**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No: 22510039**

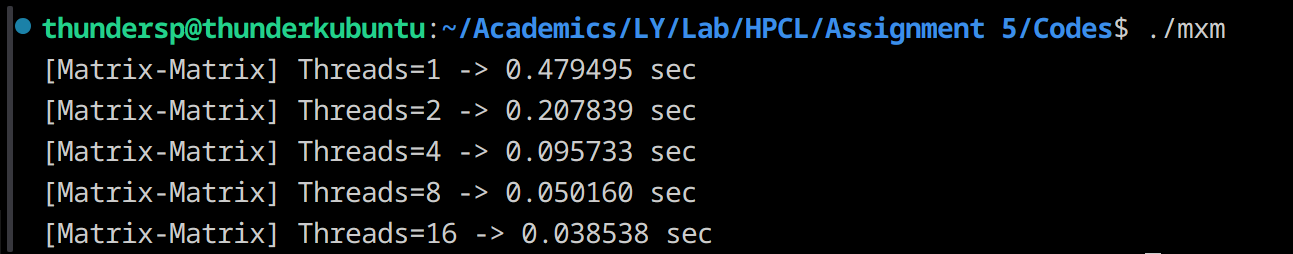
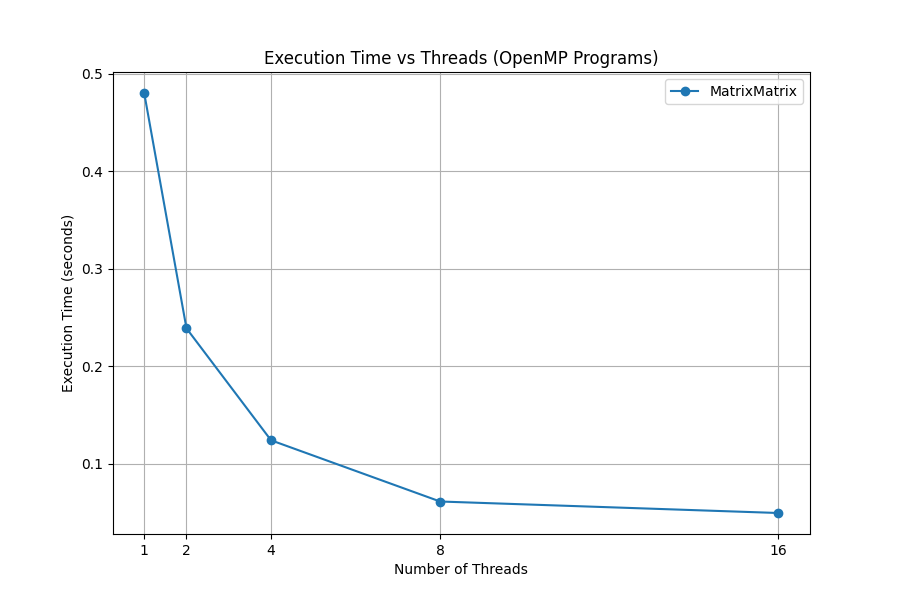
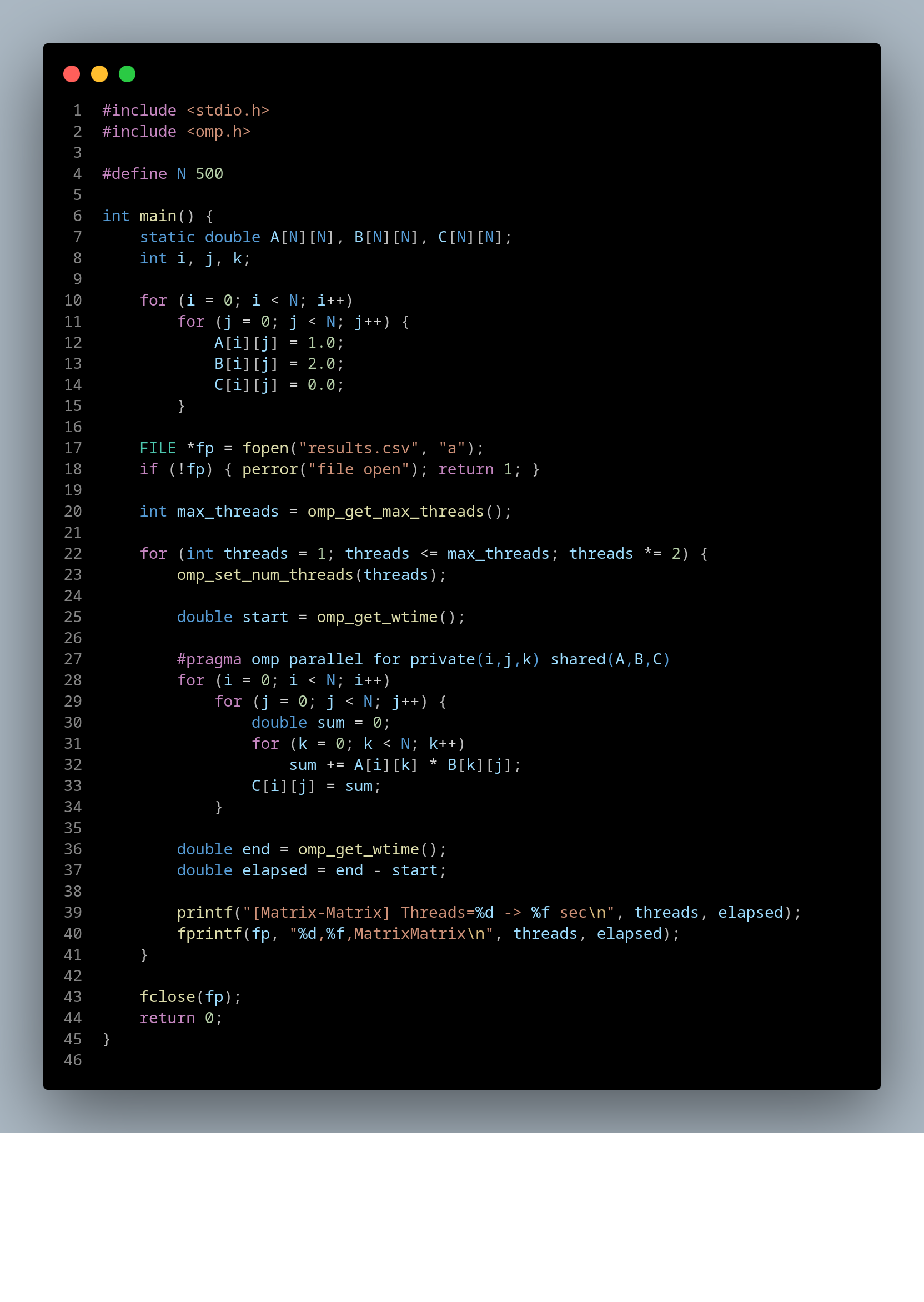
**Title of practical: Implementation of OpenMP programs.**

