Exploring Synchronization and Communication

Anamay Brahme
Department of Business
AI Research Group
Univ. of Europe for Applied Sciences
Konrad-Ruse Ring 11, 14469 Potsdam, Germany.
email address or ORCID

A. Problem Statement

Matrix multiplication is a fundamental operation in scientific computing, but it becomes computationally expensive for large matrices. This project explores the use of OpenMP to parallelize matrix multiplication in order to improve performance. The goal is to compare single-threaded and multi-threaded execution and evaluate the effectiveness of parallelization for different matrix sizes.

I. Methodology

A. Overall Workflow

In this project, I implemented matrix multiplication using the dot product approach. The operation was executed in two modes: a single-threaded version and a multi-threaded version using OpenMP. OpenMP was chosen for its ability to simplify parallelism by automatically creating and assigning threads to loop iterations, which is particularly useful in matrix operations.

Two types of input matrices were used: a large 1000x1000 matrix (with randomly generated values) to test performance under heavy computation, and a smaller 3x3 matrix (with predefined values) for demonstration purposes and to observe the behavior of parallelism in small workloads. Both implementations were evaluated for performance (execution time) and correctness by comparing their results.

II. Results

A. Performance Comparison

A sample figure is shown in Figure 1.

Matrix Size	Method	Time Taken (seconds)	Threads Used
1000 x 1000	Single-threaded	5.855740	1
1000 x 1000	OpenMP (Parallel)	0.855992	8
3 x 3	Single-threaded	0.000001	1
3 x 3	OpenMP (Parallel)	0.000105	8

TABLE I
Performance comparison of single-threaded and OpenMP-based matrix multiplication

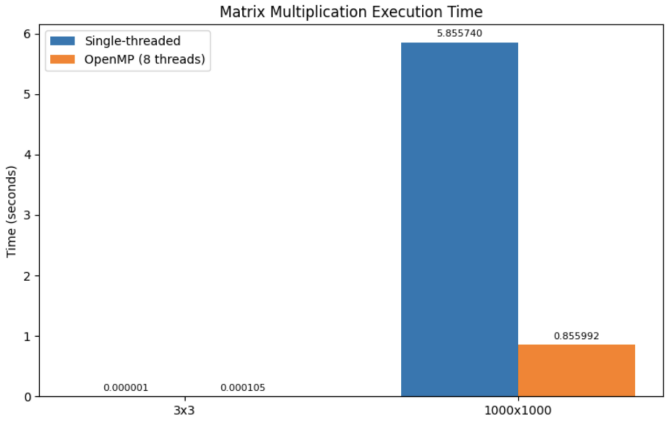

Fig. 1. Figure comparing the Execution times of different matrices for single-threaded and multi-threaded

III. Observations and Conclusions

A. Large Matrix (1000 x 1000) – Performance and Parallel Efficiency

Observation:
For large matrices, the computational workload is significantly high, making it ideal for parallel processing. OpenMP was applied in the `OMP_multiply()` function using `#pragma omp parallel for`, which parallelized the outer loop over matrix rows.

Explanation:
The matrix has 1000 rows, and using OpenMP allowed each thread to handle a subset of rows concurrently. Each thread writes to a different row in the result matrix `C`, meaning no synchronization is required for writes, thus avoiding data races. OpenMP internally manages thread spawning and workload division.

Result:

- Single-threaded time: **5.855740 seconds**
- Multi-threaded (OpenMP) time: **0.855992 seconds**
- OpenMP was approximately **584% faster**.

B. Small Matrix (3 x 3) – Inefficiency of Parallelism

Observation:
To illustrate the limits of parallelism, the same OpenMP approach was applied to a much smaller 3x3 matrix. Although the `OMP_multiply()` function was still used, it operated on just 3 rows, which was too small for meaningful task division.

Explanation:
Even though OpenMP attempted to parallelize across available threads (8 in this case), there were only 3 loop iterations (rows) to split. This led to significant overhead: thread setup,

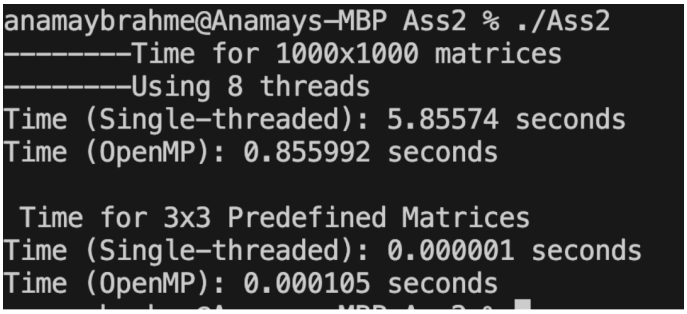

Fig. 2. Source Code Output for c++ file

coordination, and scheduling consumed more time than the actual computation. In contrast, the single-threaded version ran immediately with minimal overhead.

Result:

- Single-threaded time: **0.000001 seconds**
- Multi-threaded (OpenMP) time: **0.000105 seconds**
- Slowdown: OpenMP was approximately **105x slower**.

IV. Conclusion

A. Conclusion for Large Matrix (1000 x 1000)

The implementation of matrix multiplication using OpenMP proved to be highly effective for large matrices. By parallelizing the outer loop using the `#pragma omp parallel for` directive, each thread processed a separate row of the result matrix independently. This avoided write conflicts and made full use of multicore systems. The performance gain was substantial, with OpenMP achieving approximately **584%** speed-up compared to the single-threaded approach. This confirms that OpenMP is well-suited for compute-intensive operations where workload distribution across threads can be effectively leveraged.

B. Conclusion for Small Matrix (3 x 3)

In contrast, applying OpenMP to a small 3x3 matrix showed a performance *decline*, where parallelization incurred more overhead than benefit. The small size of the data led to inefficient thread usage, and the time spent in thread management outweighed the actual computation time. OpenMP was approximately **105 times slower** than the single-threaded version for this case. This experiment highlights that parallelization is not universally advantageous and should be applied only when the task granularity justifies the overhead.

C. Final Implementation Conclusion

Through this project, I demonstrated how OpenMP can be effectively utilized to accelerate matrix multiplication, particularly for large-scale data. The implementation validated the strengths of parallel programming in handling computationally intensive tasks, while also revealing its limitations in scenarios with small workloads. These findings underscore the importance of analyzing task size and thread overhead when designing parallel solutions. Ultimately, OpenMP provided a powerful and scalable approach for enhancing performance in appropriate use cases.

References