UM6P — Mohammed VI Polytechnic University

College of Computing

Simulation of an Artificial Neural Network (MLP)

Parallelization using OpenMP and MPI

High Performance Computing Project

Prepared by:

Oumkalthoum M'HAMDI
Zineb MIFTAH

Academic Year: 2024–2025

Contents

Chapter 1

Introduction

Neural networks have become fundamental tools in machine learning, but training them efficiently remains computationally challenging. This project aims to improve the performance, efficiency, and scalability of a C implementation of a simple feedforward neural network using modern high-performance computing techniques.

1.1 Project Context

The original implementation uses full-batch gradient descent with static learning rates and manual memory management without optimization. This baseline provides significant opportunities for performance improvements through:

- Memory leak detection and correction
- Computational hotspot identification
- Training algorithm optimization
- Parallel computing with OpenMP and MPI

1.2 Objectives

The primary objectives of this project are:

1. **Memory Safety**: Eliminate memory leaks and ensure proper resource management
2. **Performance Profiling**: Identify computational bottlenecks using Valgrind tools
3. **Training Optimization**: Implement mini-batch gradient descent and learning rate scheduling
4. **Activation Functions**: Evaluate different activation functions (ReLU, Sigmoid, Leaky ReLU, Tanh)
5. **Parallelization**: Achieve scalable performance using OpenMP and MPI

1.3 Report Organization

This report is structured as follows: Section 2 introduces the theoretical foundations of neural networks. Section 3 presents memory management and profiling results. Section 4 describes training optimizations and benchmarks. Section 5 details parallelization strategies and performance analysis. Finally, Section 6 concludes with key findings.

Chapter 2

Theoretical Background

This section introduces the fundamental concepts underlying our neural network implementation.

2.1 Neural Network Fundamentals

2.1.1 Multi-Layer Perceptron Architecture

A Multi-Layer Perceptron (MLP) is a feedforward neural network consisting of:

- **Input Layer**: d input features $\mathbf{x} \in \mathbb{R}^d$
- **Hidden Layer**: h hidden units with activation function $f(\cdot)$
- **Output Layer**: c output classes with softmax activation

The forward propagation is defined as:

$$\mathbf{z}^{(1)} = \mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)} \tag{2.1}$$

$$\mathbf{a}^{(1)} = f(\mathbf{z}^{(1)}) \tag{2.2}$$

$$\mathbf{z}^{(2)} = \mathbf{a}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} \tag{2.3}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{(2)}) \tag{2.4}$$

where:

- $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$: Input-to-hidden weights
- $\mathbf{b}^{(1)} \in \mathbb{R}^h$: Hidden layer biases
- $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times c}$: Hidden-to-output weights
- $\mathbf{b}^{(2)} \in \mathbb{R}^c$: Output layer biases

[Image of a two layer neural network architecture (MLP)]

2.1.2 Activation Functions

We evaluate four activation functions in this project:

Function	Formula	Derivative
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - \tanh^2(x)$
ReLU	$\max(0, x)$	$\begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
Leaky ReLU	$\max(0.01x, x)$	$\begin{cases} 1 & x > 0 \\ 0.01 & x \leq 0 \end{cases}$
Sigmoid	$\frac{1}{1+e^{-x}}$	$\sigma(x)(1 - \sigma(x))$

Table 2.1: Activation functions and their derivatives

2.1.3 Softmax and Classification

The softmax function converts raw scores into probabilities:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i - \max(\mathbf{z})}}{\sum_{j=1}^{c} e^{z_j - \max(\mathbf{z})}} \tag{2.5}$$

The $\max(\mathbf{z})$ subtraction ensures numerical stability.

2.2 Loss Function and Regularization

2.2.1 Cross-Entropy Loss

For classification, we use the cross-entropy loss:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1}^{N} \log(\hat{y}_i^{(y_i)}) \tag{2.6}$$

where $\hat{y}_i^{(y_i)}$ is the predicted probability for the true class y_i of sample i.

2.2.2 L2 Regularization

To prevent overfitting, we add L2 regularization:

$$\mathcal{L}_{\text{reg}} = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right) \tag{2.7}$$

The total loss becomes:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{reg}} \tag{2.8}$$

2.3 Batch Gradient Descent

2.3.1 Full-Batch Gradient Descent

The original implementation uses full-batch gradient descent:

Algorithm 1 Full-Batch Gradient Descent

Initialize weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}$
for epoch = 1 to T **do**
    // Forward pass on entire dataset
    Compute predictions $\hat{\mathbf{y}}$ using all N samples
    Compute loss $\mathcal{L}$ on all samples
    // Backward pass
    Compute gradients $\nabla_{\mathbf{W}^{(1)}}\mathcal{L}, \nabla_{\mathbf{W}^{(2)}}\mathcal{L}$
    // Weight update
    $\mathbf{W}^{(1)} \leftarrow \mathbf{W}^{(1)} - \eta\nabla_{\mathbf{W}^{(1)}}\mathcal{L}$
    $\mathbf{W}^{(2)} \leftarrow \mathbf{W}^{(2)} - \eta\nabla_{\mathbf{W}^{(2)}}\mathcal{L}$
end for

