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

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 5**

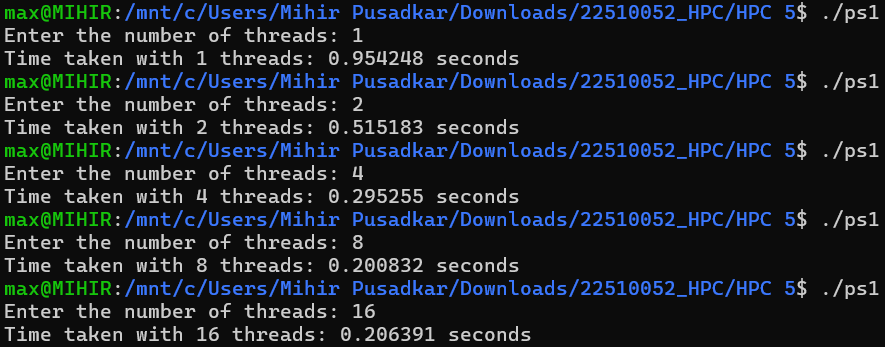
**Exam Seat No: 22510052**

**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: Matrix-Matrix Multiplication**

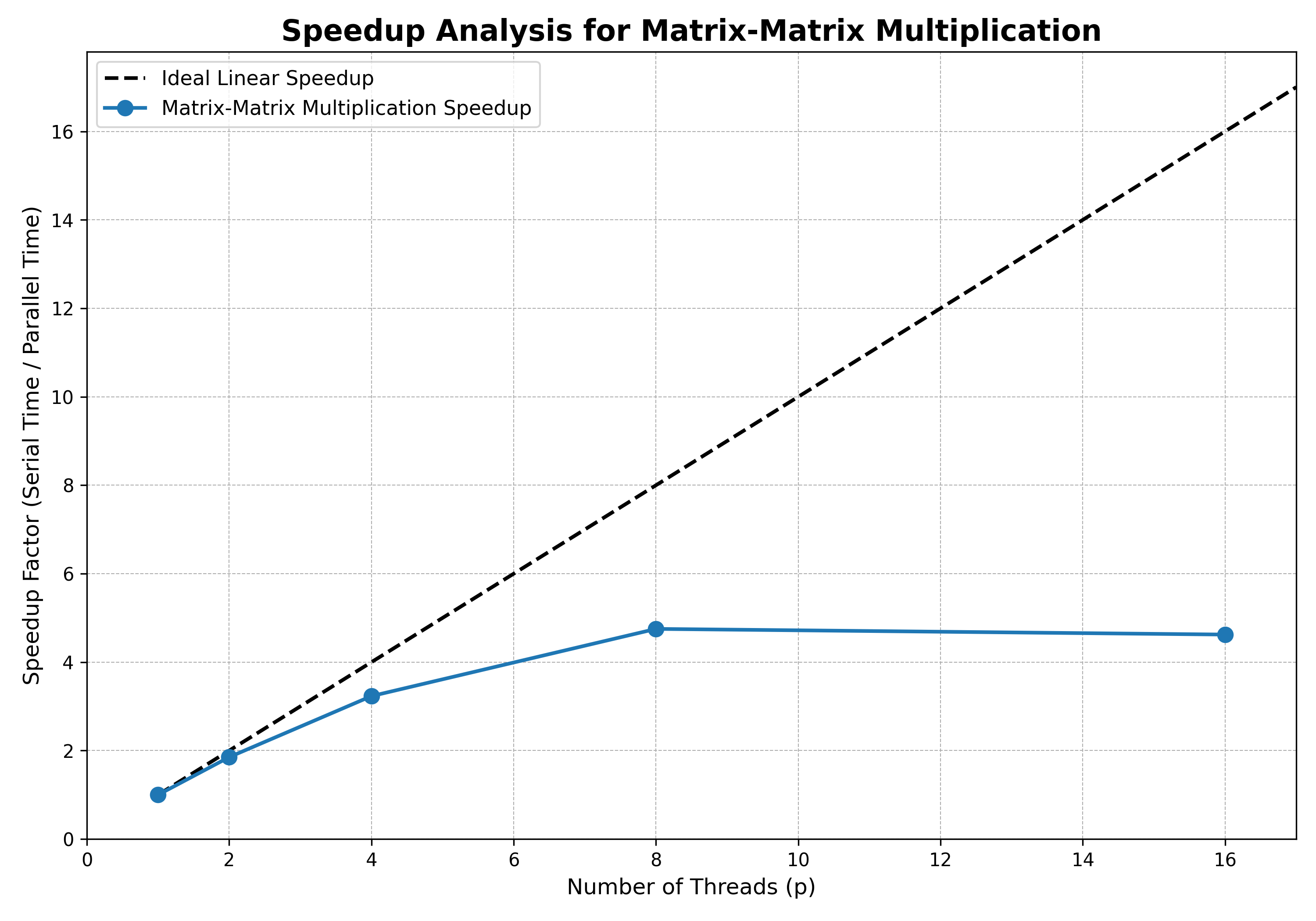
**Screenshots: **

**Information:**

Given the high computational cost of the serial algorithm, the parallel implementation is expected to yield a very substantial speedup. The problem is data-parallel, and the workload is large enough that the overhead of thread management is small in comparison.

