# IN3030, Oblig two

## Introduction

In this assignment I have implemented a sequential and a parallel solution to matrix multiplication and tested them according to time of execution and speedup. I will also briefly talk about how the sequential algorithm works, and what steps I made to make it work in parallel.

## Sequential matrix multiplication

The algorithm produces a single matrix from the two input matrixes, a and b. The number of columns in the first matrix must be equal to the number of rows of the second matrix, if not the product can not be calculated. The resulting matrix has rows = number of rows of a, and columns = number of columns in b. Then we iterate over the result matrix empty cells, for each cell, the algorithm multiplies the values in the current row of A with the values in the current col of b, adds the results together and places the value in the current cell of the result matrix.

Meaning that the index 0,0 in the result matrix will be filled with the value of the calculation: all values in row 0 multiplied with the values in colum 0 in b, then adding them together. Meaning that index 0,0 in a is multiplied with 0,0 in b, 0,1 in a is multiplied with 1,0 in b, 0,2 in a is multiplied with 2,0 in b…, then adding them all together, and placing the value in 0,0 in the result matrix. When all cells in the result matrix have been visited, the result of a * b = ab is stored in the result matrix. If the mode is a transpose of a or b, the matrix is transposed before these steps, not sure if I actually have to physically transpose the matrix, or if I can just work around it with indexing under the multiplication.

## Parallel matrix multiplication

To implement the algorithm as a parallel one utilizing threads was rather easy. The two input matrixes a and b are not modified in any way. Only the result matrix will be altered. By dividing the result matrix into chunks, where a single thread has the responsibility for only a certain part of the result matrix, the threads will never walk in each other’s feet or block each other.

The easiest approach would have been to give a certain number of rows to each thread, but there is no guarantee that the result matrix will be symmetrical. There could be cases where there are few rows, and a very large number of columns. So, I chose to divide the result matrix by the number of cells, meaning that each thread has the responsibly from a certain start position for example 0,0 to 2,5, while another thread has responsibility from 2,5 to 5,10.

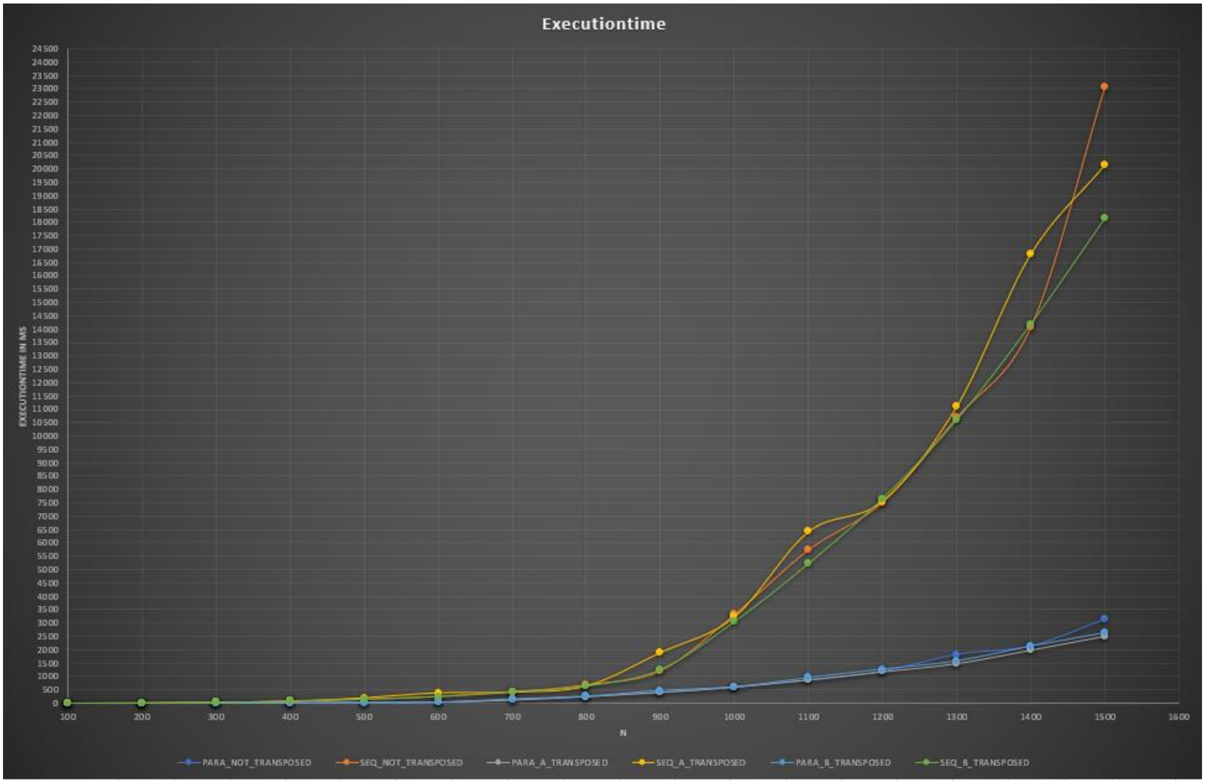
Since the threads don’t modify a or b, and only work in their own part of the result matrix, there is very little synchronization needed. The only active synchronization I use is a cyclic barrier, so that the main thread and all finished threads wait for the entire product to be calculated, before continuing execution. By giving each thread a start position and an end position, and only allowing it to work in that interval, could be a form of synchronization too, but not an active one. Also, since the result of each cell of the result matrix is individually calculated, and not reliant of previous calculations of other cells, the matrix multiplication scales very well with parallelization.

