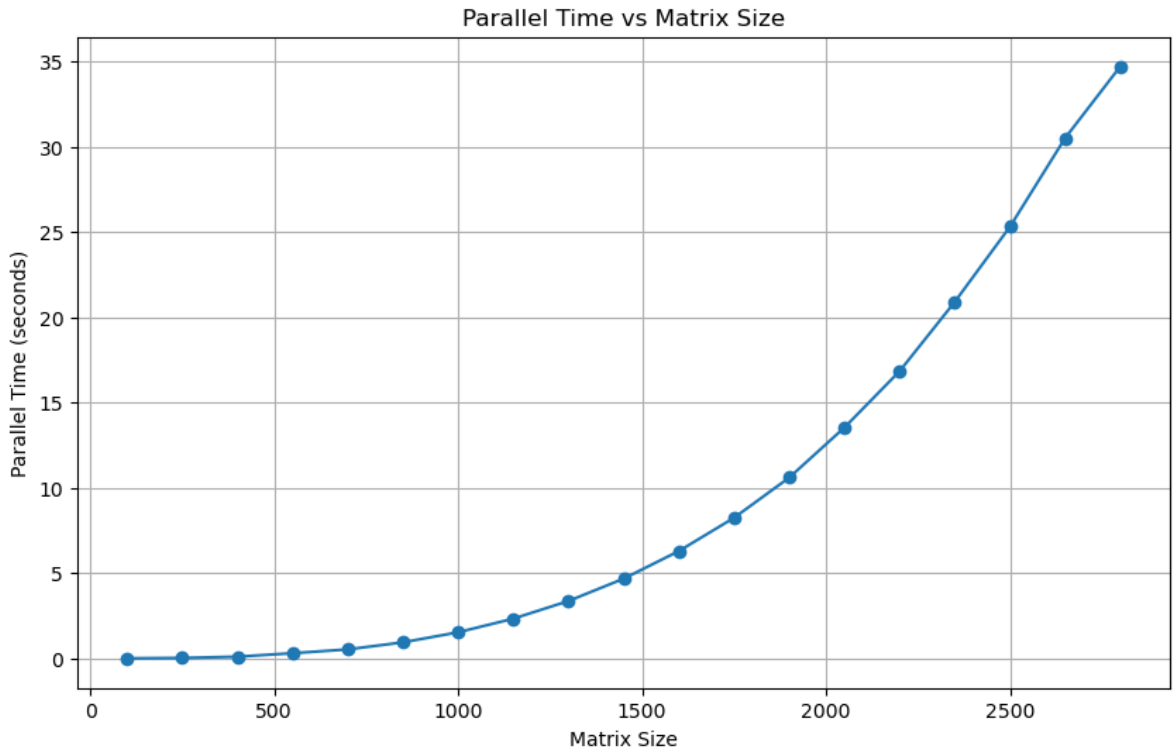
Analysis

The dependence of the execution time of the program on the matrix size n



Examining the graph, we can easily see that as the matrix size *n*, grows, so does the time required for computation. This trend is expected since larger matrices demand more processing power. We can also see that the increase in time is non-linear, resembling a curve, possibly polynomial or exponential, as *n* increases linearly.

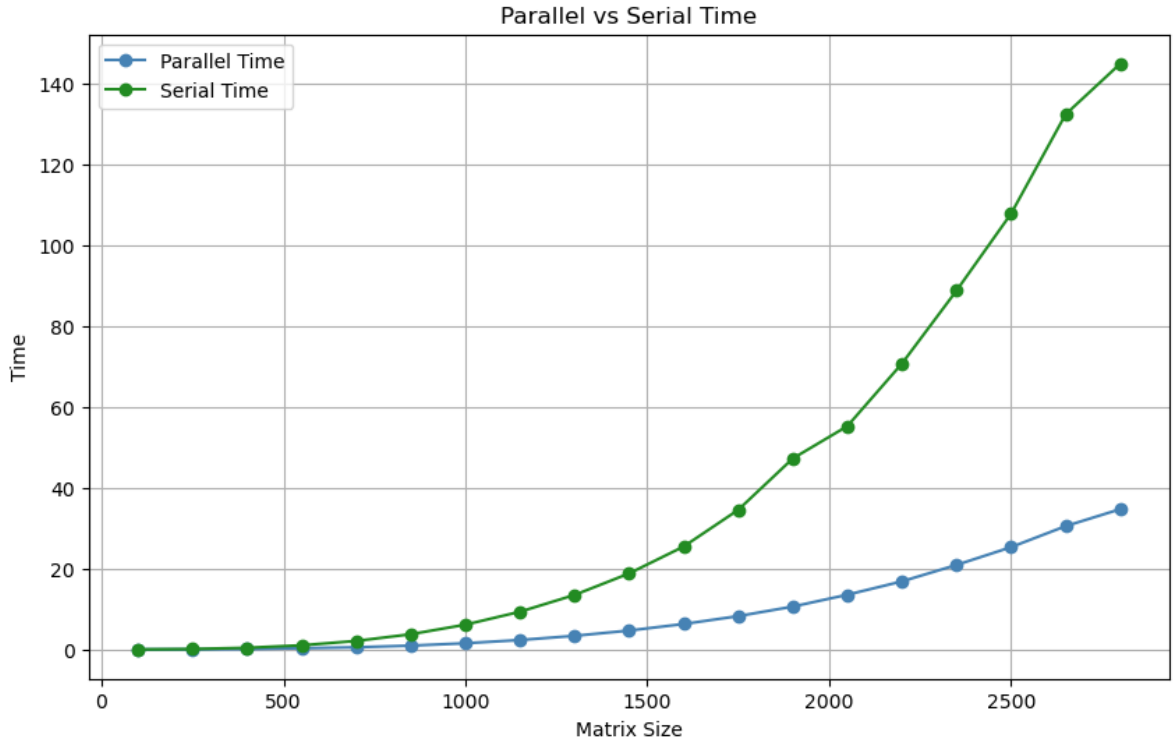
However, the graph does not provide a comprehensive view of the efficiency of the parallel method. To explore of the performance further, I will discuss the efficiency in relation to the serial method.

