IN4330 mandatory 2

Erlend Kristensen

21.02.2022

1 Matrix multiplication

For this mandatory project we wanted to take a look into how to speed up matrix multiplication in java. The classic way of doing matrix multiplication given the form

$$C = AB \tag{1}$$

is to multiply the rows of the A matrix with the columns of the B matrix. This however causes problems in memory fetching, seeing as we have to fetch a new cache line each time we go one element down in the columns of the B matrix. We therefore want to first transpose the B matrix to B^T, and then multiply rows of A with the rows of B^T. We will also do the opposite, where we transpose A to A^T and multiply the columns of A^T with the columns of B. We do this so we can compare all three and see the effect smart memory fetching has on the speed of our program. We then parallelize all 3 methods and see what speedup we can achieve.

When parallelizing, I chose to do a similar approach for all methods. The parallelization works by dividing the C matrix so each string gets its own set of rows. This is so we can keep the functions looking as similar as possible, and also try to maintain as much computational speed as possible. This means when synchronizing the strings, we need to fetch a part of the C matrix from each of the strings, which is also easily done (just a bit of smart indexing).

With this method of parallelizing we could also split A up as well, since the individual strings will not need the entire matrix. However, i chose to not do this since the matrices are not so big that it should matter too much on the memory allocation, but would be smart if we were to do bigger sizes than 1000×1000.

For our tests we used 4 different matrix sizes: $n = 100, 200, 500, 1000$. The code was run on two different machines, one with a AMD Ryzen 5 5500U cpu with 6 cores and a base speed of 2.1 GHz, and the other on IFI's own machines via virtual connection. The IFI machines only had two cores available when i tested, so the speedup will differ.

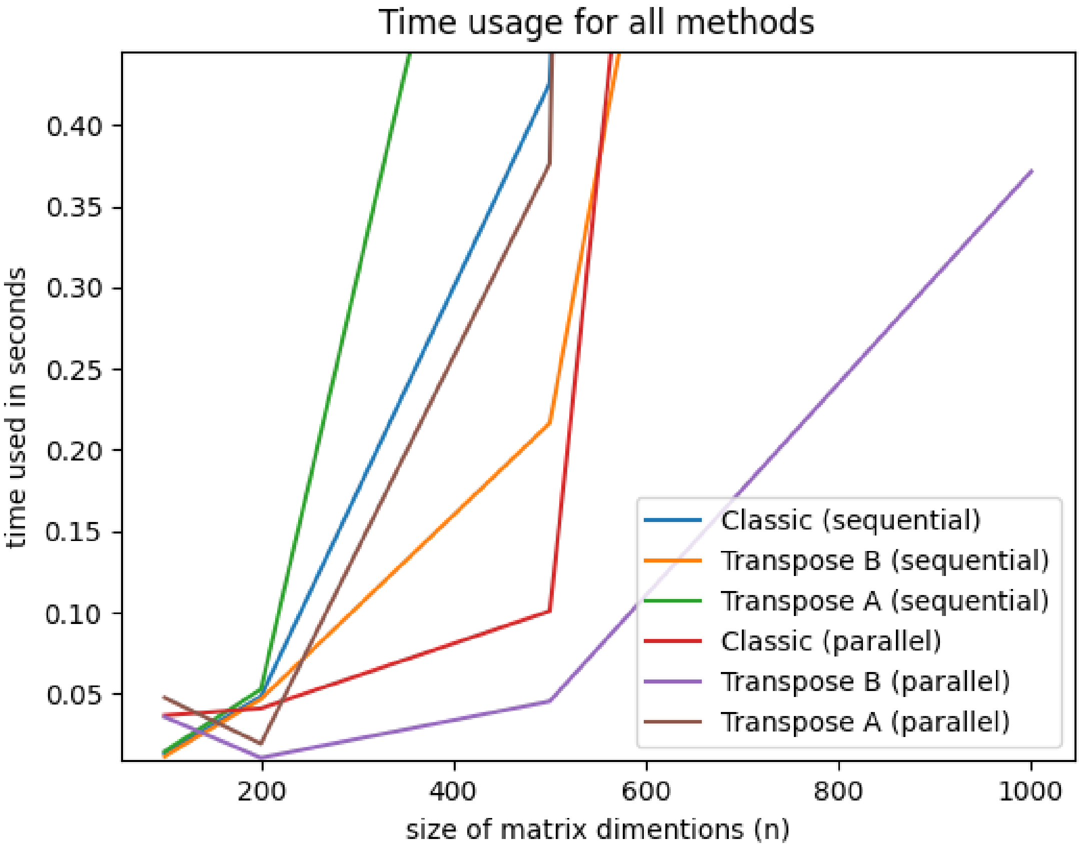
Here are the times we got:

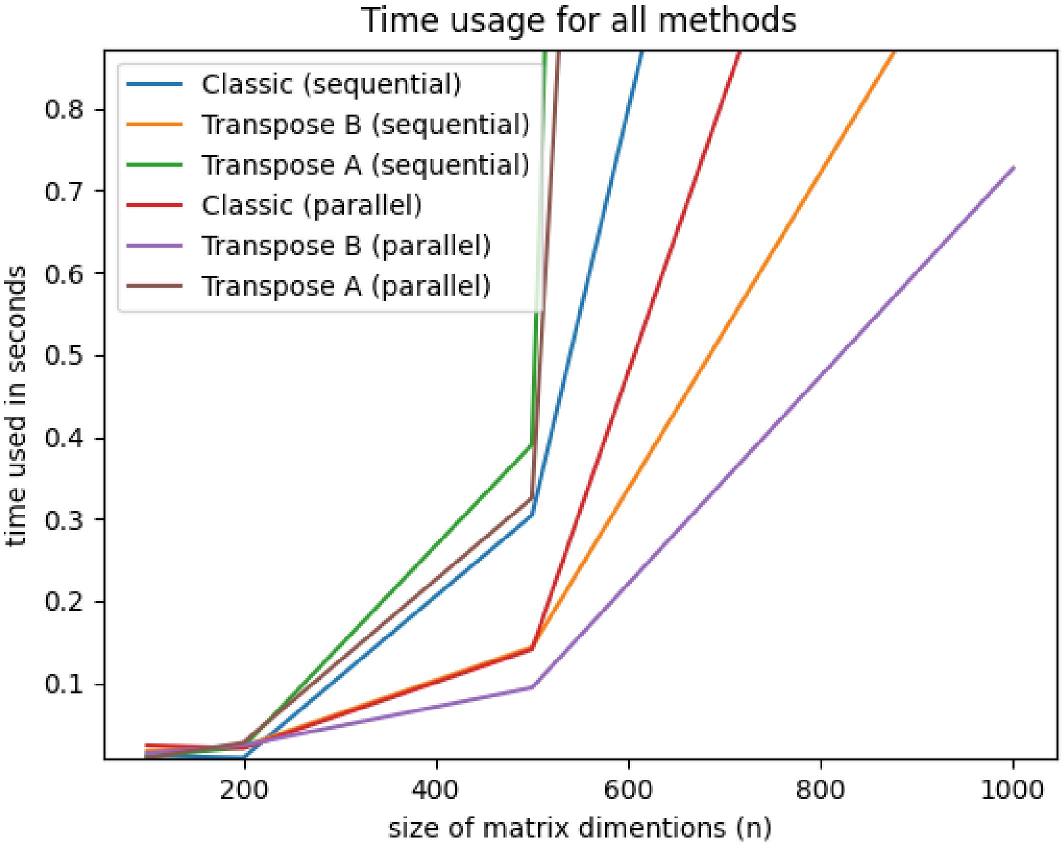
time in seconds my own laptop (6 cores)			
Size of array (n)	Classic	B-Transposed	A-Transposed
100	0.0096 (0.0400)	0.0103 (0.0416)	0.0104 (0.0640)
200	0.0506 (0.0128)	0.0469 (0.0476)	0.0524 (0.0557)
500	0.3348 (0.1080)	0.2175 (0.0513)	0.7508 (0.3430)
1000	7.5750 (3.0135)	1.8052 (0.3627)	40.8968 (14.9550)

