About:

To build the serial version use “gcc mat_mul_serial_part.c”.
To build the MPI version just type make. The executable will be called prog.

To run do the following: “mpirun ./prog MAT_SIZE NUM_DIVISIONS”
The A and B matrices will be of size MAT_SIZE x MAT_SIZE, and will be divided into NUM_DIVISIONS grids along their rows and columns.
So if you run “mpirun ./prog 90 3” this means that MAT_SIZE is 90, NUM_DIVISIONS is 3, and therefore each matrix will be divided into 9 parts of size 30x30 each.
NUM_DIVISIONS should be the square root of the number of nodes - 3 divisions for 9 nodes, 2 divisions for 4 nodes etc.
MAT_SIZE should be evenly divisible by the number of nodes.

My code has the option of using a control matrix to ensure the result is correct. The control matrix is calculated using a serial approach. To enable/disable the control matrix open *common.h* and set *USE_CONTROL_MATRIX* to 1 or 0.

My code also has the option to time the multiplication. Process 0 is responsible for the timing. Timing includes all communication and multiplication that takes place. If the control matrix is enabled then its calculation will be timed separately to provide a comparison to the parallel version. To enable/disable timing open *common.h* and set *TIME_EXECUTION* to 1 or 0.

Finally my code has the option to print out the A and B matrices and the control matrix if used. This is useful for debugging, but is not recommended when the matrices are large. To enable/disable this printing open *common.h* and set *PRINT_INITIAL_MATRICES* to 1 or 0.

The file print_functions.c contains print functions that were invaluable during debugging. I have commented these calls however, as they were no longer needed.

The file mat_util.c contains useful utility functions for working with matrices.

Testing method:

I tested my code using both the TCD HPC cluster, and my own 8 core CPU.
In both cases a serial multiplication was performed to get a baseline result, and then the parallel multiplication was timed.

Results on TCD HPC cluster:

With 4 nodes and 2000x2000 matrices:

Serial: 58’321’968 microseconds
MPI: 18’352’004 microseconds

Here we see a ~3.1x speedup, which is quite good.

With 9 nodes and 3000x3000 matrices:

Serial: 184’210’393 microseconds
MPI: 78’108’520 microseconds.

Here I got a speedup of ~2.35x. This was lower than expected. Perhaps high communication costs are the reason for this relatively low speedup.

Results on a single Intel i7-7820X:

For the following results I used my own computer.
The CPU used was an Intel i7-7820X with 8 cores, 1MB of L2 cache, 11MB of L3 cache and a clock speed of 3.6GHz.
Each MPI node was a core on this CPU. Because of this communication costs are lower for these results.

With 4 nodes and 800x800 matrices:

Serial: 1’974’352 microseconds
MPI: 480’207 microseconds

Here I got a speedup of almost 4x, which is close to the ideal as I was using 4x more cores in the parallel version.

With 4 nodes and 2000x2000 matrices:

Serial: 80’382’862 microseconds
MPI: 13’877’486 microseconds

Here the speedup is ~5.8x. This is strange as when doing the work with 4 cores we should expect a max speedup of 4x.
I believe the reason the speedup was so high is that the cache usage was unintentionally improved.
In the serial version a huge amount of pressure is put on the cache when the matrices are this large. There is a lot of cache thrashing as many addresses will map to the same cache location.
In the serial version each core only deals with 1/4th of the total data, leading to less cache pressure.

Possible improvements:

My functions for extracting the matrix parts from the original A and B matrices is not very efficient.

Likewise my function for inserting the partial results into the final result matrix is not very efficient.
Finally my matrix multiplication is not cache friendly. This could be resolved by transposing the matrices, but this would also involve changing the order in which parts are sent between nodes and I did not have time to do this.