## 1. Introduction:

This is my attempt at solving the mandatory assignment number 2 in IN3030, I ran my program on an Intel I7700k with 4 cores and 8 logical processors (I don't exactly know what this is, but by using Runtime.getRuntime().availableProcessors(), it gave me 8 cores, so I presumed that it is running on 8 cores).

## 2. Sequential Matrix:

I did nothing remarkable with the sequential multiplication, just 3 for loops which loops over N, one being the x coordinate, the second being the y coordinate, and the last being k. And for the transposed multiplication, it was the same for loop just with some small changes to be able to work on the transposed matrix.

## 3. Parallel Matrix Multiplication:

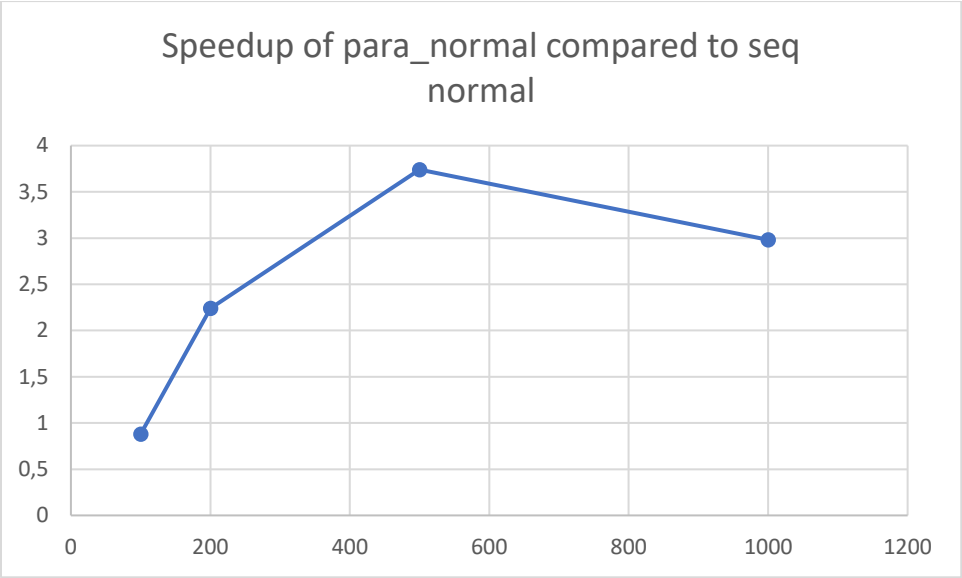
First I check how many cores are available then I divide N by the number of core and assign each thread a vertical slice of the array (I.E thread zero takes M[0][0] – M[24][99], next takes M[25][0] – M[49][99], given a 100x100 matrix) And I do some Modula operations to assure that if M % threads != 0, then we get the corrects bounds for the thread. And then the threads worked sequentially on its own sub matrix, and adds the output to global output, since no two threads are accessing the same cells inside the output matrix.

## 4. Measurements:

Below are the measurements done, the measurements were taken by running the algorithms seven times for each N and then taking the median time value of the runs. The speed up of the parallelization varies for the different sizes of N due to the startup time for the individual threads, for small numbers of N the startup time + solving time would be longer than the sequential solving time, making the parallelization redundant. And we also can see that we only get around a 3 times speedup when we only parallelize the algorithm without transposing the B Matrix, but we get 30 times speedup when we parallelize and transpose. The reason that we get such a huge speedup when we transpose B is because how java stores 2d arrays, with each row being loaded into the cache (with prefetching being utilized for large Ns) so if we don't transpose the B matrix then we have load in a new row for each k in the inner loop. And when we transpose A we have to also load in an extra row for each k in the inner loop, so in total two rows loaded for each iteration of k.

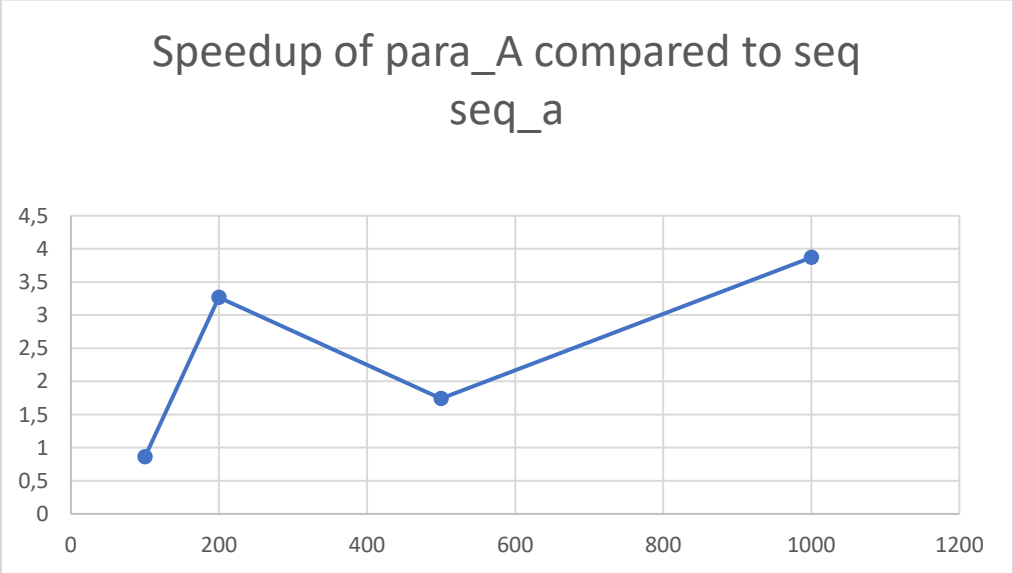
A table showing the time elapsed and speed up of the parallel solution

