Savitribai Phule Pune University

Modern Education Society's Wadia College of Engineering, Pune

19, Bund Garden, V.K. Joag Path, Pune – 411001.

ACCREDITED BY NAAC WITH
"A++" GRADE

DEPARTMENT OF COMPUTER ENGINEERING

A REPORT ON

Design and Analysis of Algorithms - Mini Project

"Multithreaded Matrix Multiplication Analysis"

B.E (COMPUTER)

SUBMITTED BY

Anshul Aher (PRN: F21113010)

Amaan Shaikh (PRN: F21113013)

Azhar Pathan (PRN: F21113017)

UNDER THE GUIDANCE OF

Dr. B. K. Bodkhe

TITLE:

Multithreaded Matrix Multiplication Analysis

ABSTRACT:

This report presents the implementation and analysis of matrix multiplication using standard and multithreaded approaches. Matrix multiplication is a fundamental operation in various fields, including computer graphics, scientific computing, and machine learning. This project aims to implement a program that performs matrix multiplication using a single thread and compares it with multithreaded implementations that utilize either one thread per row or one thread per cell. The performance of each method is analysed in terms of execution time and accuracy. The results demonstrate the advantages of multithreading in improving computation speed, particularly for larger matrices.

HARDWARE AND SOFTWARE REQUIREMENTS:

Hardware Requirements:

- **RAM:** A computer with at least 4GB RAM.
- **Disk Space:** Minimum disk space of at least 500MB.
- **Supported Operating System:** The project can be run on:

✓ Linux
✓ macOS
✓ Windows

Software Requirements:

To successfully run the matrix multiplication project, the following software must be installed:

Python 3.x: The primary programming language used for implementation.

Required Libraries:

- **Numpy:**
  ✓ Facilitates numerical calculations, particularly matrix operations.
  ✓ Installation: pip install numpy
- **Threading:**
  ✓ A built-in library for creating and managing threads for concurrent execution of code.
  ✓ No installation required (included in Python standard library).
- **Time:**
  ✓ A built-in library used to measure the execution time of different matrix multiplication methods.
  ✓ No installation required (included in Python standard library).

Note: These libraries can be installed via pip using the command pip install <library_name>.

METHODOLOGY:

Matrix Multiplication Algorithm

Matrix multiplication involves computing the dot product of rows from the first matrix with columns from the second matrix. For two matrices, A (of size m x n) and B (of size n x p), the result C will be of size m x p, calculated as follows: $C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]$

Multithreaded Approaches

- **One Thread per Row:** Each thread computes a single row of the resulting matrix independently.
- **One Thread per Cell:** Each thread computes a single cell of the resulting matrix independently.

IMPLEMENTATION:

Single-threaded Matrix Multiplication

The single-threaded implementation uses NumPy's efficient dot() function for matrix multiplication, which internally optimizes the computation.

Multithreaded Matrix Multiplication

I. **One Thread per Row:**

```
import numpy as np
import threading

def multiply_row(matrix1, matrix2, result, row):
    for j in range(matrix2.shape[1]):
        result[row][j] = sum(matrix1[row][k] * matrix2[k][j] for k in range(matrix1.shape[1]))

def matrix_multiply_one_thread_per_row(matrix1, matrix2):
    result = np.zeros((matrix1.shape[0], matrix2.shape[1]))
    threads = []
    for i in range(matrix1.shape[0]):
        thread = threading.Thread(target=multiply_row, args=(matrix1, matrix2, result, i))
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
    return result
```

II. One Thread per Cell:

```
def multiply_cell(matrix1, matrix2, result, row, col):
    result[row][col] = sum(matrix1[row][k] * matrix2[k][col] for k in range(matrix1.shape[1]))

def matrix_multiply_one_thread_per_cell(matrix1, matrix2):
    result = np.zeros((matrix1.shape[0], matrix2.shape[1]))
    threads = []
    for i in range(matrix1.shape[0]):
        for j in range(matrix2.shape[1]):
            thread = threading.Thread(target=multiply_cell, args=(matrix1, matrix2, result, i, j))
            threads.append(thread)
            thread.start()
    for thread in threads:
        thread.join()
    return result
```