## Measurements and results

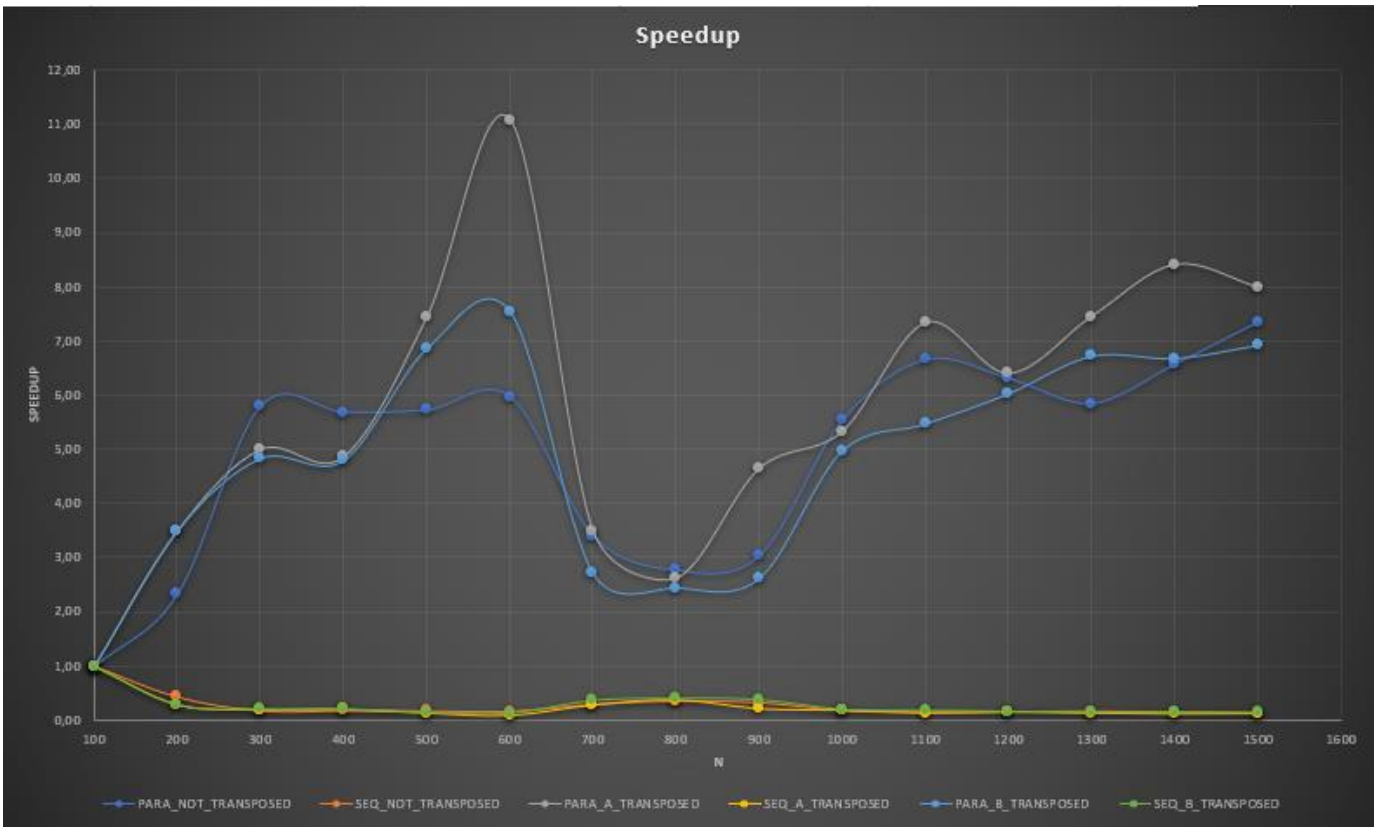
Under this paragraph you can see my measurements represented as graphs and tables. What I can tell from these is that the threaded application will run significantly faster than the sequential ones, with N sizes a little above 100.  However I can see a little dip in the speed up after N = 600, however I think this I because my CPU was working at 100%, and naturally will have to slow down the clock when running all threads at close to maximum clock speed. In shorter bursts, such as N up to 500, the CPU can maybe handle the load while running at max frequencies across all cores, but with larger workloads I believe it will have to cut down.

