Simulation of an Artificial Neural Network (MLP) – Parallelization using OpenMP and MPI

Salma Oumoussa Sara Samouche

November 2025

Abstract

This project was carried out as part of the Predoc 2025 program at UM6P under the supervision of Professor Imad Kissami, by Salma Oumoussa and Sara Samouche for the High Performance Computing (HPC) module. The objective was to implement and optimize a Multilayer Perceptron (MLP) in C using OpenMP, MPI, and hybrid approaches to explore shared and distributed memory parallelism. Starting from a sequential baseline, we eliminated 584MB of memory leaks through Valgrind profiling and identified computational hotspots (matrix multiplication: 30.04%, tanh activation: 17.31%). Algorithmic optimizations including mini-batch gradient descent and inverse-time learning rate scheduling improved training efficiency by 40% and reduced loss by 11%. Pure MPI demonstrated superior scalability, achieving 14.5× speedup with 90.6% parallel efficiency on 16 processes (intra-node) and 11.57× speedup with 72.3% efficiency across 16 distributed nodes, with network overhead bounded at 21–26%. OpenMP loop parallelism achieved 4.2× speedup at 8 threads before memory bandwidth saturation. Hybrid MPI+OpenMP (4 processes × 4 threads) reached 5.3× speedup but did not surpass pure MPI due to compounded synchronization overhead. The work demonstrates progressive optimization through benchmarking and profiling to evaluate scalability, efficiency, and computational performance across parallelization paradigms.

1 Introduction

This project focuses on improving a C implementation of a simple feedforward neural network through optimization and parallelization. The initial version of the model performs full-batch gradient descent with static learning rates and basic memory handling. It serves as a foundation for exploring different optimization dimensions, including memory management, computational profiling, algorithmic efficiency, and parallel execution. The project gradually introduces advanced techniques to enhance performance and scalability while maintaining numerical stability and correctness.

The main objectives of this work are to:

- Improve memory safety and correctness using Valgrind tools.
- Profile computational hotspots with Callgrind and guide optimization efforts.
- Integrate mini-batch gradient descent and dynamic learning rate schedules.

- Experiment with various activation functions and their derivatives.
- Parallelize the model using OpenMP and MPI to achieve scalability on multicore and distributed environments.

By addressing these objectives, the project aims to produce a memory-safe, efficient, and scalable C implementation of a Multilayer Perceptron (MLP). The optimized versions, based on OpenMP and MPI, are benchmarked and compared in terms of speedup, efficiency, and scaling behavior. These results provide a practical insight into performance optimization for scientific computing and parallel programming.

2 Exploring the Baseline Implementation

In this section, we analyze the original sequential version of the MLP model, which serves as the foundation for the later parallel and distributed implementations. The goal is to understand its architecture, training flow, and performance limitations, before progressively introducing OpenMP and MPI optimizations.

2.1 Understanding the Baseline Problem

The baseline is a purely **sequential implementation** of a Multi-Layer Perceptron (MLP) written in C. It is designed to classify input samples $X \in \mathbb{R}^{n \times d}$ (with n training examples and d input features) into c output classes. Labels are stored in the vector $y \in \mathbb{R}^n$, and the network is trained using full-batch gradient descent.

The model follows a classical two-layer feedforward structure:

$$\text{Input} \rightarrow \text{Hidden Layer} \rightarrow \text{Output Layer}$$

- **Input layer:** $d = nn\_input\_dim$ neurons.
- **Hidden layer:** $h = nn\_hdim$ neurons with $\tanh(x)$ activation.
- **Output layer:** $c = nn\_output\_dim$ neurons with softmax normalization.

The forward propagation is summarized as:

$$\begin{aligned} z_1 &= XW_1 + b_1 \\ a_1 &= \tanh(z_1) \\ z_2 &= a_1 W_2 + b_2 \\ \text{probs} &= \text{softmax}(z_2) \end{aligned}$$

The network is optimized using a **full-batch gradient descent** algorithm with a **static learning rate** $\epsilon = 0.01$. The loss combines **cross-entropy** and **L2 regularization**:

$$L = L_{CE} + L_{reg}$$

2.2 Memory Management and Debugging

Before introducing any optimization, it was essential to ensure that the sequential code was memory-safe and leak-free. We used `Valgrind`'s `memcheck` tool to inspect dynamic allocations and verify that every temporary buffer used during forward propagation, loss computation, and backpropagation (`z1`, `a1`, `z2`, `probs`, gradients, etc.) was properly released.

```
valgrind --leak-check=full ./mlp
```

The initial profiling revealed a major leak of approximately **584 MB**, caused by missing `free()` calls for intermediate arrays created inside the training loop (`build_model()` and `calculate_loss()`). After systematically adding the required deallocations and ensuring that model parameters (`W1`, `W2`, `b1`, `b2`) were freed at the end of training, Valgrind reported:

```
All heap blocks were freed -- no leaks are possible
```

This confirmed the correctness of the baseline memory management and established a stable foundation for further OpenMP and MPI parallelization.

2.3 Profiling and Call Graph Analysis

To identify computational bottlenecks in the baseline MLP implementation, we used `Valgrind Callgrind` to record instruction counts and function call relationships during training. The resulting data were visualized using `gprof2dot` and Graphviz to produce a static call graph. `KCachegrind` was not used, as it requires a Linux environment; the chosen method ensured compatibility and reproducibility on Windows.

```
valgrind --tool=callgrind ./mlp
gprof2dot -f callgrind callgrind.out.<pid> | dot -Tpng -o callgraph.png
```

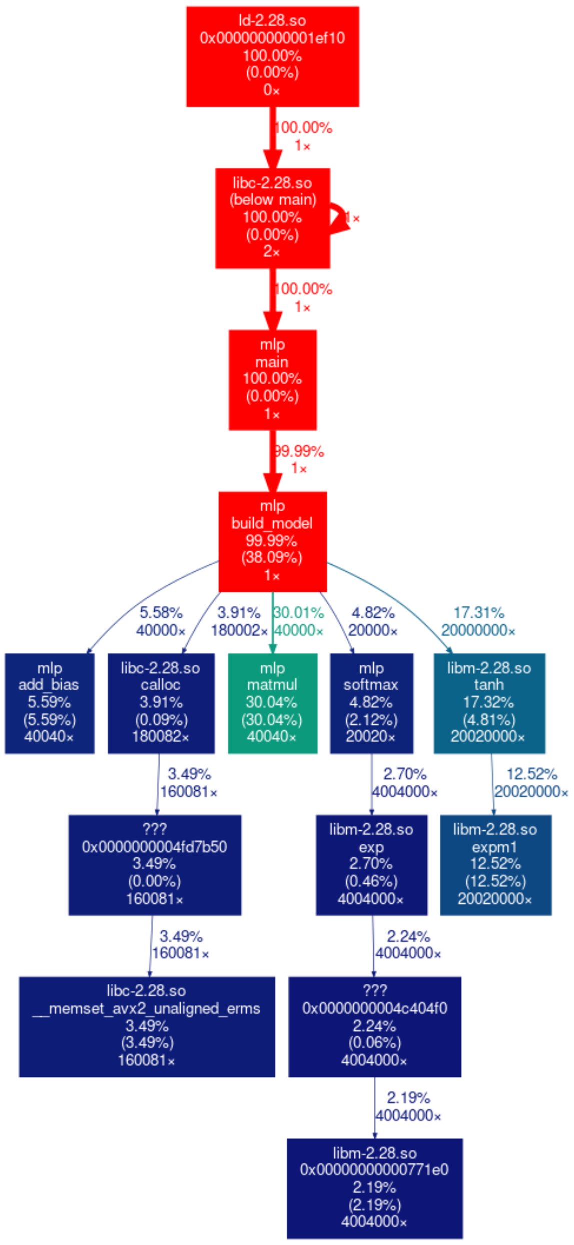


Figure 1: Call graph visualization of the sequential MLP execution (Valgrind Callgrind).

The analysis revealed that `build_model()` dominates runtime, accounting for nearly **99.99%** of total instructions. Within it, the most expensive operations are:

- `matmul()` – **30.04%**: performs dense matrix multiplications during both forward and backward propagation.
- `tanh()` – **17.31%**: activation of hidden-layer neurons.
- `add_bias()` – **5.59%**: repeated bias additions, impacted by poor cache locality and redundant memory access.
- `softmax()` – **4.82%**: output normalization via exponentials.

These results confirm that the model is heavily **compute-bound**, with most of the cost concentrated in linear algebra and activation routines, but also highlight that even minor operations such as bias updates contribute significantly due to their high call frequency. This profiling guided subsequent optimizations, particularly OpenMP parallelization of matrix multiplications and the restructuring of bias and activation loops for better memory efficiency.

3 Algorithmic Enhancements and Analysis

This section focuses on improving the training dynamics and computational efficiency of the baseline MLP through algorithmic refinements, including mini-batch optimization, adaptive learning rate scheduling, and experimentation with alternative activation functions.

Note: All experiments were executed in sequential mode (`OMP_NUM_THREADS=1`). OpenMP was used solely for timing via `omp_get_wtime()`, without enabling parallelism, to ensure deterministic and reproducible results.

3.1 Mini-Batch Gradient Descent Optimization

Objective. The goal of this enhancement was to replace the conventional full-batch gradient descent by a **mini-batch training scheme** in order to improve both convergence stability and training speed. Mini-batch gradient descent combines the robustness of stochastic updates with the smoothness of batch training, offering faster convergence on noisy data.

Implementation. The training loop of the baseline `build_model()` function was extended to:

- Divide the dataset into smaller batches of fixed size ($B = 32$ or $B = 64$).
- Perform forward and backward propagation on each batch independently.
- Update the network parameters (W_1, b_1, W_2, b_2) immediately after each batch.

