Documentation
Language Benchmark of matrix multiplication 3

Olga Kalisiak 00325481

November 30, 2024

Github repository: link

Contents

1 Abstract

Matrix multiplication is essential in machine learning, scientific computing, and graphics, where efficiency directly impacts system performance. This study evaluates six Java-based approaches: Parallel, Vectorized, Atomic, Fixed Threads, Fork/Join, and Synchronized. Using metrics like execution time, CPU utilization, and memory consumption, the methods were tested on matrices ranging from 500x500 to 2000x2000.

The Parallel method emerged as the most efficient for large matrices, achieving the lowest execution times and scaling well with increased matrix size. The Fixed Threads approach provided competitive results, particularly for mid-sized matrices, balancing speed and low memory use. Fork/Join excelled in minimizing memory consumption but struggled to maintain time efficiency for larger matrices.

In contrast, Atomic and Synchronized methods faced significant bottlenecks due to synchronization and thread safety overhead, making them less viable for large-scale computations. The Vectorized method, though fast for smaller matrices, exhibited substantial memory overhead, limiting its scalability.

This study concludes that the Parallel method is ideal for optimizing performance on large datasets, with Fixed Threads as a strong alternative for mid-sized matrices. Fork/Join is recommended in memory-constrained environments, while Atomic and Synchronized methods highlight the trade-offs between safety and computational efficiency. Future research could integrate hybrid methods or extend these approaches to distributed systems for handling even larger datasets.

2 Introduction

Matrix multiplication is a fundamental operation in data processing, commonly applied in areas such as machine learning, computer graphics, and scientific computing. The choice of programming language and the method used for performing matrix multiplication can significantly impact performance, especially when dealing with large or sparse matrices.

In this project, I explore the performance of matrix multiplication in Java using several advanced techniques, focusing on opportunities for improving execution time through parallelism and synchronization mechanisms. The methods under investigation include atomic operations, the Fork / Join framework, fixed thread pools, parallel streams, synchronized blocks, and vectorized computations. By comparing these approaches, my aim is to understand their strengths and limitations in optimizing matrix multiplication performance, especially for large-scale matrices.

3 Problem statement

Matrix multiplication is computationally expensive, and its performance varies significantly depending on the implementation and optimization techniques used. Java offers numerous opportunities for performance enhancements through parallelism, such as the Fork/Join framework and parallel streams, and synchronization mechanisms like atomic operations and synchronized blocks.

This project investigates how methods like atomic, Fork/Join, fixed threads, parallel streams, synchronized, and vectorized implementations can optimize matrix multiplication performance in Java. The primary goal is to identify which method delivers the best performance for large-scale matrix multiplication, focusing on efficient utilization of computational resources and exploring the impact of parallelism and synchronization on execution time.

4 Methodology

To benchmark the matrix multiplication methods, I conducted tests on a machine with the following specifications:

- Laptop model: Acer Nitro 5
- Processor: AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz
- Installed RAM: 16.0 GB (available: 15.4 GB)
- Product ID: 00328-00801-21601-AA448
- System Type: 64-bit operating system, x64 processor
- Operating system: Windows 11

Languages and compilers:

- Java (version 20.0.1, IntelliJ IDEA Community Edition 2024.2.2)

Each matrix multiplication method in Java was tested using approaches that leverage parallelization and synchronization mechanisms. The methods implemented include standard multiplication, blocked multiplication, and various parallelized approaches such as atomic operations, Fork/Join framework, fixed thread pools, parallel streams, synchronized blocks, and vectorized computations.

The performance of these methods was evaluated by measuring execution time, memory usage, and CPU usage, providing a comprehensive analysis of resource utilization during matrix multiplication.

4.1 Experiment Setup

4.1.1 Programming Languages and Libraries

- Java: The implementations include several parallelized methods tested with varying matrix sizes. The experiments were benchmarked using Java Microbenchmark Harness (JMH) in IntelliJ IDEA.

4.2 Performance Metrics

The following metrics were measured during the experiments:

- Execution Time: The time taken to complete matrix multiplication operations, recorded in milliseconds.
- Memory Usage: Real-time memory consumption during computation.
- CPU Usage: The processor load was monitored to evaluate parallel efficiency and resource utilization.

4.3 Experimental Procedure

Matrices of various sizes were used to observe how performance scales with increasing matrix dimensions. The following sizes were tested: 500x500, 1000x1000, 1500x1500, 2000x2000. The experiments were repeated multiple times to ensure consistency in results, and the average execution time, memory consumption, and CPU utilization were recorded.

5 Experiments

Codes from these experiments are available in the repository: Github repository

At first I tested 2 methods which depend on the number of threads.

5.1 Parallel Matrix Multipliaction

5.1.1 Method description

The ParallelMatrixMultiplication method employs Java's ExecutorService with a fixed thread pool to perform matrix multiplication in parallel. Each row of the result matrix is processed as a separate task, submitted to the executor. This approach allows efficient utilization of multiple CPU cores, making it suitable for large matrices. The method waits for all threads to complete using Future objects before consolidating the results.

The results of the execution time experiment are presented in the Table 1 and in the chart Figures 1, 2 & 3 below.

Matrix Size	Threads	Execution Time (ms)	CPU Time (ms)	Memory Used (KB)
500	2	146.63	1.059	2928.25
500	4	84.63	1.302	3982.71
500	8	54.06	0.292	5341.62
500	16	47.50	0.513	6215.65
500	32	46.77	0.558	7158.69
1000	2	2004.91	0	7769.95
1000	4	999.53	0.814	12931.47
1000	8	768.10	0.762	10734.34
1000	16	800.94	4.607	13578.79
1000	32	768.13	3.239	13946.50
1500	2	10195.31	0	15798.00
1500	4	6084.84	1.302	16675.70
1500	8	3842.15	0	17999.35
1500	16	3561.59	6.944	20273.24
1500	32	3592.48	7.813	21195.03
2000	2	26848.32	0	29114.39
2000	4	16366.22	7.813	30005.50
2000	8	8951.56	9.115	34341.00
2000	16	9181.01	12.135	36728.53
2000	32	9253.70	13.287	38954.23

Table 1: Performance metrics for Parallel Matrix Multiplication by matrix size and thread count.

