# Matrix Multiplication

**Name:** Róisín Ní Bhriain
**Student Number:** 23269640
**Email:** roisin.nibhriain3@mail.dcu.ie
**Program:** MCM Secure Software Engineering
**Module Code:** CA670
**Submission:** Java Threads vs. OpenMP - Matrix Multiplication

**Róisín Ní Bhriain**

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references. Any use of generative AI or search will be described in a one page appendix including prompt queries.

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy.

**Signed Name:** Róisín Ní Bhriain
**Date**: 07-04-2024

# Java Threads Design

For the Java threads design I decided to use the Strassen's Multiplication algorithm which is essentially this basic concept in figure 1:

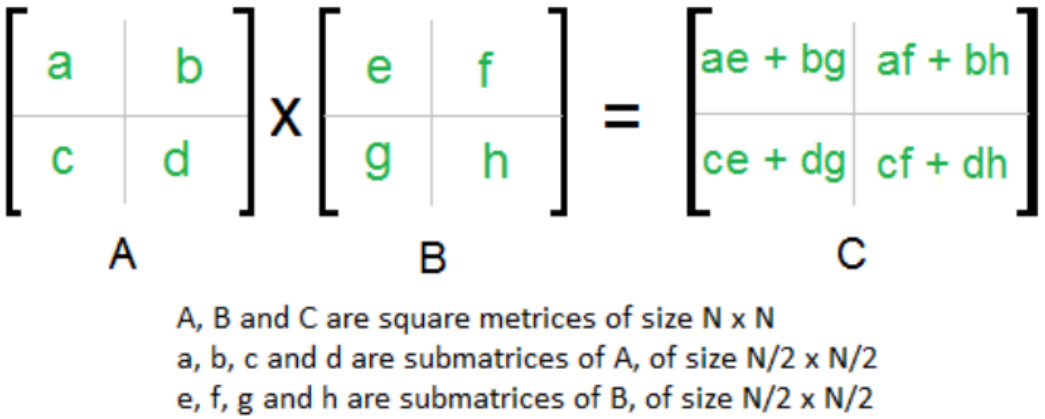


Figure 1: Strassen's Matrix Multiplication Algorithm (*Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)*, 2014).

The general idea I started with was to do one thread per row of the first matrix which is obviously not feasible for some of the larger matrices that I tested with so I had to rework the design and spread rows amongst threads. For each thread the design was this:

```
class Multiplier implements Runnable {
        int i;
        MatrixMultiplication mats;

        Multiplier ( int i, MatrixMultiplication mats ) {
                this.i = i;
                this.mats = mats;
        }


        public void run () {
                for ( int l = i; l < mats.N; l += mats.MAX_THREADS ) {
                        for ( int j = 0; j < mats.N; j++ ) {
                                for ( int k = 0; k < mats.N; k++ ) {
                                        mats.matC[l][j] += mats.matA[l][k] * mats.matB[k][j];
                                }
                        }

                        mats.matC[ l ][ mats.N ] = 1;
                }
                System.out.println( "thread " + i + " finished");
        }
}
```

Where the thread ID is used to start the index for the rows of the A matrix and the output matrix. This is incremented by the maximum number of threads to ensure that each of the

rows is only accessed by one thread. I also had to add another extra column on the resultant matrix as I ran into an issue where the threads were not all completed before the executor was finished and thus there were a lot of uncomputed cells in the matrix where there shouldn't have been. This extra column has a 1 assigned to it for each row when it is finished executing.

```
        static void waitFor ( int mat [][] ) {

                for (int i = 0; i < N; i++ ) {
                        while (mat[i][N] != 1)  {
                                try {
                                        Thread.sleep(10);
                                        // System.out.printf( "Row %d not finished ...\n", i);
                                } catch (Exception e) {
                                        System.out.println( "exception sleeping " + e);
                                }
                        }
                }
                System.out.printf( "All done\n" );
        }
```

This method above I used to ensure that all of the threads had finished executing and thus the rows had been complete. To ensure they're all finished the one that was assigned to the final column on each row is checked to ensure that it is a one and if it is not yet then the thread sleeps for 10 milliseconds and retries until it is a one and the row has finished processing.

# OpenMP Design

The algorithm for this design I first implemented in a sequential manner to get the test results as shown below to compare them. This was the sequential design where A and B were the input matrices and C is the output matrix. The idea I had was to parallelise the outer loop using OpenMP.

```
for ( i = 0; i < N; ++i ) {
        for ( j = 0; j < N; ++j ) {
            for ( k = 0; k < N; ++k ) {
                C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
            }
        }
}
```

To ensure that this was parallel using OpenMP I simply inserted this line before the first for loop:

```
#pragma omp parallel for private( i, j, k ) shared( matA, matB, matC ) num_threads(
omp_get_num_procs() )
```

This meant that the private variables consisted of the iterators and the shared variables were the matrices with the number of threads set to the number of cores available.

# Results of Both Designs

The tests were done on matrices of N * N with different N sizes for each test.

