IN4330 Oblig 2 report

Introduction

In this assignment we were tasked to multiply a set of two matrices A and B. These matrices were to be ran in three runs: One where neither matrix is transposed, One where matrix A is transposed and lastly one where matrix B is transposed. These three operations were done both sequentially and using parallelization creating 6 run times results for us to analyze and compare.

Sequential solution

Sequential solution is fairly straightforward. Operations are applied to a given matrix either A,B or None. From there the matrices are then simply multiplied. This is ran 7 times and time measurement results are a median of results from the 7 runs.

Parallelized solution

Parallelization was done by splitting matrices rows, and giving each thread in the computer an almost equally(last chunk might be slightly smaller/larger) divided parts of the whole matrix to work with. Once every worker is done doing matrix multiplication on their chunk of the entire matrix, all workers are simply joined. After joining a sequential solution with the same parameters is generated so that those results from the matrix multiplication can be directly compared to the parallelized solution. The time results are measured using the same technique described in the sequential solution.

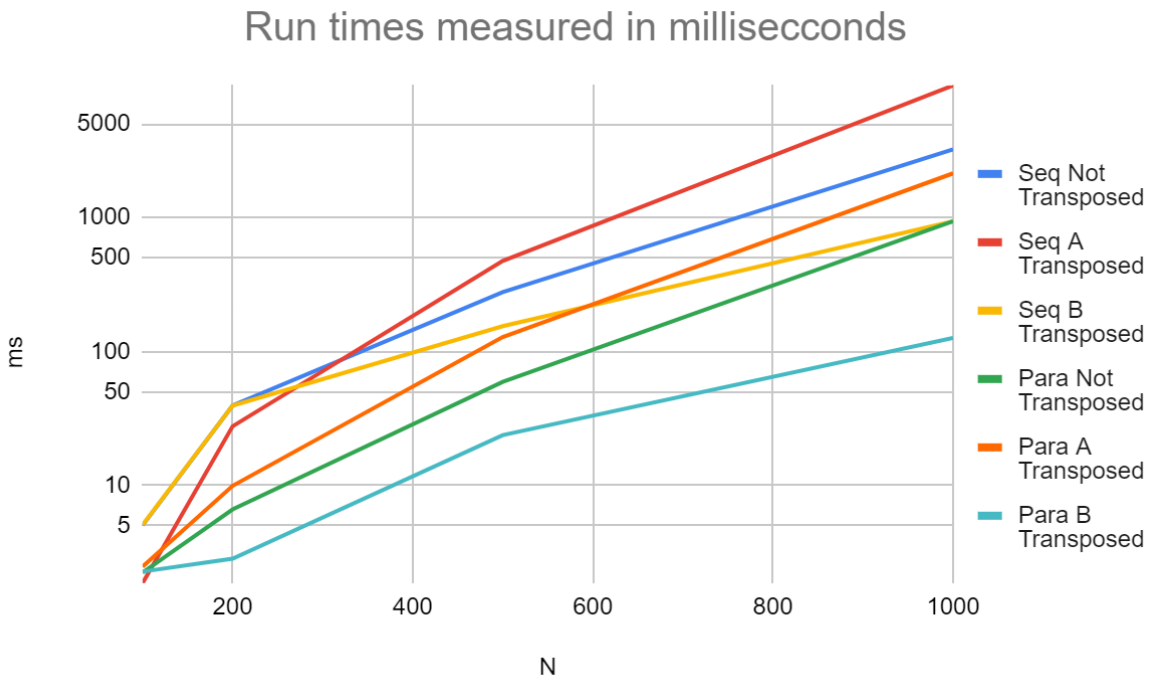
Speed up was achieved in pretty much all parameters and operations, with only really one exception. One outlier in the test results is the speed ups for B Transposed operation for 200. This can be due to many reasons, firstly i suspect that 200 is a very fine tuned number where dividing the work creates efficiency with no overhead from other threads lagging behind. In larger matrices some threads might be lagging behind creating speed up but not one as efficient as in size 200. That being said, larger matrices still see very effective speed up, showing that the solution very clearly resolves the problem of many cache misses that the sequential solution shows with its long run times.

