Oblig 2 IN3030 – Parallelization of Matrix Multiplication

Introduction

The objective of this report is to present the implementation and performance evaluation of three variants of matrix multiplication: classic algorithm, transposed matrix A and transposed matrix B. Bort sequential and parallel versions of these variants are implemented and assessed for different matrix sizes. The report includes a discussion on parallelization strategies, speedup measurements, and comparisons between different variants.

Sequential Matrix Multiplication

Sequential matrix multiplication involves multiplying two matrices together in a step-by-step manner, single-threaded, where each element of the resulting matrix is calculated. This process iterates through each row and column of the matrices, multiplying the corresponding elements and summing the products to compute each element of the resulting matrix. The sequential approach executes the multiplication in a linear fashion, making it more suitable for small to moderately sized matrices where parallelization may not provide significant performance benefits.

Parallel Matrix Multiplication

For the parallelization, the program is utilizing multithreading to distribute the workload across available processor cores. There are three separate worker classes are implemented for each matrix multiplication variant, each is responsible for computing a segment of the resulting matrix. The matrices are partitioned into equally large segments where each thread handles the computation of a segment in parallel. The segments are calculated by dividing the matrices by available cores and finding a start and end value for each segment that a thread can work on.

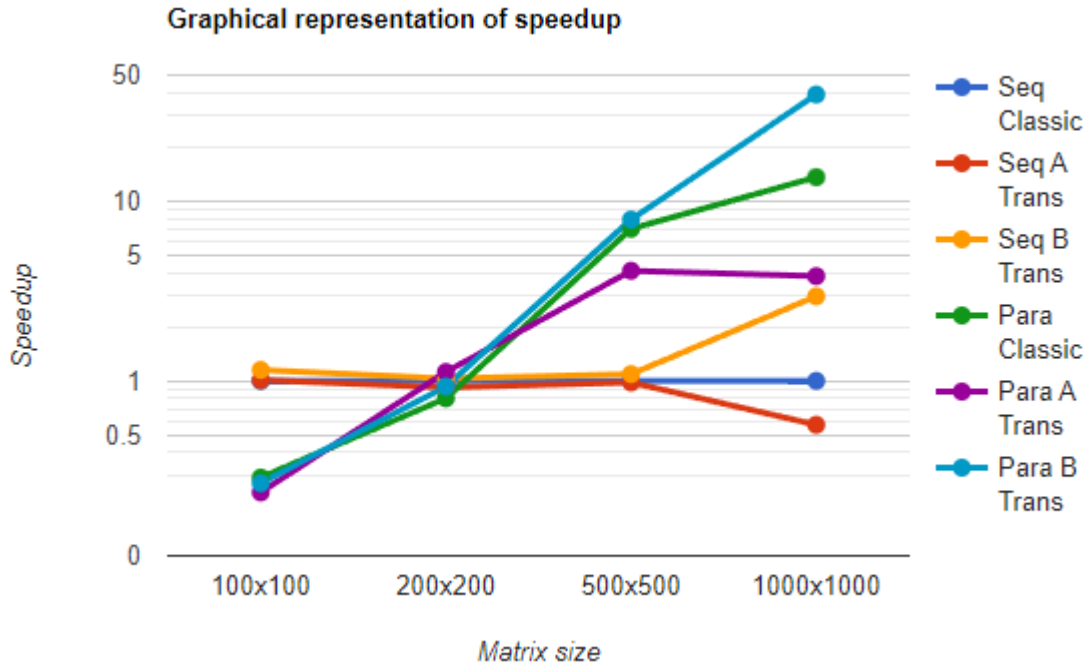
Measurements

The performance of each matrix multiplication variant is evaluated using four different matrix sizes: 100x100, 200x200, 500x500 and 1000x1000. Both sequential and parallel versions are timed, and speedup is calculated.

Table of speedup where Sequential classic is the basis, and everything is compared to that:

	100x100	200x200	500x500	1000x1000
Sequential Classic	1.00	1.00	1.00	1.00
Sequential A Trans	1.01	0.92	0.98	0.57
Sequential B Trans	1.15	1.03	1.09	2.95
Parallel Classic	0.29	0.80	7.00	13.55
Parallel A Trans	0.24	1.12	4.09	3.82
Parallel B Trans	0.27	0.93	7.88	39.02

Graphical representation of speedup:



For all the parallel methods we see that they are slower than the sequel when the matrices are 100x100 and the reason for that is probably that it takes time to create the threads and start them. But in contrary when the matrices are at the largest, i.e. 1000x1000, all the parallel methods beat out the sequential by having greater speedup. The reason is that more work is done at the same time and over large matrix multiplications a parallel solution is the best.

For the individual methods used, we can see that the one where transposing the B matrix is giving a greater speedup. That is pointing back to when we learned about matrix multiplication and cache miss. By transposing matrix B, we will have fewer cache-misses and therefore a greater speedup. We can see the opposite for when we transpose matrix A. This makes it harder for the CPU to prefetch the needed data concluding in a lot of cache-misses. It takes time to travel to the main memory got get the data needed and we can see that in the graph.

User guide

To execute the code, use the following command-line arguments:

- java Oblig2 <Seed for random class> <matrix size>

Here is an example of how I ran my code for matrices 1000x1000:

- java Oblig2 42 1000

Ensure that the precode and the submitted program is in the same folder before compiling and executing. Also, you need the command-prompt to be in the same folder to be able to compile and execute.

Conclusion

In conclusion, the parallelization of matrix multiplication demonstrates significant speedup compared to the sequential approach, especially for larger matrix sizes. The transposing variants also contribute to performance improvement by optimizing memory access patterns. However, the speedup achieved varies with different matrix sizes due to factors such as thread overhead and memory bandwidth limitations.