Report Oblig 2 IN3030

Haiyuec

Processor:

Intel I7-8550u

Base frequency: 1.80 GHz

Max Turbo frequency: 4.00 GHz

4 cores 8 Threads

How to run the program:

javac *.java

java Main

- This will run all 6 implementations 7 times each on each of the matrix sizes. The program will also use all of the processors on the pc when running the parallel implementations.
- The median running time of the different combinations is shown at the end of the run in ms.

Implementation explanation:

Sequential:

- Normal: Multiply each row in matrix a with every column in matrix b.
- A transposed: First transpose matrix a by “swapping” the indices. Then multiply each column in matrix a with every column in matrix b.
- B transposed: Transpose matrix b. Then multiply each row in matrix a with every row in matrix b.

Parallel:

- Divide the columns or rows of matrix a, depending on the sequential counter part that is being parallelized, between the threads, and carry out the same sequential algorithm in the local area.
- Transposing is parallelized in the same manner. Divide the rows in matrix a to the threads, and execute the sequential algorithm.

Runtime data:

Runtime in ms:

size	Seq normal	Seq a transposed	Seq b transosed	Para normal	Para a transposed	Para b transposed
100x100	3	4	3	3	6	5
200x200	19	41	19	8	15	7
500x500	524	1075	306	133	469	55
1000x1000	9399	29420	2586	3283	7170	344

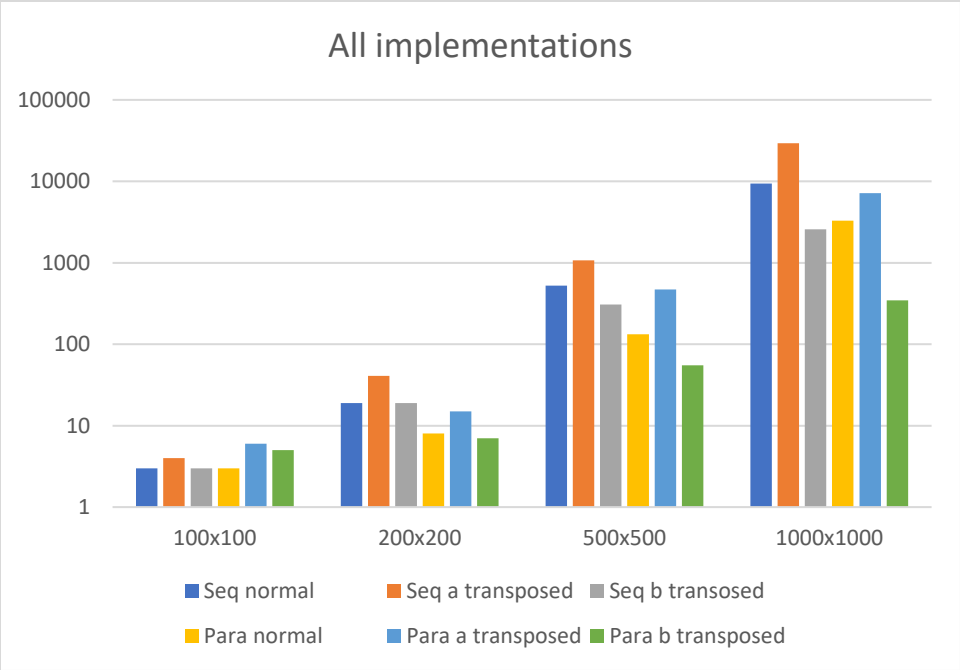
Runtime increase compared to the normal sequential implementation:

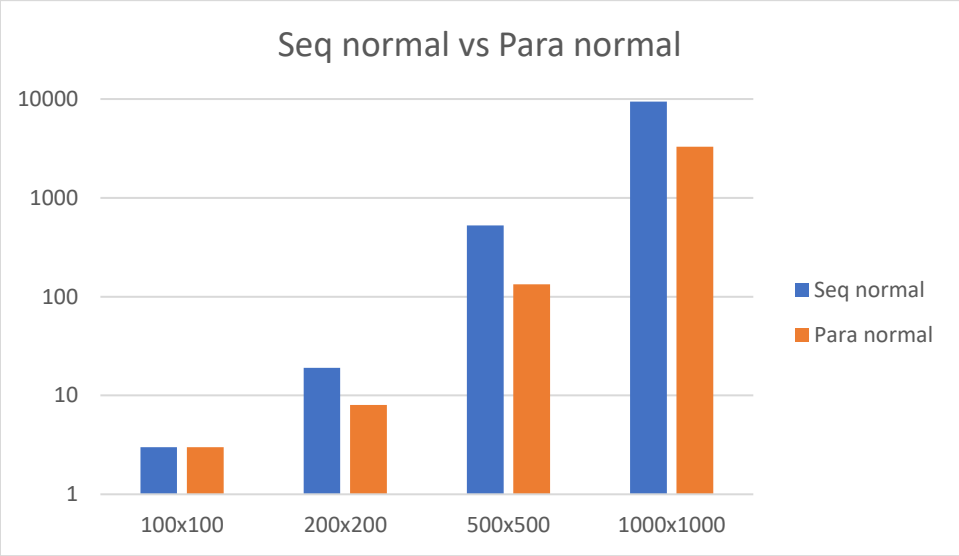
size	Seq normal	Seq a transposed	Seq b transosed	Para normal	Para a transposed	Para b transposed
100x100	0%	-25%	0%	0%	-50%	-40%
200x200	0%	-54%	0%	138%	27%	171%
500x500	0%	-51%	71%	294%	12%	853%
1000x1000	0%	-68%	263%	186%	31%	2632%

