Introducion
This exercise is about combining parallelization and cache-friendly code in order to create a fast algorithm that solves a matrix multiplication.
For the tests I used my laptop equipped with an Intel Core i7-1165G7 2.80GHz 4 cores with 8 threads, but it is 4.70 GHz while it's charging. I'm using it at 4.70 GHz.
The data are gathered on the spreadsheet attached, where I made the graphs also.
For the seed number I used 42.

Sequential Matrix Multiplication
The sequential algorithms are pretty straightforward:
- In the *not transposed* I just do the classic matrix multiplication without any changes
- In *B transposed* the B matrix is transposed. Then, when you make the actual multiplication, you have to swap the B indexes (column and rows) in order to have the correct result.
  This is due the fact that if you transpose the matrix you are swapping columns with rows, so you have to do the same when you multiply.
- In *A transposed* the A matrix is transposed, instead. The logic is the same as the B matrix but you have to do it with the A matrix.

Parallel Matrix Multiplication
The parallel algorithms use the same trick to have the correct result, precisely if the matrix is transposed, the corresponding indexes during the multiplications are swapped.
About the parallelization part, I divide the matrix in *threadNumber* sets of rows (8 in my case, because I have 8 threads) and then I use the same algorithm as in the sequential multiplication.
I divide the total matrix in rows, and not in columns, because it's more cache friendly.
I didn't have any synchronisation problem, that is because the result of a single cell in the resultant matrix is always written by a single thread only.

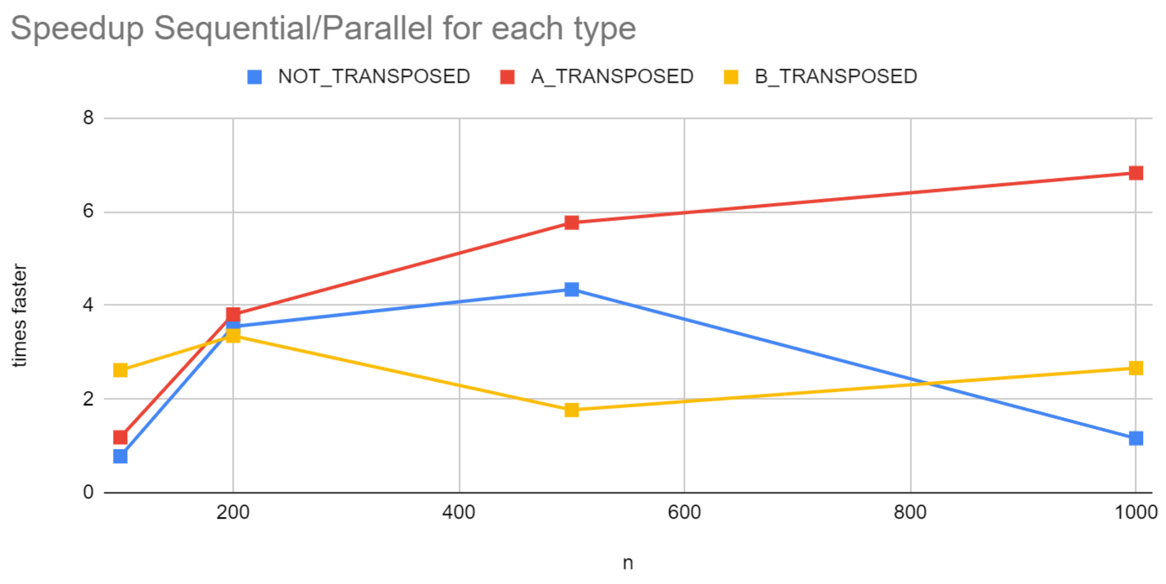
Measurements

Medians						
	SEQUENTIAL			PARALLEL		
n	NOT TR.	B TR.	A TR.	NOT TR.	B TR.	B TR.
100	0.855	0.879	2.424	1.099	0.743	0.926
200	18.331	12.870	21.306	5.168	3.380	6.359
500	180.313	104.883	282.497	41.558	18.190	159.932
1000	2685.994	986.624	14409.909	2311.717	144.555	5414.298

This table contains the summary of all the medians time expressed in milliseconds

	Sequential/Parallel		
n	NOT_TRANSPOSED	A_TRANSPOSED	B_TRANSPOSED
100	0.778	1.182	2.617
200	3.547	3.807	3.350
500	4.339	5.766	1.766
1000	1.162	6.825	2.661

In this table is calculated the speed up between sequential and parallel algorithms foreach n and method combination.
Apart from the n = 100 in the *not transposed* solution, the parallel algorithm is always the best option.
As we can see, sometimes I have some measurements that are close to 1 but they are still better than the sequential algorithm.

