1. INTRODUCTION

This is a program that has implemented Matrice Multiplication A x B, where A and B are matrices of size NxN.

There are 6 algorithms used in this solution:

1. Sequential; not transposed
2. Sequential; A transposed
3. Sequential; B transposed
4. Parallel; not transposed
5. Parallel; A transposed
6. Parallel; B transposed

Transposing a matrice is the process of “turning” the matrice such that its original values on the rows appear on the columns and vice versa.

The parallel solutions uses multiple threads to solve the same problem, increasing the speedup.

The program uses a global variable LOOPS, which is set to 7. This is the amount of loops the program does before proceeding to the next algorithm. This is used to figure out the median time of each implementation.

Every matrice values are randomly chosen by a random number generator. This seed is provided in the command line interface, so that you can use the same seed multiple times, making sure you always get the desired result.

The program was tested and programmed on a Ubuntu 18.04.1 running Intel Core i5-6200U CPU @ 2.30GHz x 4.

2. SEQUENTIAL MATRIX MULTIPLICATION

The sequential algorithms uses a single thread, and thus performs at a much slower rate. However the sequential solutions are much easier to write, and much easier to read.

The sequential solution is also used as a double check for the parallel algorithms, as it’s most likely programmed correctly.

File: Sequential.java

3. PARALLEL MATRIX MULTIPLICATION

The global variable NUM_CORES is set to 4 by default, as that’s how many cores I have on my CPU, however this number can be changed to any number.

The algorithm is almost just like the sequential one, however every thread starts the loop at ‘i = id’, where ‘id’ is the nth thread that started (from 0). For every iteration, the thread jumps NUM_CORES steps into the next ‘i’ value. This way, no threads reads the same values, making synchronization almost obsolete. There is only one synchronization, which is joining the main thread at the end of execution.

It’s also worth noting that the transposing is done sequentially before the matrices are handled in parallel.

File: Parallell.java

4. MEASUREMENTS

N	100	200	500	1000	SPEEDUP
SEQ_NOT_TRANSPOSED	2	15	325	5686	10.93
SEQ_A_TRANSPOSED	3	27	488	18458	35.50
SEQ_B_TRANSPOSED	8	13	193	1633	3.14
PARA_NOT_TRANSPOSED	23	9	141	4126	7.93
PARA_A_TRANSPOSED	7	13	676	8845	17.01
PARA_B_TRANSPOSED	4	6	65	520	1

Speedup based on PARA_B_TRANSPOSED as baseline, using N=1000

Every solution has been run 7 times, marking the median time taken in milliseconds (ms). As expected, the parallel solutions run a lot faster than the sequential ones.

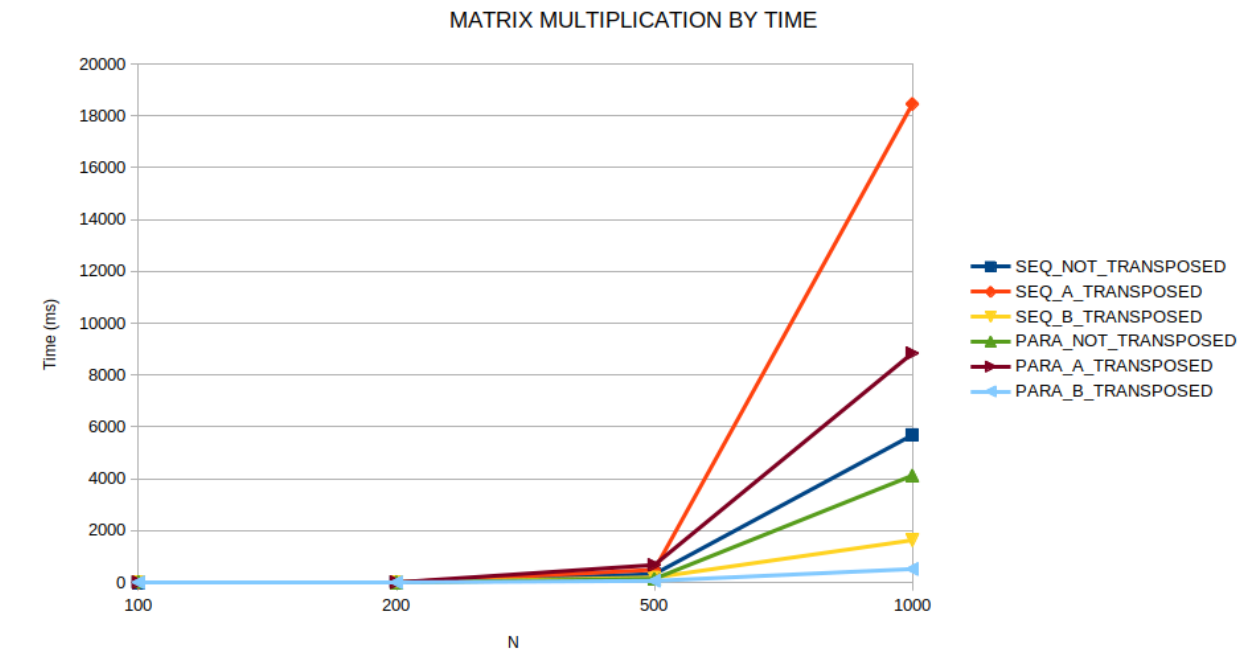
We also notice for low N, the parallel solutions take longer than sequential ones. This is due to the fact that it takes some time to start every thread. For low N, this time is crucial for such a small task.