Implement following Programs using OpenMP with C:

1. Implementation of Matrix-Matrix Multiplication.
2. Implementation of Matrix-scalar Multiplication.
3. Implementation of Matrix-Vector Multiplication.
4. Implementation of Prefix sum.

**Problem Statement 1:** Implementation of Matrix-Matrix Multiplication

**Screenshots:**



**Information:**

Computes C = A × B with complexity O(N³).

Very compute-intensive → benefits a lot from parallelization.

Each element C[i][j] is independent → easy to parallelize.

**Analysis:**

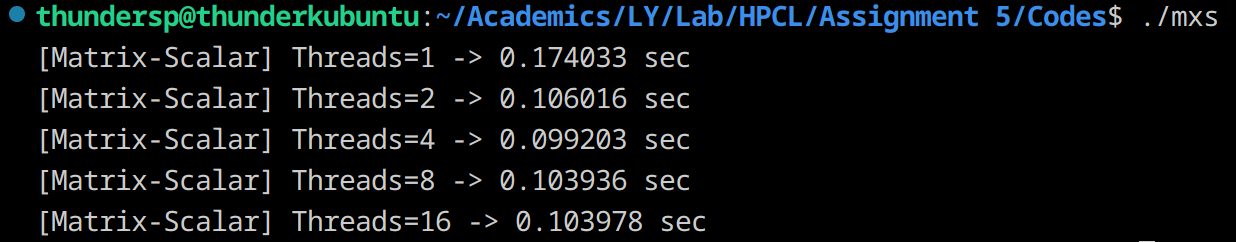
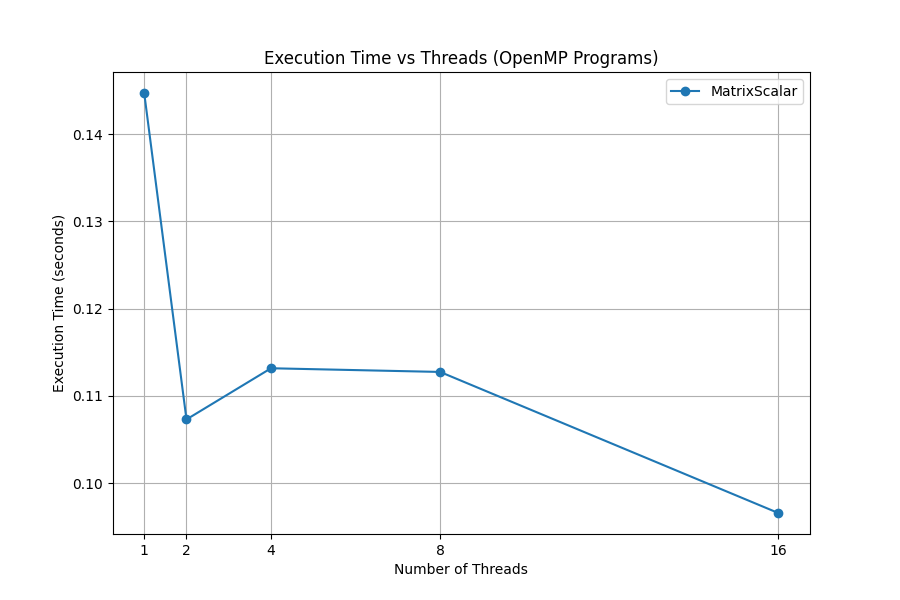
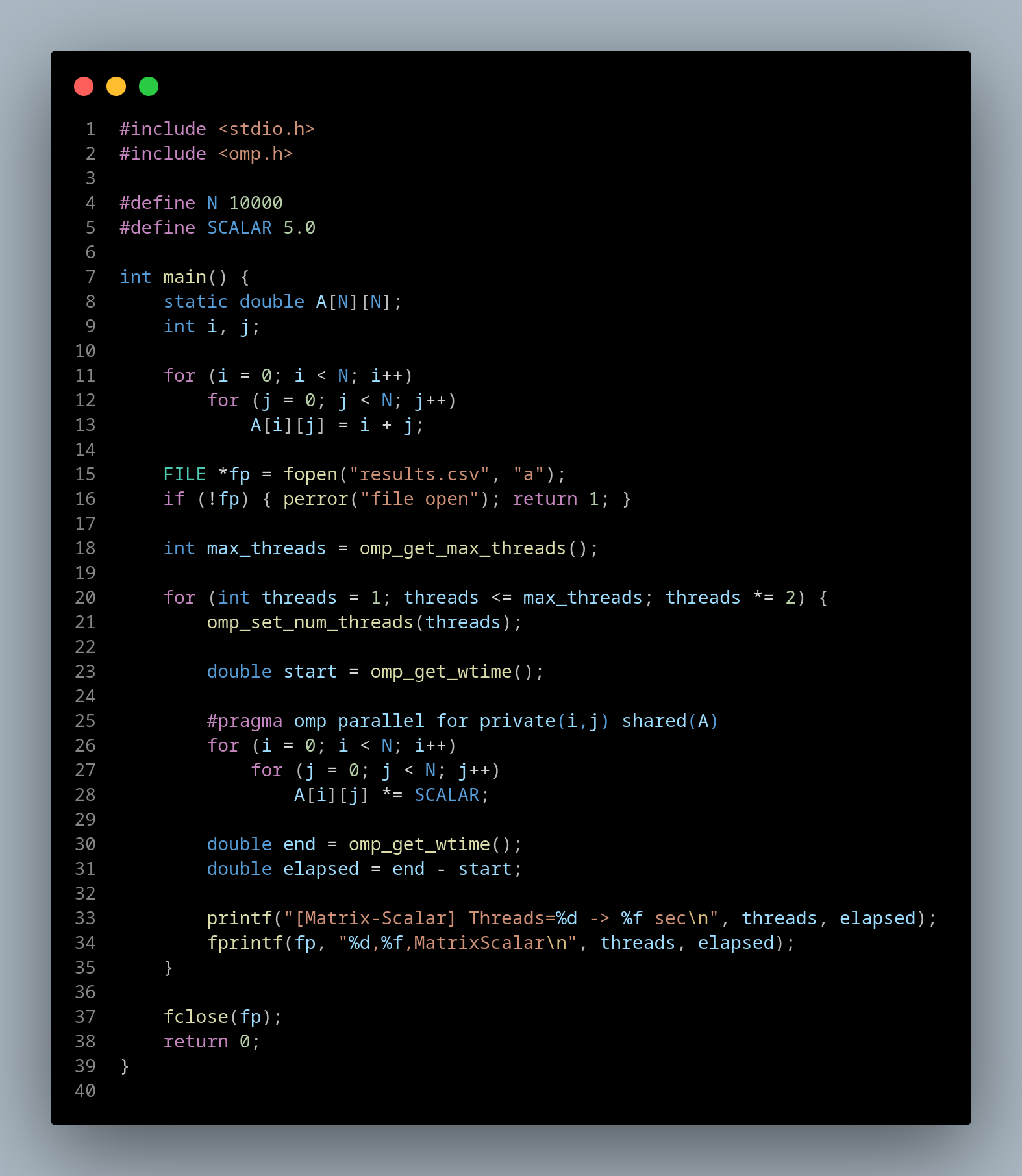
Strong scaling: As threads increase, execution time drops significantly (up to memory bandwidth limit).