| N | Seq_normal | Para_normal | Speedup |
|---|---|---|---|
| 100 | 1,02 | 1,16 | 0,87931 |
| 200 | 8,31 | 3,71 | 2,239892 |
| 500 | 228,24 | 61,02 | 3,740413 |
| 1000 | 4606,36 | 1544,52 | 2,982389 |



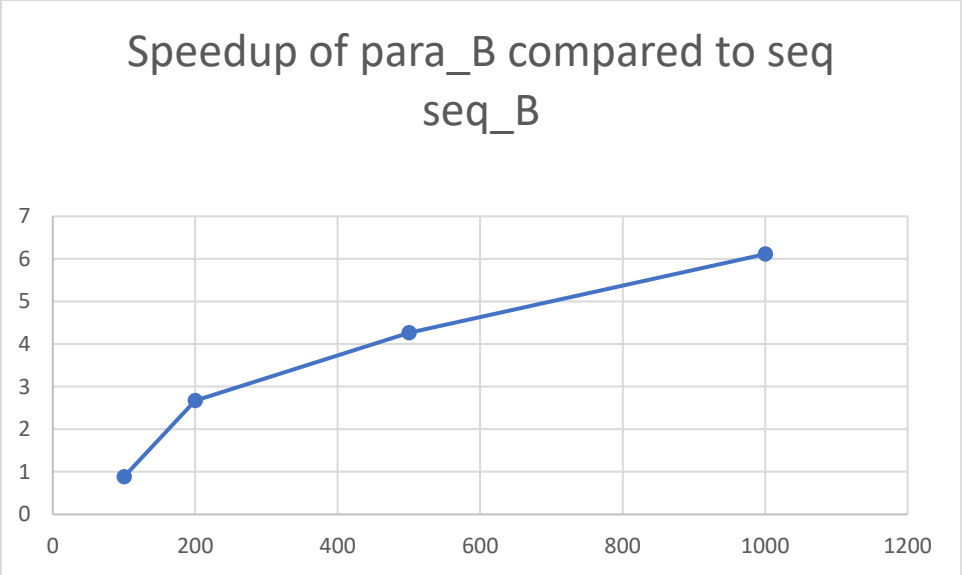

A table showing the time elapsed and speed up of the parallel solution

| N | Seq_A_Trans | Para_A_Trans | SpeedUp |
|---|---|---|---|
| 100 | 1,35 | 1,57 | 0,859873 |
| 200 | 15,39 | 4,71 | 3,267516 |
| 500 | 353,21 | 203,11 | 1,739008 |
| 1000 | 12566,21 | 3247,52 | 3,869479 |



A table showing the time elapsed and speed up of the parallel solution

| N | Seq_B_Trans | Para_B_Trans | Speedup |
|---|---|---|---|
| 100 | 0,84 | 0,95 | 0,884211 |
| 200 | 6,99 | 2,62 | 2,667939 |
| 500 | 110,1 | 25,8 | 4,267442 |
| 1000 | 961,86 | 157,42 | 6,110151 |




All the different runs of the algorithm

| N | Seq_normal | Seq_A_Trans | Seq_B_Trans | Para_normal | Para_A_Trans | Para_B_Trans |
|---|---|---|---|---|---|---|
| 100 | 1,02 | 1,35 | 0,84 | 1,16 | 1,57 | 0,95 |
| 200 | 8,31 | 15,39 | 6,99 | 3,71 | 4,71 | 2,62 |
| 500 | 228,24 | 353,21 | 110,1 | 61,02 | 203,11 | 25,8 |
| 1000 | 4606,36 | 12566,21 | 961,86 | 1544,52 | 3247,52 | 157,42 |

Speedup compared to Sequential Normal solution

| N | Seq_A_Trans | Seq_B_Trans | Para_normal | Para_A_Trans | Para_B_Trans |
|---|---|---|---|---|---|
| 100 | 0,755555556 | 1,214285714 | 0,879310345 | 0,649681529 | 1,073684211 |
| 200 | 0,539961014 | 1,188841202 | 2,239892183 | 1,76433121 | 3,171755725 |
| 500 | 0,64618782 | 2,073024523 | 3,740412979 | 1,12372606 | 8,846511628 |
| 1000 | 0,366567167 | 4,789012954 | 2,982389351 | 1,418423905 | 29,26159319 |


## 5. User guide:

Compile with javac *.java

Run with java Oblig2 [N] [seed], where N is the dimension of the matrix and seed is the seed used for the pseudo random number generation.

## 6. Conclusion:

As we saw from the data I provided, the greatest speed up was due to the more cache friendly solution, rather than the parallelization, so it is important to find the best sequential solution, and then try to parallelize it, than just try and parallelize the first solution you find. And also the larger the N the more impact the transposing seems to have (which makes sense given that transposing is N^2, while the multiplication is N^3).