For large N, the timesave is substantial, where every parallel solution is faster than its corresponding sequential solution.

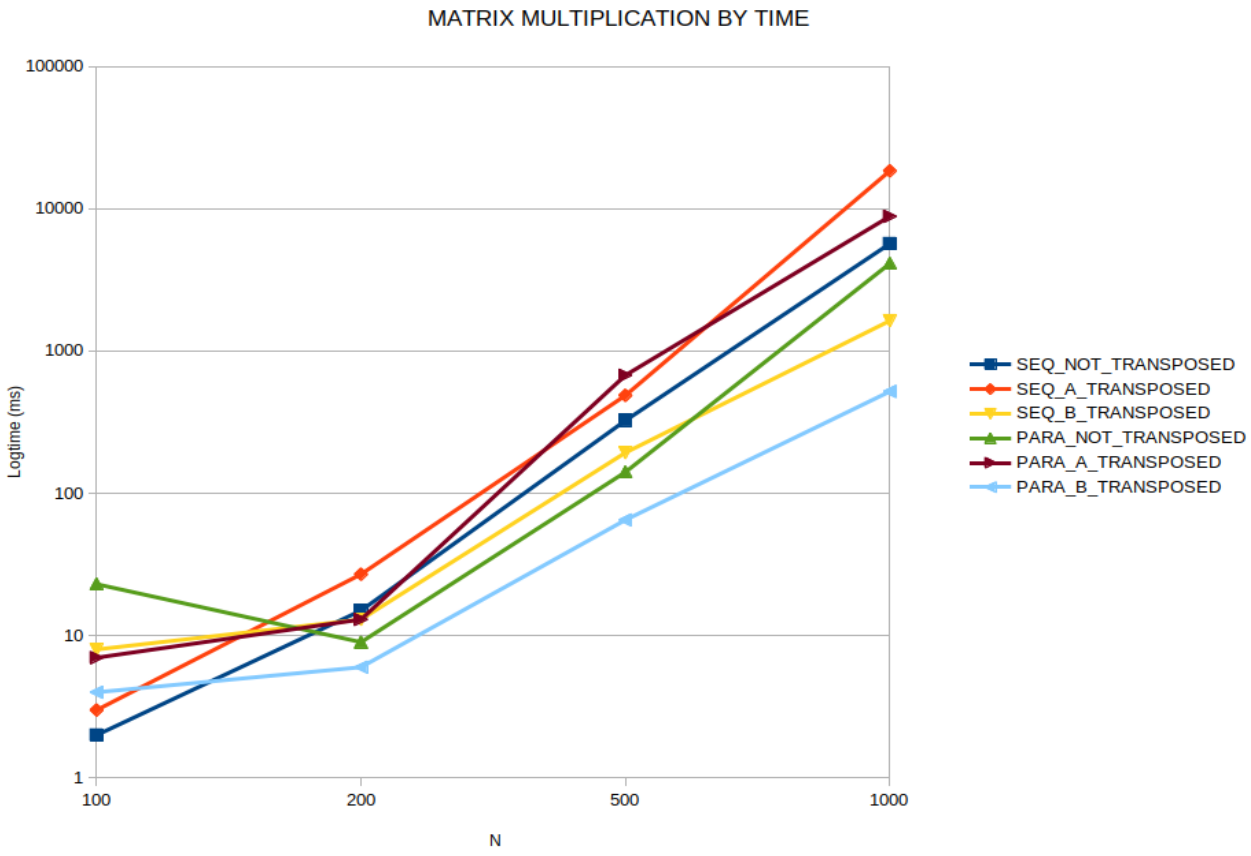
This table also shows the speedup, comparing the fastest algorithm:
Parallel - B transposed, to every other solution. Compared to the sequential one, we get a 3,14x speedup, and compared to Sequential – A transposed we get a 35,5x speedup!

On the next page we have the same values shown in two different graphs. Graph A1 uses a linear Y-axis, and A2 uses a logarithmic Y-axis.

Files: Table.png, Data1.png, Data2.png


MATRIX MULTIPLICATION BY TIME
Time (ms)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
100
200
500
1000
N
SEQ_NOT_TRANSPOSED
SEQ_A_TRANSPOSED
SEQ_B_TRANSPOSED
PARA_NOT_TRANSPOSED
PARA_A_TRANSPOSED
PARA_B_TRANSPOSED

A 1


MATRIX MULTIPLICATION BY TIME
Logtime (ms)
100000
10000
1000
100
10
1
100
200
500
1000
N
SEQ_NOT_TRANSPOSED
SEQ_A_TRANSPOSED
SEQ_B_TRANSPOSED
PARA_NOT_TRANSPOSED
PARA_A_TRANSPOSED
PARA_B_TRANSPOSED

A 2

5. USER GUIDE

Run this program by using the command:
java Oblig2 <N> <SEED>

You can also change the following variables:
NUM_CORES (Default 4)
LOOPS (Default 7)

File: Oblig2.java

6. CONCLUSION

The program works perfectly and gives expected results.

I didn't get the full potential 4x speedup, but this is due to the extra time it takes to start up different threads. Memory accesses are also a crucial factor, as the time it takes to access memory can vary. The transposing is done sequentially, so no timesave there. However, a 3.14x speedup with 4 cores is not bad!