Bottleneck: Cache misses. For large N, matrices may not fit in cache, slowing speedup.

Performance expectation: Usually scales well up to ~8–16 threads, then plateaus.

**Problem Statement 2:** Implementation of Matrix-scalar Multiplication.

**Screenshots:**



**Information:**

Simple operation: each element A[i][j] = A[i][j] × scalar.

Complexity O(N²).

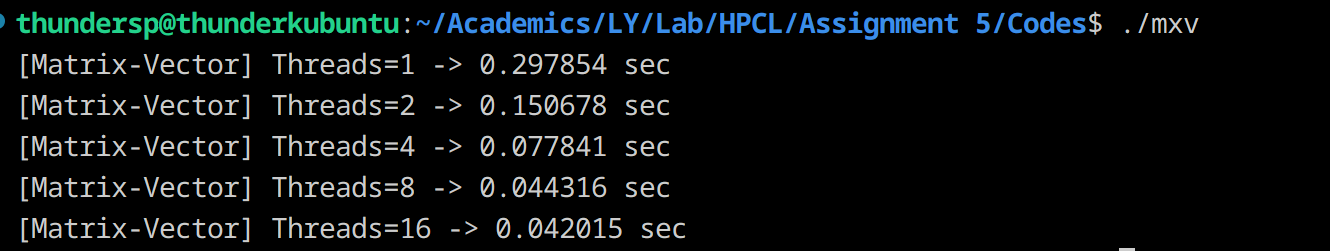
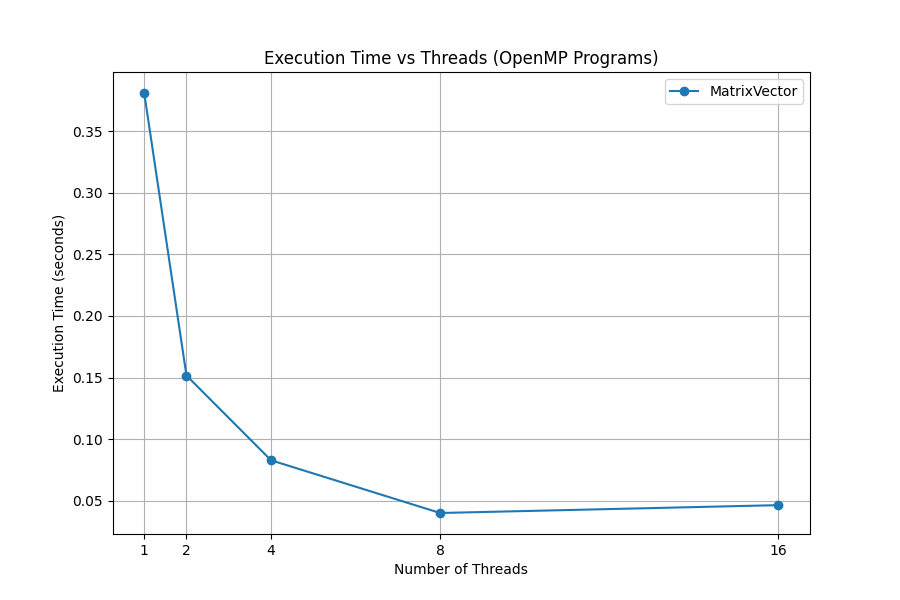
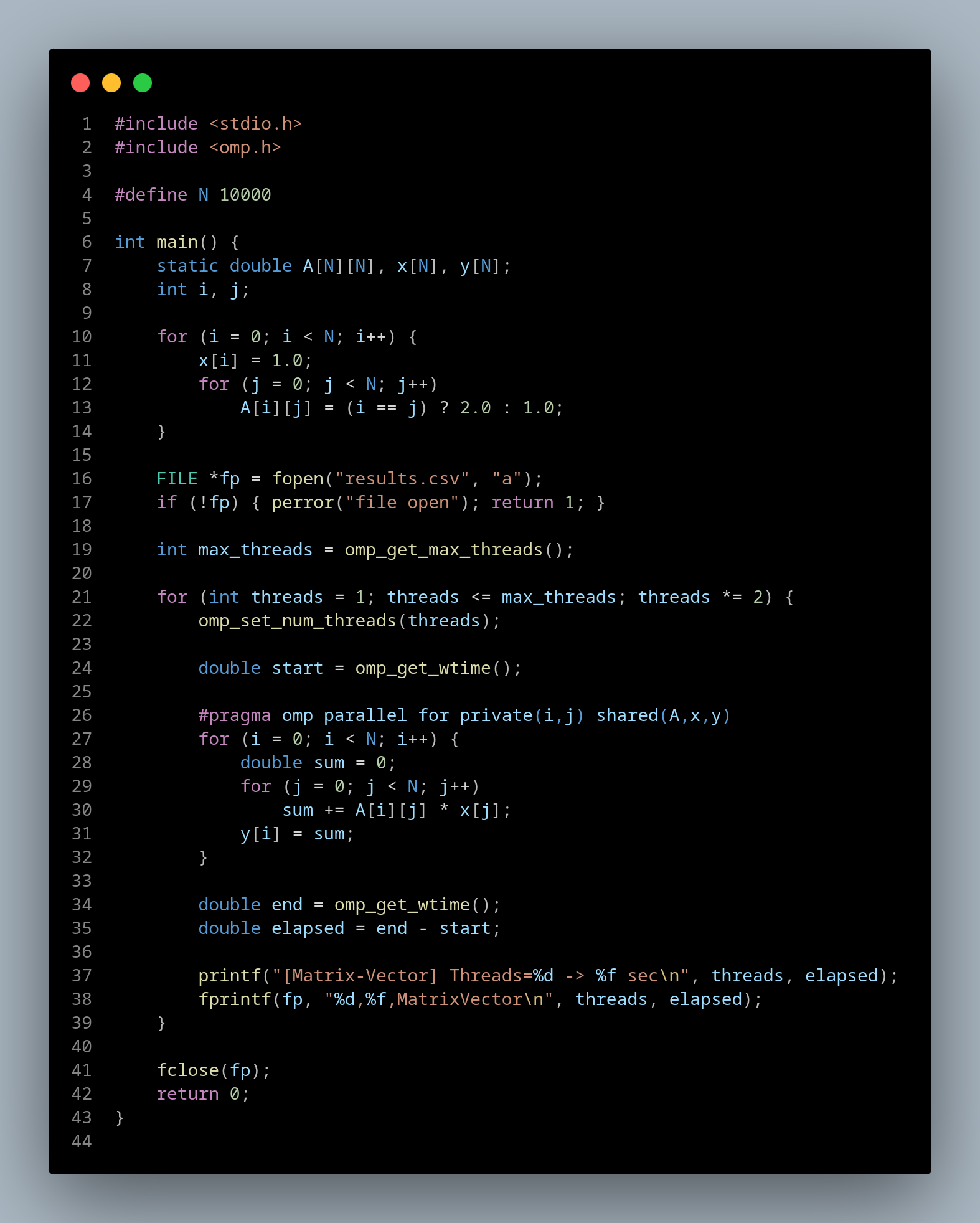
Memory-bound, not compute-bound: lots of memory access, very little arithmetic.