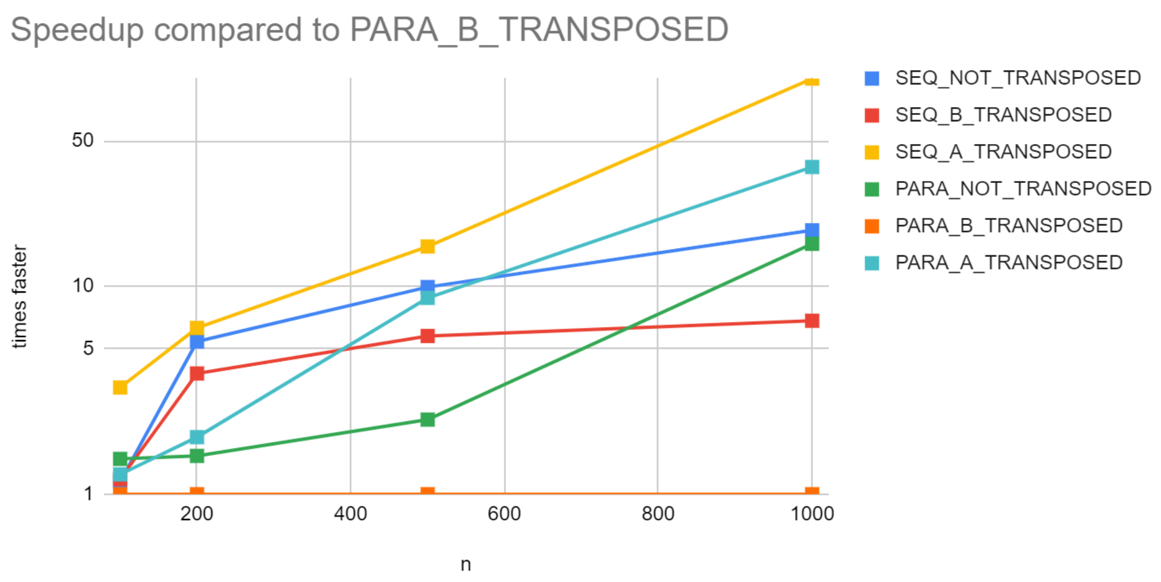


This is the graph of this data where we can clearly see that the parallelization of the *non-transposed* solution is not so useful when you are dealing with bigger n.

Speed up parallel on b transposed						
	SEQUENTIAL			PARALLEL		
n	NOT TR.	B TR.	A TR.	NOT TR.	B TR.	B TR.
100	1.150	1.182	3.261	1.478	1.000	1.246
200	5.423	3.807	6.303	1.529	1.000	1.881
500	9.913	5.766	15.530	2.285	1.000	8.792
1000	18.581	6.825	99.685	15.992	1.000	37.455

This table shows the speed up between the best solution (*i.e. parallel not transposed*) and all the others.

We can see that the *b transposed* used with the *parallelization* beats every other solution. This is because it combines the parallelization with the b transposed matrix that, since it is scrolled by row instead of columns, is more “cache friendly”.
On the contrary in *a transposed* you are making not cache friendly the A matrix, so you will have worse results.



This is the graph showing the data just discussed.

User Guide
In order to use the program the user can/must specify:

- The number of thread that the program will use (mandatory)
- The length/height of the matrix (mandatory)
- The seed of the random number (optional)
  If you don’t write the seed number it will randomly choose a number between 0 and 9999999. This number will be shown in the terminal.

If you put a wrong number of arguments the program will stop and will say to input -h to show the helper.
The helper will remind you what values you should input.

Conclusion
This exercise put together parallelization and cache.
The result is that we can clearly see how a good use of those concepts can improve your program speed, but on the other hand we have to be more careful, because the code can be tricky to write and edit when dealing with those things.