```
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.

MPI with 3 slaves:
Results:

Matrix size 15x15 no threading: 7500 Nanoseconds
Matrix size 15x15 MPI: 859700 Nanoseconds
Difference: 114.6x

Small matrix multiplication is terrible due to the added tasks of sending and receiving the matrix data to other nodes
or processes in this case.

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 MPI: 142913000 Nanoseconds
Difference: 8.1x

Multiplying using MPI with larger scale matrix of 200 x 200 was significantly more efficient that the small matrix, but
still proved to be less efficient than other methods of splitting multi-threading. I believe this is due to the fact that
the program needs to send and receive data multiple times in order to finish the multiplication.

MPI with 5 slaves:
Results:

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 MPI (5 slaves):116575000 Nanoseconds
Difference: 6.6x

Increasing the number of slaves (threads) to 5 resulted in faster speeds, so would indicate that with enough seperate
slaves, and a big enough task that this method would be faster than no threading at all.

MPI & OpenMP with 5 slaves:
Results:

Matrix size 15x15 no threading: 7500 Nanoseconds
Matrix size 15x15 MPI & OpenMP: 933100 Nanoseconds
Difference: 124.4x

Small matrix multiplication is terrible due to the added tasks of simulating sending and receiving the matrix data to
other nodes as well as attempting to multithread using OpenMP.

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 MPI & OpenMP: 166638900 Nanoseconds
Difference: 9.4x

Multiplying using MPI as well as OpenMP with a larger scale matrix of 200 x 200 was worse than with purely using MPI
(tested multiple times). I beleive this is due to the fact that this was just a simulated example of using multiple
nodes on the same PC, as opposed to multiple PCs which would have more than a single CPU attempting to do all the
work.

MPI & OpenCL with 5 slaves:
Results:

Matrix size 15x15 no threading: 7500 Nanoseconds
Matrix size 15x15 MPI & OpenCL: 1004700 Nanoseconds
Difference: 133.96x

This was the worst example so far, not only does my PC now have to simulate 6 PCs, it also has to allocate and run
the program 5 times using the GPU and all that entails. Let's see what a larger matrix is like.

Matrix size 200x200 no threading: 17721100 Nanoseconds
Matrix size 200x200 MPI & OpenCL: 1814600 Nanoseconds
Difference: 0.102x

WOW! This is the best result yet, so much better that i had to run it multiple times, and i'll try it using an even
larger set of matrix multiplications. I'm unsure what makes it so much more efficient, as the 200x200 matrix time
is close to the 15x15 time.

Matrix size 1000x1000 no threading: 2463404200 Nanoseconds

Matrix size 1000x1000 MPI & OpenCL: 39674700 Nanoseconds
Difference: 0.016x

This result shows definitively that using MPI and OpenCL together are the best possible method so far. I could
only imagine how much better it would be using multiple PCs as slaves as opposed to using threads on a single
CPU.