LP3 (DAA) Mini Project

Guide: Prof. Amruta Aphale

Name: Manish Godbole (31226)

Kaustubh Joshi (31233)

Aditya Kadu (31234)

Title: Matrix Multiplication

Problem Statement:

Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyse and compare their performance.

Learning Objectives:

1. Understand the principles of matrix multiplication and its computational complexity.
2. Learn to implement matrix multiplication using both standard and multi-threaded approaches.
3. Explore the use of multi-threading to optimize performance by dividing tasks across multiple threads.
4. Develop skills in applying thread management techniques for efficient matrix computation

Learning Outcomes:

By the end of this project, participants will have:

1. Ability to implement matrix multiplication using both standard and multi-threaded methods.
2. Proficiency in optimizing matrix operations through multi-threading, including thread-per-row and thread-per-cell approaches.
3. Improved understanding of parallel processing and its impact on computational efficiency.
4. Enhanced skills in managing concurrency and synchronizing threads for efficient execution in matrix operations.

Theory:

1. Matrix Multiplication

- Matrix multiplication is a fundamental operation in linear algebra, widely used in fields such as computer graphics, machine learning, and scientific computing. Given two matrices, matrix multiplication involves calculating the dot product of rows from the first matrix with columns from the second matrix to produce a result matrix. The time complexity for multiplying two matrices of dimensions m×nm \times nm×n and n×pn \times pn×p is O(m×n×p) O(m \times n \times p)O(m×n×p), which can become computationally expensive for large matrices.

2. Standard Matrix Multiplication

- The standard method for matrix multiplication involves using three nested loops. The outer loop iterates over the rows of the first matrix, the middle loop iterates over the columns of the second matrix, and the innermost loop computes the sum of the products of corresponding elements. This method, while easy to implement, can be inefficient for large datasets because it performs operations sequentially, resulting in longer computation times.

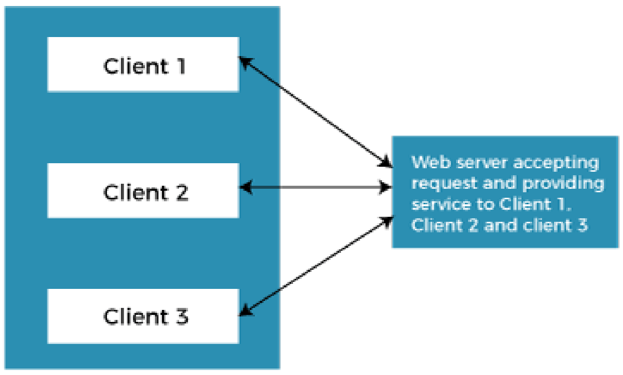
3. Multi-threading in Matrix Multiplication

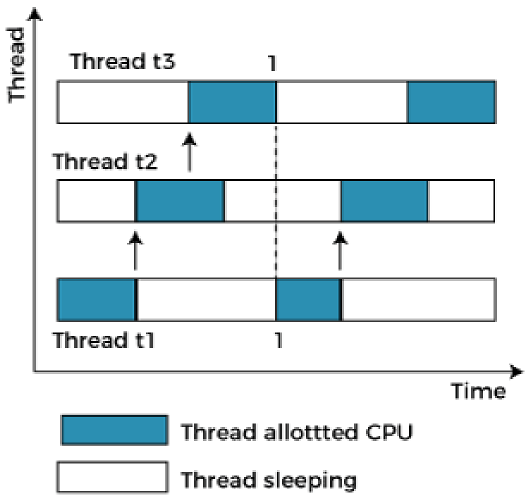
- Multi-threading is a technique used to improve performance by breaking down a task into smaller sub-tasks that can be executed simultaneously on multiple threads. In matrix multiplication, multi-threading can be used to divide the computation into smaller units, with each thread responsible for a specific part of the result matrix. There are two primary approaches:
- Thread per Row: In this approach, each thread is responsible for computing the entire row of the result matrix. This method parallelizes the operation at the row level, distributing the computation across multiple threads to reduce overall runtime.
- Thread per Cell: This more granular approach assigns each thread to compute a single element (cell) in the result matrix. Since each cell is calculated independently, this method allows for fine-grained parallelism, making it possible to fully utilize the computational resources available.