First set of test results

Computing Specification used for first set of testing

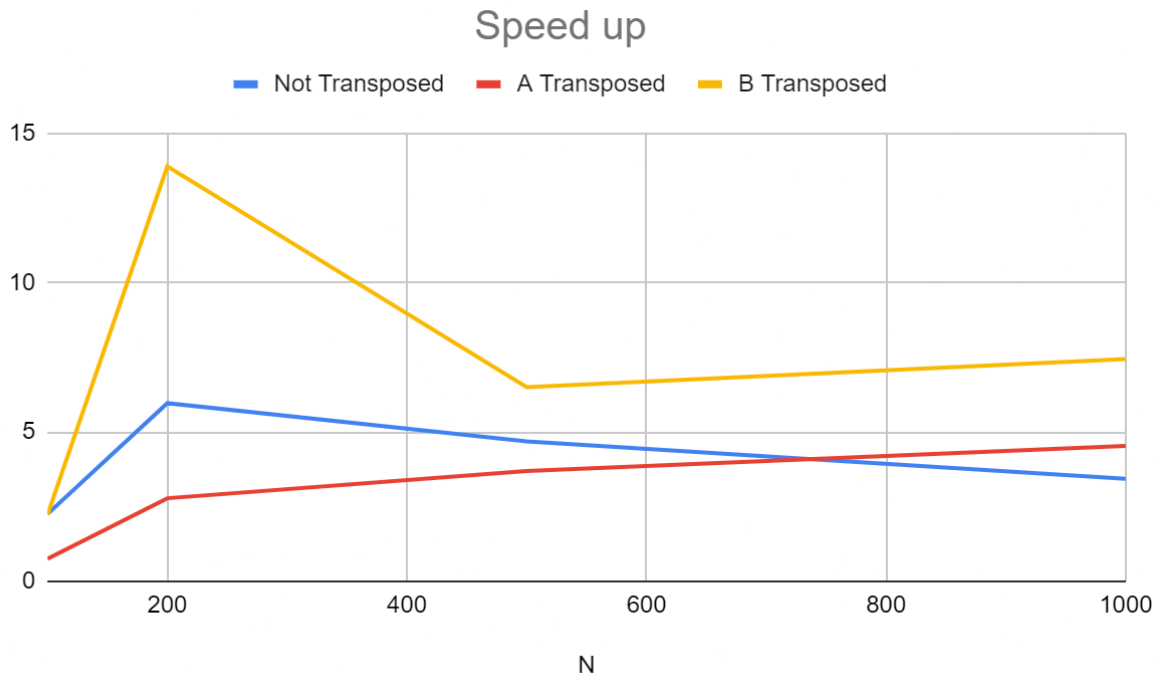
Processor is an AMD Ryzen 5 5625U, 2,30 GHZ with 6 cores and 12 logical processors

Tests measured in milliseconds						
N	Sequential Not Transposed	Seq A Transposed	Seq B Transposed	Para Not Transposed	Para A Transposed	Para B Transposed
100	5,0269	1,8601	5,0665	2,2283	2,4704	2,2701
200	39,4407	27,6207	39,3565	6,6124	9,9274	2,833
500	277,4721	473,9443	154,2993	59,2158	128,2015	23,7307
1000	3225,228	9687,5781	937,1583	939,0549	2137,7327	125,9043



Speed up table Seq/Para			
N	Not Transposed	A Transposed	B Transposed
100	2,256	0,753	2,232
200	5,965	2,782	13,892
500	4,686	3,697	6,502
1000	3,435	4,532	7,443

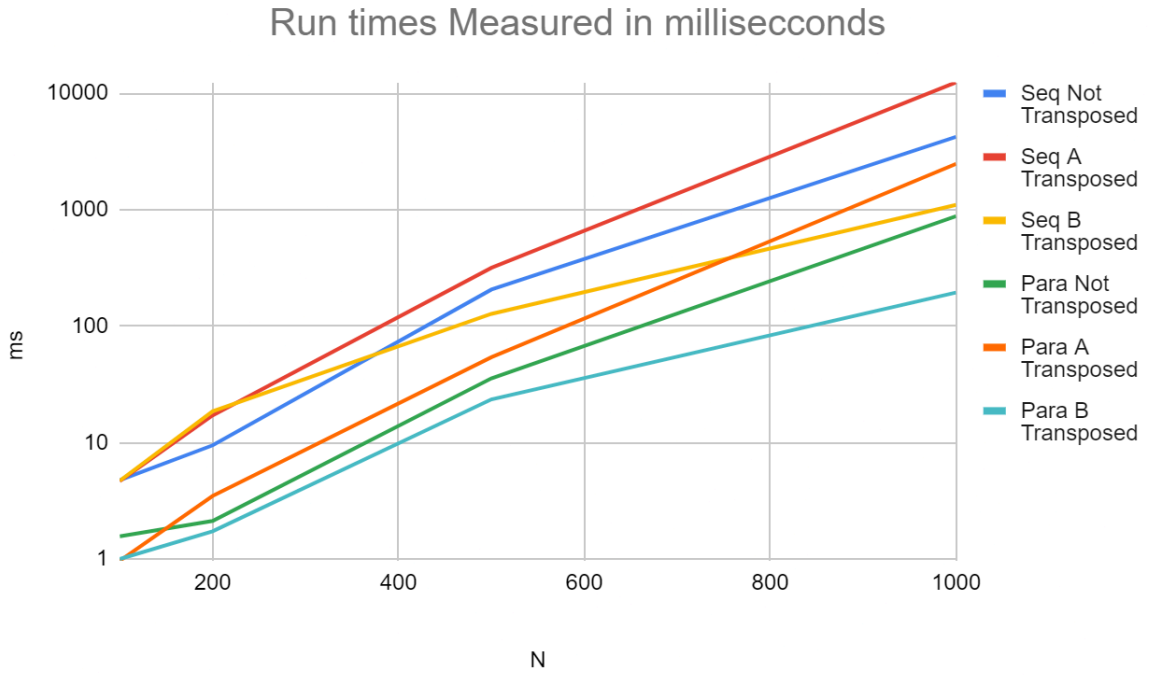
Results were as expected with speed up being achieved and a general thread of B transposed being superior in terms of effective run times.