Note:
The number of processors p is dynamically calculated at runtime using the MPI function <MPI_Comm_size(MPI_COMM_WORLD, &p)> , which queries the MPI environment to determine the number of processors available in the current execution context.

As a result, the exact number of processors utilised varies depending on the specific configuration and availability of resources in the MPI environment during each run of the program. This approach ensures that our program adapts to the available parallel computing resource.

For my particular experiments I set p as 4 using the openMPI command <mpirun -np 4 ./main>.

The speedup over a serial counterpart of the program



Looking at the graph, we can clearly see that parallel computation has a substantial advantage, particularly when it comes to larger matrices. As the size of the matrix increases, the line for serial computation begins to rise steeply, in contrast to the more gradual ascent of the parallel computation line. This marked difference demonstrates the superior efficiency of parallel processing for handling complex tasks involving large datasets. We especially see how they begin to diverge around matrix size=500.

The reason for the parallel method having a more consistent increase in time lies in its approach to splitting the matrix into smaller sections, which are then processed concurrently. This effectively utilises the multi-core design of contemporary processors, distributing the computational load across several threads. In contrast, the serial method, which handles the entire matrix in a singular sequence, finds itself increasingly challenged by larger matrices, leading to a much sharper increase in computation time. However, when I ran this programme on a different machine, the speedup (caused by increased serial times) appeared to improve, suggesting that my machine potentially optimises the serial multiplication somewhat.

It is worth noting from the values in the experiment2.csv that the speedup consistently floated around 4, suggesting there is a limit with the hardware of my processor. That being said 4 is still a great improvement with the parallel method and shows that the parallelisation of my code scales with the increase in matrix size.

This graph provides a good illustration of the strengths of parallel computing, especially for demanding computational tasks. It demonstrates that as the size of the data grows, parallel processing can significantly mitigate the impact on computation time, making it a more efficient choice for intensive calculations.