4. Concurrency and Thread Synchronization

- Managing multiple threads requires careful handling of concurrency to avoid issues such as race conditions. In matrix multiplication, each thread works on a separate portion of the result matrix, minimizing the need for synchronization. However, thread management and resource allocation must be done efficiently to prevent overhead and ensure balanced workload distribution across threads.
- By combining matrix multiplication with multi-threading, computational performance can be significantly enhanced, especially when working with large matrices or on multi-core processors. This project explores both the standard method and multi-threaded approaches to understand their trade-offs in terms of performance and complexity.

System Architecture:


Client 1
Client 2
Client 3
Web server accepting request and providing service to Client 1, Client 2 and client 3



The system architecture for the Multithreading in Matrix Multiplication. Here's an overview of the system architecture:

1. Task Division

In the first step, the main task is broken down into smaller, independent sub-tasks. These tasks should ideally not depend on each other, allowing them to run concurrently. This division is crucial for efficient parallel execution, as it minimizes the need for synchronization between threads.

2. Thread Creation

Once the tasks are divided, multiple threads are created to execute them. In Python, the threading module is used to create and manage these threads. Each thread operates independently, allowing the system to perform multiple operations at the same time, taking advantage of multi-core processors.

3. Task Assignment

Each thread is assigned a specific sub-task. This involves allocating the tasks such that each thread knows what part of the overall job it will handle. Proper distribution of tasks helps in balancing the load across threads, ensuring efficient use of system resources.

4. Thread Synchronization

In multi-threading, some threads may need to share resources or data. To prevent issues like race conditions, synchronization mechanisms like locks or semaphores are used. This ensures that only one thread accesses critical resources at a time, preventing conflicts and ensuring data integrity.

5. Thread Execution

All threads are started and run concurrently. The system scheduler manages the execution of threads, allocating CPU time to each thread. Threads may be executed on different cores of a processor, enabling true parallelism in multi-core systems, which significantly improves performance for tasks that can be parallelized.

6. Thread Joining

Once the threads complete their assigned tasks, they are joined back to the main thread. This ensures that the main program waits for all threads to finish before proceeding further. This step is essential to ensure that all sub-tasks are completed before compiling the results.

7. Result Compilation

After all threads have completed their work, the results of their computations or operations are gathered and compiled into a final output. This may involve merging the results of individual threads into a unified result that reflects the completion of the overall task.

Methodology/Algorithm Details:

1. Matrix Input and Initialization

Objective: Prepare the matrices for multiplication and verify that they meet the required dimensional constraints.

Method: Two matrices are input in the form of 2D arrays. The system checks whether the number of columns in the first matrix matches the number of rows in the second matrix. If the dimensions are compatible, an empty result matrix is initialized to store the output.

2. Standard Matrix Multiplication

Objective: Multiply the two matrices using the standard method**.**

Method: Three nested loops are used to perform the multiplication. The outer loop iterates over the rows of the first matrix, the middle loop iterates over the columns of the second matrix, and the inner loop calculates the dot product of the corresponding row and column to produce an element in the result matrix.

3. Multi-threading: Thread per Row

Objective: Optimize matrix multiplication by assigning one thread per row of the result matrix.

Method: For each row of the result matrix, a separate thread is created. Each thread computes all the elements in its corresponding row by iterating over the columns of the second matrix. Once the threads complete their work, the result matrix is updated accordingly.

4. Multi-threading: Thread per Cell

Objective: Enhance parallelism by creating one thread for each element (cell) in the result matrix.

Method: A thread is assigned to calculate each cell in the result matrix. Each thread performs the dot product between the corresponding row from the first matrix and the column from the second matrix. This method increases parallelism and makes the computation faster for larger matrices.