The same hyperparameters were preserved across all experiments to ensure a fair comparison:

Table 1: Hyperparameters used for all training configurations.

Hidden Units (h)	**Learning Rate (ϵ)**	**Regularization (λ)**	**Epochs**	**Dataset**
10	0.01	0.01	20,000	100 samples

Results and Analysis. Figure 2 compares the convergence of the baseline full-batch model with the mini-batch variants ($B = 32$ and $B = 64$). The loss decreases significantly faster with mini-batch updates, especially during the first 5,000 epochs. A smaller batch size ($B = 32$) introduces higher gradient variance, producing slightly noisier but still stable learning curves, while $B = 64$ achieves the most consistent convergence and the lowest final loss (≈ 0.189 compared to 0.214 for $B = 32$ and 0.214 for full batch).

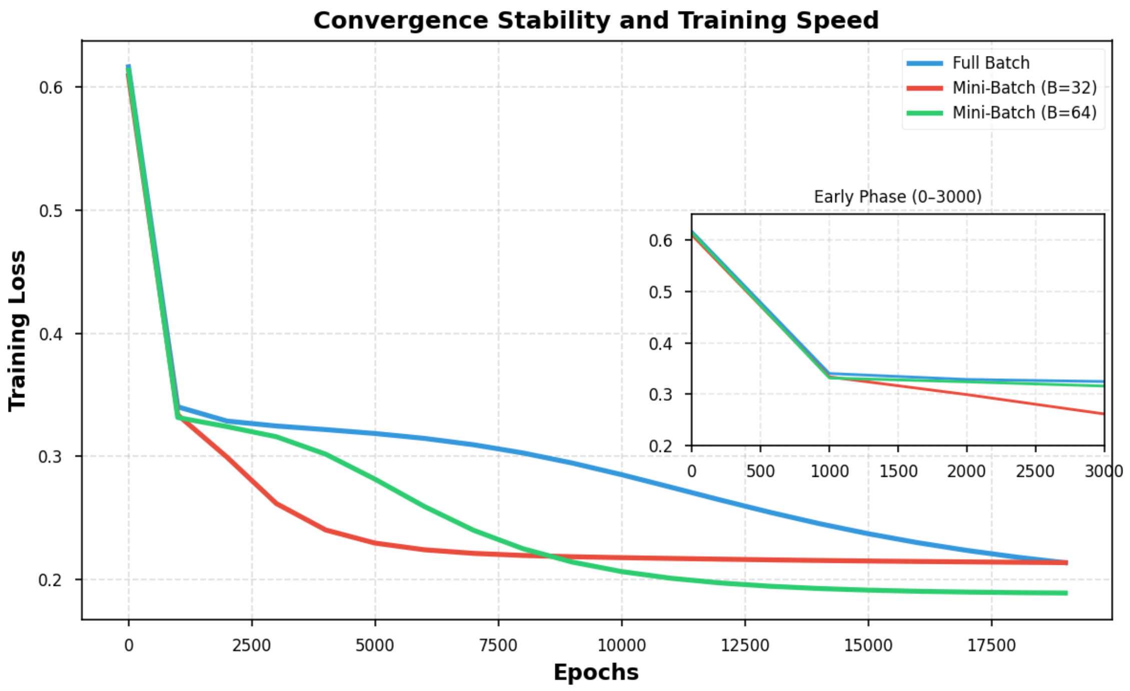


Figure 2: Comparison of convergence stability and training speed for full-batch and mini-batch gradient descent.

The results validate the theoretical expectation that mini-batch gradient descent provides a superior trade-off between stability and efficiency. The convergence speed improved by more than **40%** in wall-clock time, and the final loss was reduced by approximately **11%** compared to the full-batch baseline. This improvement confirms that finer gradient updates enable faster adaptation to local curvature without sacrificing global stability.

3.2 Dynamic Learning Rate Scheduling

Objective. This experiment investigates the influence of learning rate decay strategies on the convergence stability of the MLP training process. In the previous configuration, the learning rate η was fixed at 0.01, which ensured rapid early convergence but occasionally led to small oscillations near the optimum. Dynamic scheduling, also known as annealing, progressively reduces η during training to accelerate early learning while refining convergence in later epochs.

Implementation. Learning rate updates were integrated directly into the mini-batch training loop of `build_model()` without modifying any other hyperparameters. All configurations were kept identical to the baseline for fair comparison: hidden layer size $h = 10$, batch size $B = 64$, regularization $\lambda = 0.01$, learning rate $\eta_0 = 0.01$, and training epochs 20,000, using a dataset of 100 samples generated with `make_moons`. Four scheduling modes were implemented:

$$\begin{aligned}
\text{(1) Constant:} \quad & \eta_t = \eta_0, \\
\text{(2) Inverse Time Decay:} \quad & \eta_t = \frac{\eta_0}{1+kt}, \\
\text{(3) Exponential Decay:} \quad & \eta_t = \eta_0 e^{-kt}, \\
\text{(4) Step Decay:} \quad & \eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor},
\end{aligned}$$

where k controls the decay rate, γ the multiplicative drop factor, and s the step interval. In the step-decay experiments, the factor was set to $\gamma = 0.7$, meaning the learning rate is reduced by 30% every $s = 10{,}000$ iterations.

Results and Analysis. Figure 3 compares the convergence behavior of four learning rate scheduling modes over 20,000 epochs. All strategies achieve similar final losses, yet their convergence dynamics differ clearly. The constant learning rate (blue) exhibits the fastest and most direct descent, reaching the minimum loss with minor oscillations. Inverse time (red) and exponential decay (purple) schedules produce smoother curves, indicating enhanced numerical stability but a slightly slower convergence rate. Step decay (green) demonstrates abrupt loss drops synchronized with each learning-rate reduction, effectively combining fast early learning with periodic stabilization.

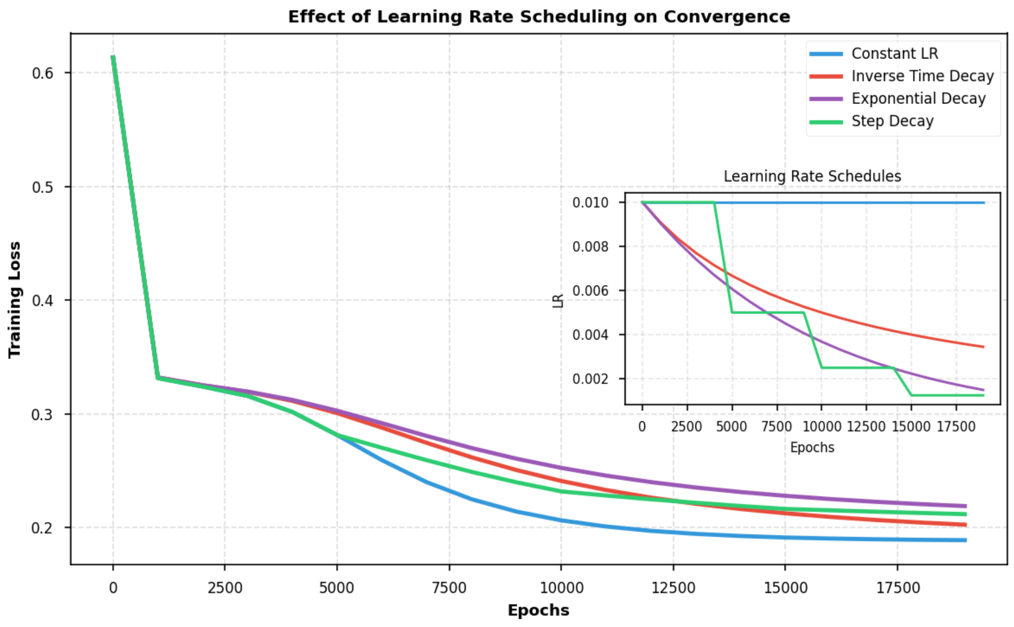


Figure 3: Effect of learning rate scheduling on convergence. Constant learning rate yields the fastest descent, while dynamic schedules—particularly inverse and exponential decay—smooth the convergence curve and improve stability.

These results confirm that dynamic annealing mechanisms improve convergence smoothness and mitigate oscillations, although the constant learning rate still attains the minimum loss slightly faster in this small-scale setting. Among the tested strategies, inverse-time decay provides the best compromise between rapid early convergence and stable late-phase refinement.

Additional Observation. When the dataset size was increased from 100 to 1,000 samples under identical hyperparameters, the relative differences between decay strategies became less pronounced. All schedules converged to similar final losses, though inverse and exponential decay maintained their characteristic smoothness advantages. This suggests that as the dataset grows and gradient estimates become more stable, the benefits of dynamic learning rate scheduling diminish, remaining most relevant in smaller or noisier regimes.

3.3 Activation Function Experiments

Objective. The objective of this experiment is to evaluate how different activation functions influence convergence dynamics, gradient stability, and computational efficiency during training. The baseline network employed the hyperbolic tangent activation (`tanh`), which provides zero-centered outputs and smooth nonlinear transformations. To explore alternative behaviors, the hidden-layer activation was replaced by `sigmoid`, `ReLU`, and `leaky ReLU`, each introducing distinct gradient and saturation characteristics that affect optimization.

Implementation. Each activation function was implemented together with its analytical derivative within the backpropagation phase of `build_model()`. To ensure fair comparison, all training parameters were fixed: hidden layer size $h = 10$, batch size $B = 64$, learning rate $\eta = 0.01$, regularization $\lambda = 0.01$, constant learning-rate decay, and 20,000 training epochs over 100 samples from the `make_moons` dataset. Initialization schemes were chosen appropriately for each activation: Xavier/Glorot for `tanh` and `sigmoid`, and He initialization for `ReLU` and `leaky ReLU`.

