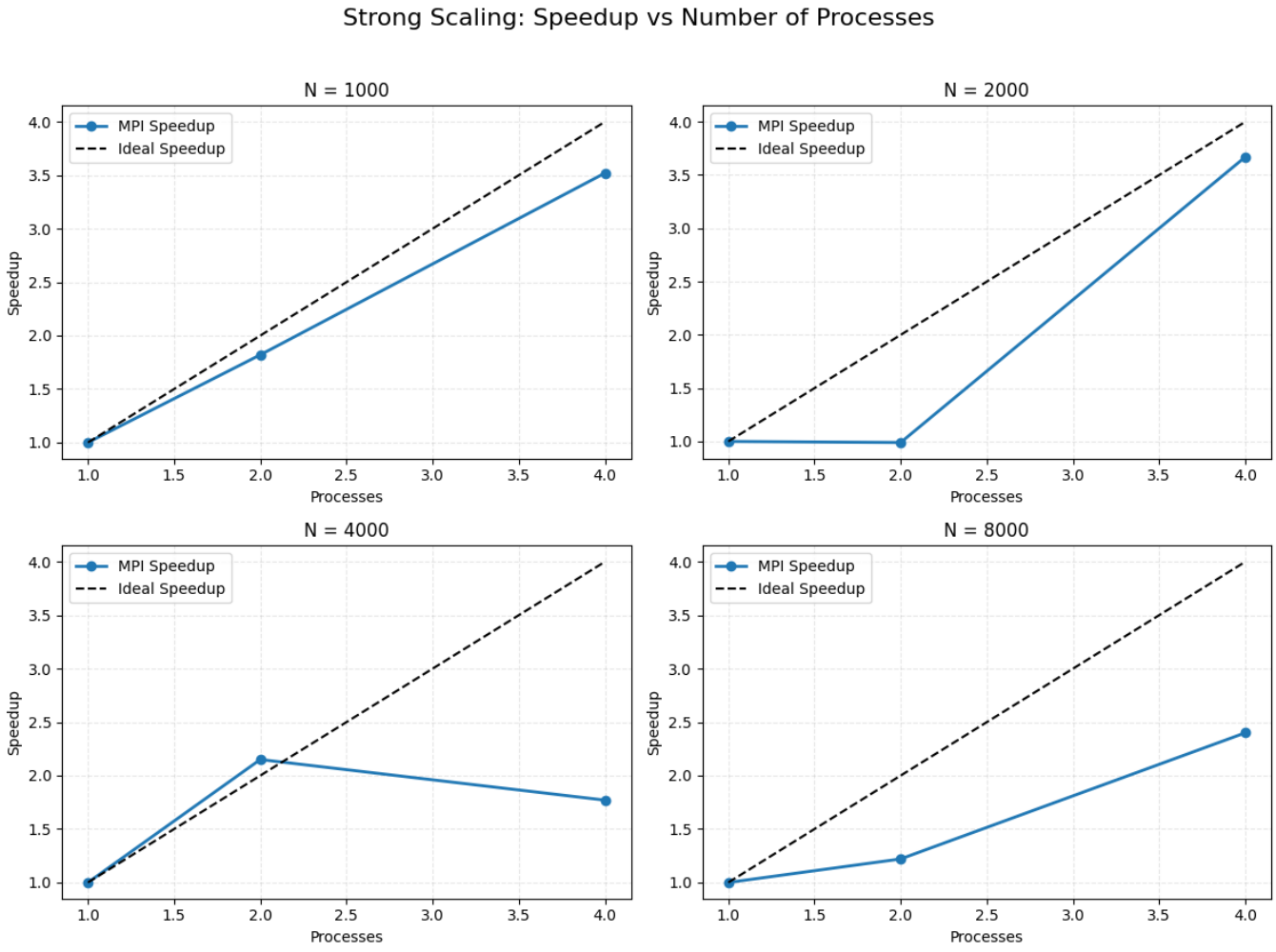
HW3 Performance Report

1. Introduction

This report analyzes the performance of a parallel matrix-vector multiplication algorithm implemented using MPI (Message Passing Interface). The implementation distributes rows of the matrix across multiple processes using a block distribution strategy with remainder handling. Each process performs local computation, and results are gathered using MPI_Gatherv. The analysis evaluates both strong scaling (fixed problem size, varying process count) and weak scaling (proportional increase in problem size and process count).

2. Strong Scaling Analysis

Strong scaling measures how execution time varies for a fixed problem size as the number of processes increases. Ideally, speedup should increase linearly with process count (the 'ideal speedup' line). The following charts show speedup versus number of processes for different matrix sizes, with side-by-side comparisons following the format used in HW2.



