IN3030 – oblig 2

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Daniel Sharifi – danish

Contents

1. Introduction 3
2. Sequential Matrix Multiplication 3
3. Parallel Matrix Multiplication 4
4. Measurements 5
User guide 7
Conclusion 8

1. Introduction

The interface for my solution is all within the abstract class Multiplier. It has a method multiply(), which takes two matrices and an operation mode as an enum from Oblig2Precode.Mode.

2. Sequential Matrix Multiplication

The logic for my sequential solution is all within Multiplier. And will execute one of the following code, depending on the input mode, before returning output matric c.

Not transposed:

```
    for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
```

Transpose B:

```
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[i][k] * b[j][k];
                }
            }
        }
```

Transpose A:

```
       for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[k][i] * b[k][j];
                }
            }
        }
```

3. Parallel Matrix Multiplication

My solution has only parallelized the multiplication of the matrices. In hindsight I've realized it would be possible to parallelize the transposition as well.

The sequential solution is parallelized by partitioning matrix A for each thread with a starting and end index. The logic for this partitioning is offered by ParaMultpiplier, which has a method multiply() that will initialize the threads and run them.

Depending on the mode each thread does following computation. The threads will only synchronize once using a cyclic barrier after all the threads are done with the computation.

PARA_NOT_TRANSPOSED

```
        for (int i = aStart; i < aEnd; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
```

PARA_B_TRANSPOSED

```
        for (int i = aStart; i < aEnd; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[i][k] * b[j][k];
                }
            }
        }
```

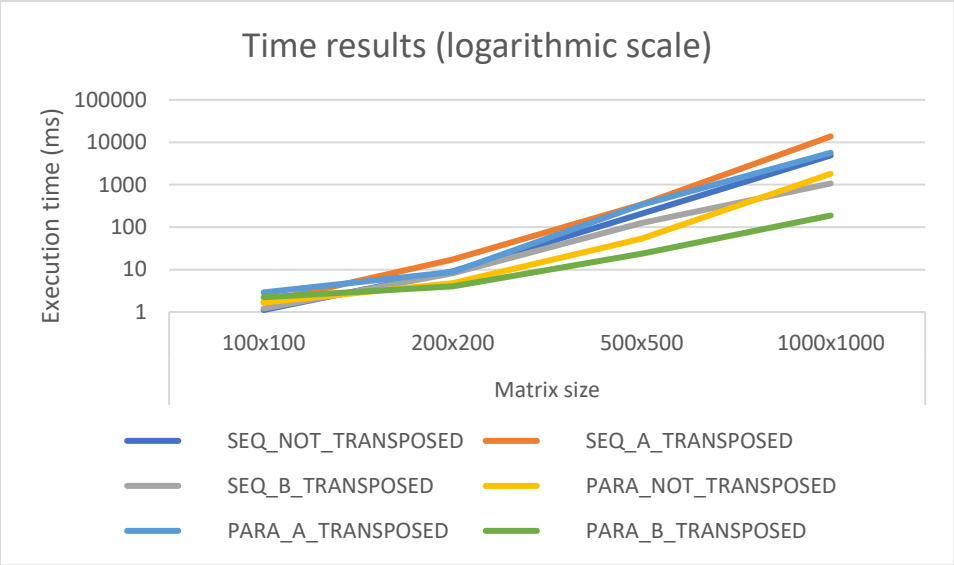
PARA_B_TRANSPOSED

```
        for (int i = aStart; i < aEnd; i++) {
            for (int j = 0; j < a.length; j++) {
                for (int k = 0; k < a.length; k++) {
                    c[i][j] += a[k][i] * b[k][j];
                }
            }
        }
```

4. Measurements

Execution times in ms. The data is plotted with logarithmic scale in the chart, because of the huge increase in execution time as n grows.

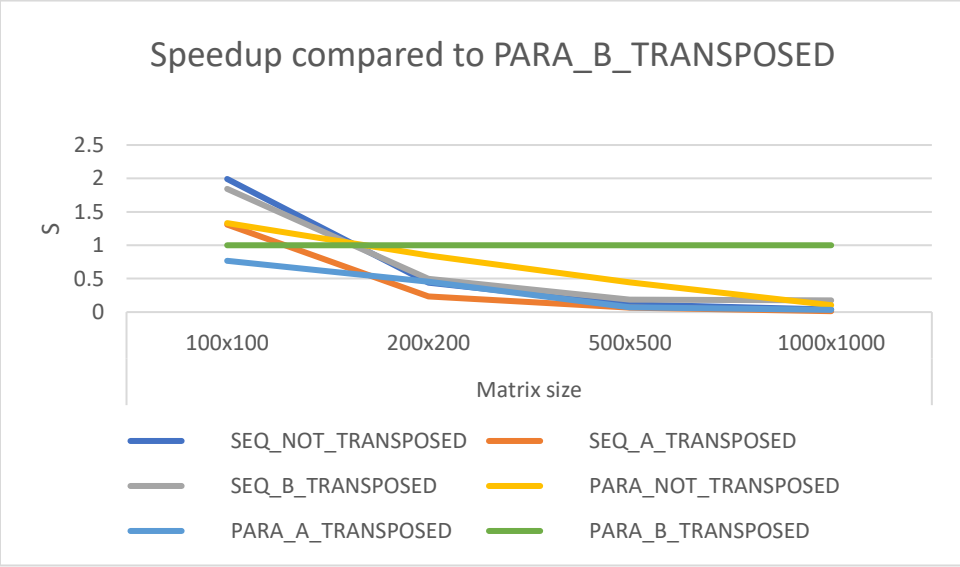
	Matrix size			
	100x100	200x200	500x500	1000x1000
SEQ_NOT_TRANSPOSED	1.1149	9.1391	206.8472	4894.72
SEQ_A_TRANSPOSED	1.6963	17.2919	342.5943	13744.7921
SEQ_B_TRANSPOSED	1.2049	8.1195	125.4342	1071.0958
PARA_NOT_TRANSPOSED	1.6679	4.7577	53.0847	1810.9159
PARA_A_TRANSPOSED	2.8952	8.8516	336.6347	5662.0554
PARA_B_TRANSPOSED	2.2211	4.0312	23.6286	187.7181



