IN3030 Oblig 2

Marius Fredrichsen (mafredri)

Introduction

In this report I will compare a sequential and a parallel implementation of matrix multiplication, testing with different sized matrices and how it was done.

Sequential Matrix Multiplication

For this part of the task I just implemented the classical variant of the matrix multiplication algorithm. However I adjusted the order of the three for-loops so it would be more cache friendly, iterating over the same arrays as much as possible.

I use the results of this implementation to be the “answer” so I can later check if the parallel implementation is equal to the “answer”.

Note: One thing I was confused about is are we supposed to make the best solution possible for each algorithm? If we were to do that then I feel like the purpose of including transposing B would be pointless, to make it even faster I would change the order of the for-loops to make it as fast as seq and seqA. For now I just kept it all the same.

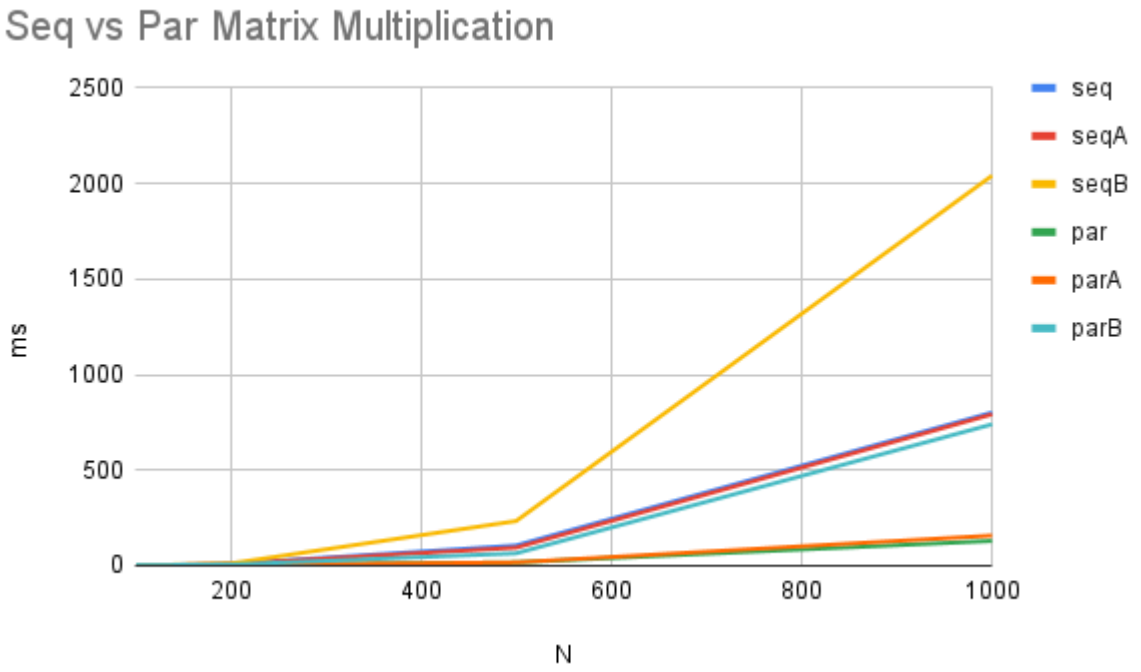
Parallel Matrix Multiplication

The parallel version of my sequential algorithm is pretty much the same. The way I adjusted for the parallel version is just to give the different threads a start and an end index so it would calculate different parts of the matrix. By separating the first matrix and multiplying it with the second matrix I didn’t need any sort of lock or thread safety because I separated the matrix in chunks that don’t overlap. I made sure that every thread was done until I returned the resulting matrix.

Measurements

N	seq	seqA	seqB	par	parA	parB	s	sA	sB
100	1,251421	0,875019	1,293023	2,064551	2,641538	2,837046	0,6061468087	0,3312536106	0,4557638473
200	9,85053	9,05788	11,4908	3,97497	4,48305	5,88565	2,47813	2,02047	1,95235

	5	1	84	9	5	9	5105	0639	2999
500	105,844 008	94,4114	233,470 702	17,3944 71	17,3597 75	64,5138 44	6,08492 2502	5,43851 5188	3,61892 4056
1000	801,664 264	792,353 841	2041,30 919	129,657 422	157,510 772	739,933 958	6,18294 1567	5,03047 3986	2,75877 2142



For the measurements I took the time including the runtime of the algorithm, the transposing of matrices, and creation, starting and joining threads.

By looking at the table/graph above we see a pattern of the parallel implementation being faster. However transposing B gives us almost the same runtime as seq and seqA. I believe this to do something with cache misses since we iterate over different arrays constantly in transposing B. We see the same pattern for seqB being significantly slower than the other sequential implementations. As for transposing A and the classical implementation we get a much faster time in general.

For smaller sized matrices the runtime for the parallel implementations seems to be slower, but I believe this has to do something with the time used for creating and starting threads. The speedup starts below 1 which means the parallel implementation is slower than the sequential one. For bigger matrices it goes above 1.

When transposing the first matrix we see a small increase in speed due to working on the same array more often which causes less cache misses.

User guide – how to run your program

After unpacking the zip file:

- cd <path to unzipped file>
- javac *.java
- java Main

Conclusion

From this experiment we see that the parallel implementation of matrix multiplication has a speedup for bigger sized matrices and that transposing the second matrix is causing a lot of cache misses which takes more time running the algorithm. Transposing the first matrix gives a slight speedup in the implementation because of less cache misses.