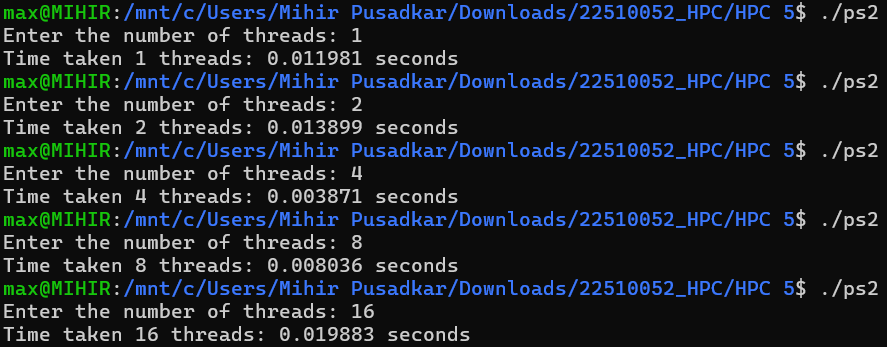
**Analysis:**

* Significant Speedup: Because the serial version is very slow (O(N³)), the parallel implementation will show a very large and impactful performance improvement.
* Strong Scalability: The algorithm is data-parallel, meaning its performance scales well as more threads are added to the problem.
* Independent Row Computation: The strategy of assigning entire rows of the result matrix to different threads ensures the work is independent and avoids conflicts.