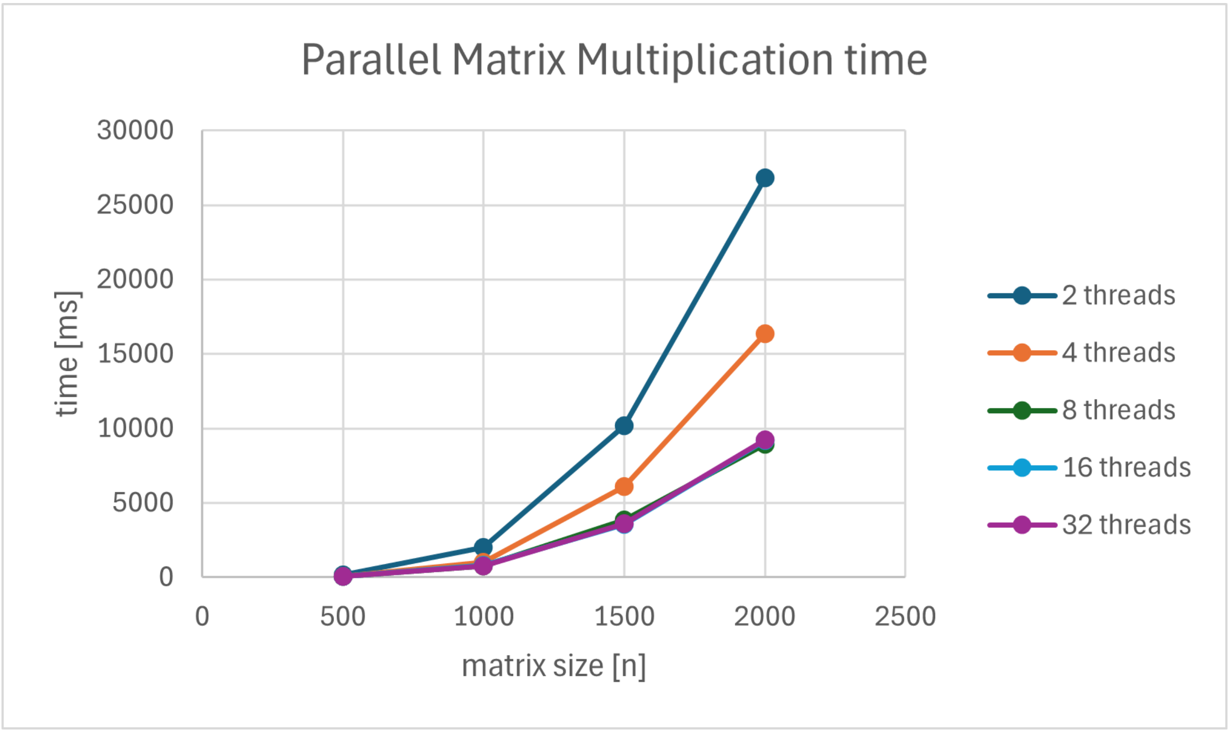

Figure 1: Execution Time for Parallel Matrix Multiplication with varying thread counts.

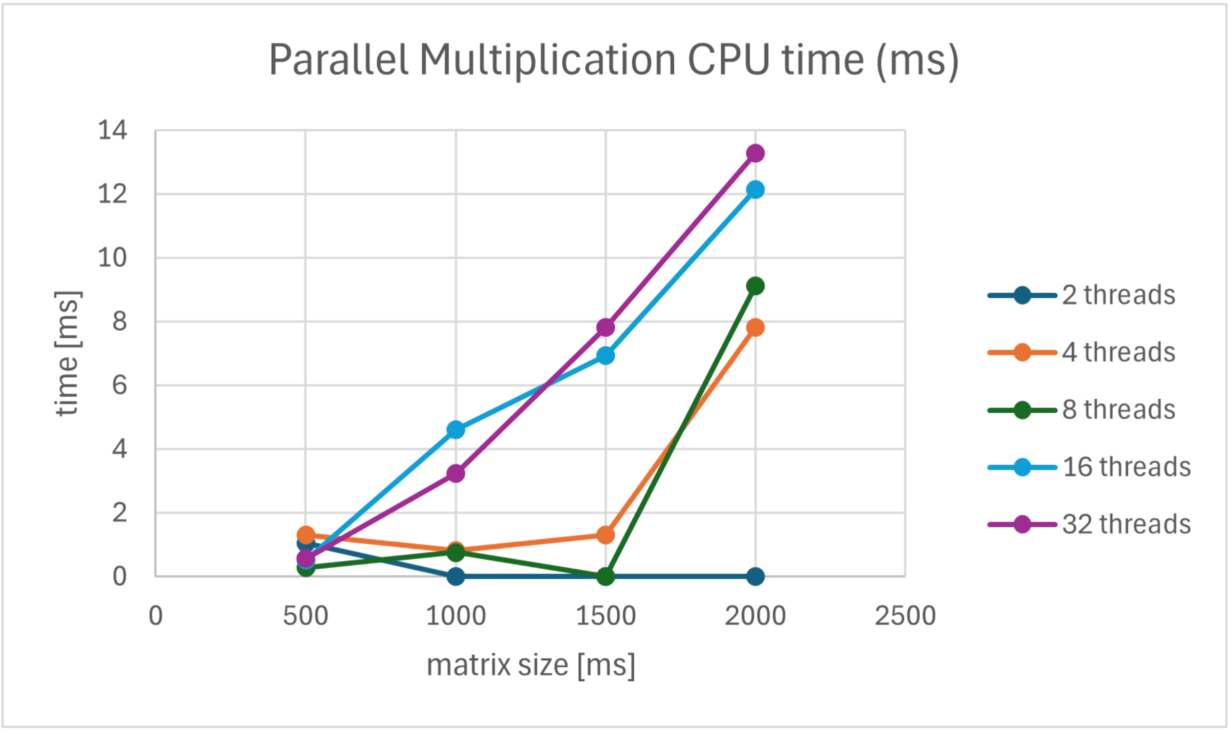

Figure 2: CPU Time for Parallel Matrix Multiplication with varying thread counts.