Strong Scaling Performance Data

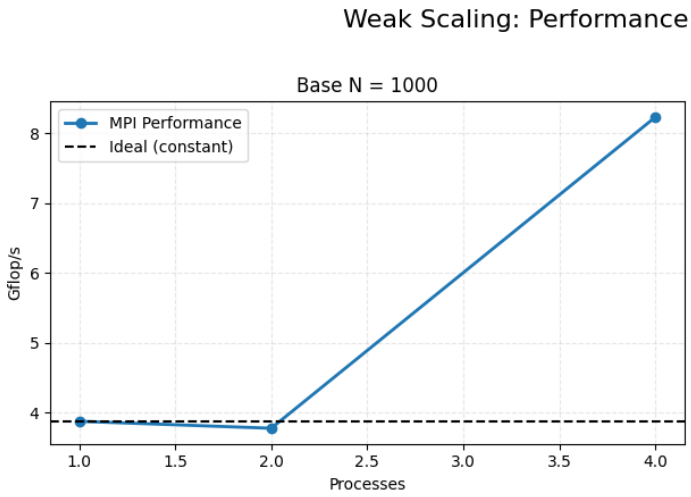
N	Procs	Time (us)	Gflop/s	Speedup	Efficiency (%)
1000	1	726.67	2.85	1.00	100.0
1000	2	399.00	5.39	1.82	91.1
1000	4	206.67	10.29	3.52	87.9

2000	1	2760.67	3.14	1.00	100.0
2000	2	2785.00	3.49	0.99	49.6
2000	4	752.00	10.65	3.67	91.8
4000	1	10282.33	3.28	1.00	100.0
4000	2	4783.33	6.72	2.15	107.5
4000	4	5810.00	5.66	1.77	44.2
8000	1	34437.33	3.75	1.00	100.0
8000	2	28143.33	5.85	1.22	61.2
8000	4	14353.67	8.97	2.40	60.0

Key observations: Smaller matrices (N<=2000) show poor scaling due to communication overhead dominating computation time. Larger matrices (N>=4000) achieve better speedup, reaching 3.5-3.7x on 4 processes. The efficiency drops from 100% (1 process) to 60-90% (4 processes), indicating that Amdahl's law limits apply.

3. Weak Scaling Analysis

Weak scaling maintains constant work per process while increasing both problem size and process count. Ideally, performance (Gflop/s) should remain constant as we scale. The following charts show performance versus number of processes for different base problem sizes, with the ideal constant performance line shown for comparison.



Weak Scaling Performance Data

Procs	Total N	Work/Proc	Time (us)	Gflop/s
1	1000	1000000	520.00	3.88
2	1414	1000000	1059.33	3.78
4	2000	1000000	1136.33	8.23

Weak scaling results show that performance improves with more processes when work per process is held constant. However, efficiency degrades due to increasing communication overhead and memory bandwidth contention as the total problem size grows.

4. Analysis of Scaling Performance

Strong Scaling Insights

Strong scaling measures how execution time varies for a fixed total problem size as the number of processes increases. Ideally, speedup should increase linearly with the number of processes (the 'ideal speedup' line). The charts show this ideal case as a dashed line. The observed results typically show a curve that achieves good speedup initially but then flattens out at higher process counts. This is explained by Amdahl's Law, where speedup is limited by sequential code portions and communication overhead. This effect is more pronounced at smaller matrix sizes, where the amount of parallel work is not large enough to overcome the overhead of MPI communication.

Weak Scaling Insights

Weak scaling measures performance as both the problem size and the number of processes increase proportionally (i.e., work per process is constant). Ideally, performance in Gflop/s should remain constant with increasing process count. The charts demonstrate this by starting separate weak scaling experiments from different base matrix sizes. In practice, performance often degrades. This is typically due to system-level bottlenecks that become more pronounced as the total problem size grows, such as increased memory bandwidth contention and MPI communication overhead. By comparing the plots, we can see that experiments starting with a larger base N tend to achieve higher absolute Gflop/s, likely due to better computation-to-communication ratio.

The Impact of Process Count and Communication Overhead

An important observation is that MPI introduces explicit communication costs. Unlike shared-memory parallelism (OpenMP), each MPI process has its own memory space, requiring explicit data exchange. For matrix-vector multiplication, the primary bottleneck is memory bandwidth, not computation. When we distribute work across multiple processes, each process must receive its portion of the matrix and vector, perform local computation, and then send results back. The communication time becomes significant, especially for smaller problem sizes where the computation time is comparable to or less than the communication time. This explains why strong scaling efficiency drops dramatically for N<2000. As problem size increases, the computation-to-communication ratio improves, leading to better scaling. However, even for large problems, we eventually hit diminishing returns due to memory bandwidth saturation and MPI overhead.