**Problem Statement 2: Matrix-Scalar Multiplication**

**Screenshot:**

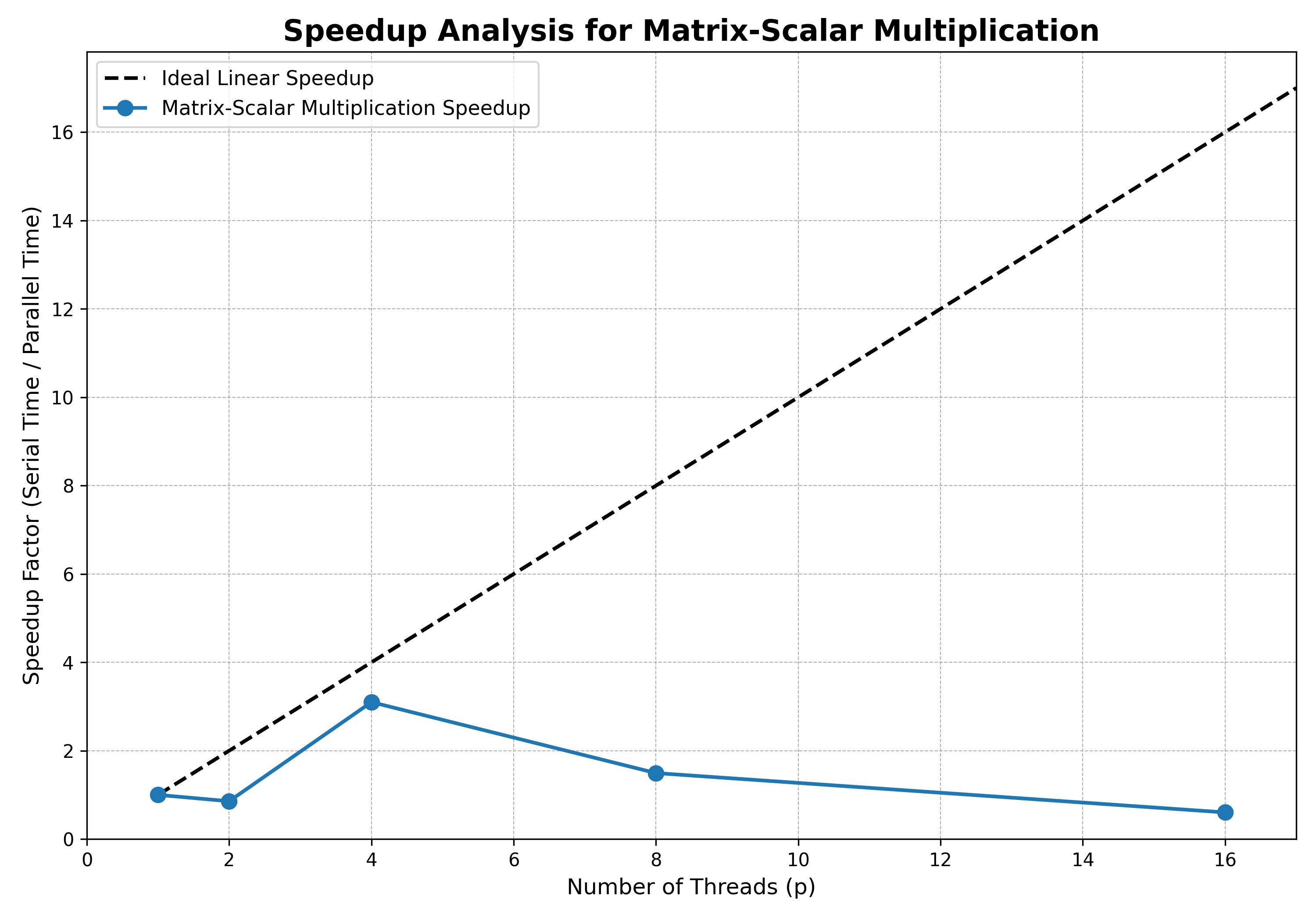
****

**Information:**

The program implements the multiplication of a matrix A by a single scalar value s. This is a fundamental linear algebra operation where every element Aij​ of the input matrix is multiplied by s to produce the corresponding element Bij​ in the output matrix. The parallel strategy leverages the fact that this is an "embarrassingly parallel" problem, meaning the calculation for each element is completely independent of all others. The implementation uses the OpenMP #pragma omp parallel for directive to automatically divide the rows of the matrix among the available threads, allowing each thread to compute its assigned portion simultaneously.