**Analysis:**

* Parallelization overhead may outweigh gains for small matrices.
* Speedup is modest because memory bandwidth is the limiting factor.
* Scaling: Good up to a few cores, then flattens (no more bandwidth).

**Problem Statement 3:** Implementation of Matrix-Vector Multiplication.

**Screenshots:**



**Information:**

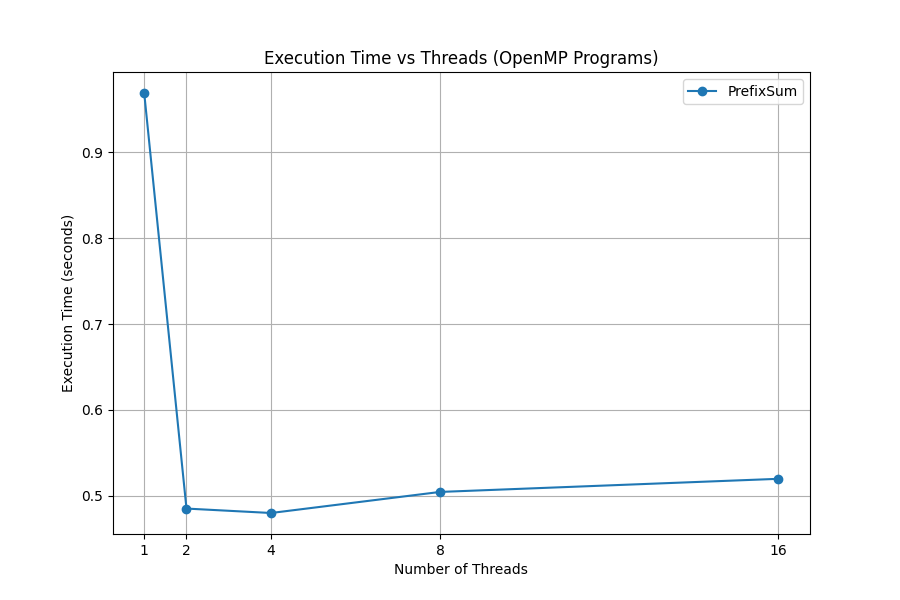
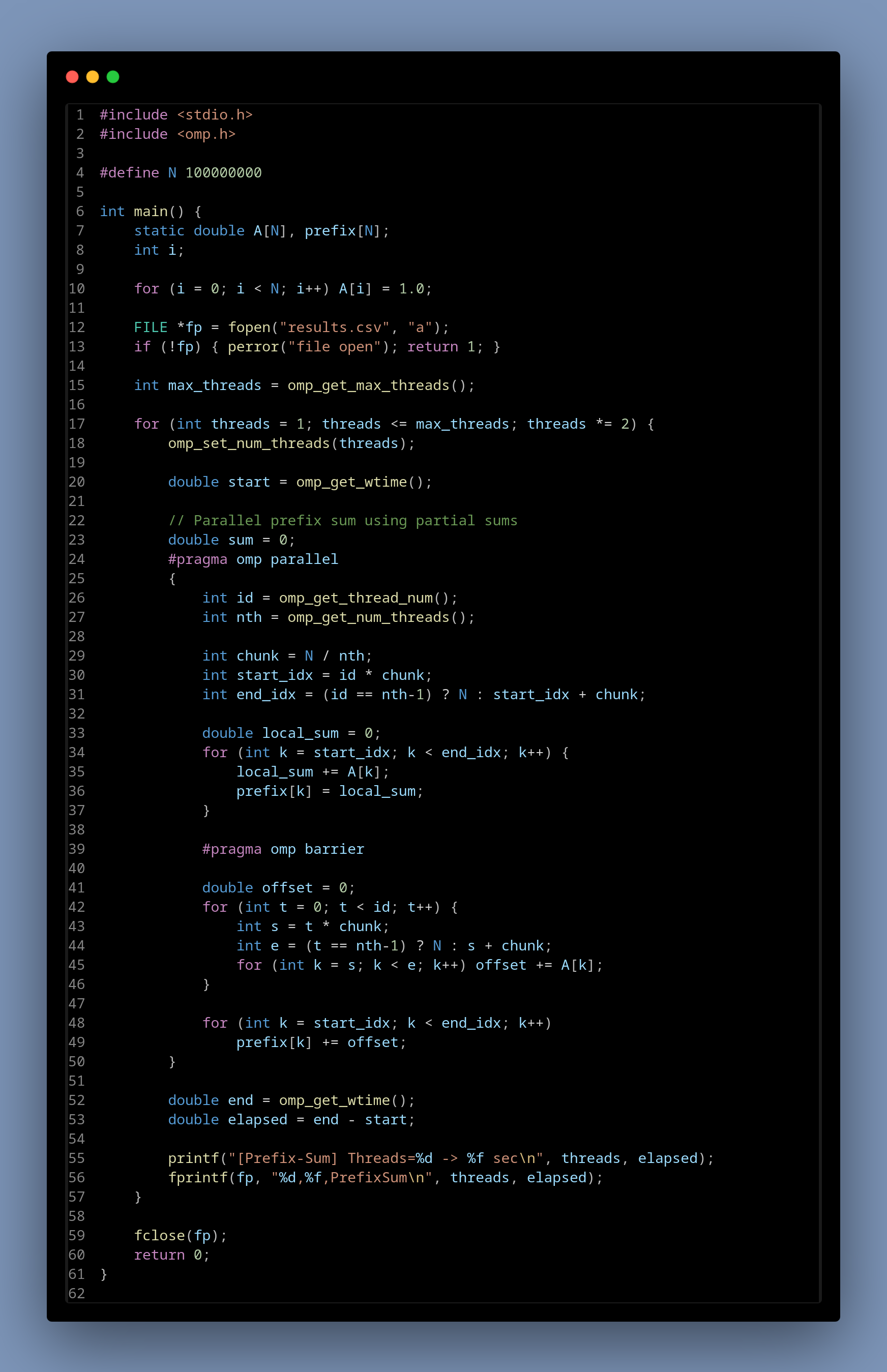
* Computes y = A × x.
* Complexity O(N²).
* Each output element y[i] is independent → parallelizable.
* Common in linear algebra & ML workloads.

**Analysis:**

* Faster than matrix–matrix (fewer operations).
* Memory-bound like scalar multiplication.
* Speedup is moderate. Scaling may flatten quickly for large N.
* Often used as a benchmark for memory performance.

**Problem Statement 4:** Implementation of Prefix sum.

**Screenshots:**



**Information:**

* Compute prefix: prefix[i] = A[0] + A[1] + ... + A[i].
* Serial complexity: O(N).
* Parallelization is tricky because values depend on earlier ones.
* Uses work-efficient parallel scan algorithms.

**Analysis:**

* Compute prefix: prefix[i] = A[0] + A[1] + ... + A[i].
* Serial complexity: O(N).
* Parallelization is tricky because values depend on earlier ones.
* Uses work-efficient parallel scan algorithms.

**Github Link:** [**https://github.com/thundersp/hpcl**](https://github.com/thundersp/hpcl)