Implementing matrix multiplication using several approaches and languages.

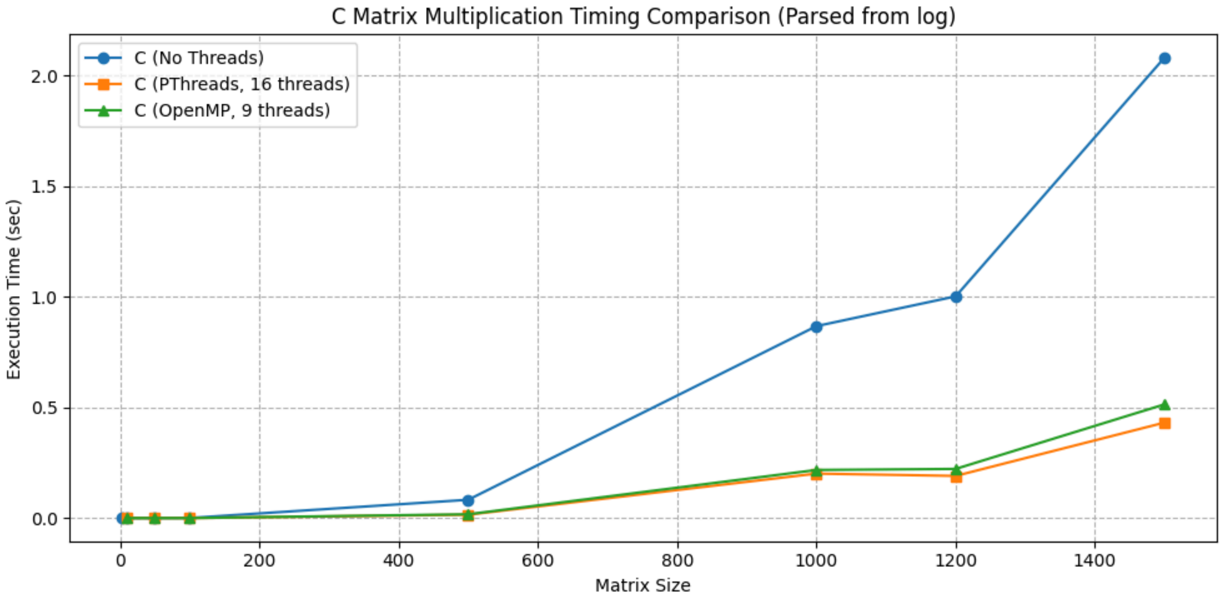
Single-Threaded Implementations:
C (matrix_multi_no_threads.c):
Implements matrix multiplication with a standard three-loop algorithm. Performs multiplication with random generated matrices, and measures execution time for increasing matrix sizes.
Python (matrix_multi_no_threads.py):
Nested loops to multiply matrices in Python. A small 3×3 case using matrices filled with ones is used for correctness verification, and then larger random matrices are multiplied while timing the execution.

Multithreaded Implementations:
C with OpenMP (matrix_multi_openmp.c):
OpenMP is used to parallelize the matrix multiplication loops. The number of threads adjusts dynamically, and results are logged for different thread counts.
C with Pthreads (matrix_multi_pthreads.c):
Partitions the work by dividing the matrix rows among threads
Python with Threads (matrix_multi_threads.py):
Uses Python's concurrent.futures.ThreadPoolExecutor to split the multiplication task among multiple threads

Automation
run_tests_og.sh script compiles and runs the C programs and executes the Python programs. Aggregates the results into a log file.