5. Thread Management and Synchronization

Objective: Ensure smooth execution of threads and avoid conflicts or race conditions.

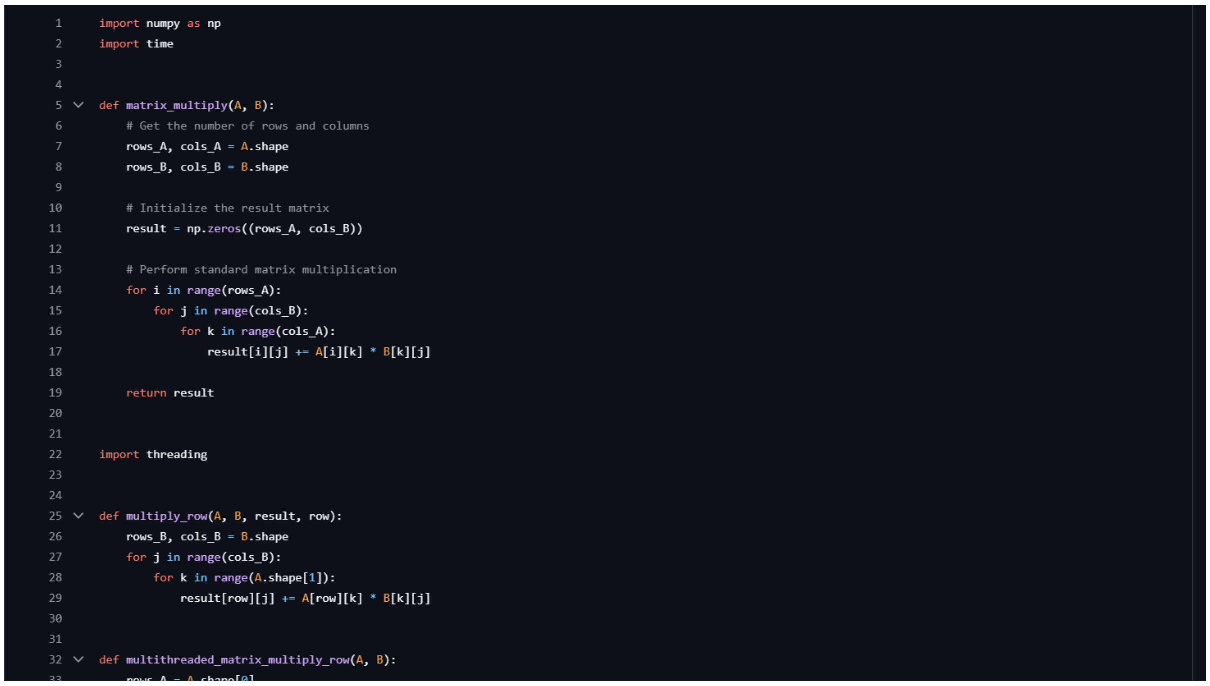
Method: Using Python's threading library, threads are created and executed. Synchronization is managed to ensure that no two threads write to the same location simultaneously. The program waits (joins) for all threads to complete before compiling the final result matrix.