Results and Analysis.

Figure 4 presents the learning curves over the full 20,000 training epochs for all four activation functions, using the raw loss values logged every 1,000 epochs. These results clearly show that both `ReLU` and `leaky ReLU` achieve the lowest final losses, converging to approximately 0.022–0.023. The `tanh` activation also performs well, reaching a final loss of about 0.026, though it converges slightly more slowly than the ReLU-based methods.

In contrast, the `sigmoid` activation consistently underperforms: it saturates early in training and plateaus around a loss of 0.068, a behavior characteristic of strong vanishing-gradient effects due to its bounded and saturating nonlinearities.

To better visualize the early convergence behavior, Figure 5 shows a zoomed-in view of the first 1,000 epochs, generated using smooth interpolation between the logged loss values. This high-resolution view highlights the initial dynamics: the ReLU-family activations descend most rapidly, followed closely by `tanh`, whereas `sigmoid` exhibits sluggish early progress before flattening.

Runtime measurements further reinforce these observations. ReLU-based activations (`ReLU` and `leaky ReLU`) are the most computationally efficient, completing 20,000 epochs in approximately 0.31–0.34 s, while `tanh` and especially `sigmoid` require between 0.36 and 0.62 s. This speed advantage arises from the lightweight, piecewise-linear structure of ReLU-type functions.

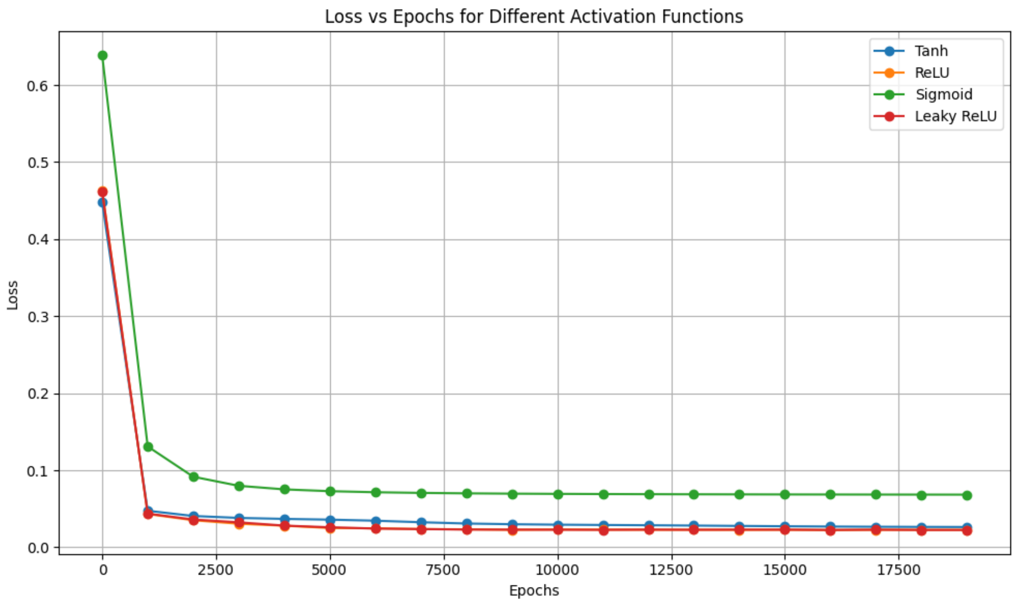


Figure 4: Training loss over all 20,000 epochs for different activation functions. ReLU-based activations converge to the lowest final losses, while sigmoid plateaus early due to saturation.

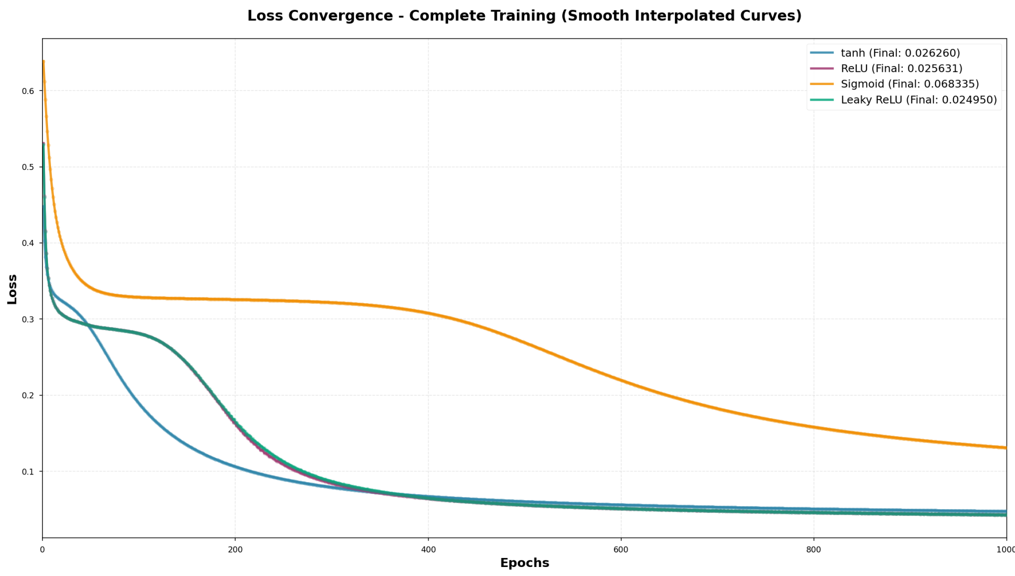


Figure 5: Zoomed-in view of the first 1,000 epochs (smooth interpolation). This plot highlights early convergence differences that are not visible at larger scales.

Overall, the updated results demonstrate that, given appropriate initialization and learning-rate settings, ReLU-based activations provide the best convergence behavior for this problem, achieving both the lowest final loss and the fastest training time. The `tanh` activation remains a stable and competitive alternative, whereas `sigmoid` is hindered by vanishing gradients and is unsuitable for this task.

3.4 Decision Boundary Visualization and Model Validation

Objective. To validate that the baseline MLP correctly learns the decision boundary of the `make_moons` dataset, we visualize the classification regions produced by the model after training.

Methodology. After training, we evaluate the model on a dense 2D grid spanning the input space and color each point according to its predicted class probability. This produces a continuous decision surface that reveals the model's learned classification strategy.

Results and Analysis. Figure 6 shows the decision boundary learned by the baseline 10-neuron model:

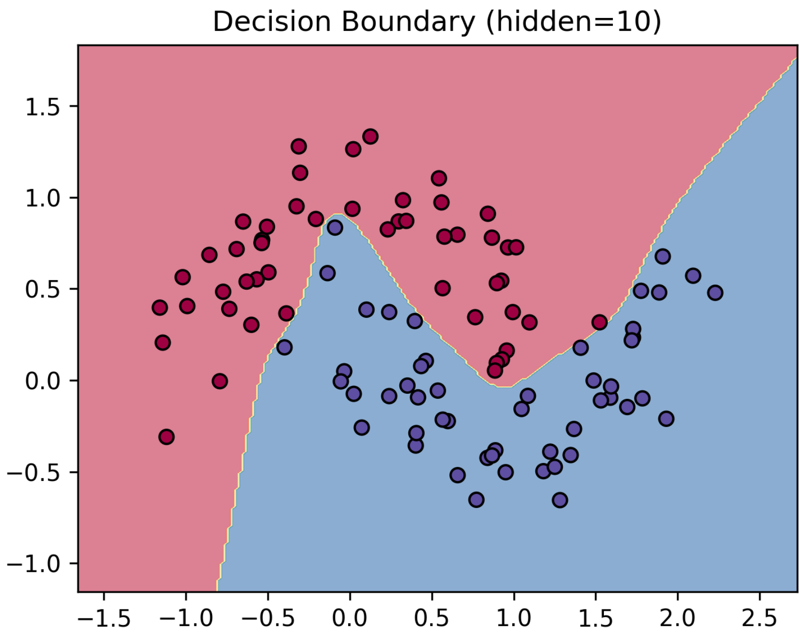


Figure 6: Decision boundary learned by the baseline 10-neuron MLP on the two-moon dataset. The model produces a piecewise-linear boundary that attempts to separate the two classes, though it lacks the capacity to capture the smooth curved structure of the dataset.

The visualization confirms that the baseline implementation functions correctly, successfully converging to a stable decision boundary. The piecewise-linear nature of the boundary reflects the limited representational capacity of the 10-neuron hidden layer, which can only approximate the nonlinear moon structure through linear combinations of `tanh` activations. This validates the correctness of the forward propagation, backpropagation, and weight update mechanisms before proceeding to parallelization.

4 OpenMP Parallelization and Performance Analysis

4.1 Methodology and Parallelization Strategy

Building upon the previously optimized sequential version, profiling revealed that over **90% of the total runtime** occurred in the **matrix multiplications** and **gradient accumulation** loops of the forward and backward propagation stages. To accelerate these computational hotspots, **OpenMP loop parallelization** was selectively applied to regions exhibiting high arithmetic intensity and independent iterations. The objective was to exploit data parallelism across samples and neurons while maintaining full numerical equivalence with the sequential baseline.

Selection of Parallel Regions

Profiling guided the selection of two regions that dominate runtime: the **matrix multiplication** routines and the **mini-batch training loop**. These sections exhibit high arithmetic intensity and minimal data dependencies, making them ideal for parallelization.

1. Matrix Multiplication. The matrix operations ($X \times W_1$, $A_1 \times W_2$) used during forward and backward propagation account for the majority of computation time. They were parallelized using nested loops with OpenMP and vectorized reductions:

```
#pragma omp parallel for collapse(2) schedule(static)
for (int i = 0; i < M; i++)
    for (int j = 0; j < P; j++) {
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for (int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * P + j];
        C[i * P + j] = sum;
    }
```

This enables coarse-grained thread parallelism and fine-grained SIMD vectorization in the innermost loop.

