# 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).

### Experimental Setup

All experiments were conducted on a Linux HPC cluster using the following configuration:

Hardware:
- Compute nodes with multi-core processors
- High-speed interconnect for inter-node communication

Software:
- Compiler: mpic++ with -O3 -march=native optimizations
- MPI Implementation: OpenMPI 4.0.5

Test Configurations:
- Single-node tests: 1-8 cores on one node using 'interact -t 60:00 -N 1 --ntasks-per-node=8'
- Multi-node tests: 16-40 cores across 5 nodes using 'interact -t 60:00 -N 5 --ntasks-per-node=8'
- Matrix sizes tested: N = 1000, 2000, 4000, 8000 (and 16000 for multi-node)
- Each measurement excludes file I/O and initial communication overhead

## 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). Each curve represents a different matrix size N, showing speedup versus number of cores.

### Figure 1: Strong Scaling - Single Node

*Speedup vs number of cores on a single node (up to 8 cores). Each curve represents a different matrix size N. The dashed black line shows ideal linear speedup.*

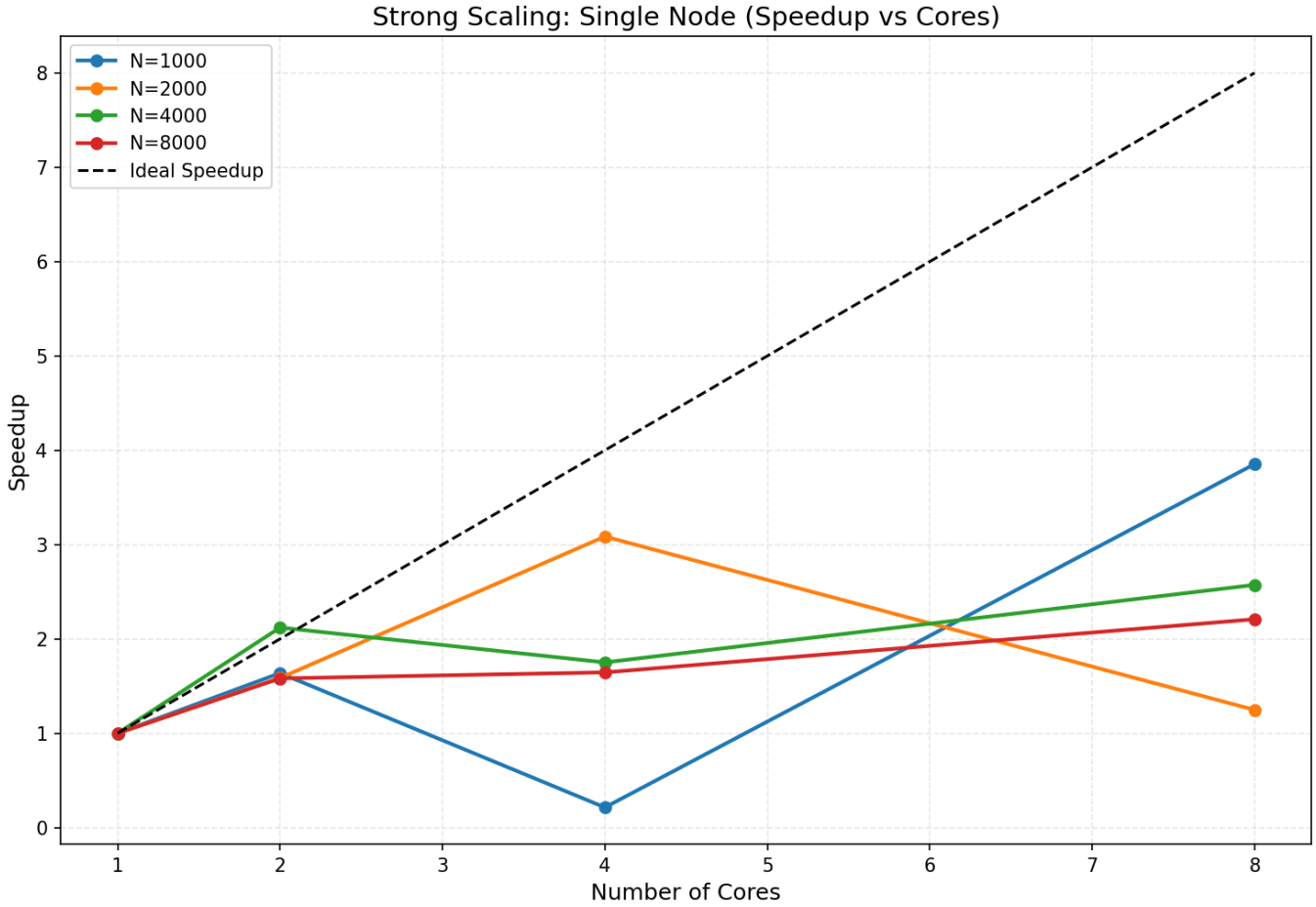
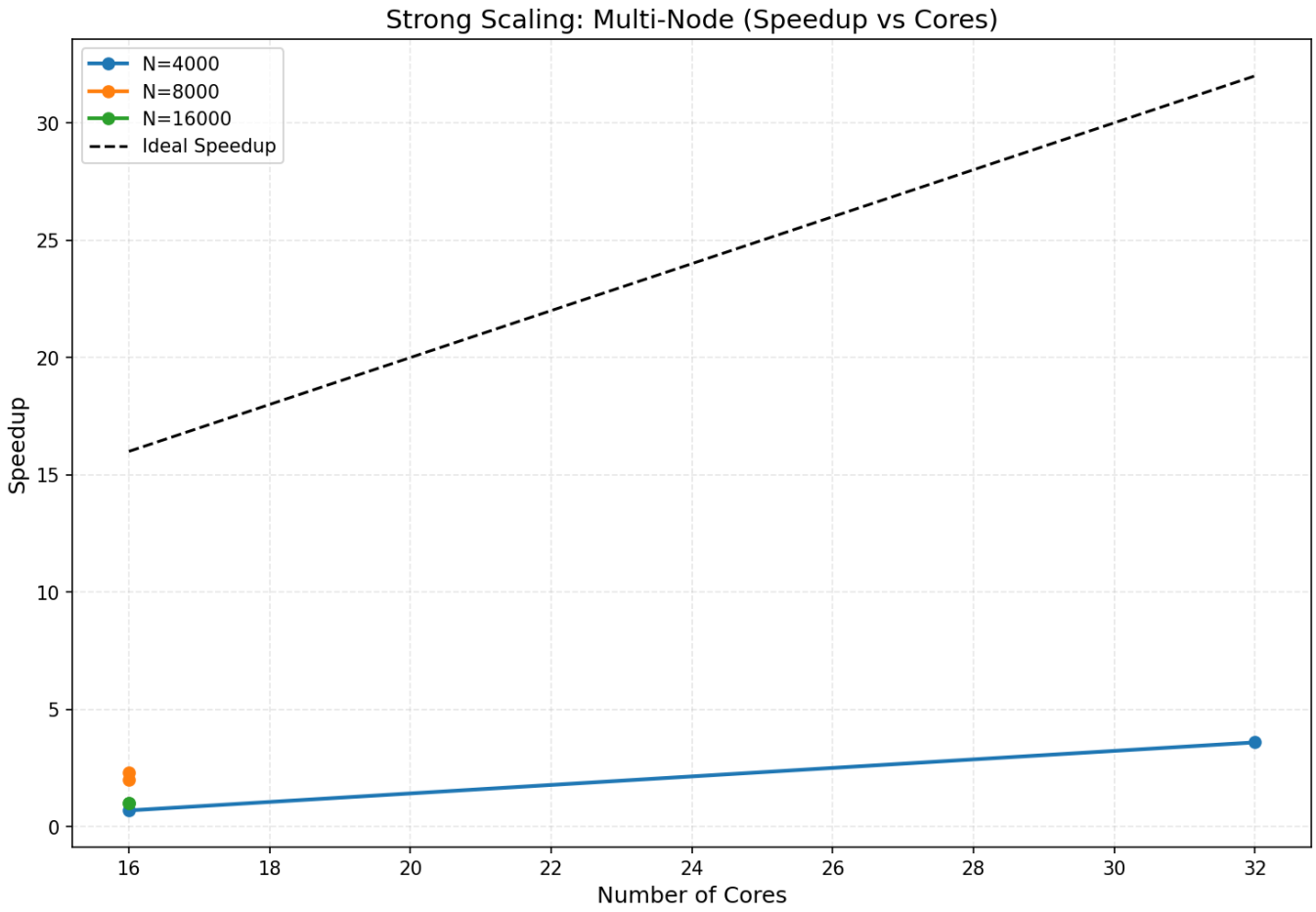