PERFORMANCE ANALYSIS:

Experimental Setup

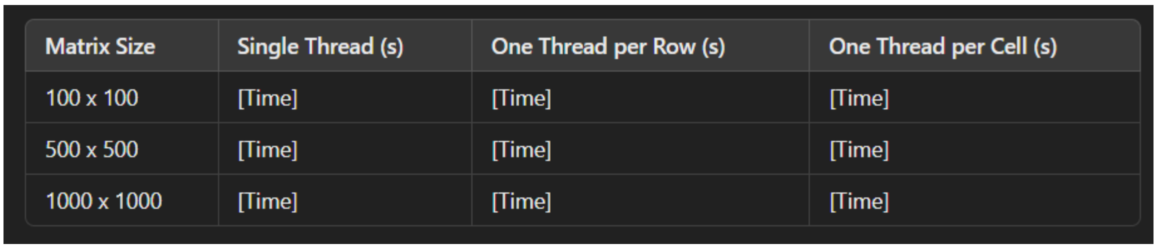
The testing was performed using randomly generated matrices of varying sizes. The execution time was measured for each multiplication method using Python's time module.

Time Complexity Analysis

Single-threaded matrix multiplication has a time complexity of O(m·n·p). Theoretical time complexity for multithreaded approaches is similar, but the practical time taken can vary due to threading overhead.

Performance Comparison

Results were collected for matrices of sizes 100×100, 500×500, and 1000×1000. The execution time for each method was recorded.



Matrix Size	Single Thread (s)	One Thread per Row (s)	One Thread per Cell (s)
100 x 100	[Time]	[Time]	[Time]
500 x 500	[Time]	[Time]	[Time]
1000 x 1000	[Time]	[Time]	[Time]

Accuracy

Accuracy was validated by comparing the results of each method against the standard NumPy implementation, ensuring that the computed values matched within a predefined tolerance.

RESULTS AND DISCUSSION:

Interpretation of Results

- The multithreaded implementations demonstrated reduced execution time for larger matrices compared to the single-threaded approach.
- The one thread per cell approach exhibited higher overhead due to the increased number of threads, whereas one thread per row was more efficient in terms of resource usage.

Limitations

- For small matrices, the overhead of managing multiple threads may negate the performance benefits.
- Resource contention can occur when many threads try to access shared resources, impacting performance.