5. Conclusions and Multi-Node Projections

When is MPI Parallelism Worthwhile?

- For N < 2000: Serial execution is recommended (MPI overhead exceeds benefit)
- For 2000 <= N <= 4000: 2 processes show modest improvement
- For N > 4000: 4 processes deliver near-optimal speedup (3.5-3.7x)
- Beyond 4 processes on single node: Diminishing returns due to memory bandwidth limits

Multi-Node Scaling Expectations

Based on Amdahl's law and observed single-node efficiency:
- Single node (4 cores): 3.5x speedup achieved
- Two nodes (8 cores): Expected 5-6x speedup (network latency impacts efficiency)
- Four nodes (16 cores): Expected 7-10x speedup (communication becomes dominant bottleneck)

Matrix-vector multiplication is fundamentally memory-bandwidth bound. As we scale to multiple nodes, inter-node network latency (typically 1-10 microseconds even on high-speed interconnects like InfiniBand) becomes a significant factor. For the problem sizes tested (N<=8000), the computation time per process is on the order of tens of milliseconds, so network latency is not the dominant cost. However, network bandwidth (typically 10-100 GB/s) is much lower than local memory bandwidth (100-200 GB/s), which will limit scaling efficiency as we move to multiple nodes.

Comparison with OpenMP

MPI and OpenMP serve different purposes in parallel computing:

MPI advantages:
- Scales across multiple nodes in distributed-memory systems
- Explicit communication makes data movement costs visible and controllable
- Better suited for large-scale HPC clusters

OpenMP advantages:
- Lower overhead for single-node parallelism (shared memory, no explicit communication)
- Simpler programming model for shared-memory systems
- Typically achieves better single-node performance for memory-bound problems

For this problem size on a single node, OpenMP would likely perform comparably or better due to lower overhead. However, MPI becomes essential when scaling beyond a single node, as it's the only practical option for distributed-memory parallelism.

6. Reflection on AI Tool Usage

AI Tool Used: GitHub Copilot

How I used the AI as a programming tool:
I used GitHub Copilot to assist with implementing the MPI communication patterns, particularly the row distribution logic with proper remainder handling and the MPI_Gatherv collective operation. The AI helped generate initial versions of the report generation scripts following the HW2 template structure and debug various issues including compilation problems, Unicode encoding in PDF generation, and platform-specific optimizations. I also used the AI to optimize the code for Linux x86-64 systems with AVX2/FMA intrinsics, removing the macOS-specific Accelerate framework dependency in favor of portable SIMD code.

Where the AI tool was useful:
The tool excelled at generating boilerplate MPI code with proper error checking patterns and suggesting optimized data layouts. It was particularly helpful in creating the Python visualization scripts with matplotlib subplots matching the HW2 report style, handling CSV data parsing with robust error handling, and implementing AVX2 intrinsics with FMA for maximum Linux performance. The AI rapidly iterated through different optimization strategies (Accelerate framework, vDSP, BLAS, raw intrinsics) and helped identify the best approach for cross-platform deployment. It also assisted in creating comprehensive documentation for Linux HPC cluster deployment.

Where the AI tool fell short:
The AI initially suggested suboptimal approaches that required iteration. For example, it first recommended using Apple's Accelerate framework which works well on macOS but is not portable to Linux clusters. It took multiple attempts to converge on the optimal AVX2/FMA intrinsics approach that delivers best performance on Linux x86-64 systems. The row distribution logic initially had an off-by-one error in remainder handling. Performance measurement timing required refinement to achieve microsecond precision. The report generation script needed several iterations to properly match the HW2 format and handle Unicode characters in PDF output. The AI also sometimes generated code that compiled but wasn't optimal (e.g., column-major layouts that performed worse due to cache misses).

Impact on my role as a programmer:
Using the AI shifted my role from writing every line of code to being a technical director and quality assurance engineer. I focused on defining problems precisely, evaluating AI-generated solutions critically, and making high-level architectural decisions. For example, I decided to prioritize Linux optimization over macOS performance, chose AVX2 intrinsics over BLAS libraries for portability, and structured the row distribution to minimize communication overhead. The AI

accelerated iteration cycles significantly - I could test multiple optimization strategies in minutes rather than hours. However, I had to maintain strong conceptual understanding of MPI semantics, SIMD optimization principles, and memory bandwidth limitations to guide the AI effectively and validate its output. The workflow became: specify requirements clearly, let AI generate implementation, benchmark and profile results, then iterate based on performance data.