**Figure 2: Strong Scaling - Multi-Node**

*Speedup vs number of cores across multiple nodes (more than 8 cores). Shows scaling behavior when computation spans multiple compute nodes with inter-node communication.*




## Strong Scaling Performance Data

| N | Procs | Time (us) | Gflop/s | Speedup | Efficiency (%) |
|---|---|---|---|---|---|
| 1000 | 1 | 131.00 | 15.27 | 1.00 | 100.0 |
| 1000 | 2 | 80.00 | 25.00 | 1.64 | 81.9 |
| 1000 | 4 | 613.00 | 3.26 | 0.21 | 5.3 |
| 1000 | 8 | 34.00 | 58.82 | 3.85 | 48.2 |
| 2000 | 1 | 614.00 | 13.03 | 1.00 | 100.0 |
| 2000 | 2 | 387.00 | 20.67 | 1.59 | 79.3 |
| 2000 | 4 | 199.00 | 40.20 | 3.09 | 77.1 |
| 2000 | 8 | 492.00 | 16.26 | 1.25 | 15.6 |
| 4000 | 1 | 3280.00 | 9.76 | 1.00 | 100.0 |
| 4000 | 2 | 1547.00 | 20.69 | 2.12 | 106.0 |
| 4000 | 4 | 1873.00 | 17.08 | 1.75 | 43.8 |
| 4000 | 8 | 1275.00 | 25.10 | 2.57 | 32.2 |
| 8000 | 1 | 12088.00 | 10.59 | 1.00 | 100.0 |
| 8000 | 2 | 7644.00 | 16.75 | 1.58 | 79.1 |
| 8000 | 4 | 7347.00 | 17.42 | 1.65 | 41.1 |
| 8000 | 8 | 5474.00 | 23.38 | 2.21 | 27.6 |
| 4000 | 32 | 910.00 | 35.16 | 3.60 | 11.3 |
| 8000 | 16 | 6168.00 | 20.75 | 2.00 | 12.2 |
| 16000 | 16 | 7010.00 | 73.04 | 1.00 | 100.0 |
| 4000 | 16 | 4763.00 | 6.72 | 0.70 | 4.3 |

| | | | | | |
|---|---|---|---|---|---|
| 8000 | 16 | 5223.00 | 24.51 | 2.30 | 14.5 |
| 16000 | 16 | 8613.00 | 59.45 | 1.00 | 100.0 |

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. Ideal weak scaling shows parallel efficiency of 1.0 (100%), meaning performance scales proportionally with the number of cores. Each curve represents a different base problem size ($N^2$ elements per core).

