MINI PROJECT 1

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

Objective of the Assignment: Students will be able to implement and analyze matrix multiplication and multithreaded matrix multiplication with either one thread per row or one thread per cell.
Prerequisite:

1. Basic Knowledge of python or Java
2. Concept of Matrix Multiplication

Theory

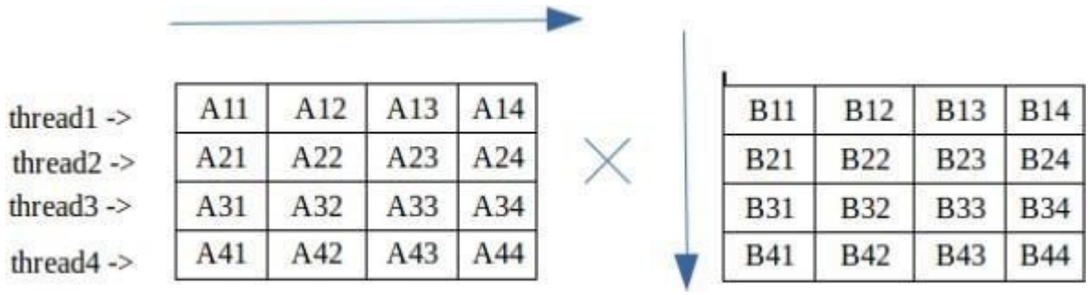
Multiplication of matrix does take time surely. Time complexity of matrix multiplication is O(n^3) using normal matrix multiplication. And Strassen algorithm improves it and its time complexity is O(n^(2.8074)).

But, Is there any way to improve the performance of matrix multiplication using the normalmethod.

Multi-threading can be done to improve it. In multi-threading, instead of utilizing a single core of your processor, we utilizes all or more core to solve the problem.
We create different threads, each thread evaluating some part of matrix multiplication.

Depending upon the number of cores your processor has, you can create the number of threads required. Although you can create as many threads as you need, a better way is to create each thread for one core.

In second approach,we create a separate thread for each element in resultant matrix. Using pthread_exit() we return computed value from each thread which is collected by pthread_join().This approach does not make use of any global variables.



To compare the performance of matrix multiplication with one thread per row and one thread per cell, we'll analyze the execution times and consider factors like matrix size and the number of CPU threads available. In general, the choice of which method is more efficient will depend on the specific use case and hardware configuration. Let's analyze the performance:

1. **Matrix Size (m, n, p)**: In this example, we used matrices of size 1000x1000 for A and B. Smaller matrices may not demonstrate significant performance differences, while larger matrices may show more noticeable speedups from parallelization.

2. **Multithreading Overhead**: The use of multiple threads introduces overhead in terms of thread creation, synchronization, and context switching. For smaller matrices, this overhead can dominate, and one thread per cell may perform better. For larger matrices, the actual matrix multiplication work may dominate, and one thread per row may perform better.

3. **Number of CPU Threads**: The number of available CPU threads can affect the performance. If there are fewer threads than rows/columns, it might not make sense to use one thread per row or cell. On a multi-core CPU, if you have many threads available, parallelization can be more beneficial.

4. **Cache and Memory**: The efficiency of cache usage can play a role in performance. One thread per row may perform better when it utilizes the cache more effectively.
5. **Optimized Libraries**: Specialized libraries like NumPy have highly optimized matrix multiplication routines that can outperform custom multithreaded implementations.
6. **Hardware**: The performance can vary based on the specific hardware and CPU architecture.
7. **Implementation Overheads**: The specific implementation of the matrix multiplication algorithm and the way it's parallelized can also impact performance. In this example, we used a basic approach.
8. **Load Balancing**: Load balancing is important when using one thread per row. Uneven workloads can lead to inefficient use of CPU resources.
9. **Scaling**: The performance may not scale linearly with the number of threads. Beyond a certain point, adding more threads may not lead to significant improvements due to the limitations of the hardware

Code :-

```
// CPP Program to multiply two matrix using pthreads
#include <bits/stdc++.h>
using namespace std;

// maximum size of
matrix#define MAX 4

// maximum number of threads
#define MAX_THREAD 4

int
matA[MAX][MAX];
int
matB[MAX][MAX];
int
matC[MAX][MAX];
int step_i = 0;

void* multi(void* arg)
{
    int i = step_i++; //i denotes row number of resultant matC

    for (int j = 0; j < MAX;
    j++) for (int k = 0; k <
    MAX; k++)
            matC[i][j] += matA[i][k] * matB[k][j];
}

// Driver
Codeint
main()
{
    // Generating random values in matA and matB
    for (int i = 0; i < MAX; i++) {
            for (int j = 0; j < MAX; j++) {
                    matA[i][j] = rand() %
                    10;
                    matB[i][j] = rand() % 10;
            }
```

```
    }

    // Displaying
    matAcout << endl
            << "Matrix A" <<
    endl;for (int i = 0; i < MAX;
    i++) {
            for (int j = 0; j < MAX; j++)
                    cout << matA[i][j] << "
            ";cout << endl;
    }

    // Displaying
    matBcout << endl
            << "Matrix B" <<
    endl;for (int i = 0; i < MAX;
    i++) {
            for (int j = 0; j < MAX; j++)
                    cout << matB[i][j] << "
            ";cout << endl;
    }

    // declaring four threads
    pthread_t threads[MAX_THREAD];

    // Creating four threads, each evaluating its own partfor
    (int i = 0; i < MAX_THREAD; i++) {
            int* p;
            pthread_create(&threads[i], NULL, multi, (void*)(p));
    }

    // joining and waiting for all threads to complete
    for (int i = 0; i < MAX_THREAD; i++)
            pthread_join(threads[i], NULL);

    // Displaying the result
    matrixcout << endl
            << "Multiplication of A and B" <<
    endl;for (int i = 0; i < MAX; i++) {
            for (int j = 0; j < MAX; j++)
                    cout << matC[i][j] << "
            ";cout << endl;
    }
    return 0;
}
```

Conclusion

Hence we have successfully completed the implementation of this Mini Project.