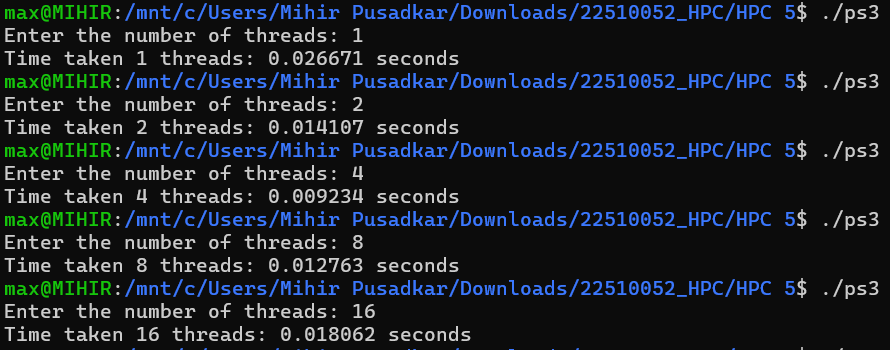
**Analysis:**

* High Efficiency: The parallel algorithm is expected to be extremely efficient.
* Near-Linear Speedup: Performance should scale almost directly with the number of threads (e.g., 4 threads result in nearly 4x speedup).
* No Data Dependency: The calculation for each matrix element is completely independent of all others.
* Minimal Overhead: Threads work on distinct rows and do not need to communicate or wait for each other, leading to very low parallel overhead.



