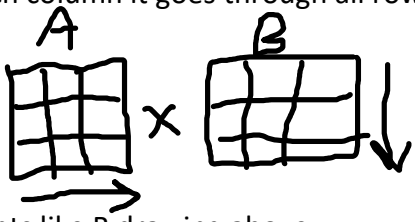
2ND MANDATORY ASSINMENT

Full Name: Pooria Mehregan
Username: pooriam
Machine:

- Macbook Pro 16, 2.6 GHz 6-Core Intel Core i7
- L2 Cache (per Core): 256 KB
- L3 Cache: 12 MB
- Hyper-Threading Technology: Enabled (cores = 12)
- Memory: 16 GB

On my machine, program took approximately 2 minutes to finish, where every algorithm is executed 7 times (and median time is taken because of Java optimizations).

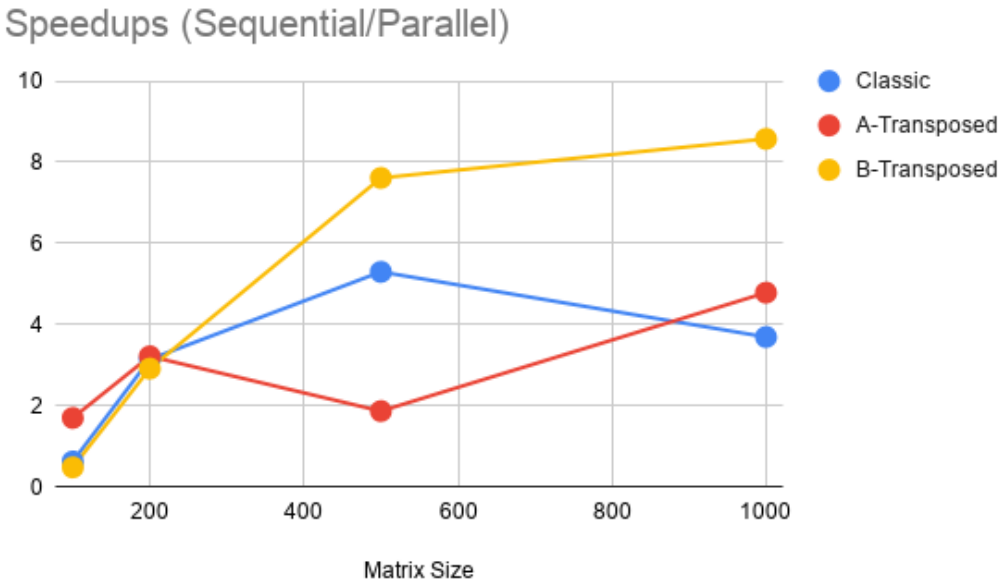
- **Introduction:**
  So, what I did in this assignment is basically to implement different methods for multiplying matrixes. There are six different methods (algorithms):
  1. A sequential classic algorithm, where array A is accessed in a way that it goes trough all array rows, and for each row it goes through all columns before moving to next row. But B goes through all columns and for each column it goes through all rows in that column, before moving to next column.



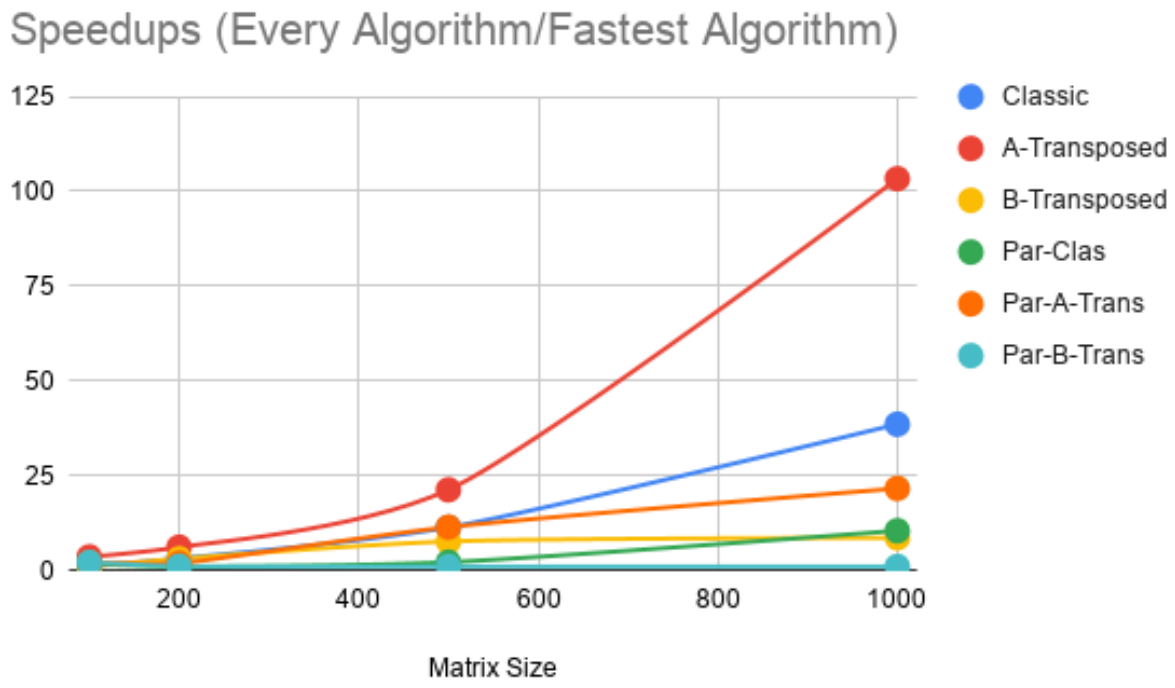
  2. A sequential algorithm where array A is Transposed, meaning both arrays access elements like B drawing above.
  3. A sequential algorithm where both arrays are accesed like A drawing above.
  4. A parallel version of classic algorithm.
  5. A parallel version of A-transposed algorithm.
  6. A parallel version of B-transposed algorithm.