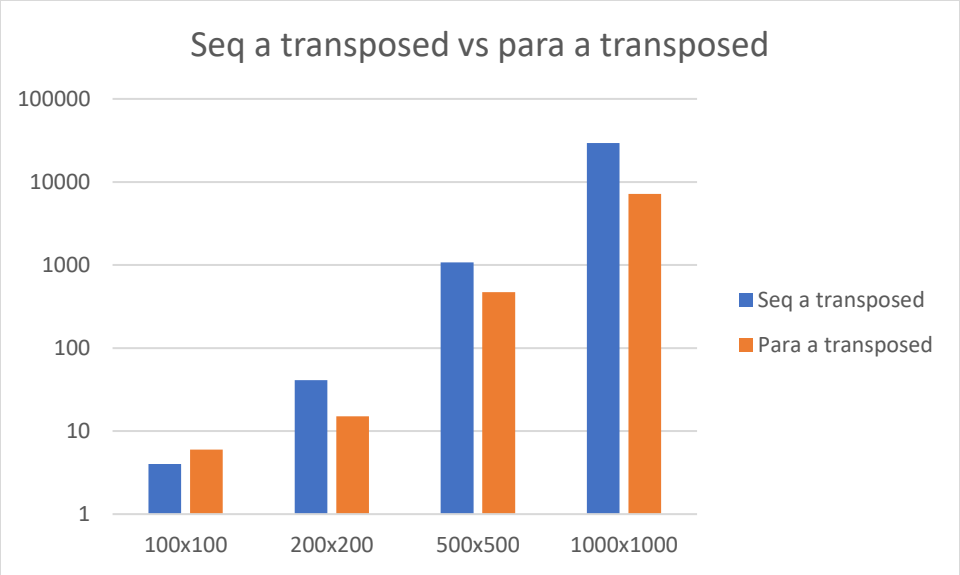
“Apples to apples” comparison:

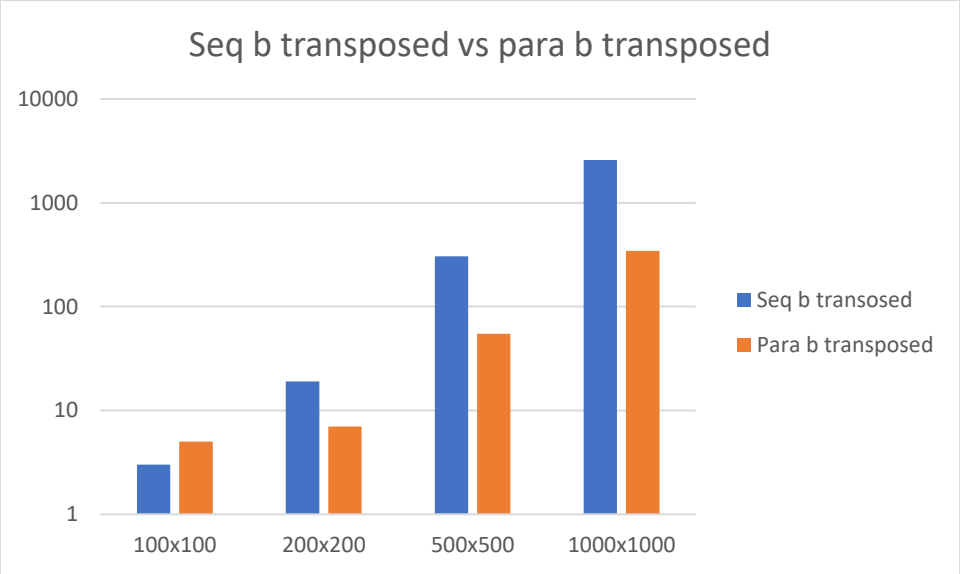
size	Seq normal	Para normal	Seq a transosed	Para a transposed	Seq b transposed	Para b transposed
100x100	0%	0%	0%	-33%	0%	-40%
200x200	0%	138%	0%	173%	0%	171%
500x500	0%	294%	0%	129%	0%	456%
1000x1000	0%	186%	0%	310%	0%	652%

Charts:


All implementations
100000
10000
1000
100
10
1
100x100
200x200
500x500
1000x1000
Seq normal
Seq a transposed
Seq b transosed
Para normal
Para a transposed
Para b transposed


Seq normal vs Para normal
10000
1000
100
10
1
100x100
200x200
500x500
1000x1000
Seq normal
Para normal


Seq a transposed vs para a transposed
100000
10000
1000
100
10
1
100x100
200x200
500x500
1000x1000
Seq a transposed
Para a transposed


Seq b transposed vs para b transposed
10000
1000
100
10
1
100x100
200x200
500x500
1000x1000
Seq b transosed
Para b transposed

Conclusion:

We can see from the apples to apples comparison that the performance difference between the parallel and sequential implementations are quite similar. As you might expect, the parallel version has a bit lower performance for smaller matrixes due to the cost of starting the threads. Because I have also implemented the transposing of the matrix in parallel, for the parallel implementations, this performance hit is higher.

However, the more interesting part is the difference between the performance of the matrix b transposed implementations and matrix a transposed implementations. We can clearly see that the implementations with matrix b transposed are much faster than the implementations with matrix a transposed. This is because by accessing the values row by row, as the implementations with matrix b transposed do, we can minimize the amount of cache miss, and thereby utilize the lightning fast CPU cache. The amount of cache miss is minimized because the data that would be fetched in to cache is fetched in as blocks of data, called cache lines. In our case, this means that when accessing a value in a particular row in a matrix, a certain amount of the values further down in the row would be taken with the value currently accessed, and stored together in the CPU cache. The implementations with matrix a transposed on the other hand, would access column by column, which is very much NOT cache friendly, and most certainly would result in a cache miss every time a value needs to be fetched. It is much slower to access data from RAM than CPU cache, therefore resulting in much lower performance.