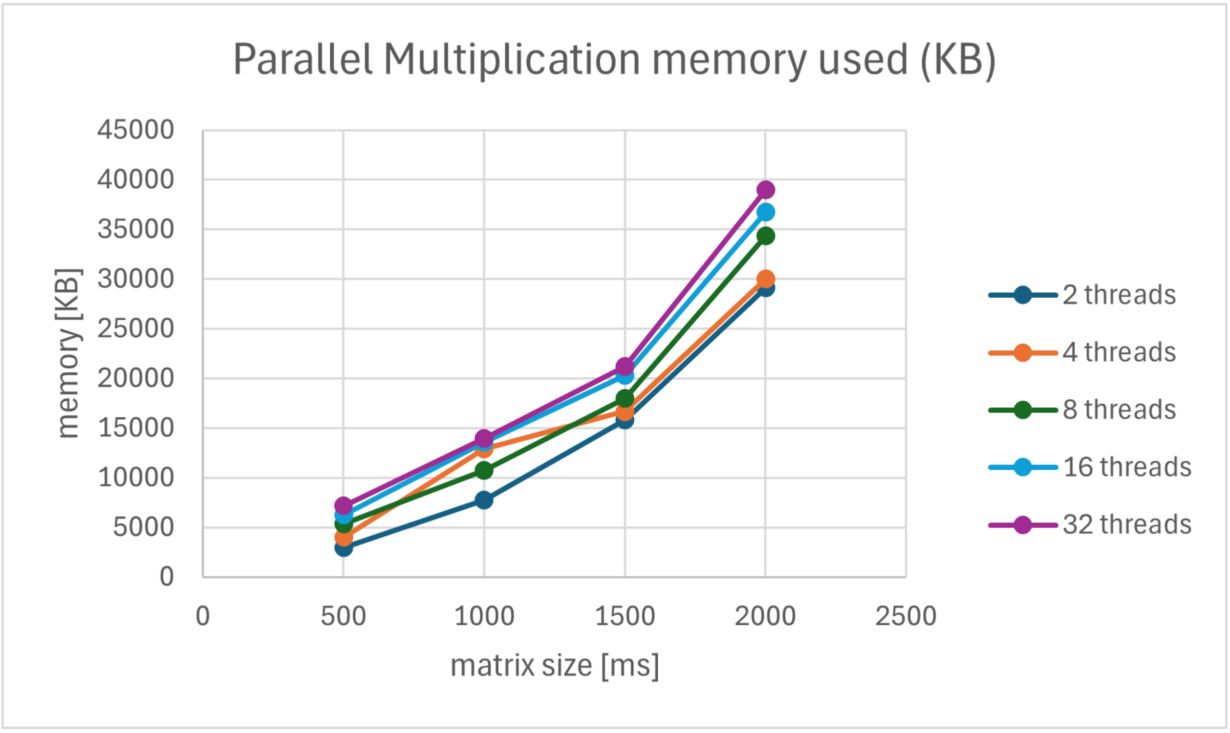

Figure 3: Memory usage for Parallel Matrix Multiplication with varying thread counts.

5.1.2 Results Analysis

The method showed significant performance improvements as the number of threads increased, especially for larger matrices.

- Execution Time: For small matrices, thread overhead somewhat negated the benefits of parallelism. However, as the matrix size grew, the method exhibited nearly linear scalability up to the CPU core count.
- CPU and Memory Usage: The method demonstrated high CPU utilization with a proportional increase in memory usage. Optimal results were observed when the thread count matched the available CPU cores.

5.2 Vectorized matrix multiplication

5.2.1 Method description

The VectorizedMatrixMultiplication method divides the workload across multiple threads using Java's ExecutorService and fixed thread pools. Each thread computes the result for a specific row of the matrix using a vectorized dot product for the row and column. This technique efficiently reduces the computational complexity for each cell by focusing on precomputed arrays and leveraging data locality.

The results of the execution time experiment are presented in the Table 2 and in the chart Figures 4, 5 & 6 below.

Matrix Size	Threads	Execution Time (ms)	CPU Time (ms)	Memory Used (KB)
500	2	178.91	0.70	23082.60
500	4	131.99	0.34	53462.96
500	8	141.41	0.76	89351.10
500	16	134.22	0.38	33119.99
500	32	146.58	0.91	95935.79
1000	2	3017.44	4.56	63633.78
1000	4	1968.03	4.14	76281.29
1000	8	1271.62	1.25	56545.21
1000	16	1175.44	3.47	111473.13
1000	32	1255.99	4.14	76281.29
1500	2	12109.09	18.23	73979.87
1500	4	7071.30	4.56	63633.78
1500	8	4661.38	4.34	120500.60
1500	16	4547.25	6.94	136597.07
1500	32	4525.20	12.15	162278.87
2000	2	29391.88	18.23	127183.13
2000	4	18495.90	10.42	153034.26
2000	8	12355.36	7.81	99763.68
2000	16	12289.38	7.81	79994.91
2000	32	12394.97	20.83	108937.13

Table 2: Performance metrics for Vectorized Matrix Multiplication with varying thread counts.