However, the increase in speedup is significant for all N above 100. From what I can tell from my graphs and observations, the larger the N value, the larger the speedup will be. The largest speedup in my graphs and tables are 11x, but I did see larger values with larger N, but these are not included in the tables/graphs.

There is some variation between the speeds of the parallel ones, however it could just be some natural variables such as the OS, the temperature of the CPU. However, as you can see, they grow at close to the same rate, so they are more or less identical in terms of speedup to their sequential counterparts. Its also important to mention that I recorded very different values in terms of time of execution when running the program with the same N. This is probably explained by different natural variables such as the OS, other processes working in the background, temperature of the CPU. But in general, the threaded workload is much faster than the sequential counterparts for all N > 100.  Its also important to mention that these results are recorded on a 9900k with 8 cores overclocked to 5ghz across all cores (it will dip under heavy load), and that these results are better than the ones I saw on my laptop with a i7 mobile with 6 cores.

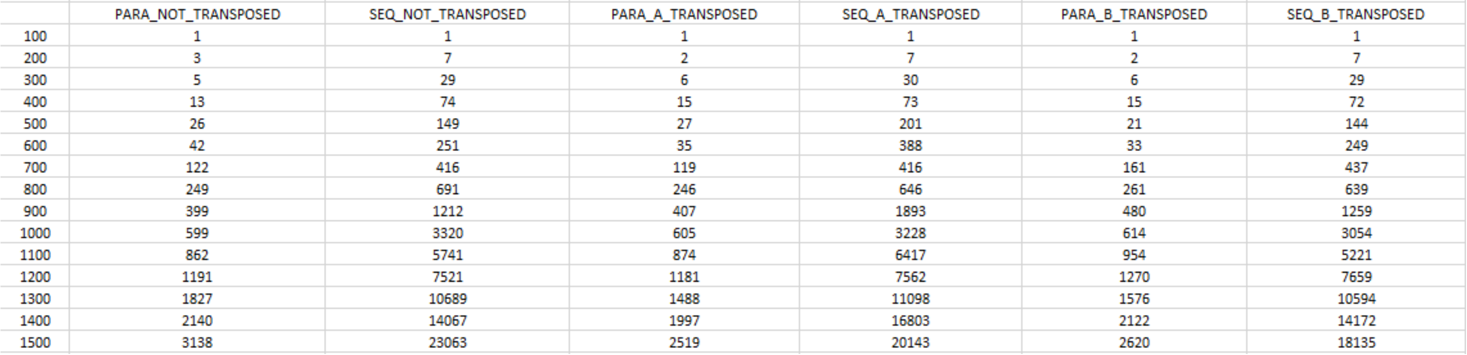


As you can tell from this graph, the time of execution grows at a much greater rate for the sequential algorithms compared to their threaded counterparts. For large N’s it was almost unbearable to wait for the sequential algorithms.



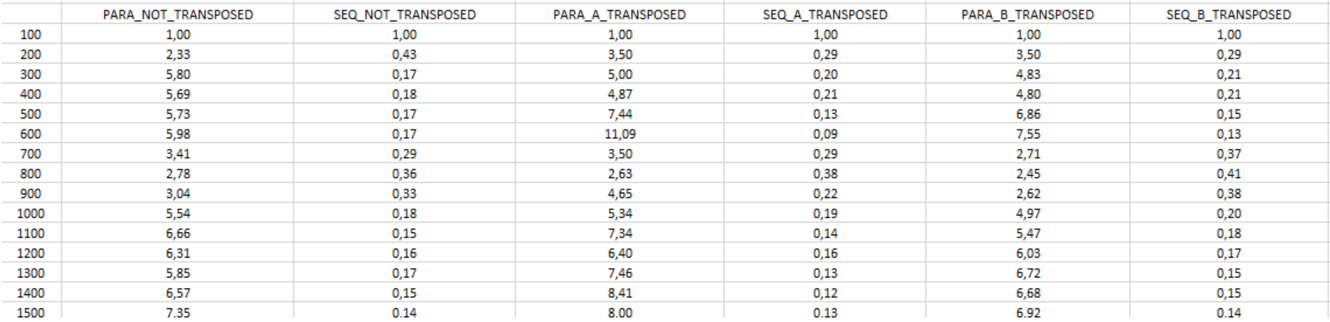
This graph compares the speedup of the parallel algorithm corresponding to the sequential one. Meaning that PARA_NOT_TRANSPOSED displays the speed up compared to SEQ_NOT_TRANSPOSED. As you can see from the graph, the largest speed up I recorded with values up to 1500, was over 11x. I only have a 8 core, so it could seem to be super linear, but the CPU has 16 logical processors, so I believe its not super linear. As you can see the speedup first raises very fast, before dipping with larger values and then stabilizing more with even larger values. I think this is because the CPU can utilize all cores on high frequencies with large, but not very large N's, before it has to drop it when the workload gets very large. Or that when the problem gets large enough and uses all resources over a long time, windows will eventually suspend threads to allow other applications to run too. However, the speedups are still significant across all values of N.