6. Result Compilation and Output

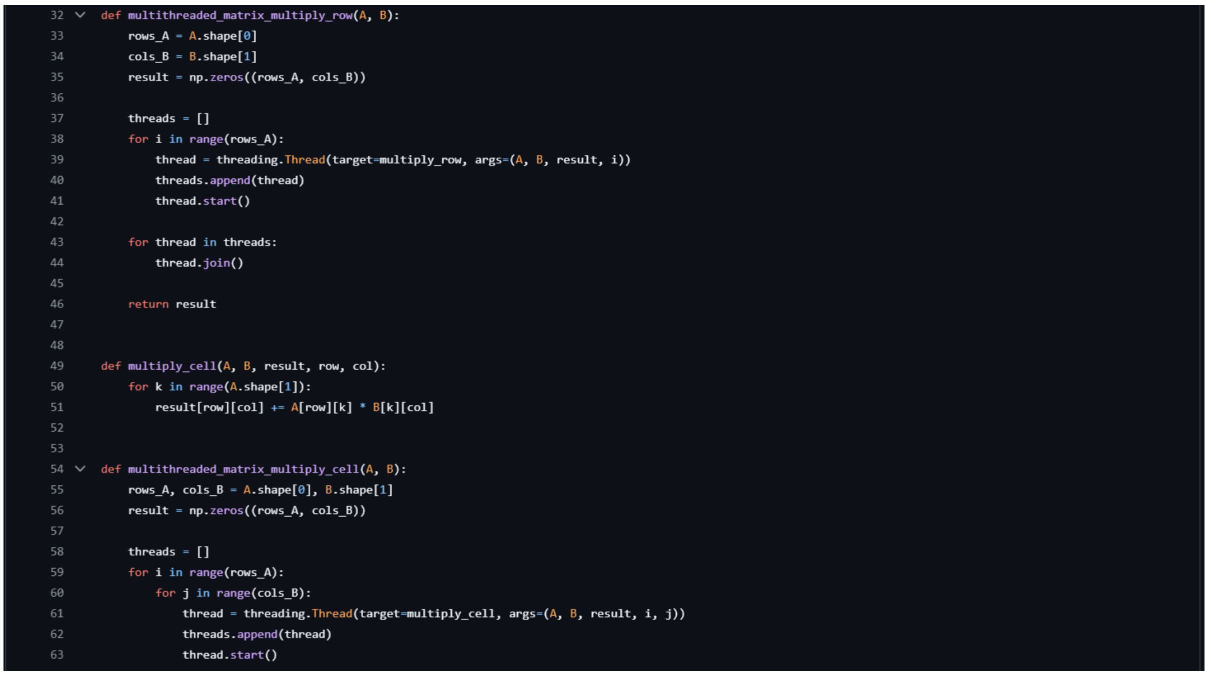
Objective: Compile and present the final matrix after all computations are completed.

Method: Once all computations—whether through the standard method or multi-threaded approaches—are finished, the result matrix is compiled and returned as the output. This matrix represents the product of the two input matrices and is then used for further analysis or output display.