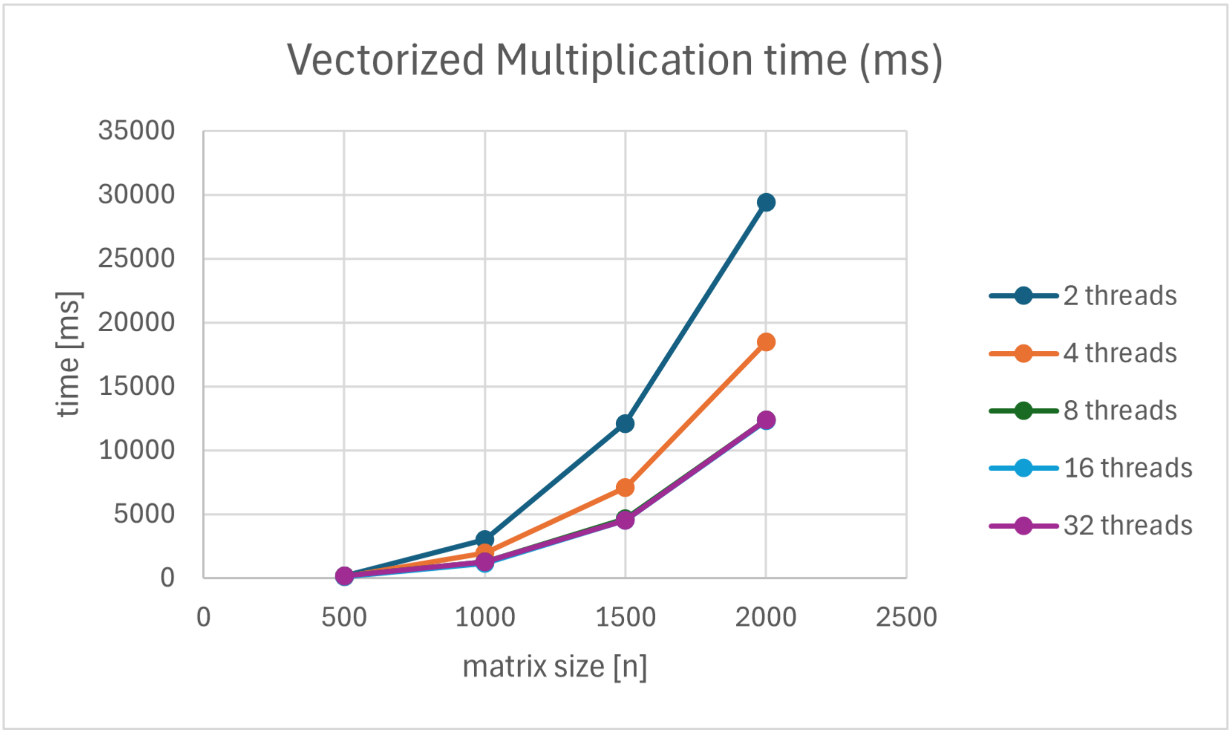

Figure 4: Execution Time for Vectorized Matrix Multiplication with varying thread counts.

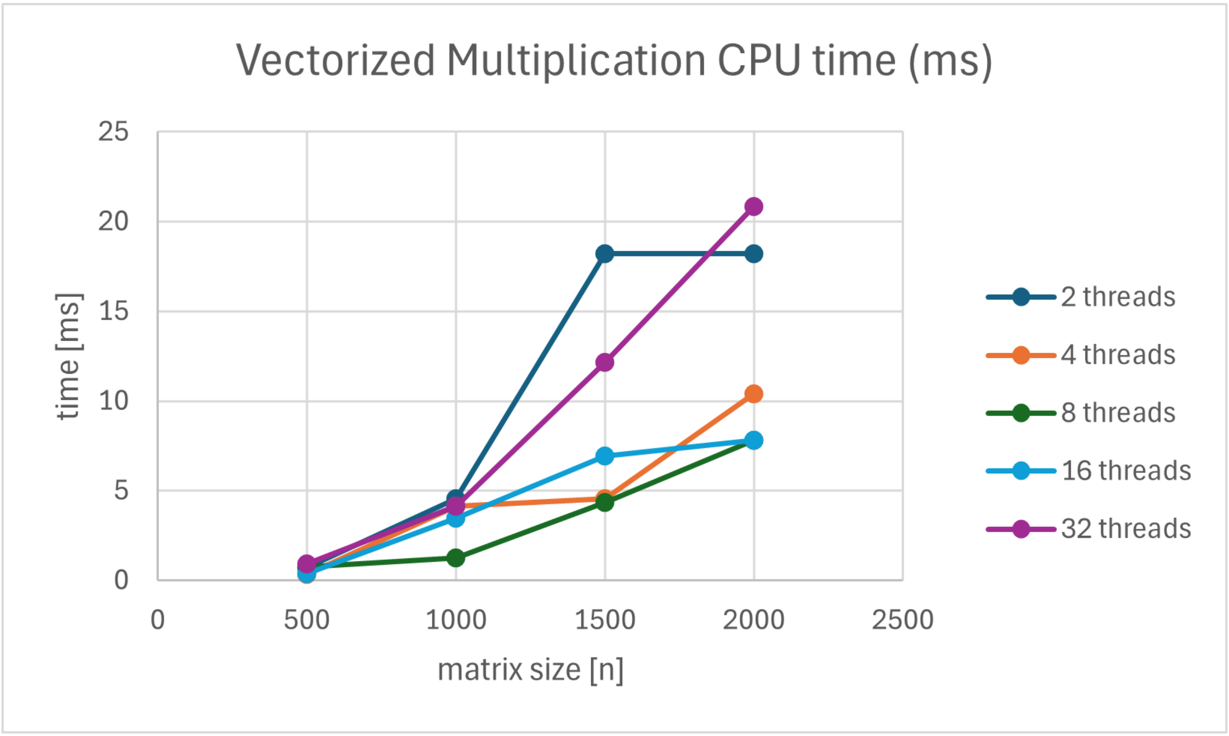

Figure 5: CPU Time for Vectorized Matrix Multiplication with varying thread counts.