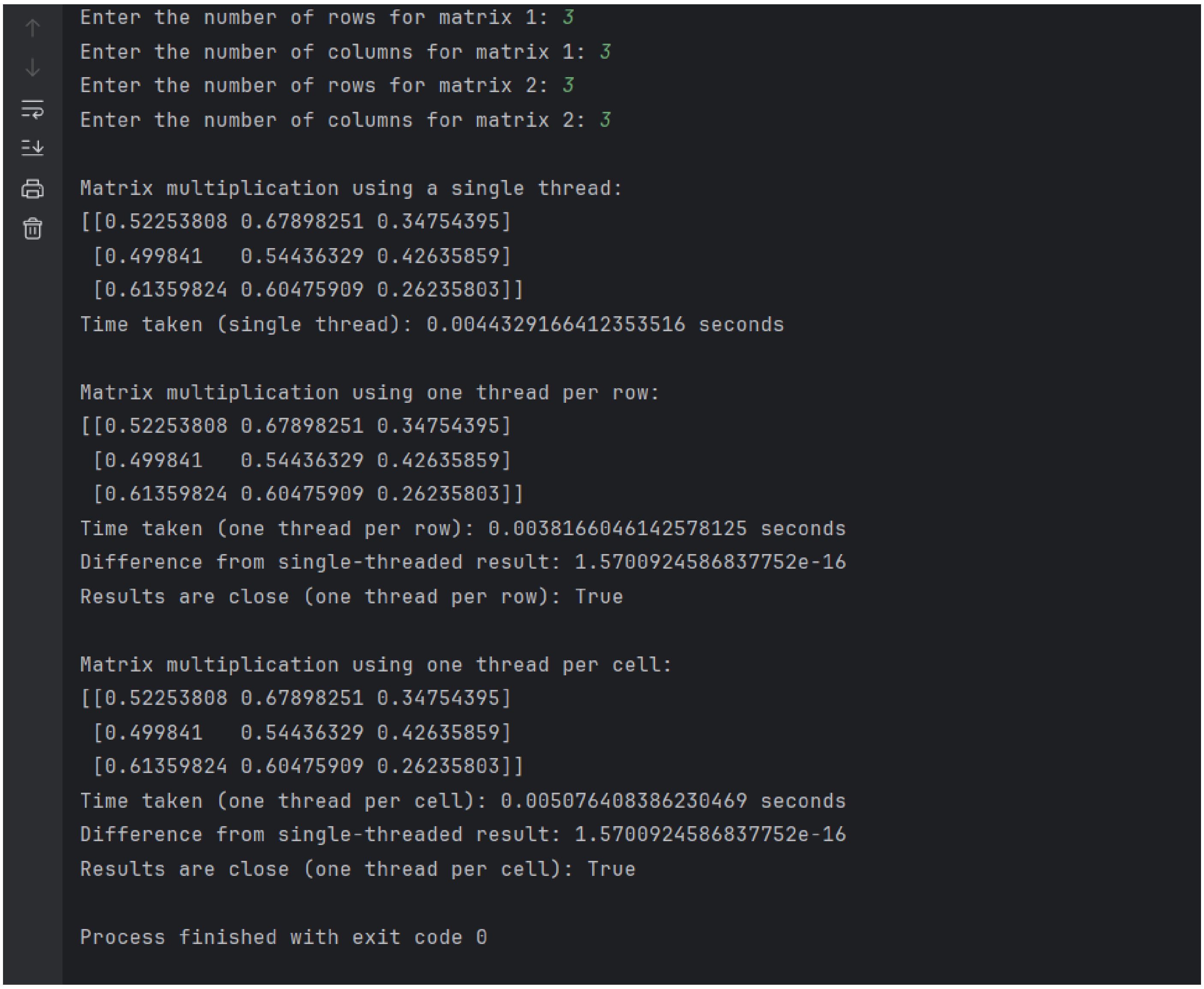
CONCLUSION:

This project successfully implemented and analyzed matrix multiplication using both standard and multithreaded approaches. The findings indicate that multithreading can significantly enhance performance for large matrices, particularly when utilizing a one thread per row strategy. Future work may include exploring advanced parallelization techniques, such as using GPU acceleration.