## Java Threads

For this first set of tests I tested using 4 threads with 8 cores.

```
Parallel time for N is 256: 0.016864 seconds
Parallel time for N is 512: 0.009503 seconds
Parallel time for N is 1024: 0.017938 seconds
Parallel time for N is 2048: 18.150785 seconds
Parallel time for N is 4096: 203.866303 seconds
```

I then increased the number of threads from to 8 threads with 8 cores.

```
Parallel time for N is 256: 0.055152 seconds
Parallel time for N is 512: 0.107369 seconds
Parallel time for N is 1024: 0.573933 seconds
Parallel time for N is 2048: 15.936117 seconds
Parallel time for N is 4096: 183.638555 seconds
```

This set of tests used only one thread and 8 cores.

```
Parallel time for N is 256: 0.026724 seconds
Parallel time for N is 512: 0.232793 seconds
Parallel time for N is 1024: 2.034004 seconds
Parallel time for N is 2048: 45.567817 seconds
Parallel time for N is 4096: 468.361185 seconds
```

This set of tests used just four cores and 4 threads.

```
Parallel time for N is 256: 0.029026 seconds
Parallel time for N is 512: 0.106139 seconds
Parallel time for N is 1024: 0.703057 seconds
Parallel time for N is 2048: 17.388358 seconds
Parallel time for N is 4096: 210.509080 seconds
```

The final set of tests used just 8 cores and N threads to see the most speedup.

```
Parallel time for N is 256: 0.100996 seconds
Parallel time for N is 512: 0.166927 seconds
Parallel time for N is 1024: 0.845867 seconds
Parallel time for N is 2048: 15.076781 seconds
Parallel time for N is 4096: 165.105417 seconds
```

These results here show that the number of threads and cores seems to plateau even for an N size of 4000. After 8 threads and 8 cores there doesn't seem to be much speedup even with a thread per row.

## OpenMP Design

In the testing of the OpenMP design I was not able to specify the number of cores as this was controlled by the OpenMP scheduler. So for these tests I decided to specify a higher number of threads. For this first set of tests I tested using 32 threads with 8 cores available.

```
Parallel time for N is 256: 0.011346 seconds
Parallel time for N is 512: 0.157532 seconds
Parallel time for N is 1024: 2.565454 seconds
Parallel time for N is 2048: 52.147949 seconds
Parallel time for N is 4096: 477.377930 seconds
```

I then increased the number of threads from 32 to 64 threads.

```
Parallel time for N is 256: 0.020604 seconds
Parallel time for N is 512: 0.168620 seconds
Parallel time for N is 1024: 2.547545 seconds
Parallel time for N is 2048: 51.518314 seconds
Parallel time for N is 4096: 478.378021 seconds
```

This set of tests used only the number of cores available as the set number of threads.

```
Parallel time for N is 256: 0.014277 seconds
Parallel time for N is 512: 0.167553 seconds
Parallel time for N is 1024: 2.809162 seconds
Parallel time for N is 2048: 54.794010 seconds
Parallel time for N is 4096: 489.000153 seconds
```

The final set of tests used just four threads.

```
Parallel time for N is 256: 0.030634 seconds
Parallel time for N is 512: 0.219969 seconds
Parallel time for N is 1024: 3.711456 seconds
Parallel time for N is 2048: 74.266014 seconds
Parallel time for N is 4096: 696.502991 seconds
```

To get around the issue of specifying cores I used a different machine with 16 cores and the used num_threads( omp_get_num_procs() ) which yielded the following results.

```
Parallel time for N is 256: 0.093671 seconds
Parallel time for N is 512: 0.489763 seconds
Parallel time for N is 1024: 1.677501 seconds
Parallel time for N is 2048: 16.847601 second
Parallel time for N is 4096: 300.776398 seconds
```

These results here show that the optimal number of threads to use is around 32 threads and after this any speedup will most likely not be useful as it has plateaued. There was also a large speedup with the larger number of cores on the 16 core machine.

### Sequential C Design

```
Sequential time for N is 256: 0.051421 seconds
Sequential time for N is 512: 0.592118 seconds
Sequential time for N is 1024: 15.614195 seconds
Sequential time for N is 2048: 263.491730 seconds
Sequential time for N is 4096: 2251.138916 seconds
```

# Conclusions

In conclusion, the comparison between Java threads and OpenMP for matrix multiplication shows different performance outcomes. Initially, the Java threads design utilised Strassen's Multiplication algorithm, employing a thread per row strategy. However, as the matrix size increased, this approach became inefficient, prompting a redesign. The OpenMP design, on the other hand, parallelised the outer loop using OpenMP directives, distributing computation across multiple threads. Performance evaluations revealed notable differences between the two approaches. The Java threads design had faster execution times compared to OpenMP, especially for larger matrix sizes exceeding 1000. Both designs had significant speedup when compared to the sequential C implementation.The optimal number of threads and cores varied between the designs. For Java threads, an increase in the number of threads beyond a certain point did not yield any worthwhile speedup, indicating a plateau in performance enhancement. OpenMP also showed diminishing returns beyond a specific number of threads, suggesting an optimal thread count for efficient execution.

# References

*Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)*. (2014, March 29).

GeeksforGeeks. https://www.geeksforgeeks.org/strassens-matrix-multiplication/