IN3030 Oblig 2 - Report

1. Introduction

In this report I will explain my solution to the sequential and parallel version of Matrix Multiplication. Then I will discuss how I did the implementation, in addition to measurements, including tables and graphs of runtime and speedup for matrix sizes of 100x100, 200x200, 500x500 and 1000x1000.

2. User guide

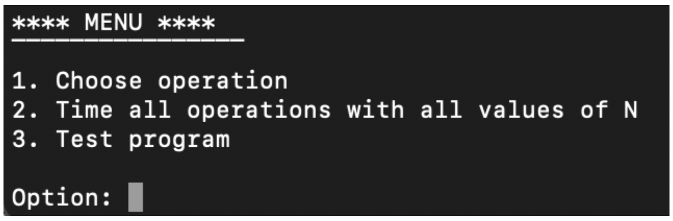
Compilation:

- javac *.java

Run program

- java Main <n>

The program will then present a menu:

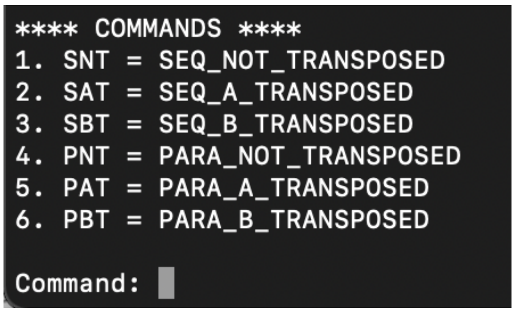


```
**** MENU ****
---------------

1. Choose operation
2. Time all operations with all values of N
3. Test program

Option:
```

Option 1:



```
**** COMMANDS ****
1. SNT = SEQ_NOT_TRANSPOSED
2. SAT = SEQ_A_TRANSPOSED
3. SBT = SEQ_B_TRANSPOSED
4. PNT = PARA_NOT_TRANSPOSED
5. PAT = PARA_A_TRANSPOSED
6. PBT = PARA_B_TRANSPOSED

Command:
```

Let the user decide which operation to run. Uses input value of N to decide the size of the matrixes.

Option 2:

- Time all operations with all values of N, including 100x100, 200x200, 500x500, 1000x1000

Option 3:

- Run program tests to check similar results from sequential and parallel versions.

3. Sequential Matrix Multiplication

The sequential matrix multiplication A X B is implemented in the class SeqMatrixMulti which contains three different version of the multiplication:

- Method no_transpose() using a classic algorithm, multiplying each row in matrix A with each column in matrix B.
- Method a_transpose() where A is transposed. Uses the method transpose() for doing the transpose of A before starting the matrix multiplication. Multiply each column in matrix A with each column in matrix B.
- Method b_transpose() where B Is transposed. Uses the method transpose() for doing the transpose of B before starting the matrix multiplication. Multiply each row in matrix A with each row in matrix B.

The methods are called by the function getResultMatrix() in Main which executes the chosen operation. The sequential operations available are SEQ_NOT_TRANSPOSED, SEQ_A_TRANSPOSED and SEQ_B_TRANSPOSED.

4. Parallel Matrix Multiplication

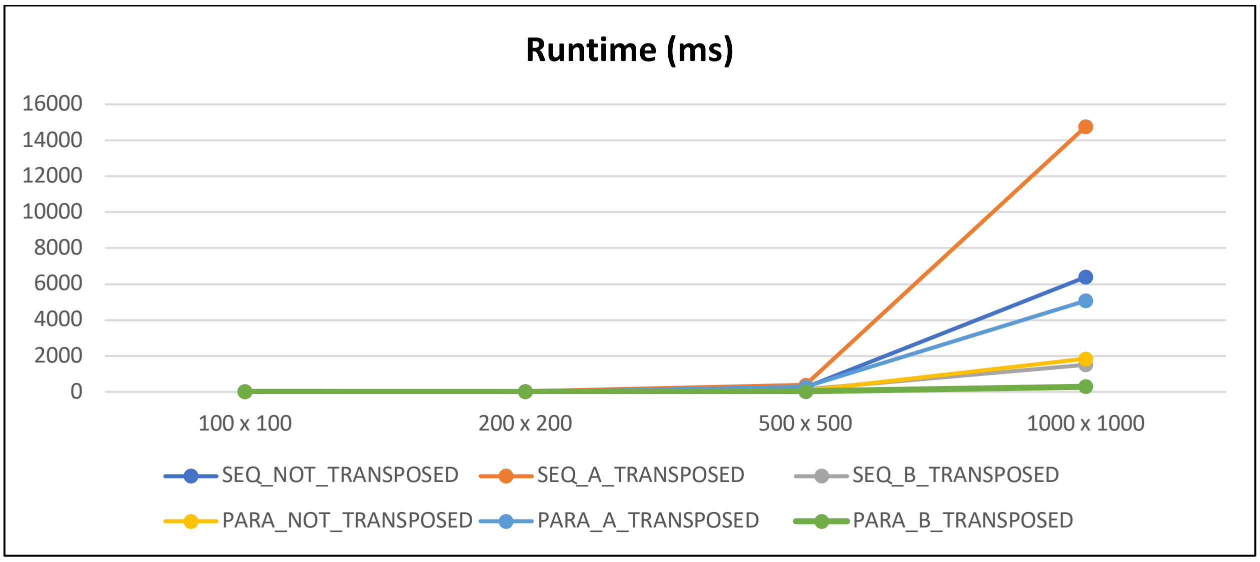
The parallel matrix multiplication is done in class ParaMatrixMulti. The class contains an inner Worker class with a run() function which calls the chosen sequential methods for multiplication. The parallel multiplication is done by the multiplyMatrix() method which create x number of threads equal to number of cores in the CPU. The parallelization is done by dividing the task of each thread into segments, by taking a.length / cores. The segments will then be used to give each thread a start and end index. A CyclicBarrier is then used for synchronization, allowing the threads to wait for each other after they are done.

