IN3030 - Obligatorisk Innlevering 2

Aleksander Rosseland Kvåle

February 28, 2023

1 Introduction

In this experiment we are working with matrix multiplication in the Java Enviornment. The task at hand is to parallelize matrix multiplication on a computer to utilize all the cores in the machine for multiplication of large matrices. All algorithms were run using an Intel Core i5-1038NG7 CPU @ 2GHz running on MAC OS Ventura 13.2.1. The CPU has L1 cache of 80KB, L2 cache of 512KB and L3 Cache of 6MB(shared between cores). This is vital to our operation on the matricies and will be the baseline for some of our conclusions further down in this report.

2 Sequential Matrix Multiplication

To create a baseline for our program we use the standard sequential algorithm for multiplying matrices within a program. The algorithm is just a double nested for loop which multiplies all the elements in $\mathcal{O}(n^3)$ runtime. As we are heavily focused on the Cache architecture of the processor we will also transpose both matrices and do two more runs. One with matrix A transposed, and one with matrix B transposed. Due to how cache misses work we should be able to have the program pre-load the next values by transposing matrix B and thus doing a row by row multiplication, instead of row by column. The result of this is a significant reduction in runtime. We use $\mathcal{O}(n^2)$ runtime to transpose the matrices, but the lack of cache miss weighs this up significantly on both large and small values of n.

By also adding a runtime with A transposed we can further emphasize what happens as we then do a column by column multiplication, effectively getting a cache miss on every pull of a value.

3 Parallel Matrix Multiplication

The implementation for the parallel version of this matrix multiplication is quite simple. We generate the amount of workers that corresponds to the machines total processors. We pull the machines logical processors so that we can use hyperthreading if the CPUs architecture supports it. In this process we also divide the workload on the matrices into equal parts, and each thread gets its own little section to work on.

To synchronize the threads we then use the method join and thus shut down all the threads once they all have completed their calculations. The method for the individual calculations themselves are identical to the sequential algorithm.

4 Measurements

Speedup	Seq a trans	Seq b trans	Para	Para a trans	Para b trans
100x100	0,85	1,35	0,88	1,48	1,56
200x200	0,60	1,08	2,53	2,02	4,80
500x500	0,70	1,66	4,41	1,79	10,95
1000x1000	0,18	1,71	1,44	0,48	9,92

Figure 1: Speedup of all algorithms

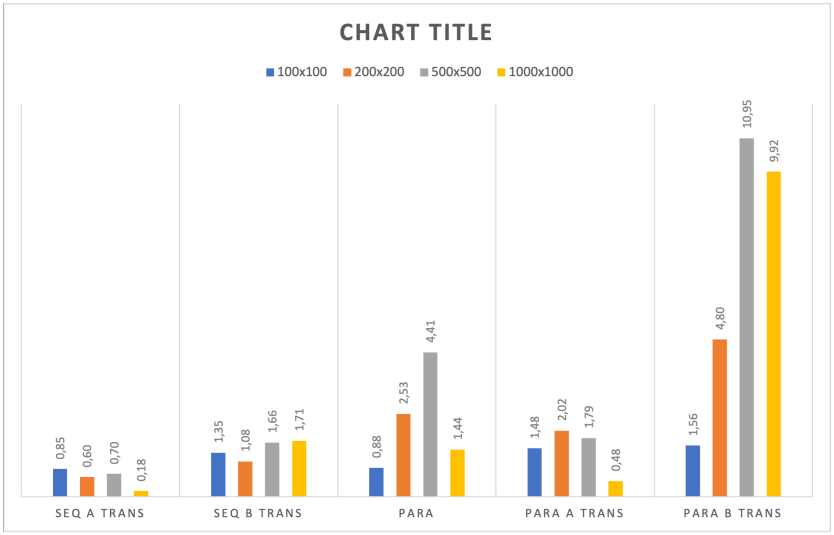

Figure 2: Speedup values charted

We can see from Figure 1 that we struggle to get any significant speedup for the smallest matrices of 100x100. There is a small amount of timings where b is transposed, and this is expected due to the cache miss explained in the Sequential algorithm section. The same pattern can be seen for every single size of matrices. Where matrix b is transposed we always get the largest speedup, for 500x500 this is almost at 11 speedup from the original sequential run. This is a very accurate representation of what we presume to get. On average we should expect 4-6 speedup comparing the regular sequential algorithm to the one where b is transposed. If we add the expected gain from multithreading of a factor of 7 on my 8 logical core machine then we are somewhere in the ballpark of 11-13 in speedup.