**Problem Statement 3: Matrix-Vector Multiplication**

**Screenshot:**

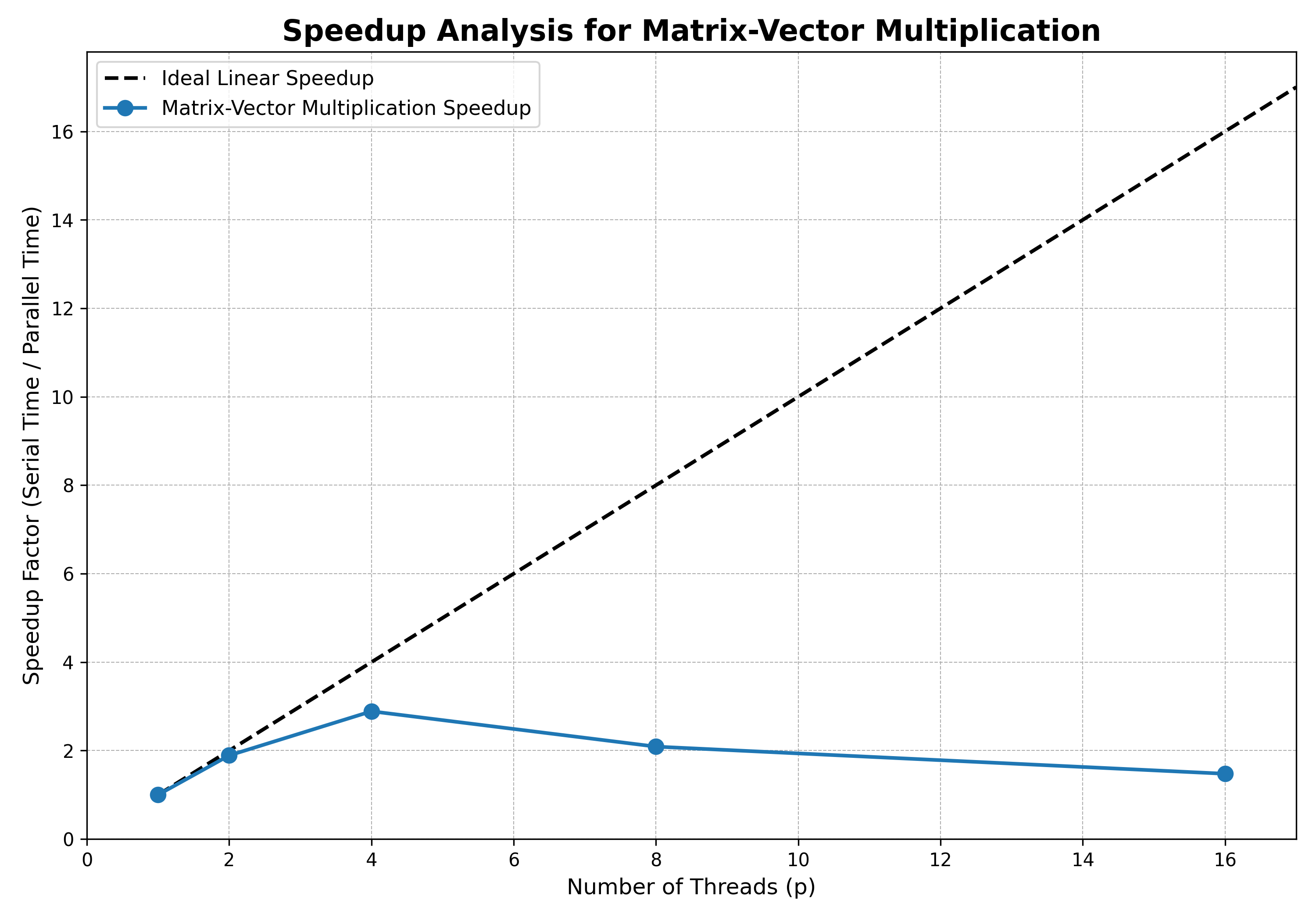
****

**Information:**

This program implements the multiplication of an N x N matrix A by a vector x of size N, resulting in an output vector y of size N. Each element yi​ of the output vector is the dot product of the i-th row of the matrix and the input vector. Like the scalar case, this problem is highly parallelizable because the calculation of each element yi​ is independent of the others. The parallel implementation uses the #pragma omp parallel for construct to assign the computation of different elements of the output vector y to different threads.

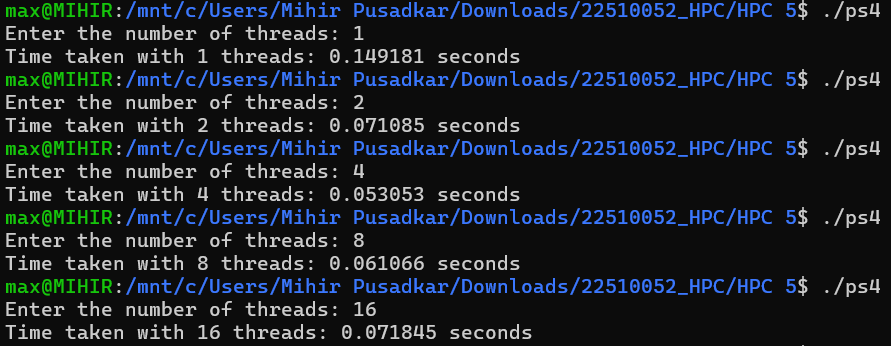
**Analysis:**

* High Efficiency: This algorithm is also very efficient due to its parallel nature.
* Near-Linear Speedup: The speedup is expected to be significant and scale well as more threads are added.
* Independent Workload: The calculation of each element in the result vector (which involves a dot product) is independent of the others.
* Low Overhead: The #pragma omp parallel for construct effectively distributes the independent work with minimal management cost.