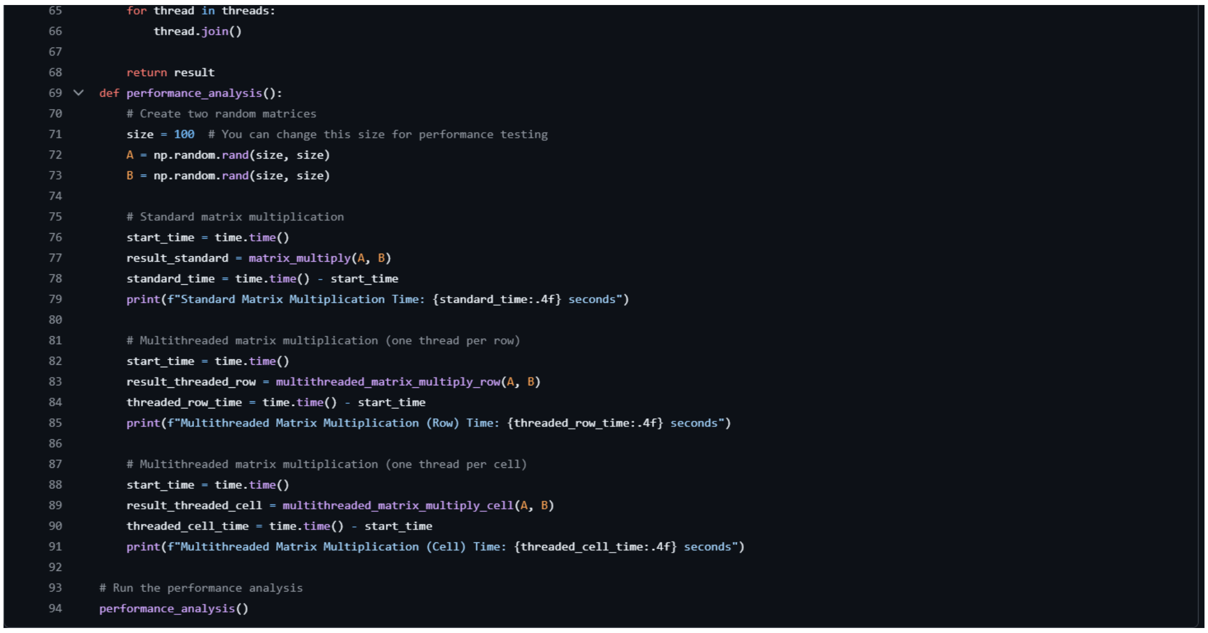
Results:



```python
import numpy as np
import time


def matrix_multiply(A, B):
    # Get the number of rows and columns
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    # Initialize the result matrix
    result = np.zeros((rows_A, cols_B))

    # Perform standard matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result


import threading


def multiply_row(A, B, result, row):
    rows_B, cols_B = B.shape
    for j in range(cols_B):
        for k in range(A.shape[1]):
            result[row][j] += A[row][k] * B[k][j]


def multithreaded_matrix_multiply_row(A, B):
    rows_A = A.shape[0]
```



```
def multithreaded_matrix_multiply_row(A, B):
    rows_A = A.shape[0]
    cols_B = B.shape[1]
    result = np.zeros((rows_A, cols_B))

    threads = []
    for i in range(rows_A):
        thread = threading.Thread(target=multiply_row, args=(A, B, result, i))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    return result


def multiply_cell(A, B, result, row, col):
    for k in range(A.shape[1]):
        result[row][col] += A[row][k] * B[k][col]


def multithreaded_matrix_multiply_cell(A, B):
    rows_A, cols_B = A.shape[0], B.shape[1]
    result = np.zeros((rows_A, cols_B))

    threads = []
    for i in range(rows_A):
        for j in range(cols_B):
            thread = threading.Thread(target=multiply_cell, args=(A, B, result, i, j))
            threads.append(thread)
            thread.start()
```



```
    for thread in threads:
        thread.join()

    return result
def performance_analysis():
    # Create two random matrices
    size = 100  # You can change this size for performance testing
    A = np.random.rand(size, size)
    B = np.random.rand(size, size)

    # Standard matrix multiplication
    start_time = time.time()
    result_standard = matrix_multiply(A, B)
    standard_time = time.time() - start_time
    print(f"Standard Matrix Multiplication Time: {standard_time:.4f} seconds")

    # Multithreaded matrix multiplication (one thread per row)
    start_time = time.time()
    result_threaded_row = multithreaded_matrix_multiply_row(A, B)
    threaded_row_time = time.time() - start_time
    print(f"Multithreaded Matrix Multiplication (Row) Time: {threaded_row_time:.4f} seconds")

    # Multithreaded matrix multiplication (one thread per cell)
    start_time = time.time()
    result_threaded_cell = multithreaded_matrix_multiply_cell(A, B)
    threaded_cell_time = time.time() - start_time
    print(f"Multithreaded Matrix Multiplication (Cell) Time: {threaded_cell_time:.4f} seconds")

# Run the performance analysis
performance_analysis()
```

Analysis Conclusion:

This project successfully demonstrates the power of multi-threading to improve the performance of computational tasks by executing them concurrently. By dividing a task into smaller, independent sub-tasks and assigning them to individual threads, we can efficiently utilize multi-core processors, significantly reducing execution time for operations that can be parallelized.

Through this project, the benefits of multi-threading became clear—tasks that would take longer in a single-threaded environment can be completed faster with proper thread management. The careful use of thread synchronization ensured data integrity and prevented issues like race conditions, highlighting the importance of concurrency control in multi-threaded applications.

While multi-threading offers significant performance improvements, it also requires careful design and consideration of the trade-offs, such as thread overhead and synchronization complexities. In the future, optimizing thread usage further, such as using thread pools or integrating with higher-level concurrency frameworks, can offer even better performance and scalability.