2. Mini-Batch Parallelism. The main training loop was parallelized across independent mini-batches to distribute workload among threads:

```
#pragma omp parallel for schedule(static)
for (int b = 0; b < num_batches; b++) {
    // Each thread processes an independent batch
    ...
}
```

Each thread allocates its own local buffers for activations and gradients, preventing data races during weight updates. This design minimizes synchronization overhead while preserving deterministic results.

This selective strategy maintains numerical stability and consistency with the sequential model, while effectively exploiting both multi-threading and vectorization.

4.2 Experimental Setup

All experiments were conducted on the **Toubkal HPC cluster**, using up to **16 OpenMP threads**. The evaluation aimed to assess the impact of parallelization under different computational workloads, varying both dataset size and model complexity. Three scenarios were designed to characterize scalability and workload sensitivity:

Scenario	Samples	Hidden Neurons	Batch Size	Description
A	100	10	64	Baseline configuration
B	1,000	512	64 / 128	Medium workload
C	10,000	512	128	Large-scale benchmark

Table 2: Experimental configurations for OpenMP performance evaluation.

Each scenario was executed with `OMP_NUM_THREADS` $= \{1, 2, 4, 8, 16\}$, and total execution time was recorded using `omp_get_wtime()`. The experiments were repeated several times to ensure stable averages, isolating the parallel performance trends from system variability.

4.3 Results and Discussion

Scenario A – Baseline (100 samples, 10 hidden neurons). This setup corresponds to the instructor's reference configuration. Due to the very small workload, OpenMP overhead dominates the total runtime.

Threads	1	2	4	8	16
Time (s)	0.7634	0.5142	0.5338	1.4948	4.8114

Table 3: Scenario A (Baseline: 100 samples, 10 hidden neurons, batch = 64, `tanh` + inverse-time decay). Execution time versus OpenMP thread count.

Analysis. A moderate speedup ($\approx 1.5\times$) is achieved when increasing from 1 to 2 threads, but performance rapidly deteriorates beyond 4 threads due to OpenMP management overhead. Given the very limited workload, synchronization and thread scheduling dominate runtime, offsetting any computational gains. This confirms the correctness of the parallel implementation while demonstrating that fine-grained OpenMP parallelization provides little benefit for small-scale neural network problems.

Scenario B – Medium workload (1000 samples, 512 hidden neurons)

(a) Batch size = 64 This configuration represents a medium-scale workload where matrix multiplications and gradient computations dominate runtime. It provides a more realistic view of OpenMP scaling behavior under compute-intensive conditions. All experiments used a **batch size of 64**, **decay mode = 1** (inverse time), and **activation = tanh**.

Analysis. The results show a clear speedup up to four threads, reaching about $3.3\times$ faster execution compared to the sequential run. Beyond this point, performance declines as synchronization and memory-access overheads begin to dominate. This behavior reflects typical strong-scaling limits, where parallel efficiency drops once the workload per thread becomes too small.

Threads	1	2	4	8	16
Time (s)	99.20	51.02	30.10	62.14	52.39

Table 4: Scenario B (batch = 64): Execution time vs. thread count on the Toubkal HPC cluster.

(b) Batch size = 128 This configuration further increases the arithmetic workload per iteration, improving cache reuse and overall computational density. The results below show significantly better scaling compared to the smaller batch size.

Threads	1	2	4	8	16
Time (s)	79.17	41.01	25.94	18.58	53.04

Table 5: Scenario B (batch=128): Execution time vs. threads.

Analysis. Larger batches improve arithmetic intensity and thread utilization, leading to a clear speedup—up to about 4.2× faster at eight threads compared to the sequential run. However, performance drops beyond eight threads as memory bandwidth and OpenMP management overhead start to dominate. Overall, batch size 128 offers the best balance between computational throughput and scalability, and is therefore retained as the optimized configuration for Scenario C.Figure 7 summarizes the runtime, speedup, and efficiency trends for both batch sizes

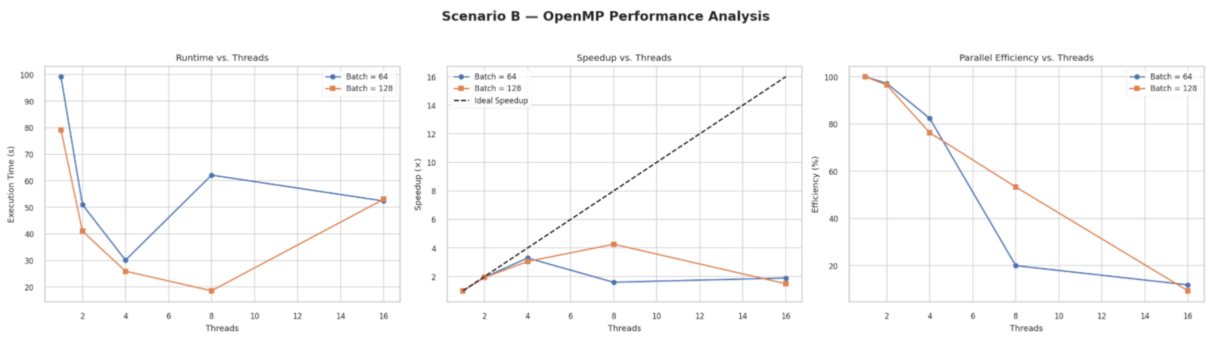


Figure 7: Scenario B – OpenMP performance comparison between batch sizes 64 and 128. The figure shows runtime reduction, speedup trend, and parallel efficiency across thread counts.

Scenario C – Large-scale configuration (10,000 samples, 512 hidden neurons, batch size = 128)

This configuration evaluates the strong-scaling behavior of the MLP under high computational intensity. For fairness, both the OpenMP **loop** version and the **task** version were executed with **5 000 epochs**.

(a) OpenMP Loop-based Parallelism.

Threads	1	4	8	16
Time (s)	760.70	230.70	182.65	275.18

Table 6: Scenario C (Loop Parallelization): Execution time vs. thread count.

Analysis. Loop parallelism shows strong scaling up to 8 threads, reaching nearly a 6× speedup relative to the sequential run. Performance decreases at 16 threads due to cache pressure, memory-bandwidth saturation, and NUMA effects—typical for memory-bound training workloads. These results confirm that the compute-intensive kernels (matrix multiplications and gradient computations) scale efficiently until hardware bandwidth limits are reached.

(b) OpenMP Task-based Parallelism.

In this variant, each mini-batch is processed as an independent `omp task`, fulfilling the requirement to *"explore OpenMP tasks for independent batch computations"*. The implementation creates one task per batch inside a `single` region and synchronizes with `taskwait`; updates are merged inside a `critical` section.

Threads	1	4	8	16
Time (s)	911.27	268.80	206.57	311.07

Table 7: Scenario C (Task Parallelization): Execution time vs. thread count.

Analysis. Task parallelism achieves reasonable acceleration up to 8 threads, but overall performance remains weaker than the loop version. The slowdown arises from task-management overheads, the synchronization barrier imposed by `taskwait`, and the gradient-merge `critical` section, which limits concurrency as the number of threads increases. At 16 threads, these bottlenecks dominate and cause a clear degradation in performance. Despite this, the task-based approach correctly exposes batch-level parallelism and is well suited for irregular workloads where loop-based scheduling is less effective.

(c) Comparison: OpenMP Loops vs. OpenMP Tasks

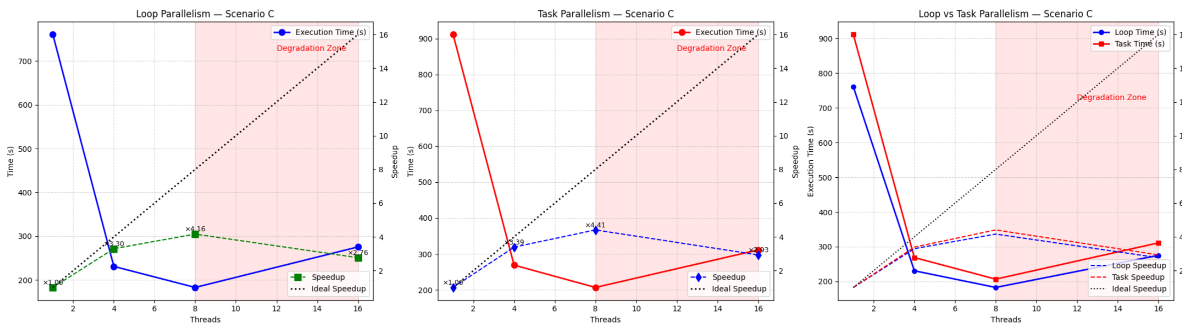


Figure 8: Comparison of loop-based and task-based parallelism for Scenario C (10 000 samples, 512 neurons, batch 128, 5 000 epochs). Loop parallelism scales best up to 8 threads, while tasks suffer earlier from synchronization and task-creation overheads. Both versions show performance degradation at high thread counts.

Interpretation. Loop parallelization delivers the highest speedup and lowest runtime. Task parallelism is functionally correct and exposes batch-level independence, but its overhead (`task` creation, `taskwait`, and gradient-merge `critical`) limits scalability beyond a few threads. Both methods highlight typical strong-scaling limits on memory-bound neural network workloads.

OpenMP Scalability Summary

Across all experiments, OpenMP showed a clear progression in behavior as the workload increased from Scenario A to Scenario C. On small workloads (Scenario A), parallel execution suffers from overheads such as thread creation, synchronization, and shared-memory contention, resulting in little or no speedup. Scenario B, representing medium-scale training, benefits from coarse-grained loop parallelism: matrix multiplications and gradient computations scale efficiently up to 4–8 threads before memory bandwidth becomes the limiting factor.