Furthermore we see a dip in performance gain when going from 500x500 to 1000x1000 matrices. It is not a significant difference, but for the parallel algorithm we see a speedup of only 1.44 for this size of matrices. This could be directly linked to the size of the cache on my machine that I introduced in the introduction section of this report. As we are getting more and more values the sizes of the L1 and L2 cache do not suffice and we pull more data from the slower L3 cache, or worst case the RAM. This then introduces cache misses that hit our runtime harder than those in the lower level of caches. This is further backed up by our parallel algorithm running with b transposed not taking as much of a performance hit when running on the 1000x1000 matrices. Remember from before that we here multiply row by row and thus preload cache lines full of data. This method is not that prone to cache miss and thus the performance is also upheld.

We can see that transposing A performs poorly across the board, which is to be expected as we here multiply column by column.

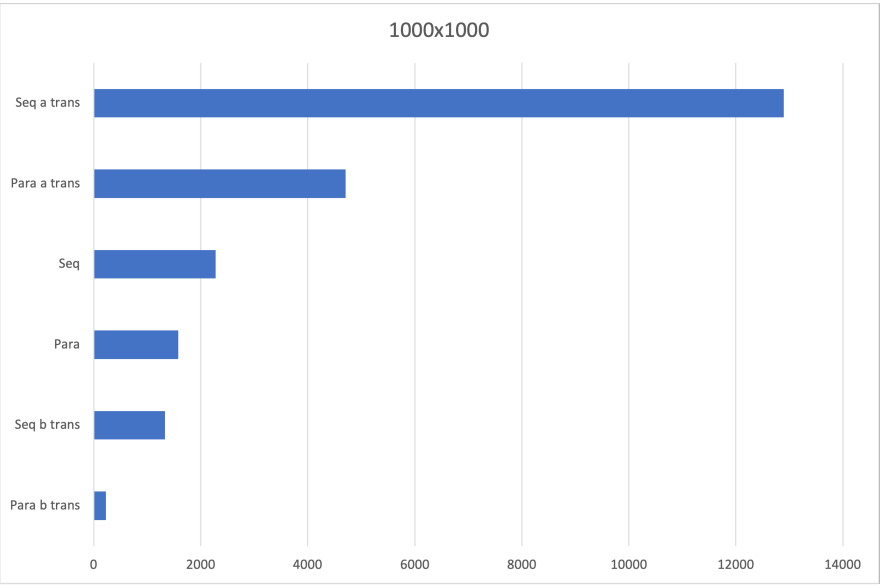

Figure 3: Comparison of time used in milliseconds

5 User guide

Running the program is pretty simple. Unzip the compressed files locally on your machine, then run the command § javac *.java(might be different depending on how you got Java setup on your machine) in a terminal. From here run § java Main n seed. Where n is the size of the matrices you wish to multiply, and seed being the seed used to generate the matrices.

6 Conclusion

To conclude we refer to Figure 3 from section 4 of this report. Here we can see the timings used comparing all 6 methods of matrix multiplication. With the parallel implementation using a transposed version of matrix b coming up miles ahead. The program itself transposes the matrix before doing the timings, so the actual value comparing the methods directly to eachother might be a little bit off. However, for such a large multiplication the time it takes to transpose this matrix is just a fraction of the time spent in total. Furthermore we see that the sequential method with b transposed actually beats out my parallel implementation. Proving furthermore that cache-misses can have a large impact on the performance of our program, and further showing how beautiful the implementation of matrix multiplication with matrix b transposed is.