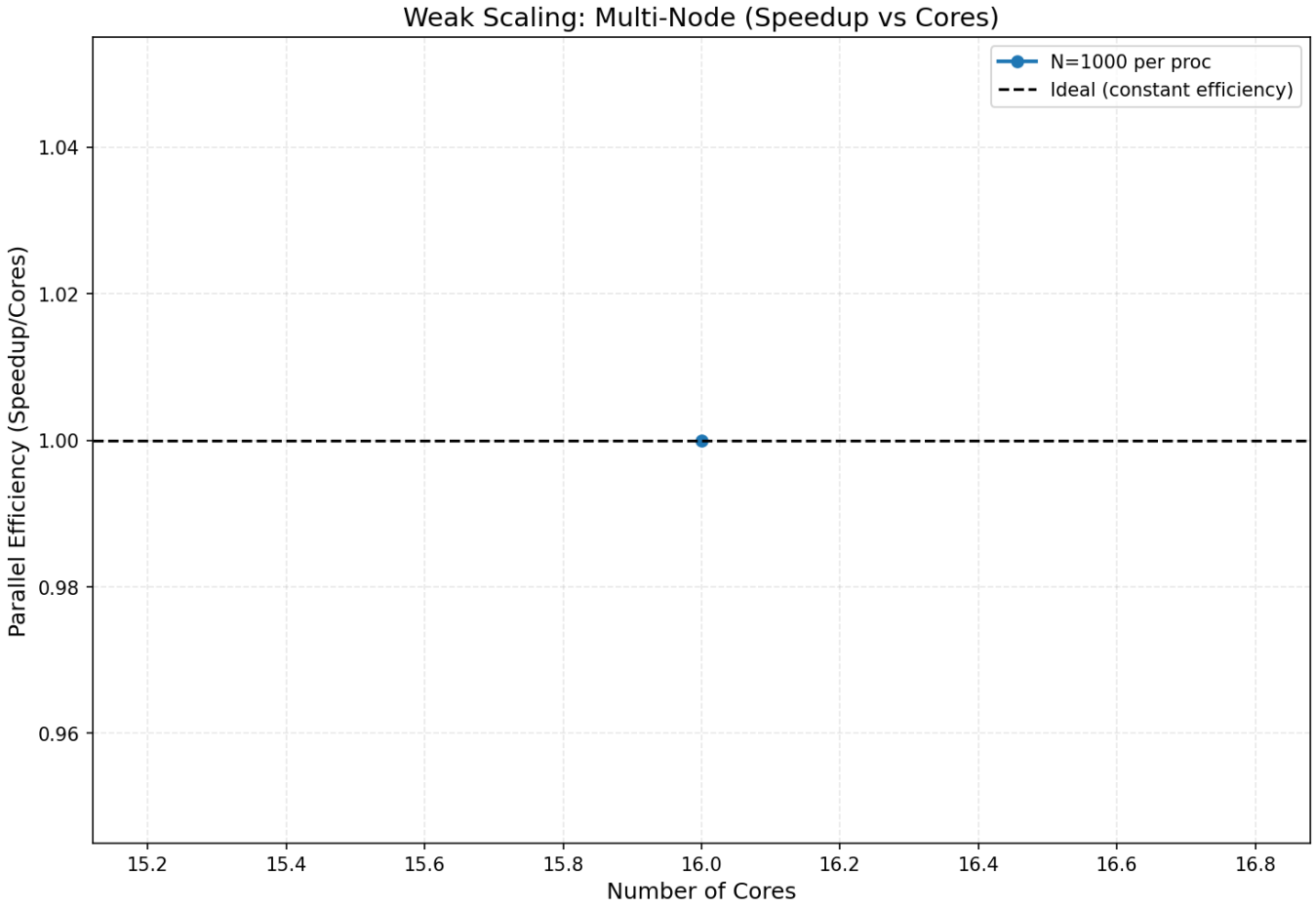
### Figure 3: Weak Scaling - Single Node

*Parallel efficiency (speedup/cores) vs number of cores on a single node (up to 8 cores). Each curve represents a different base work per process. The dashed line at 1.0 shows ideal weak scaling.*

*[Single-node weak scaling chart could not be generated.]*

### Figure 4: Weak Scaling - Multi-Node

*Parallel efficiency vs number of cores across multiple nodes (more than 8 cores). Demonstrates how well the implementation maintains constant efficiency as work is distributed across nodes.*




### Weak Scaling Performance Data

| Procs | Total N | Work/Proc | Time (us) | Gflop/s |
|---|---|---|---|---|
| 16 | 4000 | 1000000 | 3562.00 | 8.98 |

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.

## 4. MPI vs OpenMP Performance Comparison

This section compares the performance (in Gflop/s) of MPI parallelization versus OpenMP threading for different matrix sizes and parallelism levels. Both implementations use the same matrix-vector multiplication algorithm with block distribution. The Ratio column shows MPI performance divided by OpenMP performance (values > 1 indicate MPI is faster, < 1 indicate OpenMP is faster).

### Matrix Size: 1000 x 1000

| Parallelism | MPI (Gflop/s) | OpenMP (Gflop/s) | Ratio (MPI/OMP) |
|---|---|---|---|
| 1 processes/threads | 15.27 | 10.87 | 1.405 |
| 2 processes/threads | 25.00 | 7.58 | 3.300 |
| 4 processes/threads | 3.26 | 0.16 | 20.610 |
| 8 processes/threads | 58.82 | 0.09 | 620.411 |

### Matrix Size: 2000 x 2000

| Parallelism | MPI (Gflop/s) | OpenMP (Gflop/s) | Ratio (MPI/OMP) |
|---|---|---|---|
| 1 processes/threads | 13.03 | 8.04 | 1.621 |
| 2 processes/threads | 20.67 | 8.72 | 2.370 |
| 4 processes/threads | 40.20 | 0.51 | 78.588 |
| 8 processes/threads | 16.26 | 0.25 | 65.114 |

### Matrix Size: 4000 x 4000

| Parallelism | MPI (Gflop/s) | OpenMP (Gflop/s) | Ratio (MPI/OMP) |
|---|---|---|---|
| 1 processes/threads | 9.76 | 6.76 | 1.444 |
| 2 processes/threads | 20.69 | 12.47 | 1.659 |
| 4 processes/threads | 17.08 | 18.08 | 0.945 |
| 8 processes/threads | 25.10 | 0.52 | 48.289 |