Computing Specification used for second set of testing

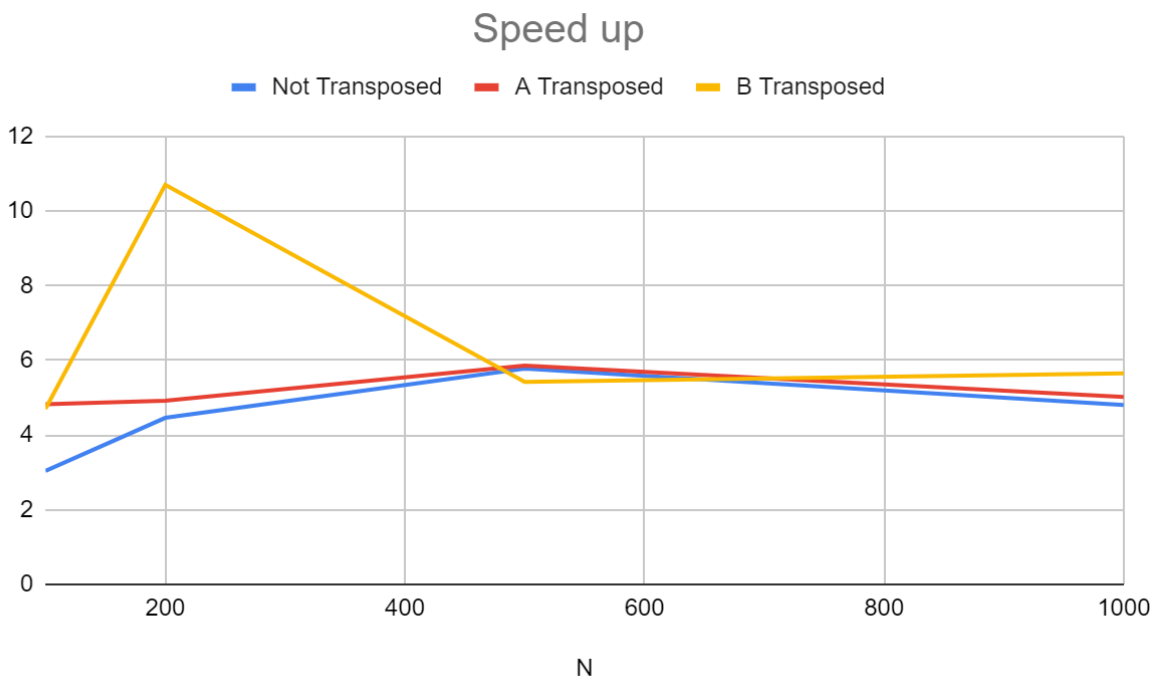
Processor is an Intel(R) Core(TM) i5-9400F , 2,90 GHZ with 6 cores and 6 logical processors

Tests measured in milliseconds						
N	Sequential Not Transposed	Seq A Transposed	Seq B Transposed	Para Not Transposed	Para A Transposed	Para B Transposed
100	4,779	4,6742	4,7152	1,5743	0,9686	1,0032
200	9,5045	17,1814	18,6065	2,1315	3,493	1,7388
500	206,5647	317,6539	127,7715	35,7456	54,2296	23,5702
1000	4229,618	12433,559	1101,6442	880,6654	2476,1677	194,9271



Speed up table Seq/Para			
N	Not Transposed	A Transposed	B Transposed
100	3,036	4,826	4,700
200	4,459	4,919	10,701
500	5,779	5,858	5,421
1000	4,803	5,021	5,652

Result trends are pretty much the same as with the first set of tests, except the machine used for the second set of testing does not seem to create such extreme speed ups as the first one did.



Instruction

The program is runned through the Oblig2.java file. The file contains a try catch with instructions on how the program should be runned. Currently the solution runs all six operations one after each other. With simple tweaking a choice in operation can be used if you don't want to run all of the operations at once.

```
try {
    seed = Integer.parseInt(args[0]);
    n = Integer.parseInt(args[1]);
    // operation as an argument
    //op = args[2];
   // operation = Oblig2Precode.Mode.valueOf(op.toUpperCase());
   } catch (Exception e) {
        System.out.println(x:"Usage: Oblig2.java <seed> <n> ");
        System.exit(status:0);
   }
```

Conclusion

Transposing matrices in order to avoid cache missing increased our performance as predicted. By adding parallelization we also see considerable speed up with the proper solution. With both paralyzation and transposing of matrix B in play speed ups and run times were excellent, with results being vastly superior to the other 5 configurations.