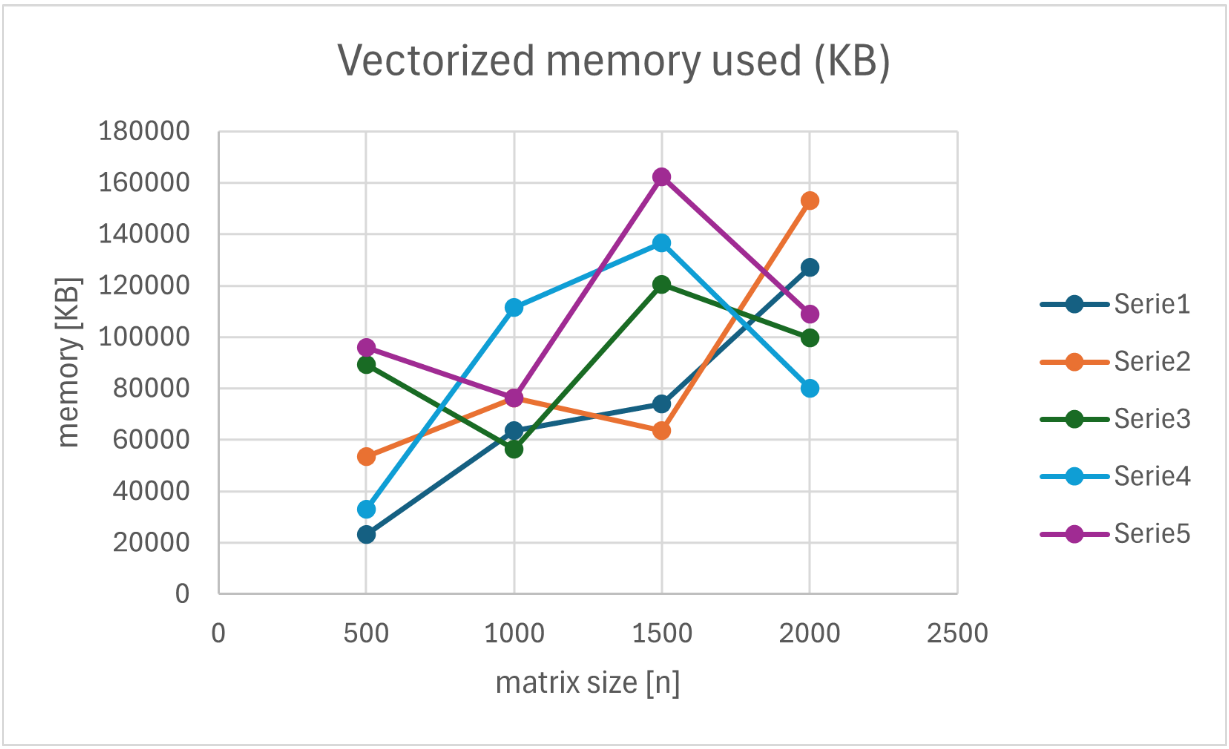

Figure 6: Memory usage for Vectorized Matrix Multiplication with varying thread counts.

5.2.2 Results Analysis

The method demonstrated excellent scalability with thread counts up to 16 for larger matrices. Beyond 16 threads, the overhead of managing threads outweighed the benefits of parallelism, particularly for matrices smaller than 1500x1500.

5.3 Comparison of all methods

5.3.1 Descriptions of methods

- Atomic Matrix Multiplication: This method utilizes AtomicDoubleArray to perform matrix multiplication in parallel, where each row of the resulting matrix is computed by a separate thread. The threads safely update the result using atomic operations to avoid race conditions during concurrent writes. This approach ensures thread safety by using atomic operations for each element in the matrix.

- Fork-Join Matrix Multiplication: This method uses Java's ForkJoinPool to recursively divide the matrix multiplication task into smaller subtasks. The matrix rows are split into manageable sections (controlled by a threshold), which are computed concurrently using the RecursiveTask mechanism. The result is then combined once all subtasks complete. This method is designed for efficient parallel execution using a divide-and-conquer strategy.

- Fixed Threads Matrix Multiplication: In this approach, the matrix multiplication task is split into multiple tasks where each row of the resulting matrix is computed by a fixed number of threads. The method uses an ExecutorService to manage a thread pool, and each thread computes a row of the resulting matrix. The fixed thread count is determined by the user, and the system waits for all tasks to complete before returning the result. In my tests, the number of threads used for the Fixed Threads method was dynamically set based on the number of available processors. On the testing machine, this resulted in a configuration of 12 threads, reflecting the system's hardware capabilities at runtime.

- Stream Matrix Multiplication: This method leverages Java Streams to perform matrix multiplication in parallel. Each row of the result matrix is computed in parallel using IntStream.range(0, rowsA).parallel(), and the matrix elements are multiplied using a functional approach. This method simplifies the parallel processing logic by using built-in stream operations and is ideal for handling large datasets.

- Synchronized Matrix Multiplication: In this method, matrix multiplication is performed using multiple threads, but the synchronized keyword is used to ensure that only one thread can update a particular element in the result matrix at a time. This method ensures thread safety, but the use of synchronization may lead to potential performance bottlenecks, especially for larger matrices, as multiple threads wait to access the shared result matrix.

The results of the execution time experiment are presented in the Table 3 and in the chart Figures 7, 8, 9, 10 & 11 below. For comparison, I chose the results of parallel and vectorized matrix multiplication for 16 threads because they were the best.

Matrix Size	Method	Threads	Time (ms)	CPU Time (ms)	Memory (KB)
500	**Parallel**	16	47.50	0.51	6215.65
500	**Vectorized**	16	134.22	0.38	33119.99
500	**Atomic**	-	56.22	7.54	4053.91
500	**Fixed Threads**	12	37.55	0.23	8189.36
500	**Fork Join**	-	46.94	0.06	1953.99
500	**Stream**	-	76.31	40.56	36545.70
500	**Synchronized**	-	254.47	3.37	2168.20
1000	**Parallel**	16	800.94	4.61	13578.80
1000	**Vectorized**	16	1175.44	3.47	111473.13
1000	**Atomic**	-	642.67	32.55	14236.96
1000	**Fixed Threads**	12	636.42	0.81	12931.47
1000	**Fork Join**	-	746.49	0.00	7344.66
1000	**Stream**	-	960.80	625.47	2411.66
1000	**Synchronized**	-	4000.89	10.69	7048.45
1500	**Parallel**	16	3561.59	6.94	20273.24
1500	**Vectorized**	16	4547.25	6.94	136597.07
1500	**Atomic**	-	3079.54	95.70	27743.80
1500	**Fixed Threads**	12	3088.11	7.81	19810.68
1500	**Fork Join**	-	3178.18	0.65	15640.03
1500	**Stream**	-	3418.41	2493.06	34303.30
1500	**Synchronized**	-	27724.76	26.04	15397.80
2000	**Parallel**	16	9181.01	12.14	36728.53
2000	**Vectorized**	16	12289.38	7.81	79994.91
2000	**Atomic**	-	8867.04	136.72	57731.55
2000	**Fixed Threads**	12	9353.86	6.51	33829.58
2000	**Fork Join**	-	10339.76	4.69	29750.46
2000	**Stream**	-	9846.70	7150.00	76362.32
2000	**Synchronized**	-	67897.01	13.02	28834.97

Table 3: Comparison of performance metrics for various matrix multiplication methods.