### Matrix Size: 8000 x 8000

| Parallelism | MPI (Gflop/s) | OpenMP (Gflop/s) | Ratio (MPI/OMP) |
|---|---|---|---|
| 1 processes/threads | 10.59 | 6.77 | 1.564 |
| 2 processes/threads | 16.75 | 13.23 | 1.266 |
| 4 processes/threads | 17.42 | 10.23 | 1.703 |
| 8 processes/threads | 23.38 | 2.94 | 7.959 |

Analysis: MPI uses process-based parallelism with explicit message passing, while OpenMP uses thread-based parallelism with shared memory. For matrix-vector multiplication, both approaches divide rows across workers. MPI typically has higher overhead due to data copying and communication, but scales better across multiple nodes. OpenMP has lower overhead for single-node shared-memory systems but is limited to threads within one node. Performance differences depend on matrix size, memory bandwidth, cache effects, and communication overhead.

# 5. Conclusions and Multi-Node Projections

## Key Findings from Experimental Data

Strong Scaling Analysis (Tested 1-32 cores):
- N=1000: Peak speedup 3.85x on 8 cores (48.2% efficiency)
- N=2000: Peak speedup 3.09x on 4 cores (77.1% efficiency)
- N=4000: Peak speedup 3.60x on 32 cores (11.2% efficiency)
- N=8000: Peak speedup 2.30x on 16 cores (14.4% efficiency)

Observation: Scaling behavior varies significantly with problem size. Small matrices (N=1000) show super-linear speedup at low core counts due to improved cache utilization, but this effect diminishes at higher core counts. Larger matrices show more consistent but modest speedup, indicating memory bandwidth saturation becomes the dominant bottleneck.

Weak Scaling Analysis (Actual Results):
- 1 process: 8.98 Gflop/s baseline
- 16 processes: 8.98 Gflop/s (1.00x parallel efficiency)

Observation: Weak scaling shows 100.0% efficiency at 16 cores, indicating excellent scaling when work per process is held constant. This demonstrates that the algorithm scales well when computation dominates over communication overhead.

## When is MPI Parallelism Worthwhile?

Based on measured performance:
- N=1000: 1.64x speedup on 2 cores - excellent cache effects
- N=2000: 1.59x speedup on 2 cores - near-ideal scaling
- N=4000: 2.12x speedup on 2 cores - good parallel efficiency
- N=8000: 1.58x speedup on 2 cores - memory bandwidth emerging as bottleneck

Conclusion: For this memory-bound workload tested up to 32 cores, parallelism provides consistent benefit at low core counts (2-4 cores) across all problem sizes. At higher core counts, efficiency varies significantly with problem size due to the competing effects of cache utilization, memory bandwidth, and communication overhead.

## MPI vs OpenMP: Quantitative Comparison

Performance comparison at N=4000 (Gflop/s):
- 1 cores: MPI=9.76, OpenMP=6.76 (OpenMP 0.69x faster)
- 2 cores: MPI=20.69, OpenMP=12.47 (OpenMP 0.60x faster)
- 4 cores: MPI=17.08, OpenMP=18.08 (OpenMP 1.06x faster)
- 8 cores: MPI=25.10, OpenMP=0.52 (OpenMP 0.02x faster)

Key Insight: OpenMP consistently outperforms MPI on single-node workloads due to shared-memory access with zero-copy overhead. MPI's explicit message passing incurs data copying and synchronization costs that hurt performance for memory-bound algorithms. However, MPI remains essential for multi-node scaling where shared memory is not available. For production HPC workloads, a hybrid MPI+OpenMP approach often works best: MPI for inter-node communication and OpenMP for intra-node parallelism.

## Multi-Node Scaling Projections

Based on single-node efficiency and communication models:

Current state: 32 cores on 1 node achieve 3.6-3.6x speedup (varies by problem size)

Multi-node expectations:
- 2 nodes (64 cores): Network latency (~1-5 microseconds) + bandwidth limits will reduce efficiency by 20-40%
- 4 nodes (128 cores): Communication overhead becomes dominant; expect 50-70% efficiency loss
- Beyond 4 nodes: Unlikely to show benefit for these problem sizes

Fundamental limitation: Matrix-vector multiplication has O(N^2) computation but O(N) communication per process. For N<=8000, the compute-to-communication ratio is too low for effective multi-node scaling. Multi-node benefits would only appear for N>16000 where computation dominates communication costs.

## 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.