**Problem Statement 4: Prefix Sum**

**Screenshots:**

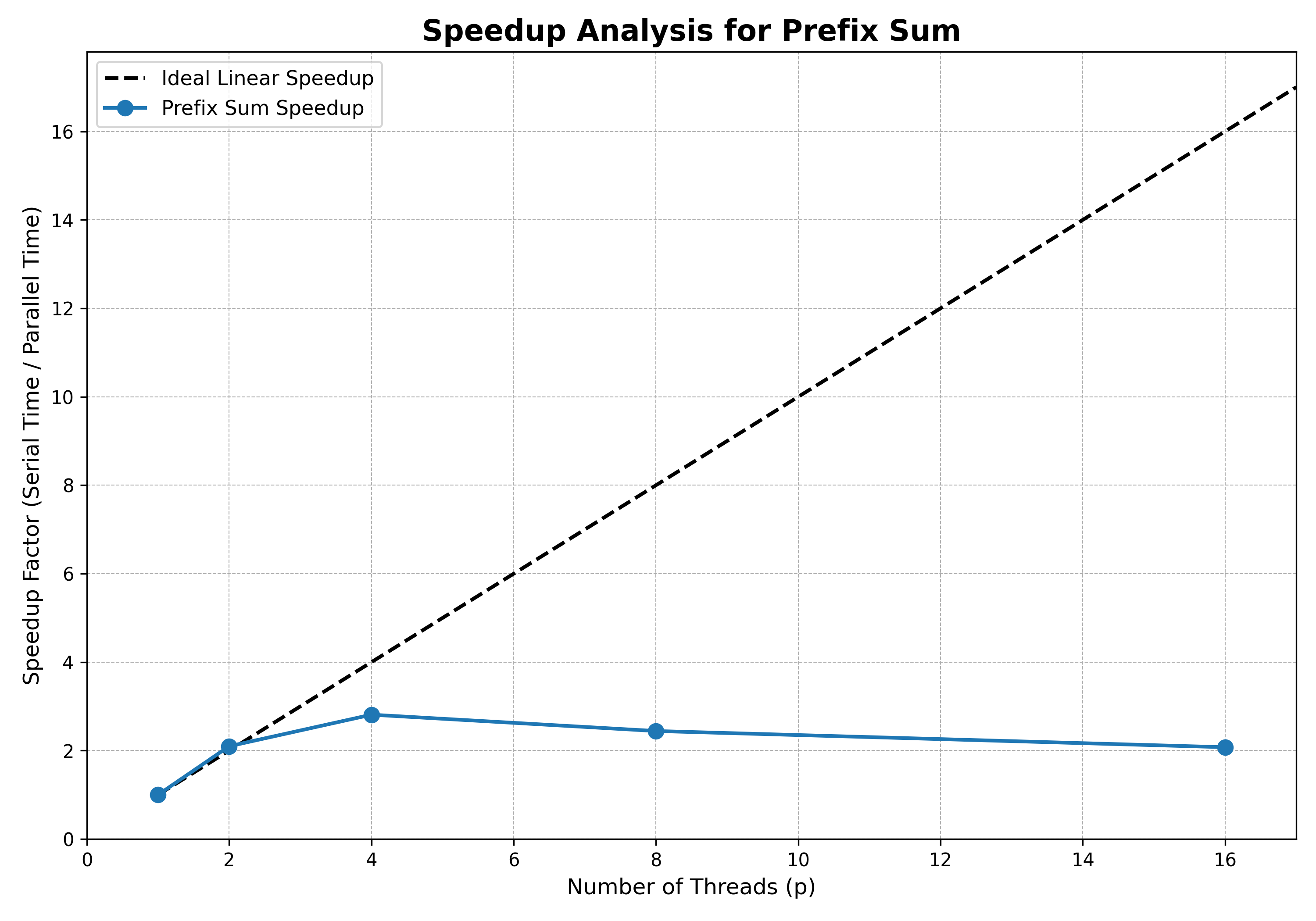
****

**Information:**

The program implements a parallel prefix sum of an array. This operation is not parallel because each element of the output array is dependent on the sum of all previous elements. A simple parallel for loop cannot be used. Hence a two-pass algorithm is implemented. In the first pass, threads compute local prefix sums on separate chunks of the array. In the second pass, these local sums are adjusted using offsets calculated from the totals of the preceding chunks. This requires explicit synchronization using #pragma omp barrier to ensure all threads complete a pass before any thread begins the next.

**Analysis:**

* Speedup: The speedup will be noticeable but will not be linear. The performance gains will become smaller as more threads are added.
* Data Dependency: This is the main reason for lower efficiency. Each output element depends on the previous one, which prevents a simple parallel approach.
* Synchronization Overhead: The use of #pragma omp barrier is necessary for correctness but introduces significant overhead. This is because threads must stop and wait for the slowest thread to finish its pass before proceeding.
* Serial Component: The algorithm contains a small but crucial serial part (calculating the prefix sum of the partial sums).

****

**Github Link:**