Design and Analysis of Algorithms

Assignment - 7

Multithread Matrix Multiplication

Aim

To implement multithreaded matrix multiplication.

Objective

The objective of implementing multithreaded matrix multiplication is to improve the performance of the matrix multiplication algorithm by utilising multiple threads to perform the computations in parallel.

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 normal method.

Multi-threading can be done to improve it. In multi-threading, instead of utilising a single core of your processor, we utilises 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. By implementing and studying multithreaded matrix multiplication, we can gain a deeper understanding of the concepts of concurrency and parallelism, and learn how to design and implement efficient, scalable, and robust concurrent programs.

Additionally, we can explore the trade-offs between performance, memory usage, and code complexity, and identify optimization techniques that can further improve the performance of the algorithm.

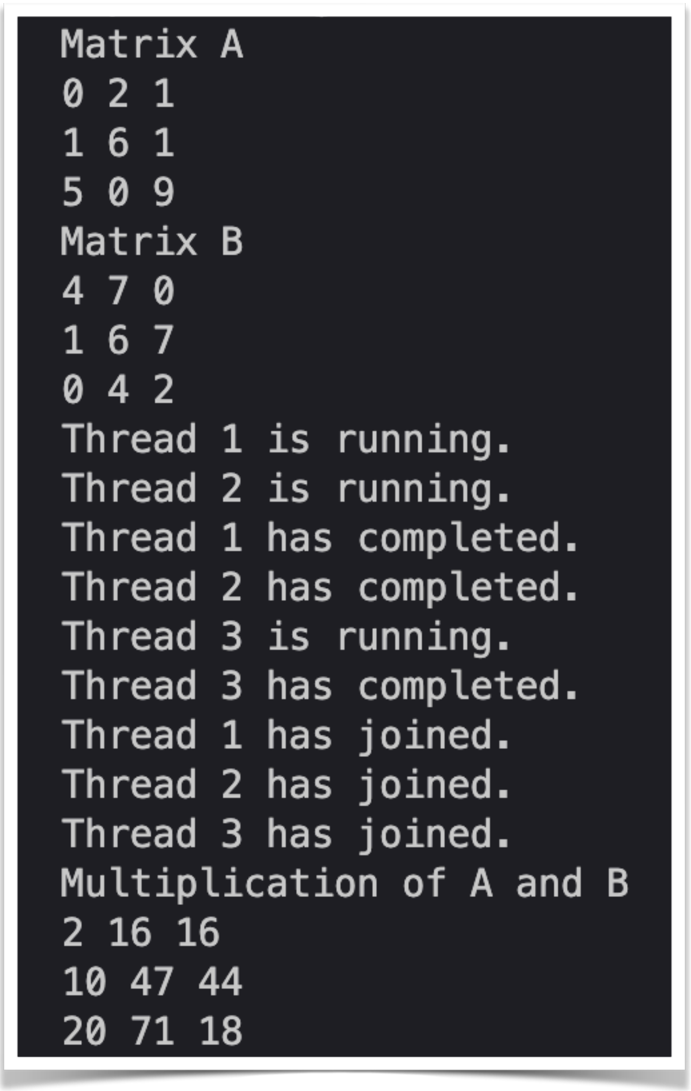
Space and Time Complexity

In general, the time complexity of the algorithm is O(N^3/P), where N is the size of the matrix and P is the number of threads. This is because the workload is divided among P threads, each of which computes a portion of the final result. The space complexity of the algorithm is O(N^2), which is the same as for sequential matrix multiplication. When compared to sequential matrix multiplication, the primary advantage of multithreaded matrix multiplication is its ability to exploit parallelism, which can significantly reduce the overall execution time for large matrices. However, the performance gains are highly dependent on the number of threads used, the size of the matrices, and the hardware specifications of the system. In some cases, the overhead associated with thread creation, synchronisation, and communication can actually result in slower performance for small matrices or on systems with low numbers of cores or slower memory access. Overall, the performance of multithreaded matrix multiplication can be greatly improved by using efficient algorithms, optimising thread management and synchronisation, and exploiting hardware features such as cache locality and SIMD instructions. The space complexity of the algorithm is the same as for sequential matrix multiplication, which makes it a feasible option for large matrices on systems with sufficient memory.

Code

```
import java.util.Random;

public class MatrixMultiplication {
    static final int MAX = 3;
    static final int MAX_THREAD = 3;
    static int[][] matA = new int[MAX][MAX];
    static int[][] matB = new int[MAX][MAX];
    static int[][] matC = new int[MAX][MAX];
    static int step_i = 0;

    static class Worker implements Runnable {
        int i;

        Worker(int i) {
            this.i = i;
        }

        @Override
        public void run() {
            System.out.println("Thread " + (i+1) + " is 
running.");
            for (int j = 0; j < MAX; j++) {
                for (int k = 0; k < MAX; k++) {
                    matC[i][j] += matA[i][k] * matB[k][j];
                    // System.out.print(matC[i][j] + "");
                }
            }
            System.out.println("Thread " + (i+1) + " has 
completed.");
        }
    }

    public static void main(String[] args) {
        Random rand = new Random();

        // 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.nextInt(10);
                matB[i][j] = rand.nextInt(10);
            }
        }

        // Displaying matA
        System.out.println("Matrix A");
        for (int i = 0; i < MAX; i++) {
            for (int j = 0; j < MAX; j++) {
                System.out.print(matA[i][j] + " ");
            }
            System.out.println();
        }

        // Displaying matB
        System.out.println("Matrix B");
        for (int i = 0; i < MAX; i++) {
            for (int j = 0; j < MAX; j++) {
                System.out.print(matB[i][j] + " ");
            }
            System.out.println();
        }

        // declaring four threads
        Thread[] threads = new Thread[MAX_THREAD];

        // Creating four threads, each evaluating its own
part
        for (int i = 0; i < MAX_THREAD; i++) {
            threads[i] = new Thread(new Worker(step_i++));
            threads[i].start();
        }

        // joining and waiting for all threads to complete
        for (int i = 0; i < MAX_THREAD; i++) {
            try {
                threads[i].join();
                System.out.println("Thread " + (i+1) + " has
joined.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Displaying the result matrix
        System.out.println("Multiplication of A and B");
        for (int i = 0; i < MAX; i++) {
            for (int j = 0; j < MAX; j++) {
                System.out.print(matC[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output



```
Matrix A
0 2 1
1 6 1
5 0 9
Matrix B
4 7 0
1 6 7
0 4 2
Thread 1 is running.
Thread 2 is running.
Thread 1 has completed.
Thread 2 has completed.
Thread 3 is running.
Thread 3 has completed.
Thread 1 has joined.
Thread 2 has joined.
Thread 3 has joined.
Multiplication of A and B
2 16 16
10 47 44
20 71 18
```

Conclusion

In conclusion, multithreaded matrix multiplication is a powerful technique for improving the performance of matrix multiplication by exploiting parallelism. The time complexity of the algorithm is O(N^3/P), where N is the size of the matrix and P is the number of threads used. This makes it a very efficient approach for large matrices on systems with multiple cores or processors. When compared to sequential matrix multiplication, multithreaded matrix multiplication provides a significant advantage for large matrices, but may result in slower performance for small matrices or on systems with low numbers of cores or slower memory access.

Overall, multithreaded matrix multiplication is a powerful tool for solving large-scale matrix problems efficiently, and is an important technique for scientific and engineering applications.