Homework 1: Critical Sections. Locks, Barriers, and Condition Variables

ID1217, Concurrent Programming

January 28, 2026

Group members:
[Adar Deprem, deprem@kth.se]
[Razmus Nilsson, Razmus@kth.se]

1 Compute the Sum, Min, and Max of Matrix Elements

The original file contained code for creating a square matrix of a given size, filled with random values between 0 and 99. It also included a barrier function used to synchronize all working threads. The main functionality of the original program was to distribute rows of the matrix among multiple threads, where each thread summed the values of its assigned rows and contributed to a global total sum.

The main task was to extend this program to also identify the smallest and largest elements in the matrix, along with their respective positions. Three different approaches were given to implement this functionality. All solutions relied on parallelization using the `pthread` library in C, but they differed in how the threads coordinated their work and combined results. In this section, the three tasks are described, along with their implementations and a comparison of their execution times.

A

The first task focused on finding the minimum and maximum values in the matrix and reporting their positions. This solution was built on top of the original code, which divided the matrix into sections assigned to different threads. Each thread summed the values in its section and stored the result in a partial sum array. After all threads had finished their work, thread 0 combined the partial sums into a single total value, which was then printed.

To extend this solution, a new structure called `matrixElement` was introduced to store both the value of a matrix element and its position. Each thread used this structure to track the minimum and maximum values within its assigned section by comparing each element in the section. Both the local minimum and maximum were initialized using the first element of the thread's matrix section. Once the entire section had been processed, the thread stored its local minimum and maximum in global arrays reserved for partial results. Since each thread wrote to a unique index in these arrays, no mutex was required at this stage.

After completing their computations, all threads synchronized using the barrier function. The barrier ensured that all partial results were available before proceeding. Once synchronization was complete, only thread 0 continued execution. This thread compared all entries in the partial min and max arrays to determine the global minimum and maximum values, and also summed the partial sums to compute the total matrix sum. Finally, thread 0 printed the total sum along with the minimum and maximum values and their positions.

B

In the second task, the program from task A was modified to remove the barrier synchronization and the use of arrays for storing partial results. Additionally, thread 0 was no longer responsible for combining the results. Instead, the main thread performed the final aggregation by joining all worker threads after they had completed their execution.

To eliminate the partial result arrays, shared variables for the global minimum, maximum, and total sum were introduced in the main function. These variables were initialized using the first element of the matrix. Each thread maintained its own local sum and `matrixElement` variables for the minimum and maximum values encountered while processing its assigned portion of the matrix. As each element was examined, it was compared against the thread's local minimum and maximum and updated accordingly.

After a thread had finished processing its assigned rows, it entered a critical section, protected by a mutex, where its local minimum and maximum values were compared against the global minimum and maximum which was updated if necessary. Additionally a mutex was used to protect the critical section where each thread added its local sum of the matrix elements to the global total sum. Once all threads had completed their work, the main thread joined them and printed the final sum, minimum, and maximum values.

C

The final task introduced a dynamic work distribution approach using a bag-of-tasks model. Instead of assigning fixed sections of the matrix to each thread, each row of the matrix was treated as an individual task. Threads repeatedly requested a new row to

process until no rows remained.

A shared row counter was used to keep track of the next unprocessed row. Access to this counter was protected by a mutex to ensure that each row was assigned to only one thread. Once a thread obtained a row, it processed all columns in that row, updating its local minimum, maximum, and contributing to the global total sum. Aside from the dynamic row assignment, the remainder of the program followed the same approach as in task B.

Discussion

In Table 1 we can see that the time difference between the different methods are close to none and most likely just because of the cache or buffer. This is most likely because all the methods parallelize the most heavy task, comparing the matrix elements, in the same way.

Type	**Workers** n	**Matrix length/height** n	**Time (ms)**
A	10	100	0.40
B	10	100	0.47
C	10	100	0.49
A	10	10 000	36.31
B	10	10 000	32.4
C	10	10 000	33.1

Table 1: Performance evaluation between method A, B and C using 10 threads

2 Quicksort

To create the parallelized version of this program we first started by creating a sequential one. Which worked in a serial way of taking the entire array as one chunk, selecting the last element in the array as a pivot, and then performing the quick sort algorithm sequentially. The purpose of this assignment was to take this serial version and turn parallel by the use of `pthreads`.

To solve this we decided that the best approach was to use a task queue. Where we first split up the array into chunks of the lower and higher quadrant. These chunks were then put into a task-queue, where the use of `pthread_cond_signal` called for available workers to indivudally handle these newly added chunks. This made the program work in parallel by having multiple threads work simultaneously on chunks of the array. We also liberally used `pthread_mutex_(un)lock` to ensure that no race condition would get in the way, and only one thread at the time would handle the chunk of data assigned to them.

Performance Evaluation

Before creating the parallel version of the quick sort algorithm, we started of by creating a simple serial one. To do a controlled performance evaluation we will compare these two programs. The following table shows the time difference between these two with a varying amount of workers. From this table we can see that the vast improvement comes from the amount of workers working on larger array sizes, the interesting stand out here is that smaller array sizes will suffer if too many workers are initialized because they won't be utilized fully.

Type	**Workers** n	**Array size** n	**Time** ms
Serial	1	10 000	86.9
Parallel	5	10 000	55.7
Parallel	10	10 000	106.0
Serial	1	10 00000	561.0
Parallel	5	10 00000	124.3
Parallel	10	10 00000	119.6

Table 2: Performance Evaluation between a serial and a parallel quick sort algorithm (Average time over 5 runs)