- **Table representation of the speedups:**

Speedups (Sequential/Parallel)			
Matrix Size	Classic	A-Transposed	B-Transposed
100	0.6233	1.6967	0.4785
200	3.1427	3.2159	2.9139
500	5.2979	1.8652	7.6133
1000	3.6919	4.7851	8.5764



Speedups (Every Algorithm/Fastest Algorithm)						
Matrix Size	Classic	A-Transposed	B-Transposed	Par-Clas	Par-A-Trans	Par-B-Trans
100	1.0211	3.5361	1.0000	1.6381	2.0841	2.0899
200	3.1427	6.1000	3.0419	1.0000	1.8968	1.0439
500	11.3100	21.2375	7.6133	2.1348	11.3864	1.0000
1000	38.5581	103.3678	8.5764	10.4441	21.6022	1.0000



- **How did I parallelize the multiplications?**
  I created a class named Workers. Inside the constructor I had to choose an effective strategy for dividing the work as much as possible between threads. So, I decided to:
  - give each row of the array to one thread if number of array rows are less than cores and greater than 2.
  - If rows are less than 2, then only one thread does the job.
  - And if there are less cores than number of rows, then number of threads is equal to number of cores and each thread handles handles *'array.length/cores'* components.

  Inside this class is an inner class MatrixMultiplier that implements Runnable, thus acting as a thread. It takes a startRowNumber and endRowNumber to work on. In addition, it takes a parameter named choice that decide which of the algorithms to run:
  - 0 for parallel classic
  - 1 for parallel A-transposed
  - 2 for parallel B-transposed.

  To escape the trap of false sharing, it creates a local array and writes the result to that, and when finished calls an inner method which copies the local array to the global array.

  The method that creates the thread is also inside class Workers. It takes an int as a choice to which algorithm to call, then decided to transpose or not transpose one of the arrays. When finished it returns the resulted array c.

  When it comes to choosing the strategy for how many threads should have handled how many rows, I could try to adjust this strategy by observing the results for each run. But I didn't, because I taught maybe this was the whole point of the assignment to show results based of expectations.

- **Observations**
  Parallel B-Transposed
  So what I observed was the parallel algorithm B-Transposed was not the fastest (even though it was still better that most) for array length less than 100-200. But it was the definite best one after length grows.

  Sequential B-Transposed
  Also the Classic Parallel algorithm beats the Sequential B-transposed for array lengths less than 1000. But it beats the classic one when length is around 1000 and greater.

- **Why speedup varies with different array size?**
  When cash mises accrue, for every time the next element the CPU needs is not in cash, additional processing time is added to the total time of the algorithm execution. But cash misses can be avoided if the programmer knows about this issue and makes sure that cash line is filled with the next elements that CPU needs.

  This is why there are variation in the results. Because we are accessing two arrays A and B, and if we only need one element from each row, we get a cash miss for that element in addition to every other element that we are going to access next, because we are

jumping from row to row. But if we make sure that the next element is on the same row, then we only get 1 cash miss for each row, thus our CPU doesn't have to wait for next element to arrive to cash from main memory.

- **How to run the code**
  First make sure you are in the src folder. Compiled version of the code is also included so you can directly call:
  java Main
  or you can compile before running the line above using:
  javac Main.java

- **Conclusion:**
  The java programs we are writing could run almost 40 times faster for n = 1000, if we take advantage of parallelization and take cash structure to consideration, so that we get fewer cash misses.