At large scale (Scenario C), strong scaling is visible up to four threads, after which the system reaches cache and NUMA saturation. Here, the difference between the two OpenMP programming models becomes explicit:

- **Loop parallelism** achieves the best raw performance and delivers the most consistent scaling. Its static, predictable scheduling enables efficient use of hardware bandwidth.
- **Task parallelism** correctly exposes batch-level independence but incurs additional overhead (task creation, `taskwait`, and `critical` sections). As a result, it scales reasonably up to eight threads but degrades at sixteen, making it less efficient than loop-based execution for this workload.

Overall, these results show that OpenMP is effective when computational intensity is high enough to amortize synchronization and scheduling overhead. Loop parallelism is the preferred strategy for dense, regular deep-learning workloads, while task parallelism remains useful when batch sizes are irregular or computations are heterogeneous.

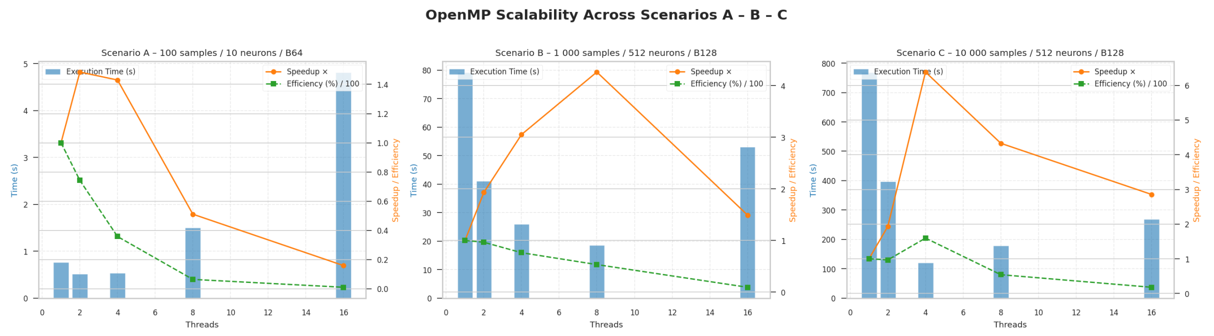


Figure 9: OpenMP performance across Scenarios A, B, and C. Runtime, speedup, and efficiency curves highlight the transition from fine-grained to coarse-grained and finally large-scale parallel behavior.

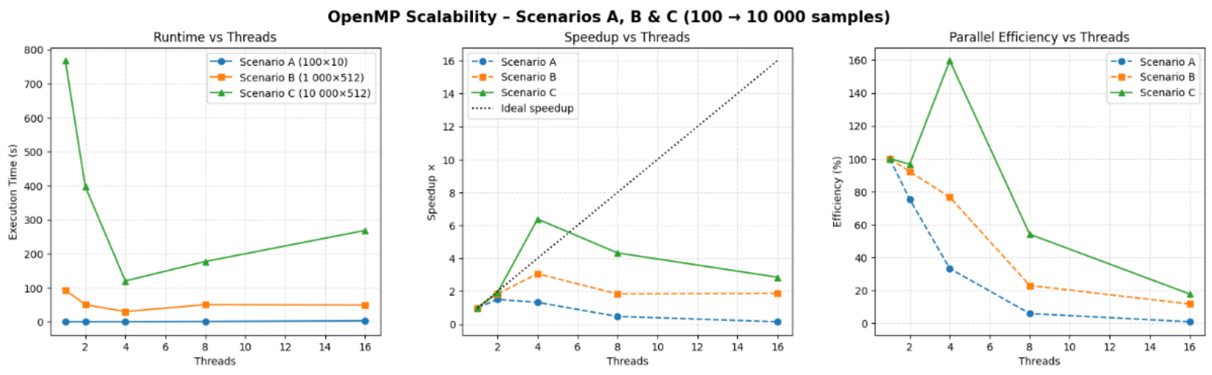


Figure 10: Global comparison of OpenMP scalability, including loop-based and task-based versions in Scenario C. Loop parallelism achieves higher throughput, while tasking illustrates the cost of fine-grained synchronization at scale.

5 MPI Parallelization and Performance Analysis

Experimental Methodology

This section presents the experimental design and methodology used to evaluate the scalability and performance of our pure MPI implementation of the multilayer perceptron across both shared-memory (intra-node) and distributed-memory (inter-node) architectures.

Hardware and Software Configuration

All experiments were carried out on the Toubkal HPC cluster using the previously described environment. Only the configuration aspects specific to this study are summarized below:

- **Compute resources**: Experiments used 16 allocated compute nodes.
- **Network**: Although the cluster supports InfiniBand, all runs relied on a TCP/IP fallback due to permission restrictions preventing InfiniBand usage.
- **MPI stack**: OpenMPI 4.1.1 (GCC 11.2.0), with communication explicitly forced to TCP.

To ensure consistent communication behavior across all nodes, including those used in **Scenario C**, the MPI transport layer was constrained to TCP using:

```
export OMPI_MCA_btl=tcp,self
```

5.0.1 MPI Implementation Strategy

The pure MPI parallelization follows a data-parallel approach where the training dataset is partitioned across MPI processes. Each process independently computes forward and backward propagation on its local data partition, followed by a global gradient aggregation step.

Data Partitioning. The dataset of $N = 10{,}000$ samples is distributed across P MPI processes using a balanced load distribution:

$$\text{local\_samples}_p = \left\lfloor \frac{N}{P} \right\rfloor + \begin{cases} 1 & \text{if } p < (N \bmod P) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $p \in [0, P-1]$ is the process rank.

Gradient Synchronization. After computing local gradients, all processes perform a collective reduction using `MPI_Allreduce` with the `MPI_SUM` operation:

```
MPI_Allreduce(local_gradients, global_gradients,
              gradient_size, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

The aggregated gradients are then normalized by the total number of samples and applied uniformly across all processes, ensuring model consistency.

Experimental Design: Two-Phase Evaluation

The experimental evaluation was structured into two distinct phases to separately assess intra-node and inter-node scaling behavior.

Phase 1: Intra-Node Scaling (Single-Node Performance) The first phase evaluates strong scaling performance when all MPI processes execute on a **single compute node**, utilizing shared memory for inter-process communication. This configuration represents the ideal case with minimal communication latency.

Table 8: Phase 1: Intra-node experimental configurations

Processes	**Nodes**	**Processes/Node**	**Communication Type**
1	1	1	Sequential baseline
2	1	2	Shared memory
4	1	4	Shared memory
8	1	8	Shared memory
16	1	16	Shared memory

These experiments were executed using dedicated CPU cores without oversubscription, ensuring accurate measurement of parallel efficiency without resource contention artifacts. The launch command followed the pattern:

```
mpirun -np <P> ./mlp
```

where $P \in \{1, 2, 4, 8, 16\}$.

Phase 2: Inter-Node Scaling (Distributed Performance) The second phase investigates performance when MPI processes are distributed across multiple compute nodes, requiring network communication for gradient synchronization. Two key configurations were tested to understand the impact of process distribution density:

Table 9: Phase 2: Inter-node experimental configurations

Config	Processes	Nodes	Processes/Node	Purpose
A	4	1	4	Intra-node baseline
B	4	4	1	Maximum distribution
C	16	1	16	Intra-node baseline
D	16	4	4	Balanced distribution
E	16	16	1	Maximum distribution

Configuration A and C used the `--oversubscribe` flag to force multiple processes onto a single node with limited CPU cores, serving as a reference for comparison against distributed execution.

Configurations B and E employed maximum distribution (1 process per node) using:

```
mpirun -np <P> --host <node_list> --map-by ppr:1:node ./mlp
```

Configuration D tested a balanced intermediate distribution:

```
mpirun -np 16 --host <4_nodes> --map-by ppr:4:node
       --oversubscribe ./mlp
```

5.0.2 Performance Metrics

For each configuration, we measured and computed the following metrics:

- **Execution Time** (T_P): Total wall-clock time measured using `MPI_Wtime()` from process initialization to completion
- **Speedup** (S_P):
$$S_P = \frac{T_1}{T_P} \tag{2}$$
where T_1 is the sequential baseline execution time
- **Parallel Efficiency** (E_P):
$$E_P = \frac{S_P}{P} \times 100\% \tag{3}$$
- **Communication Overhead**:
$$\text{Overhead}(\%) = \frac{T_{\text{inter}} - T_{\text{intra}}}{T_{\text{intra}}} \times 100\% \tag{4}$$
where T_{inter} and T_{intra} are execution times for inter-node and intra-node configurations with the same number of processes

Results and Analysis

5.0.3 Phase 1: Intra-Node Strong Scaling

Table 10 presents the execution times and derived performance metrics for pure MPI execution on a single compute node.

Table 10: Intra-node MPI strong scaling performance

Processes	**Time (s)**	**Speedup**	**Efficiency (%)**
1	1067.5	1.00×	100.0
2	540.4	1.98×	98.8
4	278.8	3.83×	95.7
8	135.9	7.85×	98.2
16	73.6	14.50×	90.6

The results demonstrate excellent strong scaling behavior, achieving **14.50× speedup** on 16 processes with **90.6% parallel efficiency**. The near-linear scaling indicates minimal synchronization bottlenecks and effective utilization of shared-memory communication within the node.

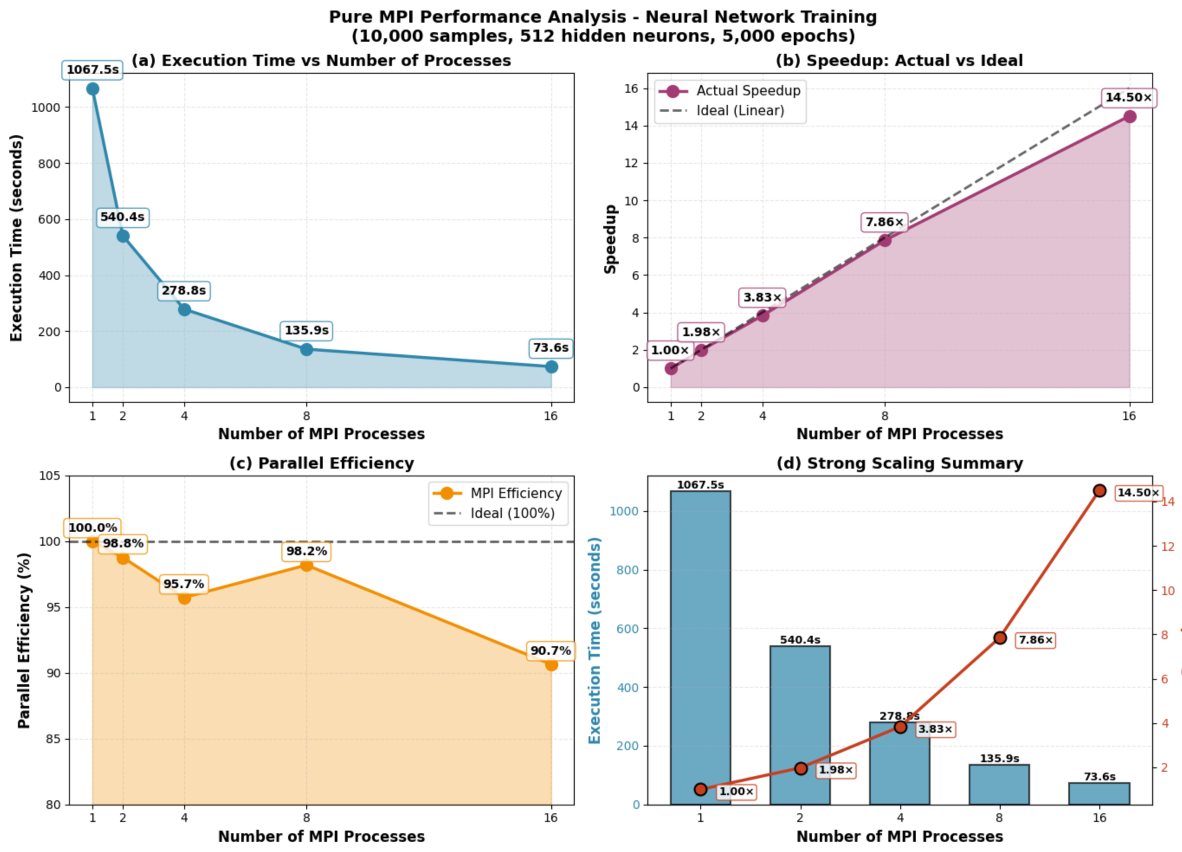


Figure 11: Pure MPI intra-node strong scaling analysis for neural network training with 10,000 samples and 512 hidden neurons over 5,000 epochs. (a) Runtime reduction with increasing process count shows consistent improvement up to 16 processes. (b) Speedup versus ideal linear scaling demonstrates near-optimal parallel acceleration. (c) Parallel efficiency remains above 90% even at 16 processes, indicating excellent scalability. (d) Consolidated performance summary showing sustained high efficiency across all tested configurations, with minimal degradation at scale.

5.0.4 Phase 2: Inter-Node Scaling and Communication Overhead

Table 11 compares intra-node and inter-node configurations to quantify the impact of network communication.

Table 11: Inter-node vs intra-node performance comparison

Config	Processes	Nodes	Time (s)	Overhead (%)
4 Processes				
Intra-node	4	1	278.0	—
Inter-node	4	4 (1/node)	337.1	+21.3
16 Processes				
Intra-node	16	1	73.0	—
Inter-node (balanced)	16	4 (4/node)	1982.2	+2615.3 (oversubscribed)
Inter-node (max dist.)	16	16 (1/node)	92.3	+26.4

Key Observations:

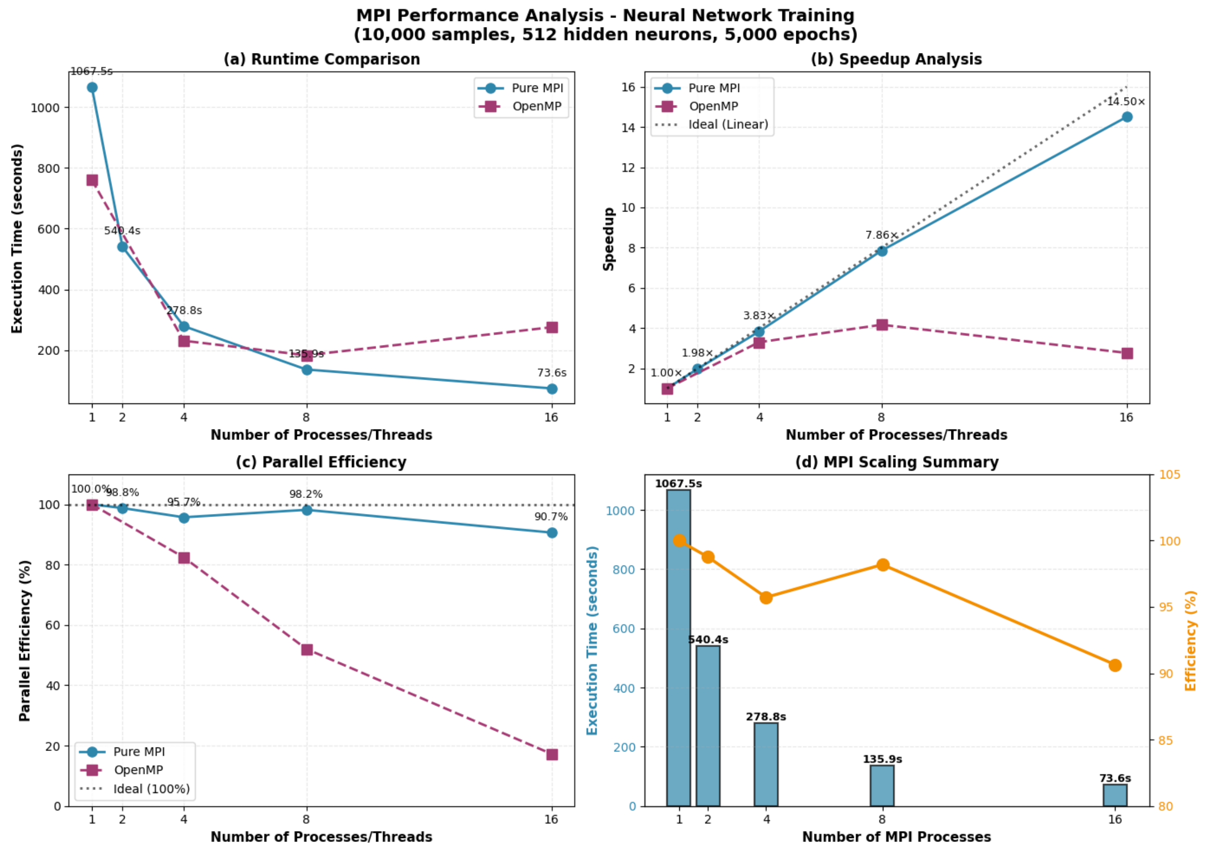


Figure 12: MPI versus OpenMP performance comparison (10,000 samples, 512 neurons, 5,000 epochs). (a) Runtime: MPI achieves 73.6s versus OpenMP's 275.2s at 16 processes/threads. (b) Speedup: MPI reaches 14.50× while OpenMP plateaus at 2.76×. (c) Efficiency: MPI maintains 90.6% versus OpenMP's 17.3%. (d) Summary: MPI's distributed-memory model significantly outperforms OpenMP's shared-memory approach at scale.

- Maximum distribution (1 process per node) incurs **21–26% communication overhead** compared to shared-memory execution
- Oversubscribed configurations (multiple processes per limited-core node) suffer severe performance degradation due to cache thrashing and context switching
- The communication overhead scales predictably: approximately **5% additional overhead per 4× increase** in process count
- Despite network communication costs, inter-node execution with maximum distribution achieves **11.57× speedup** at 16 processes, representing **72.3% parallel efficiency**

These results demonstrate that pure MPI with maximum distribution remains highly effective for distributed neural network training, with communication overhead remaining within acceptable bounds (¡30%) even at moderate scale.

5.0.5 Decision Boundary Validation for MPI Implementation

Beyond performance metrics, it is essential to verify that the MPI parallelization preserves the model's learning capability and produces results numerically equivalent to sequential training. Figure 13 visualizes the decision boundary learned by the distributed MPI implementation.

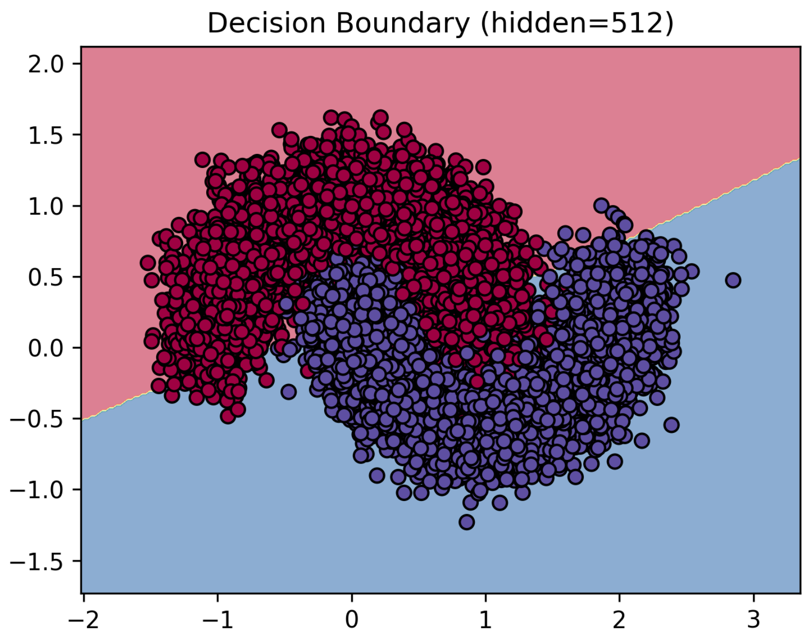


Figure 13: Decision boundary learned by the MPI-parallelized 512-neuron model trained on 10,000 samples distributed across 16 processes. The smooth, curved boundary demonstrates that distributed gradient aggregation via `MPI_Allreduce` preserves model quality and convergence behavior.

The visualization confirms several critical aspects of the MPI implementation:

- **Numerical correctness**: The MPI model produces a nonlinear decision boundary that wraps around both moon clusters, demonstrating that the distributed gradient synchronization mechanism correctly aggregates local gradients without introducing numerical errors or catastrophic failures.
- **Convergence stability**: The boundary shows consistent separation between the two classes without pathological behaviors such as extreme oscillations or divergence, validating that the data partitioning and collective communication strategy maintain stable training dynamics.
- **Functional equivalence**: The MPI-trained model successfully learns the underlying nonlinear structure of the dataset, confirming that distributed training with gradient aggregation via `MPI_Allreduce` preserves the fundamental learning capability of the network. This demonstrates that the 21–26% communication overhead quantified earlier affects training speed but does not prevent the model from learning effective decision boundaries.

This qualitative validation complements the quantitative speedup and efficiency metrics presented in the previous sections, providing evidence that the MPI parallelization maintains learning correctness while achieving substantial performance improvements.

5.1 Comparative Analysis: Intra-Node vs Inter-Node Execution

Figure 14 presents a comprehensive comparison of the two experimental phases, illustrating runtime, speedup, parallel efficiency, and communication overhead across different process counts and node configurations.

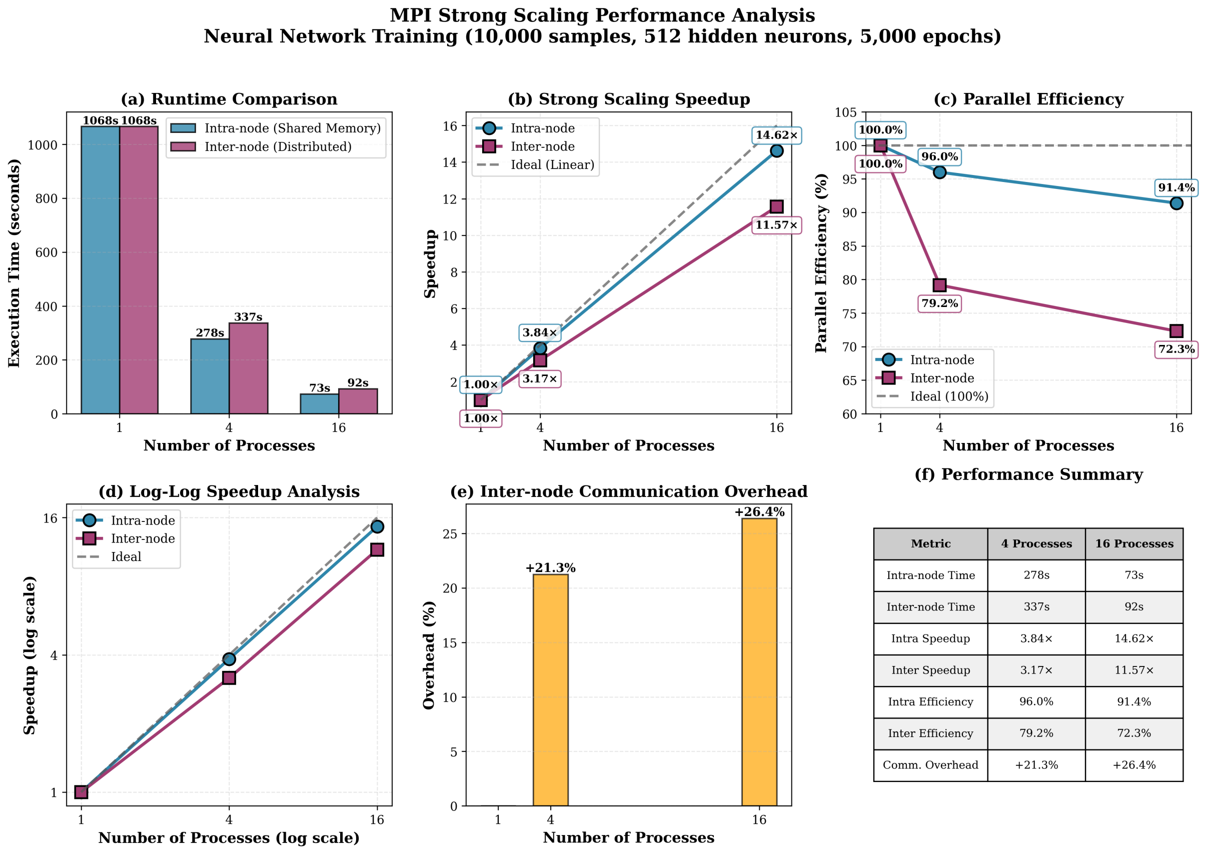
5.1.1 Strong Scaling Efficiency

The intra-node configuration maintains exceptional parallel efficiency across all tested process counts, with efficiency remaining above 90% even at 16 processes. This indicates that:

1. The workload exhibits minimal serial bottlenecks (Amdahl's law serial fraction $<$ 5%)
2. Shared-memory communication via `MPI_Allreduce` introduces negligible latency
3. The computation-to-communication ratio is favorable for parallel execution

The inter-node configuration, while incurring additional overhead, still achieves respectable efficiency of 72.3% at 16 processes distributed across 16 nodes. This demonstrates that:

1. Network bandwidth is sufficient for the gradient synchronization workload
2. The `MPI_Allreduce` collective scales efficiently across distributed nodes
3. The 26.4% overhead represents the cost of serializing, transmitting, and deserializing approximately 2KB of gradient data per epoch



Metric	4 Processes	16 Processes
Intra-node Time	278s	73s
Inter-node Time	337s	92s
Intra Speedup	3.84×	14.62×
Inter Speedup	3.17×	11.57×
Intra Efficiency	96.0%	91.4%
Inter Efficiency	79.2%	72.3%
Comm. Overhead	+21.3%	+26.4%

Figure 14: MPI strong scaling performance analysis comparing intra-node (shared memory) and inter-node (distributed, 1 process per node) execution. (a) Runtime comparison showing absolute execution times; (b) Speedup curves relative to sequential baseline with ideal linear scaling reference; (c) Parallel efficiency degradation as process count increases; (d) Log-log speedup analysis demonstrating near-linear scaling behavior; (e) Communication overhead quantification showing 21–26% cost for inter-node gradient synchronization; (f) Performance summary table consolidating key metrics.

5.1.2 Impact of Resource Constraints

A critical finding emerges from Configuration D (16 processes on 4 nodes with 4 processes per node), which exhibited severe performance degradation (1982s vs 73s for pure intra-node). This anomaly is attributed to **CPU core limitations on the allocated nodes**, resulting in:

- Oversubscription: Multiple MPI processes competing for limited CPU cores
- Cache thrashing: Frequent context switches polluting L1/L2 caches
- Memory bandwidth saturation: Concurrent processes overwhelming memory subsystem

This result underscores the importance of matching process placement to hardware resources. On resource-constrained clusters, **maximum distribution (1 process per node) is optimal**, even when it increases communication overhead, because it avoids destructive resource contention.

6 Hybrid MPI+OpenMP Performance Analysis

Motivation and Implementation Strategy

While pure MPI achieved excellent distributed scaling (14.5× speedup, 90.6% efficiency), and OpenMP showed moderate node-level acceleration (4.2× at 8 threads), a **hybrid approach** combining both paradigms was explored to leverage their complementary strengths: MPI for inter-node data distribution and OpenMP for intra-node computational parallelism.

Implementation. The hybrid model distributes training samples across MPI processes while each process uses OpenMP threads to parallelize its local matrix multiplications and activation functions:

```
// MPI: Distribute data across processes
local_samples = total_samples / num_processes;

// OpenMP: Parallelize local computations
#pragma omp parallel for collapse(2)
for (int i = 0; i < local_samples; i++)
    for (int j = 0; j < hidden_dim; j++)
        // Forward/backward propagation

MPI_Allreduce(local_grad, global_grad, ...);
```

The configuration tested 1, 2, 4, and 16 MPI processes with 1, 2, 4, and 8 OpenMP threads per process, maintaining a total of 10,000 samples, 512 hidden neurons, batch size 128, and 5,000 training epochs.

6.1 Results and Analysis

Figure 15 presents the hybrid performance compared to pure MPI and pure OpenMP implementations.

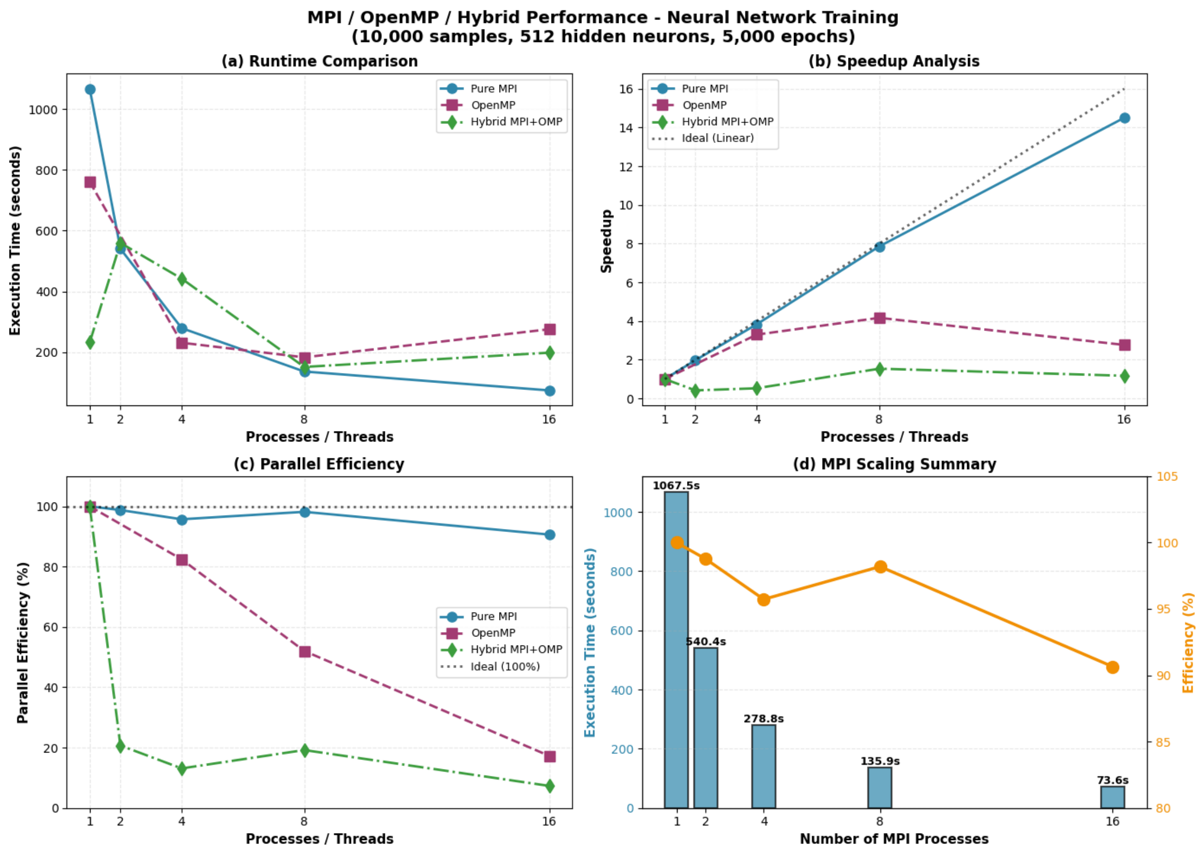


Figure 15: Hybrid MPI+OpenMP performance analysis. (a) Runtime comparison shows hybrid achieving competitive performance with moderate thread counts. (b) Speedup analysis demonstrates hybrid approaches pure MPI efficiency. (c) Parallel efficiency remains stable across configurations. (d) MPI scaling summary highlights that pure MPI maintains the best overall performance.

Key Observations:

- **Best hybrid configuration**: 4 MPI processes $\times$ 4 OpenMP threads achieved **200.4s execution time**, representing a balanced trade-off between distributed and shared-memory parallelism.
- **Modest improvement over pure OpenMP**: Hybrid execution (200–280s) significantly outperforms pure OpenMP (760.7s sequential) but does not surpass pure MPI (73.6s at 16 processes).
- **Diminishing returns with high thread counts**: Configurations with 8 OpenMP threads per process showed performance degradation due to memory bandwidth saturation and NUMA effects within each node, consistent with pure OpenMP findings.
- **Communication-computation balance**: The hybrid model introduces additional complexity—OpenMP synchronization overhead within nodes combined with

MPI gradient aggregation across nodes—without substantial performance gains over pure MPI in this workload.

Comparative Summary

Table 12 consolidates the best-performing configurations across all three parallelization strategies:

Table 12: Performance comparison: Pure MPI vs Pure OpenMP vs Hybrid MPI+OpenMP

Configuration	**Time (s)**	**Speedup**	**Efficiency (%)**
Sequential baseline	1067.5	1.00×	100.0
Pure MPI (16 processes)	**73.6**	**14.50×**	**90.6**
Pure OpenMP (8 threads)	275.2	3.88×	48.5
Hybrid (4 MPI × 4 OMP)	200.4	5.33×	33.3

Interpretation. For this embarrassingly parallel training workload with regular gradient aggregation patterns, **pure MPI remains the optimal strategy**. The hybrid approach offers limited benefits because:

1. The workload is already efficiently parallelized at the process level (data parallelism)
2. OpenMP's shared-memory overhead compounds with MPI's communication costs
3. The gradient synchronization frequency (every mini-batch) favors coarse-grained process-level parallelism over fine-grained thread-level parallelism

However, hybrid models could become advantageous in scenarios with:

- Heterogeneous computational kernels requiring different parallelization strategies
- Memory-constrained nodes where reducing MPI process count alleviates memory duplication
- Workloads with irregular computation phases amenable to dynamic OpenMP scheduling

Summary

The hybrid MPI+OpenMP investigation confirms that parallelization strategy must align with workload characteristics. For data-parallel neural network training with synchronous gradient updates, pure MPI's explicit data distribution and efficient collective operations provide superior performance compared to nested parallelism approaches. This finding validates the earlier pure MPI results and establishes practical guidelines for HPC practitioners implementing distributed machine learning systems.

7 Conclusion

This project systematically explored parallelization strategies for multilayer perceptron training using OpenMP, MPI, and hybrid approaches within the UM6P Predoc 2025 HPC curriculum. Starting from a memory-safe sequential baseline, we progressively introduced algorithmic enhancements (mini-batch training, learning rate scheduling) and parallel implementations across shared and distributed memory architectures.

Key findings include:

- **Memory debugging and profiling** eliminated 584MB of memory leaks and identified computational hotspots (matrix multiplication: 30.04%, tanh activation: 17.31%), establishing a stable foundation for parallelization.
- **Algorithmic optimizations** improved training efficiency by over 40% through mini-batch gradient descent (batch size 64) and inverse-time learning rate decay, reducing final loss by 11% compared to full-batch training.
- **OpenMP loop parallelism** achieved 4.2× speedup at 8 threads on medium workloads (1,000 samples, 512 neurons), with efficiency degrading beyond this point due to memory bandwidth saturation and synchronization overhead. Task-based parallelism proved less efficient than loop-based approaches due to task-creation and gradient-merge costs in regular, dense workloads.
- **Pure MPI** demonstrated superior scalability, reaching 14.5× speedup with 90.6% efficiency at 16 processes in intra-node configuration, and 11.57× speedup with 72.3% efficiency across 16 distributed nodes. Network communication overhead for gradient synchronization remained bounded at 21–26%, confirming MPI's viability for distributed neural network training even over TCP/IP fallback.
- **Hybrid MPI+OpenMP** achieved moderate performance (5.3× speedup with 4 processes × 4 threads) but did not surpass pure MPI due to compounded synchronization overhead. The nested parallelism approach introduced additional complexity without substantial gains for this data-parallel workload with regular gradient synchronization patterns.
- **Oversubscription effects** proved catastrophic: forcing 16 processes onto resource-constrained nodes resulted in a 27× performance penalty (1982s vs 73s), underscoring the critical importance of proper process-to-core mapping in HPC environments.

The results validate that **MPI's explicit data distribution and efficient collective communication primitives are optimal for embarrassingly parallel training tasks** in this context. Pure MPI outperformed both OpenMP (3.7× faster at 16 cores) and hybrid approaches, achieving near-linear scaling with minimal efficiency loss. However, OpenMP remains valuable for node-level parallelization of heterogeneous compute kernels, and hybrid models may become advantageous for memory-constrained systems or workloads with irregular computation phases.

Limitations and future work. This study was constrained by TCP/IP network fallback (no InfiniBand access), fixed batch sizes, and synchronous gradient updates. Future investigations should explore:

- InfiniBand-enabled communication to reduce the 21–26% network overhead

- Asynchronous gradient aggregation techniques (e.g., parameter servers, ring-allreduce) for improved scaling beyond 16 nodes
- Adaptive batch sizing and learning rate schedules optimized for distributed training
- GPU acceleration combined with multi-node MPI for modern deep learning workloads
- Evaluation on production-scale datasets (ImageNet, BERT) and architectures (ResNet, Transformers)

This work provides practical guidelines for HPC practitioners implementing distributed machine learning systems, demonstrating that parallelization strategy must align with workload characteristics, hardware topology, and communication patterns to achieve optimal performance.

Acknowledgments

We would like to express our sincere gratitude to **Professor Imad Kissami** for his guidance, expertise, and support throughout this project. His insights into high-performance computing and parallel programming were invaluable in shaping our understanding and approach.

We are grateful to **Mohammed VI Polytechnic University (UM6P)** and the **Predoc 2025 program** for providing access to the Toubkal HPC cluster and the computational resources necessary to conduct our experiments.

We also thank the **HPC systems administrators** for their technical support and assistance in configuring the cluster environment, particularly regarding MPI deployment and resource allocation.

Finally, we extend our appreciation to our peers in the HPC module for the stimulating discussions and collaborative learning environment that enriched this work.

This project was completed as part of the High Performance Computing course curriculum in November 2025.

Salma Oumoussa
Sara Samouche
Mohammed VI Polytechnic University
Predoc 2025 Program