```
auto start = chrono::steady_clock::now();

for (int i = 0; i < arrayX; ++i)
        for (int j = 0; j < arrayY; ++j)
                for (int k = 0; k < arrayY; ++k)
                {
                        matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                }

auto end = chrono::steady_clock::now();
```

This is the original multiplication code. The best way to parallelise it would be to split it by doing each row (i) at the same time. This can be done by creating a new thread in each instance of the for loop that uses the i variable, and to nest the next 2 loops into that. The reason that it is unecessary to thread each multiplication itself is because the CPU can calculate an individual matrix multiplication so fast it wouldn't be very helpful.

Threading with pthread:
Results:

Matrix size 15x15 no threading: 7500 Nanoseconds
Matrix size 15x15 threading: 6170300 Nanoseconds
Difference: 822x

Small matrix multiplication is inefficient using threads.

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 threading: 98142900 Nanoseconds
Difference: 5.5x

It seems like due to the fact that each thread is being used for such a menial task, that the effort of creating and implementing the thread still causes an increase in time, however, the difference has been reduced significantly. In the 15x15 matrix, the threading took over 800 times slower to complete the task, where the 200x200 matrix was only 5.5 times slower. This infers that as the required calculations grow threading becomes more efficient.

OpenMP:
Results:

```
Matrix size 15x15 no threading: 7500 Nanoseconds
Matrix size 15x15 OpenMP:       7400 Nanoseconds
Difference: 0.98x

Small matrix multiplication is inefficient using threads.

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 OpenMP:       16842300 Nanoseconds
Difference: 0.95x

Multiplying using openMP netted between 2% and 5% faster processing times over the
sequential calculation times.
```