The multiplyMatrix() method is called by the function getResultMatrix() in Main which executes the chosen operation. The parallel operations available are PARA_NOT_TRANSPOSED, PARA_A_TRANSPOSED and PARA_B_TRANSPOSED.

5. Measurements

Runtime (ms)

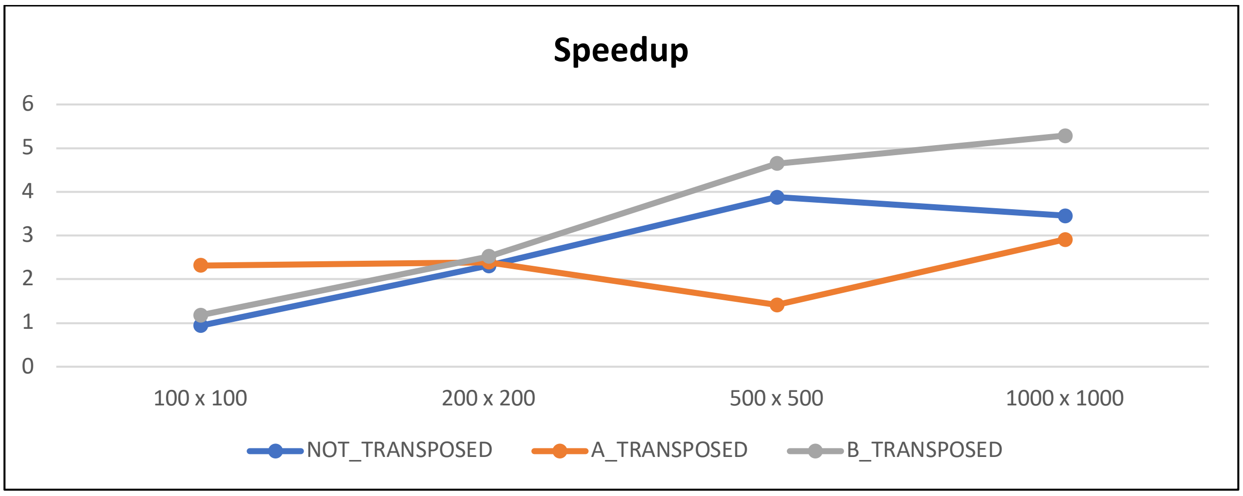
	SEQ_NOT_ TRANSPOSED	SEQ_A_ TRANSPOSED	SEQ_B_ TRANSPOSED	PARA_NOT_ TRANSPOSED	PARA_A_ TRANSPOSED	PARA_B_ TRANSPOSED
100 x 100	1.22	2.79	1.19	1.30	1.20	1.01
200 x 200	10.46	18.65	8.96	4.52	7.80	3.55
500 x 500	232.12	377.41	138.26	59.80	266.68	29.74
1000 x 1000	6390.50	14765.40	1505.34	1849.93	5070.58	284.67



The diagram shows runtime for all operations for different size of n. The results show a big difference in runtime when n is large, especially for the matrix size of 1000 x 1000.

Speedup

	NOT_TRANSPOSED	A_TRANSPOSED	B_TRANSPOSED
100 x 100	0,93	2,31	1,17
200 x 200	2,31	2,39	2,52
500 x 500	3,88	1,41	4,64
1000 x 1000	3,45	2,91	5,28



The diagram shows speedup (sequential runtime / parallel runtime) for the different operations. Based on the results we can se A_TRANSPOSED have the best speedup for n = 100 but the worst speedup for n = 1000. B_TRANSPOSED and NON_TRANSPOSED have on the other hand bad speedup for n = 100, but seems to be way better then A_TRANSPOSED for n > 200. B_TRANSPOSED have by far the best speedup for n = 1000.

6. Conclusion

From the results we can see that parallelization seems to have a big impact on the matrix multiplication. The reason for low speedup for n = 100 may be related to the time it takes for the threads to start up, which results in quite similar runtime for the sequential and parallel versions. For larger values of n we can see how parallelization makes the multiplication way more efficient. We can also see how the transpose of one of the matrixes either worsens or improves the speedup. I think the reason for that could be related to the number of cache misses which occurs when accessing the values in a particular order. It shows that simple techniques of changing the order of access to memory can have a substantial change in the performance of the problem we want to solve, highlighting the importance of writing cache friendly code. In summary, the combination of correct transpose and parallelization can make the matrix multiplication a lot more efficient than using a sequential algorithm.