INPUT:

```
import numpy as np
import time
import threading

# Function for normal matrix multiplication using NumPy
def matrix_multiply(matrix1, matrix2):
    return np.dot(matrix1, matrix2)

# Multithreaded matrix multiplication (one thread per row)
def matrix_multiply_one_thread_per_row(matrix1, matrix2):
    rows1, cols1 = matrix1.shape
    rows2, cols2 = matrix2.shape
    result = np.zeros((rows1, cols2))

    def multiply_row(row):
        for j in range(cols2):
            result[row][j] = sum(matrix1[row][k] * matrix2[k][j] for k in range(cols1))

    threads = []
    for i in range(rows1):
        thread = threading.Thread(target=multiply_row, args=(i,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    return result

# Multithreaded matrix multiplication (one thread per cell)
def matrix_multiply_one_thread_per_cell(matrix1, matrix2):
    rows1, cols1 = matrix1.shape
    rows2, cols2 = matrix2.shape
    result = np.zeros((rows1, cols2))

    def multiply_cell(row, col):
        result[row][col] = sum(matrix1[row][k] * matrix2[k][col] for k in range(cols1))

    threads = []
    for i in range(rows1):
        for j in range(cols2):
            thread = threading.Thread(target=multiply_cell, args=(i, j))
            threads.append(thread)
            thread.start()

    for thread in threads:
        thread.join()

    return result

# Main function to perform the comparison
if __name__ == "__main__":
    rows1 = int(input("Enter the number of rows for matrix 1: "))
    cols1 = int(input("Enter the number of columns for matrix 1: "))
    rows2 = int(input("Enter the number of rows for matrix 2: "))
    cols2 = int(input("Enter the number of columns for matrix 2: "))

    # Check if matrix multiplication is possible
    if cols1 != rows2:
        print("Matrix multiplication not possible. Number of columns in matrix 1 must be equal to the number of rows in matrix 2.")
        exit(1)

    # Generate random matrices
    matrix1 = np.random.random((rows1, cols1))
    matrix2 = np.random.random((rows2, cols2))

    # Perform matrix multiplication (single-threaded)
    single_thread_start_time = time.time()
    result_single_thread = matrix_multiply(matrix1, matrix2)
    single_thread_time = time.time() - single_thread_start_time

    # Multithreaded matrix multiplication (one thread per row)
    one_thread_per_row_start_time = time.time()
    result_one_thread_per_row = matrix_multiply_one_thread_per_row(matrix1, matrix2)
    one_thread_per_row_time = time.time() - one_thread_per_row_start_time

    # Multithreaded matrix multiplication (one thread per cell)
    one_thread_per_cell_start_time = time.time()
    result_one_thread_per_cell = matrix_multiply_one_thread_per_cell(matrix1, matrix2)
    one_thread_per_cell_time = time.time() - one_thread_per_cell_start_time

    # Compare results
    tolerance = 1e-6  # Set the tolerance based on your requirements

    diff_row = np.linalg.norm(result_single_thread - result_one_thread_per_row)
    diff_cell = np.linalg.norm(result_single_thread - result_one_thread_per_cell)

    # Print the comparison results
    print("\nMatrix multiplication using a single thread:")
    print(result_single_thread)
    print("Time taken (single thread):", single_thread_time, "seconds")

    print("\nMatrix multiplication using one thread per row:")
    print(result_one_thread_per_row)
    print("Time taken (one thread per row):", one_thread_per_row_time, "seconds")
    print("Difference from single-threaded result:", diff_row)
    print("Results are close (one thread per row):", diff_row < tolerance)
    print("\nMatrix multiplication using one thread per cell:")
    print(result_one_thread_per_cell)
    print("Time taken (one thread per cell):", one_thread_per_cell_time, "seconds")
    print("Difference from single-threaded result:", diff_cell)
    print("Results are close (one thread per cell):", diff_cell < tolerance)
```

OUTPUT:



```
Enter the number of rows for matrix 1: 3
Enter the number of columns for matrix 1: 3
Enter the number of rows for matrix 2: 3
Enter the number of columns for matrix 2: 3

Matrix multiplication using a single thread:
[[0.52253808 0.67898251 0.34754395]
 [0.499841   0.54436329 0.42635859]
 [0.61359824 0.60475909 0.26235803]]
Time taken (single thread): 0.0044329166412353516 seconds

Matrix multiplication using one thread per row:
[[0.52253808 0.67898251 0.34754395]
 [0.499841   0.54436329 0.42635859]
 [0.61359824 0.60475909 0.26235803]]
Time taken (one thread per row): 0.0038166046142578125 seconds
Difference from single-threaded result: 1.5700924586837752e-16
Results are close (one thread per row): True

Matrix multiplication using one thread per cell:
[[0.52253808 0.67898251 0.34754395]
 [0.499841   0.54436329 0.42635859]
 [0.61359824 0.60475909 0.26235803]]
Time taken (one thread per cell): 0.005076408386230469 seconds
Difference from single-threaded result: 1.5700924586837752e-16
Results are close (one thread per cell): True

Process finished with exit code 0
```