Oblig2 Report

CPU info

My computer has an Apple M1 Pro chip. It has 8 cores with a speed of 3.20 GHz.

Sequential Matrix Multiplication

My classic sequential method I used was just the standard one, with three nested for-loops to multiply the elements of the rows of matrix A by the corresponding elements of the columns of matrix B and summing the products. For my transposed methods I created a simple method that switch the places of the rows and columns in the matrix. After that I changed my matrix multiplication so that it multiplied the rows of matrix A with the corresponding elements of the row in matrix B, or so that it multiplied the columns of matrix A with the corresponding elements of the columns in matrix B, depending on what matrix had been transposed.

Parallel Matrix Multiplication

To make my matrix multiplication parallel I decided to make several threads that got different parts of the matrix to calculate. I used all the available cores on my computer and split the number of rows in the matrix evenly between them, then every thread just worked on these number of rows. I did not need to synchronize, other than waiting for all the threads to be done with a CyclicBarrier, because they all wrote to different parts of the result array.

User guide – how to run my program

To execute my program (after compiling it) simply write:
java Oblig2 <seed> <n>
For example: java Oblig2 42 500
This will give you an output that will show the times for all the methods, if the sequential methods and parallel methods have the same results and what the speedup is.

Measurements/Discussion

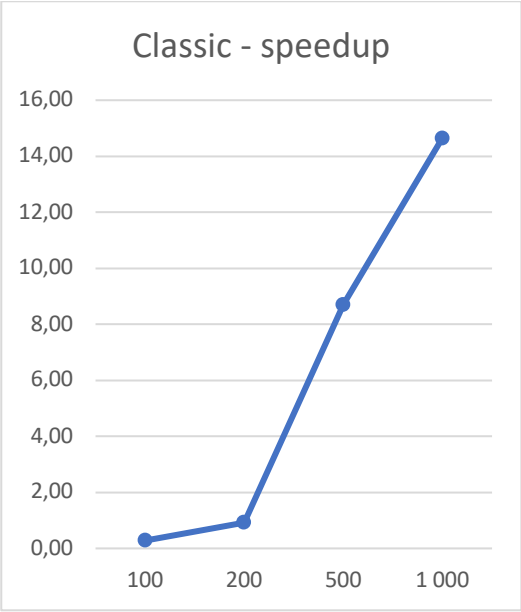
Here are my tables that show the execution times for the classic sequential method, classic parallel method, transposed A sequential, transposed A parallel, transposed B sequential and transposed B parallel. As well as the graphical representation of the speedup result for all of them.

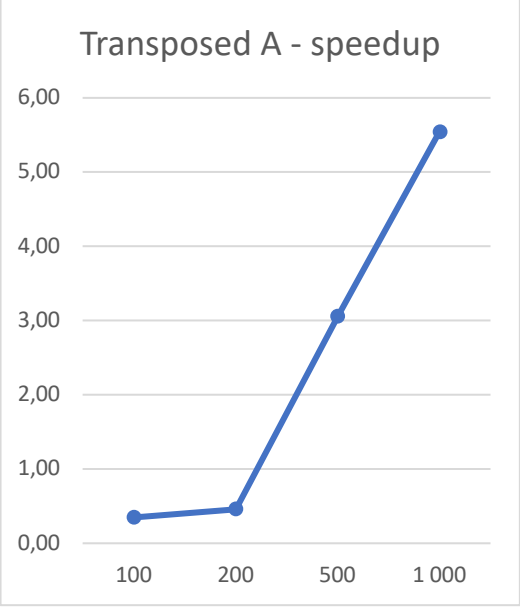
Classic

N- values	Sequential	Paralell	Speedup
100	7	25	0,28
200	38	41	0,93
500	566	65	8,71
1 000	4587	313	14,65

Transposed A

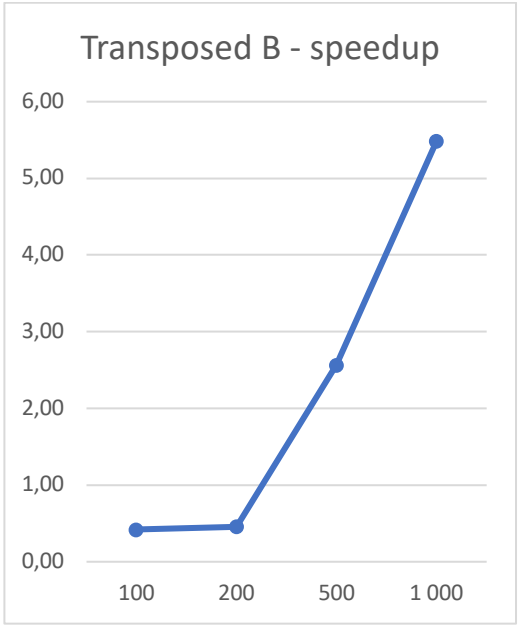
N- values	Sequential	Paralell	Speedup
100	8	23	0,35
200	17	37	0,46
500	275	90	3,06
1 000	3371	608	5,54





Transposed B

N- values	Sequential	Parallel	Speedup
100	8	19	0,42
200	17	37	0,46
500	159	62	2,56
1 000	1240	226	5,49



As you can observe in my tables and graphical representation, I do get a speedup of over >1 for when n >= 500 for all my parallel methods. The speed up varies for different sizes of the matrices because when n is larger the overhead from creating threads has less and less impact on the execution times, so the larger the n the more beneficial the parallel methods become in comparison to the sequential methods.

Here is the graph comparing all the six methods using times from n = 1000.
As seen the transposed B para method is the fastest method for n = 1000, and the classic seq is the slowest, 20,30 times slower than transposed B para.