To more clearly present the differences between the methods, I excluded the synchronized method from the second graph because it was significantly worse than the others in terms of time.

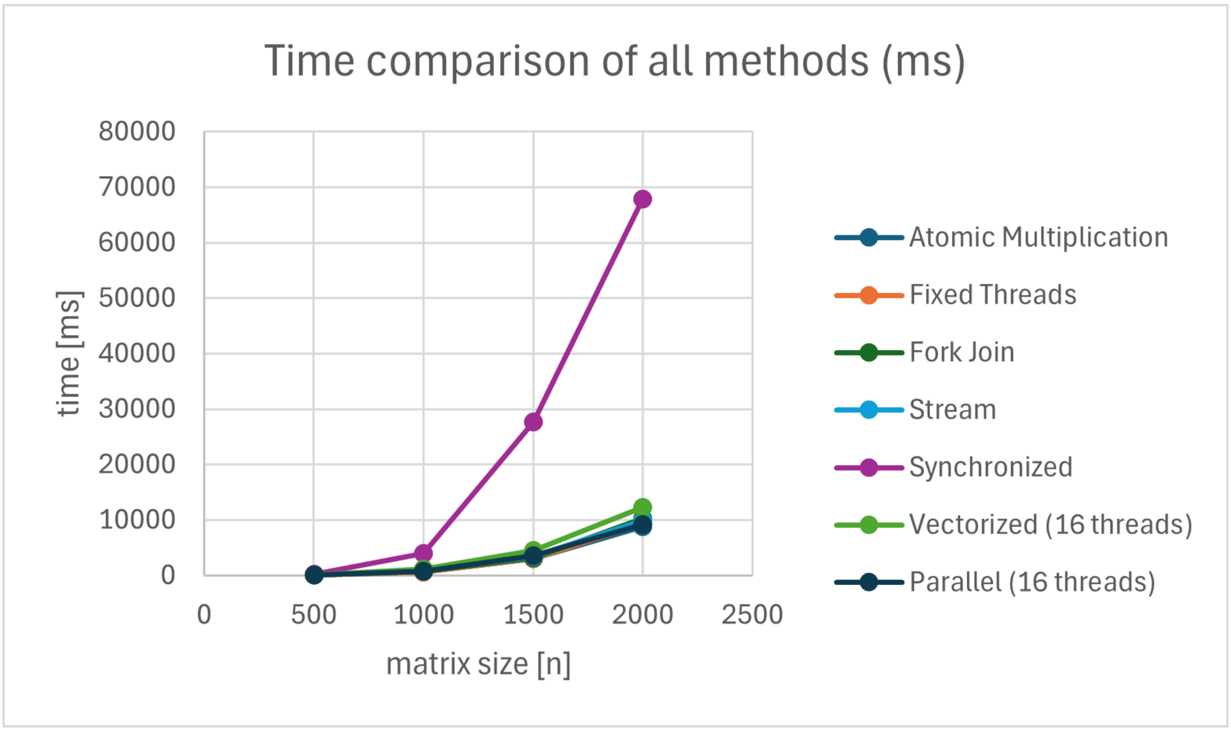

Figure 7: Execution Time for all methods

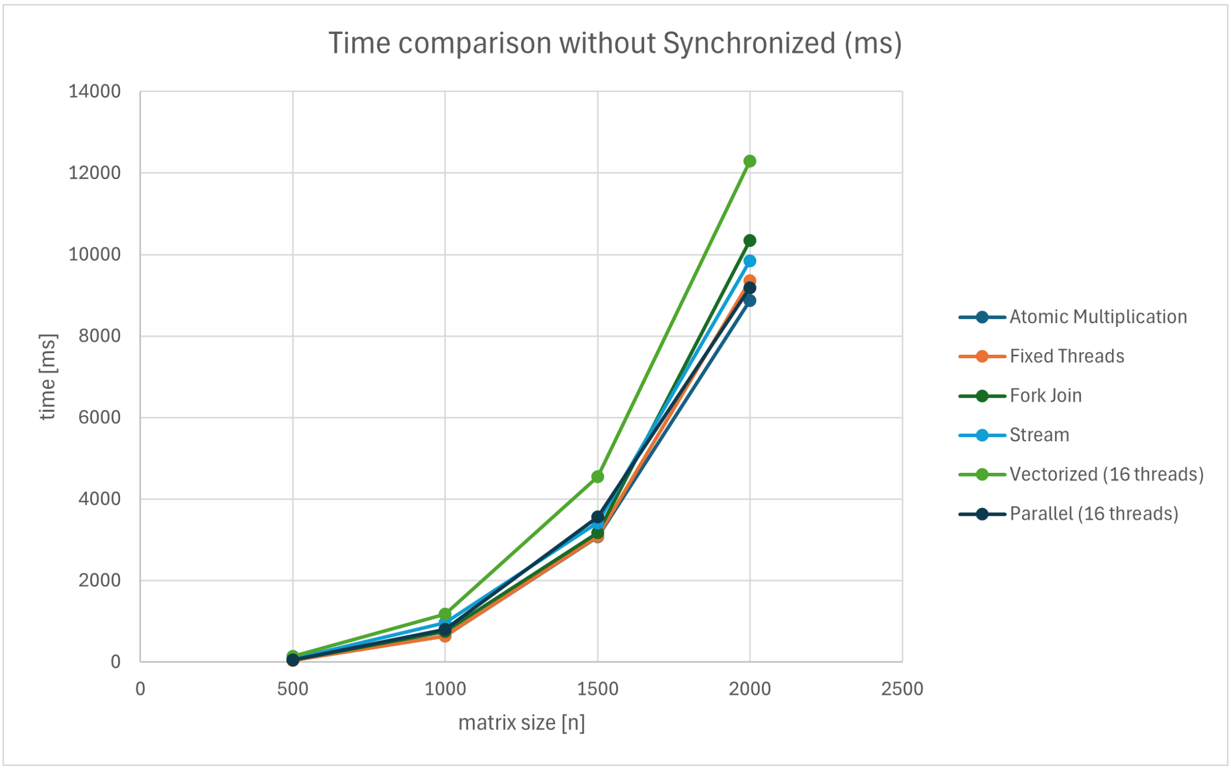

Figure 8: Execution Time Comparison for All Methods (Excluding Synchronized)

To more clearly present the differences between the methods, I excluded the Stream method from the second graph because it was significantly worse than the others in terms of CPU time.