time in seconds my own IFI (2 cores)			
Size of array (n)	Classic	B-Transposed	A-Transposed
100	0.0120 (0.0245)	0.0179 (0.0149)	0.0102 (0.0092)
200	0.0092 (0.0209)	0.0230 (0.0246)	0.0222 (0.0277)
500	0.3043 (0.1411)	0.1438 (0.0945)	0.3902 (0.3253)
1000	2.7745 (1.8277)	1.1070 (0.7270)	18.1944 (10.1696)

Here the parallel times are presented behind the sequential times. As we see, the B-transposed method is by far the fastest method. On both my own laptop and the IFI machines the speed of B-transpose method is about the same as the Classic one for $n = 100, 200$, but from $n = 500, 1000$ it is a lot faster. We also see the drastic effects of A-transposed method for $n = 1000$, where it is about 20 times slower than the B-transposed method.

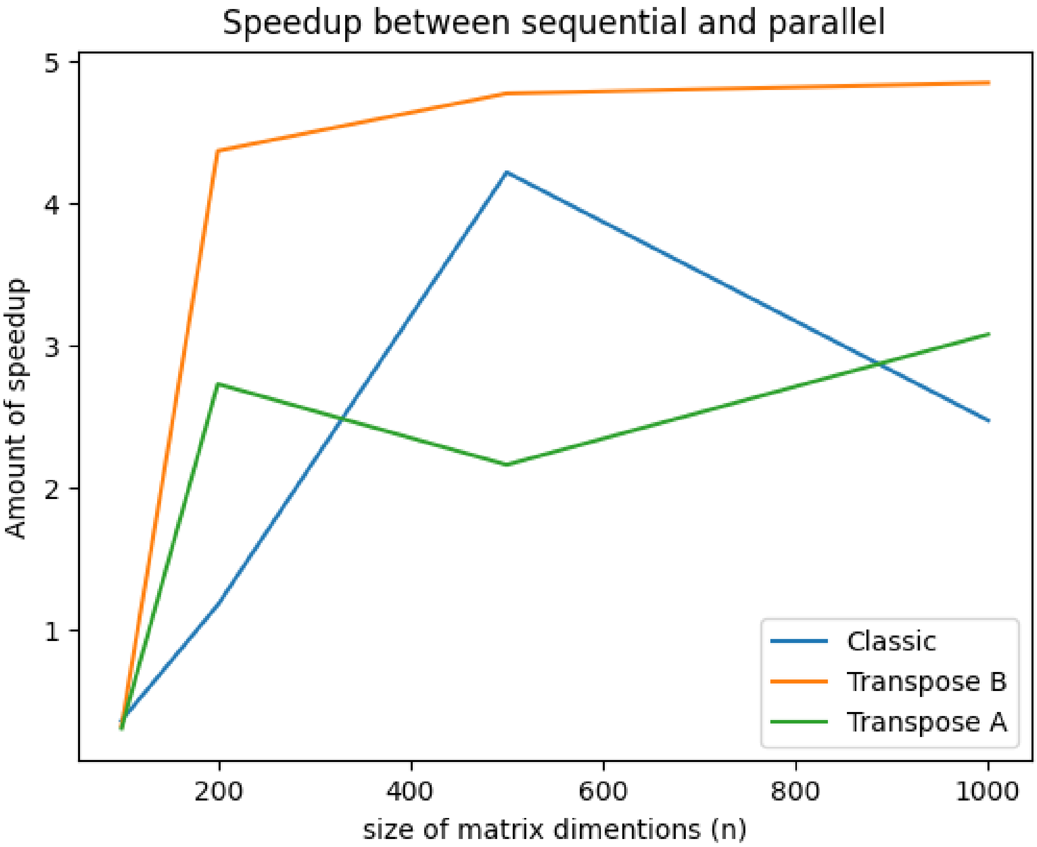
We also made plots of all the 3 methods (sequential and parallel) in a graph plot:

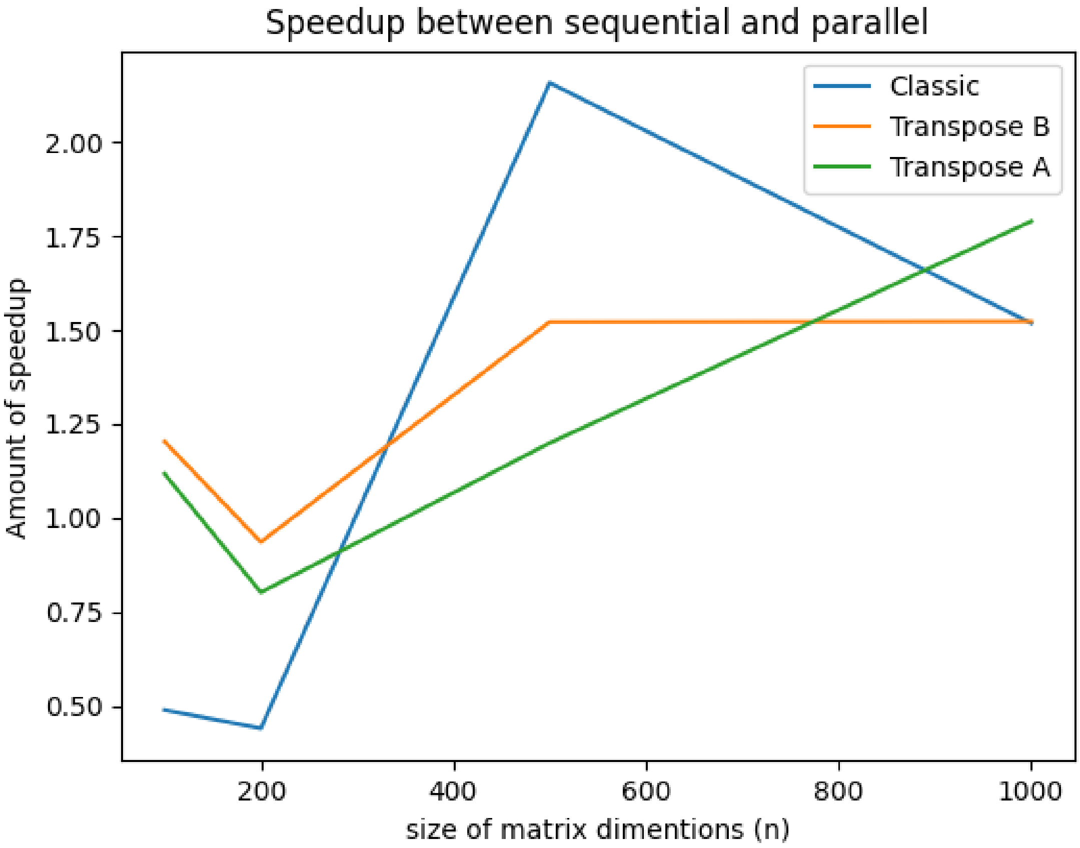

Time usage for all methods
time used in seconds
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
200
400
600
800
1000
size of matrix dimentions (n)
Classic (sequential)
Transpose B (sequential)
Transpose A (sequential)
Classic (parallel)
Transpose B (parallel)
Transpose A (parallel)



The top image is my own computer and bottom one is IFI's. As we clearly see the other methods than B-tranposed really take off at $n = 500$, and clearly shows how exponentially much slower they are as we increase n. This makes sense seeing as matrix multiplication requires $O(n^3)$ operations.

