OS Concurrency Lab Assignment: Scalable and Optimized Parallel Matrix Multiplication and Producer-Consumer Synchronization using Bounded buffer

Objective:

This lab assignment is designed to deepen your understanding of concurrent programming and parallelization using the Pthreads library. You will work on implementing an optimized and scalable parallel matrix multiplication algorithm, focusing on both parallelization and memory/cache optimization. In the second part, you will extend your implementation by introducing a producer-consumer model with a bounded buffer, where producer threads generate matrices and consumer threads perform the multiplication. The producer-consumer model is used to decouple the generation of matrices from their multiplication, allowing for better load balancing and resource utilization. This model helps in optimizing the pipeline of tasks, ensuring that producers can generate data independently while consumers process it concurrently, improving overall system throughput.

Assignment Overview:

The assignment is divided into two parts:

1. **Part 1:** Implement an optimized and scalable parallel matrix multiplication using Pthreads, focusing on maximizing performance through parallelization, load balancing, and cache optimization techniques.
2. **Part 2:** Enhance your implementation by introducing a producer-consumer scenario. Implement producer threads that generate matrices and insert them into a bounded buffer, and consumer threads that retrieve matrices from the buffer to compute their product.

Part 1: Optimized and Scalable Parallel Matrix Multiplication

Background:

You will implement the General Matrix-Matrix Multiply (GEMM) operation, which multiplies two matrices:

- **Matrix A:** Dimensions M x K
- **Matrix B:** Dimensions K x N
- **Result Matrix C:** Dimensions M x N

To leverage multicore processors, you will parallelize the computation and optimize memory and cache usage for better performance.

Tasks:

1. **Optimize Memory and Cache Access:** First experiment with optimizing memory and cache access before moving on to parallelization. The dimension K is shared between both matrices A and B. When we multiply these matrices, we use a triple-nested loop to combine their elements appropriately. Implementing GEMM involves iterating over the M and N dimensions and also co-

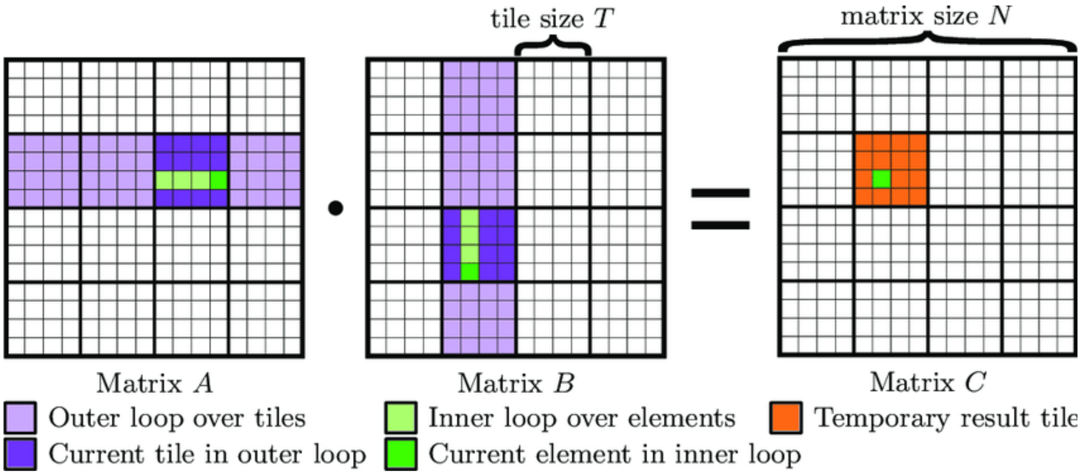
iterating over the shared dimension K. “Co-iteration” means that, as we loop through the process of multiplication, we’re iterating together over that shared dimension K for both matrices. Depending on which level (whether outer, inner or middle) of the nested loops we choose to place this co-iteration, we get different dataflows for performing the multiplication. The importance of choosing one variant over another largely comes down to “data reuse" or cache locality. You should consider the following scenarios and experiment to get the optimized performance:

- Inner Product (IP): Places the co-iteration at the innermost loop. Think of it as focusing on a single element of the resulting matrix and accumulating its value by iterating over the shared dimension. This method generates a single full sum of each resulting element at a time without merging partial sums.

- Outer Product (OP): Puts the co-iteration at the outermost loop. This involves multiplying a single row from A with a single column from B to produce an entire matrix, and then aggregating these matrices to get the final result. This method generates multiple partial sums at a time, requiring merging and potentially increased memory traffic.