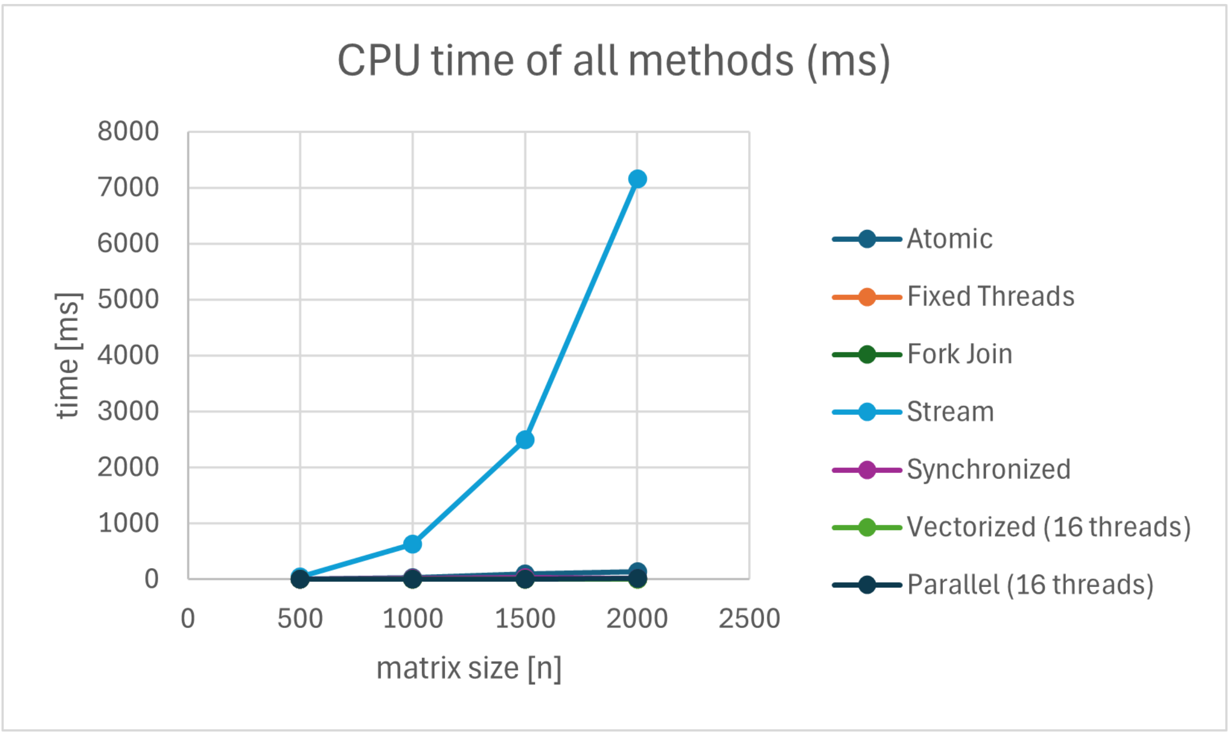

Figure 9: CPU Time for all methods

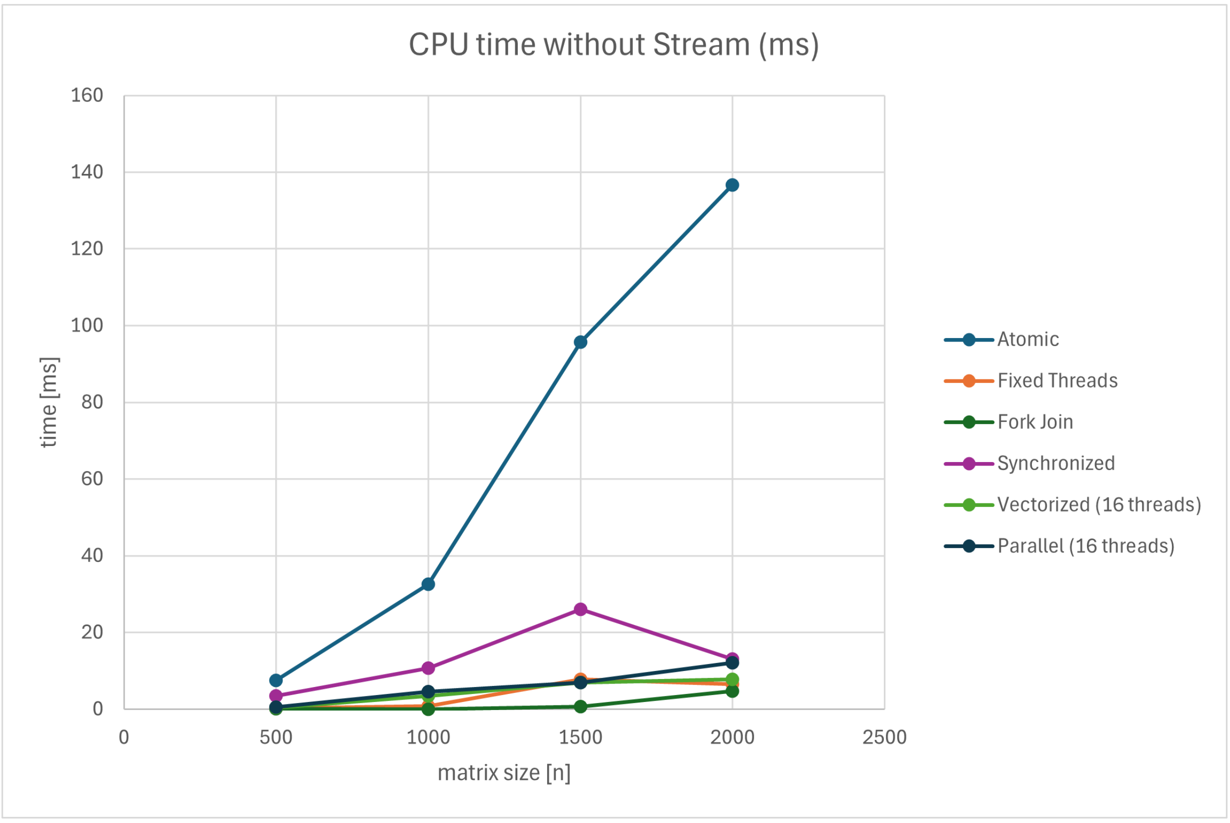

Figure 10: CPU Time Comparison for All Methods (Excluding Stream)

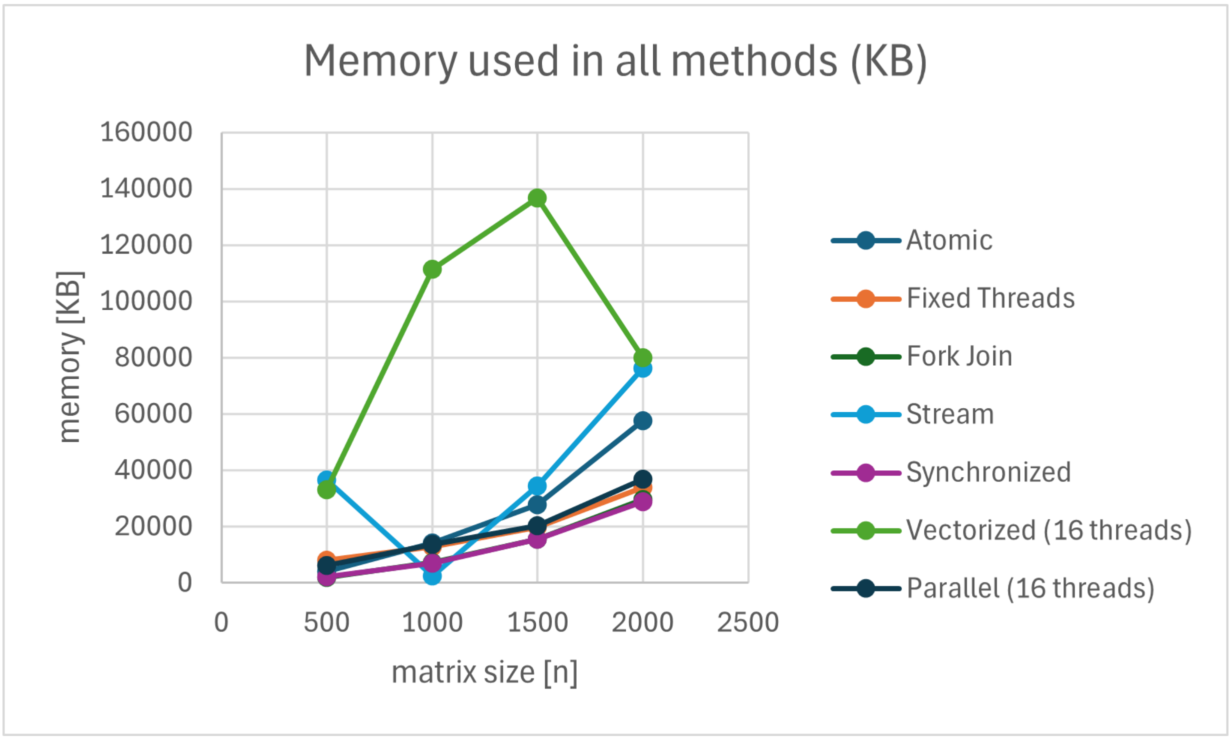

Figure 11: Memory usage for all methods

5.3.2 Comparison analysis

The experiment revealed several key performance trends across the tested methods:

- Parallel Matrix Multiplication:
  - Achieved consistent scalability with increased thread counts, particularly for larger matrices (1500x1500 and 2000x2000).
  - With 16 threads, this method showed one of the fastest execution times for most matrix sizes, outperforming many other approaches.
  - Memory usage was efficient compared to vectorized and atomic methods, though slightly higher than Fork/Join.
- Vectorized Matrix Multiplication:
  - Delivered strong performance for smaller matrices but struggled with memory overhead and execution time for large matrices.
  - For 2000x2000 matrices, memory consumption (79 MB) was significantly higher than in Parallel and Fork/Join.
- Atomic Operations:
  - Although ensuring thread safety, atomic operations introduced overhead, particularly for CPU time. This was most noticeable for large matrices, where CPU time exceeded 136 ms for 2000x2000 matrices.
- Fixed Threads:
  - Showed competitive results for both execution time and memory usage, especially for matrices of size 1000x1000 and 1500x1500. Its balance of efficiency and simplicity makes it a versatile option.
- Fork/Join Framework:
  - Demonstrated excellent memory efficiency, particularly for smaller matrices. However, the method's execution time scaled less effectively for larger sizes compared to Parallel or Fixed Threads.

- Stream and Synchronized:
  - Both methods exhibited significant performance bottlenecks. Stream operations struggled with CPU usage, while synchronized methods showed poor scalability due to locking overhead, particularly for 2000x2000 matrices.

6 Conclusion

This study demonstrates the impact of parallelization and synchronization on matrix multiplication performance. Among the tested methods, the Parallel and Fixed Threads approaches offered the best balance between execution time, memory consumption, and scalability. Fork/Join excelled in memory efficiency, while atomic and synchronized methods highlighted the costs of ensuring thread safety. Future efforts could explore hybrid approaches, such as integrating vectorized techniques with Fork/Join, or extending these methods to distributed systems for even larger-scale computations.

7 Future work

Future experimentation will explore Distributed Matrix Multiplication