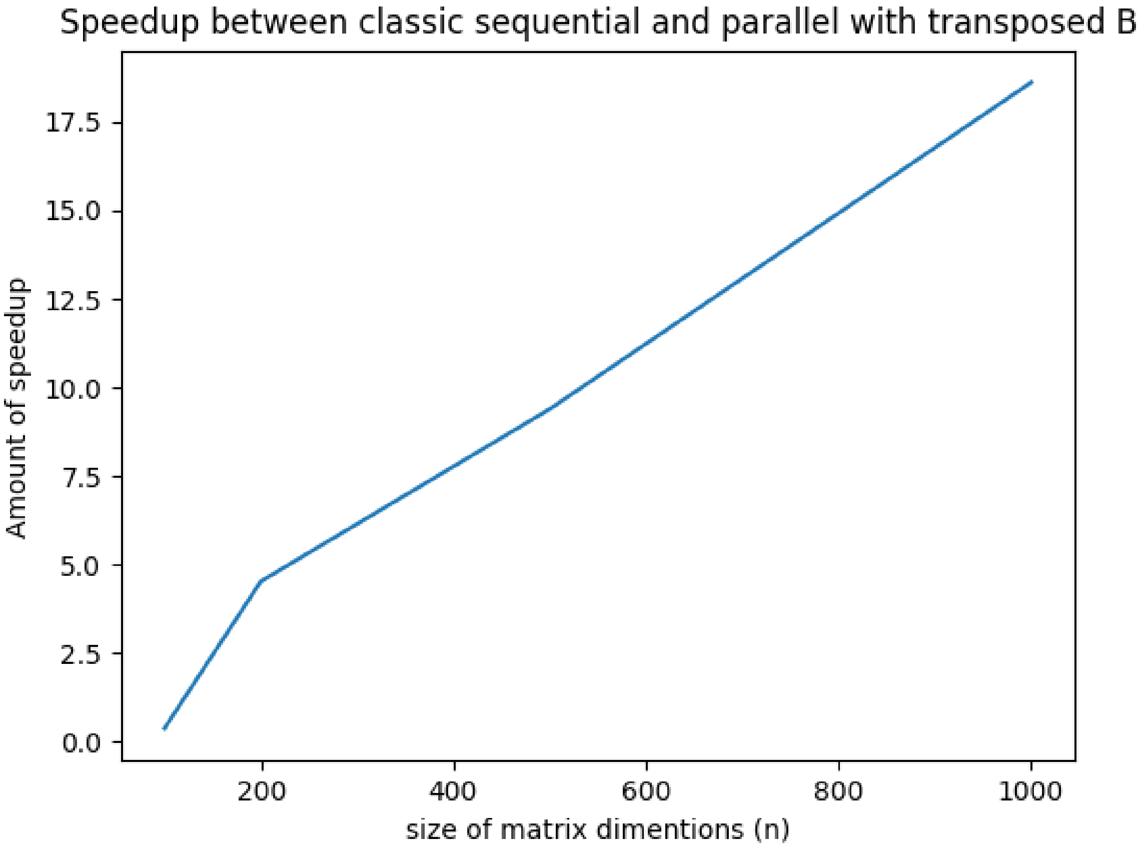
We also take a look at the speedups achieved when parallelizing our code:

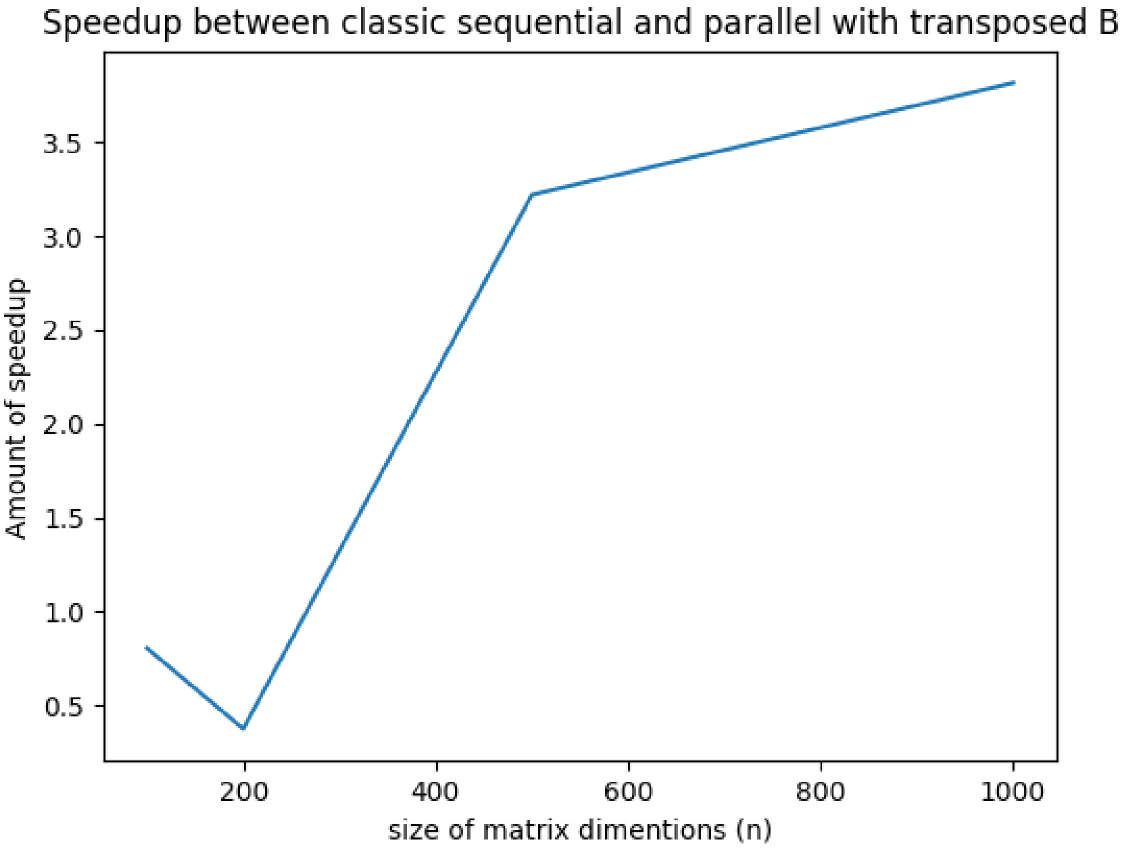

Speedup between sequential and parallel
Amount of speedup
5
4
3
2
1
200
400
600
800
1000
size of matrix dimentions (n)
Classic
Transpose B
Transpose A



Here the top picture is from my own laptop and bottom is IFI's machines.
As we see, my own laptop achieves on average about a $3-4$ times speedup (on the larger matrices), which is pretty good when using 6 cores. IFI's machines achieve about a 1.7 times speedup on the larger matrices, which also is very good seeing as we only used 2 cores. We also see that for $n = 100-200$, the speedup is either less than 1 or about 1, meaning parallelization here is useless and should be avoided.

Lastly i made a graph comparing classic matrix multiplication with B-tranposed parallelized to see just how much smarter this way of doing it is.





As we see here, the method greatly improves our computation speed. With only 2 cores on the IFI machines we achieve almost 4 times speedup, and with 6 cores we get a speedup of almost 18! This shows us that combining smart implementations of algorithms combined with parallelization can really bring drastic changes to your computation speed.

So as we now have seen, for small matrix sizes it does not really matter too much which method we choose, since just allocating the arrays will probably take more time anyways. For larger matrices however, it is really important to use smarter methods to go through memory. Combining this with good parallelization will greatly improve our computational speeds!