- Middle loop: Features the co-iteration at the middle loop. This method generates partial sums in the current fiber (i.e., the current row of A and column of B) and then aggregates them to get the final result.

- Tiling/Blocking: Furthermore Tiling, also known as blocking, is a common optimization technique used in GEMM. The idea behind tiling is to improve data locality by dividing matrices into smaller sub-matrices or “tiles”, and then performing the multiplication operation on these smaller chunks. Tiling is one method to make the multiplication more cache-friendly by ensuring that the data being operated on fits inside the fast cache memory and can be reused before it’s evicted. The following figure illustrate the concept of tiling for the inner loops of GEMM.



You may experiment implementing a tiled matrix multiplication algorithm by breaking matrices into smaller blocks (tiles) to improve data reuse. Optimize tile size to match the cache size of the processor. You can refer this (https://coffeebeforearch.github.io/2020/06/23/mmul.html) for more details.

2. **Implement Basic Multithreaded Matrix Multiplication:** Now use Pthreads to create multiple threads and divide the computation of matrix among the threads. Ensure each thread computes a distinct portion of the result matrix to avoid race conditions. Ensure that the workload is evenly distributed among threads. If required, implement strategies to prevent some threads from becoming bottlenecks. Use synchronization primitives (e.g., mutexes) if threads need to write to shared data structures. Ensure that your program is free from race conditions and deadlocks.

Implementation Guidelines:

- **Data Structures:**
    - Store matrices using dynamically allocated 2D arrays or flattened 1D arrays for better cache performance.
    - Consider alignment and padding to optimize memory access.
- **Thread Management:**
    - Decide on the number of threads based on the number of available CPU cores.
    - Use thread arguments to pass necessary data to each thread function.
- **Performance Measurement:**
    - Measure the execution time of your matrix multiplication.
    - Compare the performance of different versions of GEMM implementation to achieve the optimized and scalable version.

Expected deliverable for Part 1: Along with well-documented code implementing the optimized parallel matrix multiplication, you will be expected to explain your parallelization strategy, describe how you optimized memory and cache access, discuss any load balancing approach. You should include performance measurements and comparisons and it would be nice to show graphs showing the speedup achieved with different numbers of threads/impact of tile size etc.

Part 2: Implementing Producer-Consumer Model with Bounded Buffer

Background: Extend your matrix multiplication program by introducing a producer-consumer model:

- **Producers:** Generate random matrices and and place them into a bounded buffer.
- **Consumers:** Retrieve matrices from the buffer and compute the product .

This part emphasizes synchronization, buffer management, and thread coordination.

Tasks:

1. **Implement synchronized Bounded Buffer:** Design a thread-safe bounded buffer to hold matrix pairs. Use mutexes and condition variables to handle synchronization.

2. **Implement Producer Threads:** Create producer threads that generate random matrices and insert them into the buffer. Ensure that producers wait when the buffer is full.

3. **Implement Consumer Threads:** Create consumer threads that remove matrix pairs from the buffer and perform multiplication. Ensure that consumers wait when the buffer is empty.

4. **Load Balancing and Optimization:** Balance the workload among producer and consumer threads. Apply the same optimization techniques from Part 1 within consumer threads.

5. **Termination Condition:** Decide on a condition for stopping the producers and consumers, such as processing a fixed number of matrices. Use a shared variable (protected by a mutex) to signal completion.

Deliverables for Part 2: Along with Well-documented code implementing the producer-consumer model with the bounded buffer, you will be expected to describe how you implemented the bounded buffer, explain synchronization between producers and consumers, discuss load balancing strategies among producers and consumers and analysis of performance by varying the number of producer/consumer threads and buffer size.

For this task, you may refer following assignment (https://faculty.washington.edu/wlloyd/courses/tcss422/assignments/TCSS422_s2024_A2.pdf) for more details and feel free to use the accompanied started code if it is helpful for you.

General Guidelines and Tips:

- **Thread Safety:**
    - Ensure all shared data structures are accessed in a thread-safe manner.
    - Minimize the scope of locks to reduce contention.
- **Error Handling:**
    - Check the return values of Pthreads functions for errors.
- **Memory Management:**
    - Properly allocate and free memory for matrices.
    - Avoid memory leaks by freeing matrices after use.
- **Testing:**
    - Start with small matrices for initial testing.
    - Gradually increase matrix sizes to test performance and scalability.
- **Documentation:**
    - Comment your code for clarity.
    - Include explanations for your design choices.

Good luck with the assignment! Remember to focus on both the correctness of your implementation and the efficiency of your solution. Effective use of parallelism and optimization techniques will significantly improve performance. Better the performance, better the marks you get.