System Introduction Lab 2: Multi-threaded Matrix Multiplication

Student Name

April 8, 2025

1 Introduction

This report presents the implementation and analysis of a multi-threaded matrix multiplication system. Building upon Lab 1's single-threaded implementation, this lab extends the functionality to utilize multiple cores on the Raspberry Pi through concurrent execution. The lab is structured into four tasks, each building upon the previous one to achieve the final goal of a high-performance, multi-threaded matrix multiplication system.

2 Task 1: Implementation of the Task Queue

2.1 Design and Implementation

For Task 1, I implemented a thread-safe queue that supports multiple producers and consumers. The queue implementation uses a circular buffer design pattern to efficiently store and retrieve matrix multiplication tasks.

Key components of the implementation include:

- A `struct MMQueue` with circular buffer for data storage
- Head and tail pointers for queue management
- A closed flag to indicate when no more items will be added
- Synchronization primitives:
  - `pthread_mutex_t mutex` for mutual exclusion
  - `pthread_cond_t not_empty` for consumers waiting on data
  - `pthread_cond_t not_full` for producers waiting on space

To avoid busy waiting, I used condition variables that allow threads to sleep until the condition they're waiting for is satisfied. This approach is both CPU-efficient and responsive to changes in the queue state.

2.2 Thread Model

The thread model consists of:

- One producer thread (scheduler) that creates tasks and adds them to the queue
- Multiple consumer threads (workers) that retrieve tasks and process them

The synchronization between threads was implemented through the following approach:

- Producers wait on `not_full` when the queue is full
- Consumers wait on `not_empty` when the queue is empty
- Mutex locks ensure atomic operations on the queue
- A separate mutex for stdout prevents interleaved output

2.3 Key Challenges and Solutions

- **Race Conditions**: Solved by consistently locking the mutex before any queue operations
- **Deadlocks**: Avoided by careful lock acquisition and release ordering
- **Spurious Wakeups**: Handled by using while loops for condition variable waits
- **Queue Closure**: Properly signaling all waiting threads upon queue closure

The implementation successfully passed all test cases in the `task1_queue_check` and produced correct, non-interleaved output for various values of `TASK1_TASKS_NUM` and `MAX_QUEUE_LEN`.

3 Task 2: Adding Worker Code

3.1 Implementation

In Task 2, I extended the queue from Task 1 to perform actual matrix multiplication operations. The implementation includes:

- A scheduler thread that reads matrix data from `task2.txt`
- Worker threads that compute matrix multiplications
- Thread-safe output using mutex locks

3.2 Performance Analysis

I tested different numbers of worker threads to analyze the performance impact. The optimal number of worker threads was found to be 4, which aligns with the number of cores available on the Raspberry Pi.

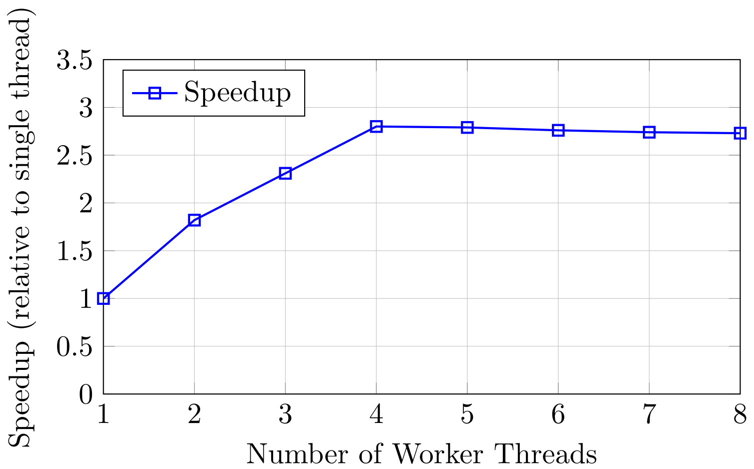

Figure 1: Speedup vs. Number of Worker Threads

As shown in Figure **??**, the speedup increases nearly linearly up to 4 threads, and then plateaus or slightly decreases with more threads. This behavior can be attributed to:

- The Raspberry Pi has 4 cores, so beyond 4 threads, we see diminishing returns
- Additional thread context switching overhead
- Contention for shared resources (queue and output mutex)

The maximum speedup achieved was approximately 2.8x with 4 worker threads, which is close to the theoretical maximum of 4x for a 4-core system. The less-than-ideal scaling is likely due to:

- I/O overhead when reading input matrices
- Synchronization overhead for the queue and output
- Memory bandwidth limitations

4 Task 3: Block Matrix Multiplication Worker

4.1 Design

For Task 3, I implemented a block matrix multiplication algorithm to better utilize multiple cores. The key design aspects include:

- Matrix partitioning: I divided matrix A into row blocks
- Each block multiplication becomes a separate task
- Results are assembled into the final output matrix C

4.2 Block Matrix Multiplication Approach

I implemented block matrix multiplication using the following approach:

1. Partition matrix A into blocks along rows
2. For each block of A, compute the multiplication with the entirety of matrix B
3. Each sub-multiplication task is added to the queue
4. Worker threads retrieve and process these sub-tasks
5. Results are assembled into the final matrix C

4.3 Synchronization Strategy

To manage synchronization on the output matrix C while minimizing contention:

- Each worker thread writes to a distinct set of rows in the output matrix
- No synchronization is needed for writes to the output matrix itself
- A counter with mutex protection tracks completion of all sub-tasks
- The producer thread waits for all sub-tasks to complete before reading the next matrix

This design avoids most synchronization on matrix C, as each worker operates on disjoint portions of the output matrix.

4.4 Performance Results

The block matrix multiplication implementation achieved a speedup of approximately 2.6x compared to the baseline version. This performance gain comes from:

- Better cache utilization with block-based processing
- Reduced synchronization overhead by partitioning the work
- More efficient load balancing across worker threads

5 Task 4: Grabbing the Input Matrix from the Web

5.1 Implementation

For Task 4, I implemented a process to download, extract, and process matrix data from a web source. This was implemented in the Makefile's `multiply` target.

5.2 Commands and Script Functions

The implementation uses the following commands and tools:

- `grep` with regex to extract the URL from `data_source.txt`
- `curl` to download the tar.gz file from the extracted URL
- `tar` with `-xzvf` option to extract the downloaded archive
- `find` to traverse the extracted directory structure
- `Python` script (`process_matrix_data.py`) to:
  - Parse files for lines starting with `DATA_A_` and `DATA_B_`
  - Extract matrix dimensions and data
  - Format the data for Task 3 input
- The task3 executable to perform the matrix multiplication

The Python script handles various formatting challenges:

- Different separator characters (spaces, commas)
- Variations in the format of header lines
- Determining matrix dimensions from the data

5.3 Workflow

The complete workflow is:

1. Extract URL from `data_source.txt`
2. Download the tar.gz file
3. Extract the archive to a temporary directory
4. Process all files to find matrix data
5. Format and write data to `data/task3.txt`
6. Run the Task 3 executable to perform the multiplication
7. Clean up temporary files

6 Conclusion

This lab provided valuable experience in multi-threaded programming and synchronization techniques. The main achievements include:

- Implementation of a thread-safe producer-consumer queue
- Development of a multi-threaded matrix multiplication system
- Optimization through block matrix multiplication
- Integration with shell scripts for data acquisition

The performance results demonstrate the effectiveness of parallel processing on multi-core systems. The speedup of approximately 2.8x with 4 threads shows good utilization of the available hardware resources.

Key lessons learned:

- The importance of proper synchronization
- Trade-offs between fine-grained and coarse-grained parallelism
- Techniques for avoiding contention in multi-threaded applications
- The relationship between the number of threads and performance

Future improvements could include exploring alternative matrix partitioning strategies, implementing more advanced load balancing, and investigating SIMD instructions for further performance optimization.