compute the midpoint of the current rectangle
      sum += f(x); // Add the function value at the midpoint to the sum
      i++; // Move to the next interval
   }

   return sum * width; // Multiply by width to get final integral approximation
}


int main(int argc, char *argv[]) {
   // Ensure correct number of command-line arguments
   if (argc != 2) {
      printf("Usage: %s <num_intervals>\n", argv[0]);
      return -1;
   }

   // Convert input argument to an integer representing the number of intervals
```

```
    int num_intervals = atoi(argv[1]);

    // Validate the number of intervals (must be positive)
    if (num_intervals <= 0) {
        printf("Error: Number of intervals must be positive.\n");
        return -1;
    }

    // Start the timer before computation
    TIMER_CLEAR;
    TIMER_START;

    // Compute the integral using the serial method
    double result = compute_integral(num_intervals);

    // Stop the timer after computation
    TIMER_STOP;

    // Print the computed integral and execution time
    printf("Computed integral (Serial): %.8f\n", result);
    printf("Execution time: %f seconds\n", TIMER_ELAPSED);

    return 0; // Indicate successful execution
}
```

Table 1: Program 1

```
// Author: Gianna Foti
// Teacher: Trevor H McClain
// Course: Operating Systems Laboratory - CPE 435-03
// Date: 12 February 2025
// Description: This program computes the numerical approximation of a definite integral
// using the rectangular decomposition method with POSIX threads (Pthreads) for parallelization.
// It approximates the integral of the function f(x) = 4 / (1 + x^2) over the interval [0,1],
// which estimates the value of π. The user specifies the number of intervals and threads via
// command-line arguments, and execution time is measured for performance analysis.

#include <stdio.h>      // Standard I/O functions
#include <stdlib.h>     // Standard library functions (e.g., strtol, malloc)
#include <math.h>       // Math functions (e.g., sqrt, pow)
#include <pthread.h>    // POSIX Threads (Pthreads) library for multi-threading
#include <sys/time.h>   // Timing functions for performance measurement

// Structure to hold data for each thread
typedef struct {
    int start;      // Start index for the thread's portion of the integral computation
    int end;        // End index for the thread's portion
    double width;   // Width of each rectangle in rectangular decomposition
    double sum;     // Partial sum computed by the thread
} ThreadData;

// Function executed by each thread to compute its portion of the integral
void *compute_integral_parallel(void *arg) {
    ThreadData *data = (ThreadData *)arg;  // Cast argument to ThreadData structure
    double localSum = 0.0;  // Local sum for the thread

    // Compute the sum of function values at midpoints of assigned intervals
    for (int i = data->start; i < data->end; i++) {
        double x = (i + 0.5) * data->width; // Midpoint of the rectangle
        localSum += 4.0 / (1.0 + x * x);  // f(x) = 4 / (1 + x^2)
```

```
    }

    data->sum = localSum * data->width;  // Multiply by width to get partial integral result
    pthread_exit(NULL);  // Exit the thread
}

// Timer utility functions for measuring execution time
struct timeval tv1, tv2;  // Structs for storing start and end times
#define TIMER_CLEAR (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0) // Reset timer values
#define TIMER_START gettimeofday(&tv1, NULL) // Start the timer
#define TIMER_STOP gettimeofday(&tv2, NULL)  // Stop the timer
#define TIMER_ELAPSED (double)(tv2.tv_usec - tv1.tv_usec) / 1000000.0 + (tv2.tv_sec - tv1.tv_sec) // Compute elapsed
time

int main(int argc, char *argv[]) {
    // Ensure correct number of command-line arguments
    if (argc != 3) {
        printf("Usage: %s <num_intervals> <num_threads>\n", argv[0]);
        return -1;
    }

    // Convert input arguments to integers safely
    char *endptr;
    long num_intervals = strtol(argv[1], &endptr, 10);
    if (*endptr != '\0' || num_intervals <= 0) {
        printf("Error: Number of intervals must be a positive integer.\n");
        return -1;
    }

    long num_threads = strtol(argv[2], &endptr, 10);
    if (*endptr != '\0' || num_threads <= 0 || num_intervals < num_threads) {
        printf("Error: Number of threads must be a positive integer and ≤ number of intervals.\n");
        return -1;
    }

    pthread_t threads[num_threads];        // Array to hold thread identifiers
    ThreadData thread_data[num_threads];   // Array to hold thread data
    double width = 1.0 / num_intervals;    // Compute width of each rectangle
    double total_sum = 0.0;                // Final sum of all partial integrals

    // Start the timer before computation
    TIMER_CLEAR;
    TIMER_START;

    // Divide work among threads
    int chunk_size = num_intervals / num_threads;  // Divide work into equal chunks

    for (int i = 0; i < num_threads; i++) {
        thread_data[i].start = i * chunk_size;  // Start index for this thread
        thread_data[i].end = (i == num_threads - 1) ? num_intervals : (i + 1) * chunk_size; // Last thread gets remaining work
        thread_data[i].width = width;  // Set the width of each interval
        thread_data[i].sum = 0.0;  // Initialize sum to 0

        // Create the thread and assign it the computation task
        if (pthread_create(&threads[i], NULL, compute_integral_parallel, (void *)&thread_data[i]) != 0) {
            printf("Error: Failed to create thread %d\n", i);
            return -1;
        }
    }

    // Wait for all threads to finish execution and accumulate their results
```

```
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
        total_sum += thread_data[i].sum;  // Accumulate partial results from all threads
    }

    // Stop the timer after computation
    TIMER_STOP;

    // Print the computed integral and execution time
    printf("Computed integral (Parallel with %ld threads): %.8f\n", num_threads, total_sum);
    printf("Execution time: %f seconds\n", TIMER_ELAPSED);

    return 0;  // Indicate successful execution
}
```

Table 2: Program 2