From the raw results PARA_B_TRANSPOSED is clearly the fastest as N grows. This is because of the parallelization and the fact that B is transposed minimizing cache misses. Thus we can plot the speed up (slow down in this case) of all the other modes using PARA_B_TRANSPOSED as base.

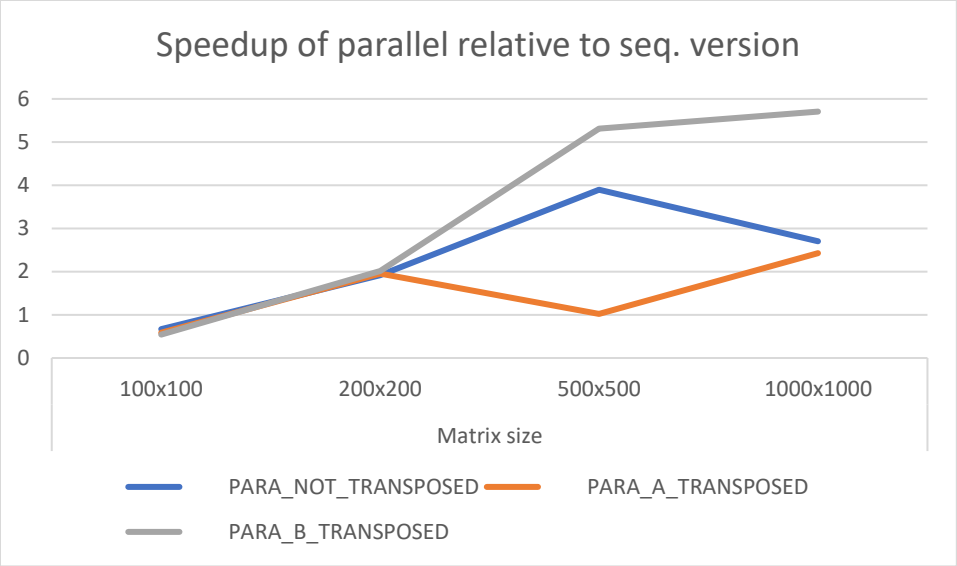
Speed up (slow down)

	Matrix size			
	100x100	200x200	500x500	1000x1000
SEQ_NOT_TRANSPOSED	1.99219661	0.44109376	0.114232148	0.038351142
SEQ_A_TRANSPOSED	1.309379237	0.23312649	0.068969624	0.013657398
SEQ_B_TRANSPOSED	1.843389493	0.49648377	0.188374462	0.175257993
PARA_NOT_TRANSPOSED	1.331674561	0.84730017	0.445111303	0.103659204
PARA_A_TRANSPOSED	0.767166344	0.45542049	0.070190625	0.033153702
PARA_B_TRANSPOSED	1	1	1	1



Lastly the chart and table for the speedup of parallel versions relatively to their sequential versions.

	Matrix size			
	100x100	200x200	500x500	1000x1000
PARA_NOT_TRANSPOSED	0.66844535	1.92090716	3.89655023	2.702897468
PARA_A_TRANSPOSED	0.585900801	1.95353382	1.017703463	2.427526954
PARA_B_TRANSPOSED	0.542478952	2.01416452	5.308575201	5.705873861



This shows that parallelization does not give any speedup until matrix size exceeds 200. This is because of the time needed to initialize the threads and the synchronization, while the work load being very small.

User guide

The main program is in Main.java. After compiling to use the program with following arguments to get execution time:

- java Main [mode] [dimensions of matrices]

example:

PARA_A_TRANSPOSED for a 200x200 matrix.

- java main 4 200

Modes:

Mode:	Code:
SEQ_NOT_TRANSPOSED	0
SEQ_A_TRANSPOSED	1
SEQ_B_TRANSPOSED	2
PARA_NOT_TRANSPOSED	3
PARA_A_TRANSPOSED	4
PARA_B_TRANSPOSED	5

Conclusion

The conclusion from the analysis of the runtimes is following for computing A x B = C.

- Transforming A will increase execution time (cache misses)
- Transforming B will reduce execution time (minimizing cache misses)
- Parallelization gives huge benefits for the execution time if the dimensions are greater than 100x100.