The effect of the number of processors *p* on speedup

p = 8

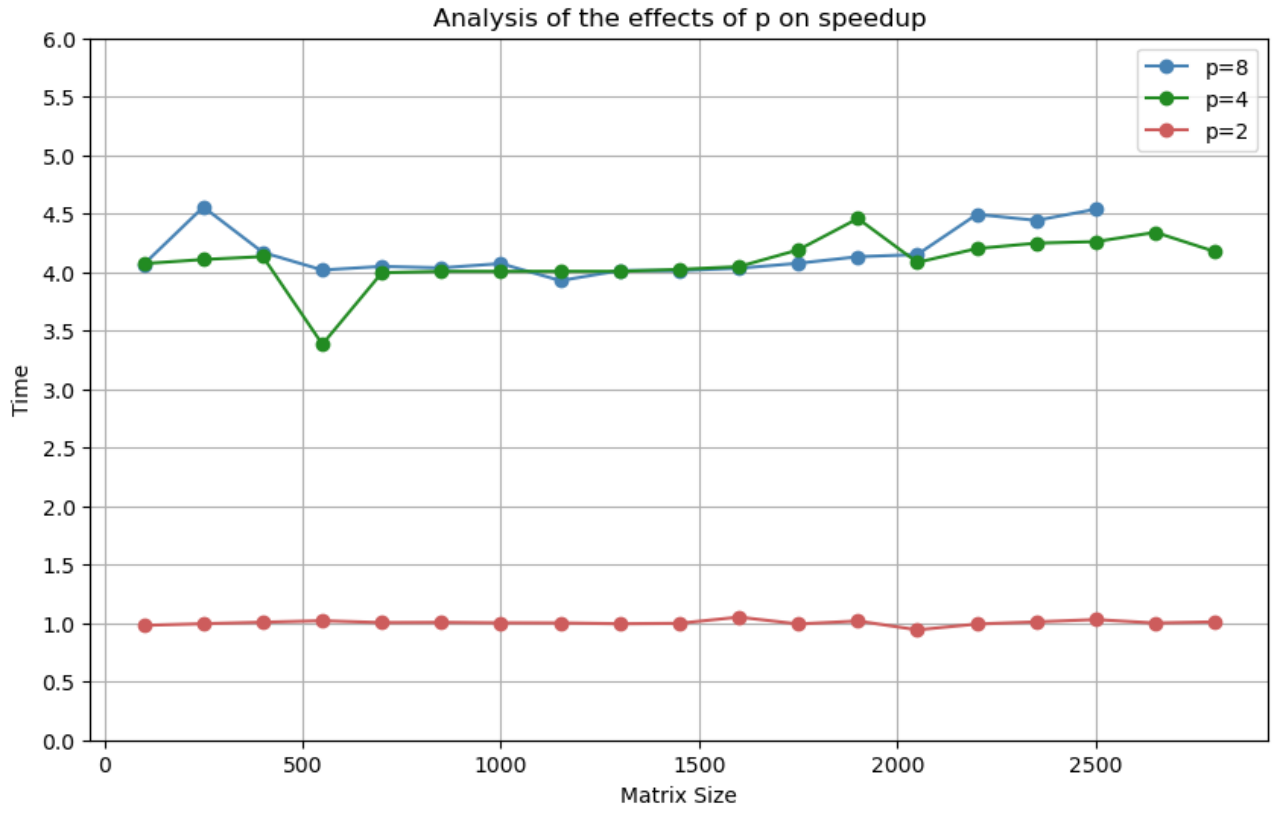
Matrix Size	Serial Time	Parallel Time	Speedup
100	0.005959	0.001465	4.067577
250	0.107519	0.023596	4.556662
400	0.416174	0.099836	4.168576
550	1.01971	0.25379	4.017928
700	2.126491	0.525188	4.049009
850	3.798794	0.941076	4.03665
1000	6.244625	1.53312	4.073148
1150	9.368154	2.385435	3.927231
1300	13.517019	3.36799	4.013379
1450	18.707884	4.662332	4.012559
1600	25.33802	6.282582	4.033058
1750	33.558891	8.233952	4.075672
1900	43.444675	10.51504	4.13167
2050	55.169559	13.292317	4.150485
2200	74.032608	16.474596	4.493743
2350	89.583397	20.166518	4.442185
2500	111.700485	24.615526	4.537806

p = 4

Matrix Size	Serial Time	Parallel Time	Speedup
100	0.005926	0.001455	4.072852
250	0.096426	0.023473	4.107954
400	0.404436	0.097861	4.13276
550	1.021478	0.301946	3.382983
700	2.099202	0.525511	3.994592
850	3.758721	0.938134	4.006593
1000	6.124197	1.528601	4.006407
1150	9.323936	2.327	4.006849
1300	13.471663	3.362487	4.006458
1450	18.798637	4.672137	4.023563
1600	25.456583	6.290354	4.046924
1750	34.532238	8.240396	4.190604
1900	47.244839	10.597695	4.45803
2050	55.255627	13.533082	4.083004
2200	70.73085	16.83509	4.201394
2350	88.780075	20.901503	4.247545
2500	107.815544	25.305041	4.260635
2650	132.40921	30.506183	4.340406
2800	144.925415	34.706657	4.175724

p = 2

Matrix Size	Serial Time	Parallel Time	Speedup
100	0.006345	0.006471	0.980529
250	0.094939	0.095457	0.994573
400	0.391468	0.388569	1.007461
550	1.037346	1.016708	1.020299
700	2.099198	2.09066	1.004084
850	3.774193	3.752295	1.005836
1000	6.129642	6.116769	1.002105
1150	9.346747	9.335935	1.001158
1300	13.461467	13.537216	0.994404
1450	18.724487	18.765273	0.997827
1600	26.548019	25.258235	1.051064
1750	34.716187	34.99048	0.992161
1900	46.852364	46.022775	1.018026
2050	55.178143	58.58279	0.941883
2200	69.513199	70.090242	0.991767
2350	86.792877	85.898994	1.010406
2500	105.928238	102.850194	1.029927
2650	124.464691	124.406422	1.000468
2800	143.566803	142.07013	1.010535



As we can see from the tables and graph above, p=4 is the best number for my hardware as it balances well the additional memory required for parallelisation and the speedup from parallelisation. P=2 showed no obvious improvements in performance over the serial computation and p=8 was largely equal to p=4 in terms of performance, meaning the increase in memory use had diminishing returns. In fact, on my computer, I was unable to go over n=2500 as numbers above this caused segmentation errors.
We can see that the parallelisation offers a speedup of around 4, with larger matrices showing more consistently over 4, suggesting even larger matrices could see greater improvements in calculation if the memory was available, and vice versa for smaller matrices.