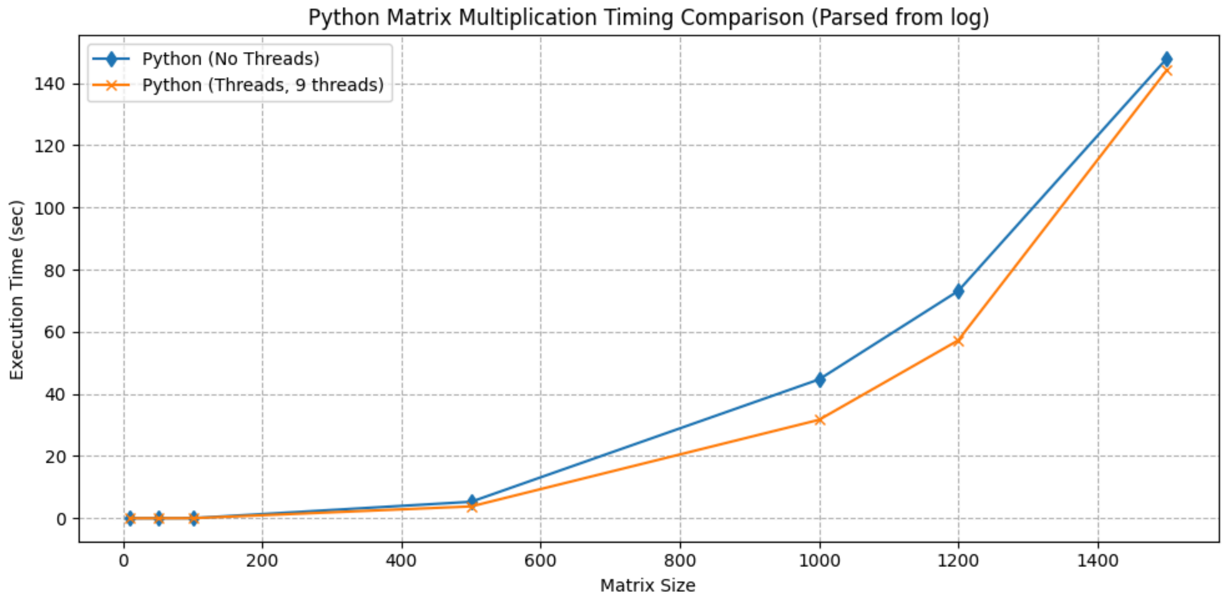
The **timing_comparison.log** file contains the detailed timing results for all tests, covering a range of matrix sizes (from 3×3 to 1500×1500) and varying thread counts.

Images:



- **Blue (C No Threads):** Single-threaded C implementation. Execution time grows significantly once the matrix size goes 400
- **Orange (C PThreads):** Using the PThreads library for multithreading, stays low even as the matrix size increases, demonstrating how parallelizing the workload can reduce time.
- **Green (C OpenMP):** Similar to PThreads, OpenMP also provides a parallelized approach. Its performance is close to PThreads for most sizes, outperforming both the single-threaded version as the matrix size grows.

Conclusion:
Multithreaded versions (PThreads and OpenMP) scale better for large matrices compared to the single-threaded C code, resulting in lower execution times.



- **Blue (Python No Threads):** Shows the timing for Python's, single-threaded implementation. It is very fast for small matrices but execution time rises quickly as the matrix size grows. Matrix 500×1500, it reaches over 100 seconds.
- **Orange (Python Threads):** Uses Python's threading (ThreadPoolExecutor). A clear improvement over the single-threaded version, the execution times are still much higher than the C versions

Conclusion:
Although threading helps reduce the time in Python, Python loops limit it's performance gains compared to C. These two figures highlight the strong advantage of multithreading in C and demonstrate that Python can benefit from threading; it is still behind C's performance for numerical computations.

Conclusion

- **Single-Threaded Implementations:**
  The C implementation (matrix_multi_no_threads.c) is efficient, even for larger matrix sizes, while the Python implementation (matrix_multi_no_threads.py) suffers as the size increases.

- **Multithreaded Implementations:**
  The Pthreads-based C implementation when multiple threads are used improves execution time. But there is an optimal range for thread count, too many threads can limit performance.

- **General Observations:**

  OpenMP or Pthreads provides clear performance advantages, python offers rapid development, but its loop-based implementations are not best for high-performance computing unless optimized libraries (NumPy) are used. Parallelism is key to scaling up the performance.