Report Homework 1

Report on computePi.c

computePi.c

Description of the Solution and Design Choices

This program calculates π using the trapezoidal rule on the unit circle's quarter-arc. We divide the interval [0,1] into **STEPS** trapezoids of equal width. Each trapezoid's area is computed by:

$$area = \frac{f(x)+f(x+widhtOfSteps)}{2} \times widthOfSteps,$$

where$f(x) = \sqrt{1 - x^2}$. By summing up these trapezoid areas, we effectively compute the integral of $\sqrt{1 - x^2}$ from 0 to 1, which produces $\pi/4$. Multiplying the final sum by four gives us the approximate value of π.

Each thread processes a portion of the interval independently and contributes to a shared global sum protected by a **pthread mutex** (sumLock). This design choice allows simultaneous computations while avoiding data races.

Development Process

This program calculates π by applying the trapezoid rule to the interval [0,1]. Each trapezoid's width is defined by the widthOfSteps variable, and the total number of trapezoids is specified by STEPS. The standard library functions (I/O, stdlib, stdbool), along with math.h, sys/time.h, and pthread.h, provide the necessary functionality for input/output, mathematical operations, timing, and threading.

We start by declaring the variables widthOfSteps, globalSum, and numberOfThreads. To prevent data races, we create a pthread mutex named sumLock and include a timer to measure execution time. The function $f(x)$ computes $\sqrt{1 - x^2}$, and $integrate(x)$ uses both $f(x)$ and $f(x + widthOfSteps)$ to calculate the area of a trapezoid.

Each thread's work is managed by the compute_pi_worker function, which identifies the thread using long worker_id and accumulates a partial sum. The workload for every thread is

determined by calculating a start and end index based on the thread's ID; a loop then computes the trapezoid areas within that range. Before adding partial results to the global sum, the program locks the mutex so only one thread can write at a time.

In the main function, the user's input is validated and converted to an integer. We then create an array of threads, initialize the mutex, compute widthOfSteps, and start the timer. Next, we launch all worker threads and wait for them (join) before stopping the timer. Finally, we multiply the global sum by four to approximate π, print the result, and return 0 in keeping with standard C conventions.

Summary

This compact report outlines how computePi.c calculates π by splitting the unit interval into trapezoids, leveraging multithreading for faster execution. Each thread computes a segment of the integral, and a mutex protects the shared global sum from race conditions. Finally, the result is multiplied by four to produce π, and the execution time is displayed.

Report on matrixSum.c

MatrixSumA.c

Description of the Solution and Design Choices

This program computes the sum of all elements in a matrix and identifies both the global maximum and minimum values (along with their coordinates) by dividing the matrix into horizontal strips. Each thread is responsible for one strip and calculates its partial sum, as well as its local minimum and maximum. A reusable barrier ensures that all threads complete their work before worker zero (the first thread) performs the final aggregation. In this design, worker zero gathers the partial sums and determines the global minimum and maximum, then prints the results. The simplicity of a barrier-based approach offers a clear resort point for threads, though frequent barrier synchronization may add overhead if the number of threads becomes large. For sufficiently large matrices, however, the partitioned workload yields strong speedups.

Development Process

Matrix size and number of workers are read from the command line. The matrix is initialized

with random values, and a barrier mechanism is set up using pthread_mutex_t and pthread_cond_t. Arrays store each worker's partial sums, minimum values, and maximum values. Each thread calculates its segment's sum and min/max, then waits at the barrier. Once every thread arrives, worker zero aggregates and prints the final sum, global min, global max, and timing information. This straightforward synchronization pattern ensures that all partial results are available before generating any output.

MatrixSumB.c

Description of the Solution and Design Choices

Here, the same objective of computing the global sum and identifying the global minimum and maximum is achieved without using a barrier. Instead, threads still process distinct sets of rows but update shared global variables (the total sum and min/max values) directly within critical sections protected by mutexes. Once a thread completes its portion of the matrix, it locks the appropriate mutex to update min, max, or sum, and then unlocks it immediately. This approach eliminates the need for partial results arrays and a barrier, potentially reducing waiting times. However, frequent updates to the global variables can introduce lock contention if the number of threads is high.

Development Process

Matrix size and number of workers are again read from the command line. After initializing the matrix with random values, the program creates two mutexes: one for safely updating the global sum and another for managing global min/max updates. Threads calculate local sums and local min/max within their assigned rows, then enter critical sections to integrate these results into global variables. Upon completion, each thread exits, and the main thread, after joining all workers, prints the final sum, global minimum, global maximum, and execution time. This structure removes any explicit synchronization barrier, making the flow more flexible but relying on careful use of mutexes to maintain correctness.

MatrixSumC.c

Description of the Solution and Design Choices

This version uses a bag-of-tasks model, where a shared row counter acts as a dynamic

workload queue. Each thread repeatedly locks the bag to obtain the index of the next unprocessed row, processes that row to compute a local sum, min, and max, and then locks global variables to update the total sum and global min/max values. By letting threads pull tasks on demand, this design balances the load more effectively if certain rows take longer to process. Although locking the bag for each new row can add overhead, it generally scales well for large matrices where dynamic scheduling prevents idle threads.

Development Process

Matrix size and worker count are read from the command line, and the matrix is filled with random values. The program initializes three mutexes: one for managing the row counter, one for updating the global min/max values, and one for updating the global sum. Each thread, running in a loop, acquires the bag lock to fetch a row index, processes the row if it is within bounds, and updates global variables for sum, min, and max using the corresponding locks. This continues until all rows have been processed. The main thread joins all workers and prints the total, global min, global max, and elapsed time once every row is accounted for. This approach provides a more adaptive workload distribution compared to static strip assignment.

Summary

All three matrixSum programs calculate the sum of a matrix's elements and identify the global minimum and maximum with their coordinates, but they use different synchronization strategies. MatrixSumA employs a barrier to ensure all threads finish before worker zero consolidates partial results. MatrixSumB removes the barrier in favor of direct mutex-protected updates to global variables, reducing the need for partial-result arrays. MatrixSumC adopts a bag-of-tasks model, where threads dynamically retrieve rows to process, aiming for better load balancing but at the cost of frequent lock acquisitions for the shared row counter.