Execution time in MS



| | PARA_NOT_TRANSPOSED | SEQ_NOT_TRANSPOSED | PARA_A_TRANSPOSED | SEQ_A_TRANSPOSED | PARA_B_TRANSPOSED | SEQ_B_TRANSPOSED |
|---|---|---|---|---|---|---|
| 100 | 1 | 1 | 1 | 1 | 1 | 1 |
| 200 | 3 | 7 | 2 | 7 | 2 | 7 |
| 300 | 5 | 29 | 6 | 30 | 6 | 29 |
| 400 | 13 | 74 | 15 | 73 | 15 | 72 |
| 500 | 26 | 149 | 27 | 201 | 21 | 144 |
| 600 | 42 | 251 | 35 | 388 | 33 | 249 |
| 700 | 122 | 416 | 119 | 416 | 161 | 437 |
| 800 | 249 | 691 | 246 | 646 | 261 | 639 |
| 900 | 399 | 1212 | 407 | 1893 | 480 | 1259 |
| 1000 | 599 | 3320 | 605 | 3228 | 614 | 3054 |
| 1100 | 862 | 5741 | 874 | 6417 | 954 | 5221 |
| 1200 | 1191 | 7521 | 1181 | 7562 | 1270 | 7659 |
| 1300 | 1827 | 10689 | 1488 | 11098 | 1576 | 10594 |
| 1400 | 2140 | 14067 | 1997 | 16803 | 2122 | 14172 |
| 1500 | 3138 | 23063 | 2519 | 20143 | 2620 | 18135 |

Speedup



| | PARA_NOT_TRANSPOSED | SEQ_NOT_TRANSPOSED | PARA_A_TRANSPOSED | SEQ_A_TRANSPOSED | PARA_B_TRANSPOSED | SEQ_B_TRANSPOSED |
|---|---|---|---|---|---|---|
| 100 | 1,00 | 1,00 | 1,00 | 1,00 | 1,00 | 1,00 |
| 200 | 2,33 | 0,43 | 3,50 | 0,29 | 3,50 | 0,29 |
| 300 | 5,80 | 0,17 | 5,00 | 0,20 | 4,83 | 0,21 |
| 400 | 5,69 | 0,18 | 4,87 | 0,21 | 4,80 | 0,21 |
| 500 | 5,73 | 0,17 | 7,44 | 0,13 | 6,86 | 0,15 |
| 600 | 5,98 | 0,17 | 11,09 | 0,09 | 7,55 | 0,13 |
| 700 | 3,41 | 0,29 | 3,50 | 0,29 | 2,71 | 0,37 |
| 800 | 2,78 | 0,36 | 2,63 | 0,38 | 2,45 | 0,41 |
| 900 | 3,04 | 0,33 | 4,65 | 0,22 | 2,62 | 0,38 |
| 1000 | 5,54 | 0,18 | 5,34 | 0,19 | 4,97 | 0,20 |
| 1100 | 6,66 | 0,15 | 7,34 | 0,14 | 5,47 | 0,18 |
| 1200 | 6,31 | 0,16 | 6,40 | 0,16 | 6,03 | 0,17 |
| 1300 | 5,85 | 0,17 | 7,46 | 0,13 | 6,72 | 0,15 |
| 1400 | 6,57 | 0,15 | 8,41 | 0,12 | 6,68 | 0,15 |
| 1500 | 7.35 | 0.14 | 8.00 | 0.13 | 6.92 | 0.14 |

## User guide

To run the program is straight forward. Compile with javac *java, and run java Main <N>. Where N is the size of the matrixes. The program will then run 7 executions of each of the algorithms and calculate the median time and write it to the console.

## Conclusion

To sum up this task. I have implemented three types of sequential matrix multiplication and attempted to implemented as best as possible as parallel ones and measured the speedup. I believe I did quite well, and my parallel algorithms runs noticeably faster than the sequential ones. Preferably I would like to test many large values, but the algorithm runs for extreme durations with large N. As a general overview, the parallel algorithms are much more efficient then the sequential ones.