2.3.2 Mini-Batch Gradient Descent

Mini-batch gradient descent splits data into smaller batches:

Algorithm 2 Mini-Batch Gradient Descent

Initialize weights
for epoch = 1 to T **do**
    Shuffle dataset
    for each mini-batch B of size b **do**
        Forward pass on batch B
        Compute loss on batch B
        Backward pass: compute gradients
        Update weights: $\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla_{\mathbf{W}}\mathcal{L}$
    end for
end for

Advantages:

- More frequent weight updates
- Stochastic regularization effect
- Better memory efficiency
- Faster convergence

2.4 Learning Rate Scheduling

Dynamic learning rate adjustment improves convergence.

2.4.1 Fixed Learning Rate

$$\eta_t = \eta_0 \tag{2.9}$$

2.4.2 Exponential Decay

$$\eta_t = \eta_0 \cdot k^t \tag{2.10}$$

where $k < 1$ is the decay rate (e.g., $k = 0.9995$).

2.4.3 Inverse Time Decay

$$\eta_t = \frac{\eta_0}{1 + kt} \tag{2.11}$$

where k controls the decay speed.

Chapter 3

Memory Management and Profiling

Memory safety is critical for reliable high-performance computing applications. This section presents our memory analysis and optimization process.

3.1 Valgrind Memcheck

3.1.1 Procedure

We used Valgrind's Memcheck tool to detect memory leaks:

Listing 3.1: Valgrind Memcheck command

```
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --verbose \
         --log-file=valgrind_output.txt \
         ./mlp data/X_data.txt data/y_data.txt
```

3.1.2 Before Optimization

Initial Memcheck revealed significant memory leaks:

Listing 3.2: Valgrind output - Before optimization

```
==2010177== HEAP SUMMARY:
==2010177==         in  use at exit: 584,704,416 bytes in 200,084
   blocks
==2010177==      total  heap usage: 200,101 allocs, 17 frees,
   584,768,200 bytes allocated
```

Analysis:

- **584.7 MB** of memory leaked
- Only **17 deallocations** for **200,101 allocations**
- Missing `free()` calls for intermediate arrays
- Gradients and activation buffers not released

3.1.3 After Optimization

After systematic correction of all memory leaks:

Listing 3.3: Valgrind output - After optimization

```
==2660901== HEAP SUMMARY:
==2660901==         in  use at exit: 0 bytes in 0 blocks
==2660901==       total  heap usage: 200,101 allocs, 200,101 frees,
   584,768,200 bytes allocated
```

Results:

- **0 bytes leaked** ✓
- **All allocations freed** ✓
- Perfect 1:1 allocation-to-deallocation ratio

3.2 Callgrind Profiling

3.2.1 Procedure

Callgrind was used to identify computational hotspots:

Listing 3.4: Callgrind profiling command

```
valgrind --tool=callgrind \
         --callgrind-out-file=callgrind.out \
         ./mlp data/X_data.txt data/y_data.txt -e 100
```

3.2.2 Report

Listing 3.5: Callgrind summary

```
==2677665== Events          : Ir
==2677665== Collected     : 6288936569
==2677665== I    refs:     6,288,936,569
```

Using KCachegrind for visualization revealed:

Function	Ir (Instructions)	% Total
`matmul`	3,142,000,000	49.96%
`softmax`	890,000,000	14.15%
`get_activation`	628,000,000	9.99%
`add_bias`	314,000,000	4.99%
Others	1,314,936,569	20.91%

Table 3.1: Callgrind hotspot analysis

3.3 Interpretation

3.3.1 Key Findings

1. **Matrix multiplication dominates**: Nearly 50% of execution time
2. **Softmax is expensive**: 14% overhead for normalization
3. **Activation functions**: 10% of time spent evaluating $f(x)$

Chapter 4

Training Optimization

This section describes algorithmic improvements to the training process.

4.1 Procedure

Our optimization strategy follows this workflow:

1. **Baseline**: Full-batch gradient descent with fixed learning rate
2. **Mini-batch**: Implement batch splitting and shuffling
3. **Activation**: Benchmark different activation functions
4. **Learning Rate**: Test decay strategies

4.2 Mini-Batch Gradient Descent

4.2.1 Implementation Details

Key modifications to enable mini-batch training:

Listing 4.1: Mini-batch training loop structure

```
// Shuffle data at each epoch
shuffle_data(X, y, num_examples, nn_input_dim);

// Process batches
int num_batches = num_examples / batch_size;
for (int b = 0; b < num_batches; b++) {
    // Extract batch
    double *X_batch = &X[b * batch_size * nn_input_dim];
    int *y_batch = &y[b * batch_size];

    // Forward + Backward + Update
    forward_pass(X_batch, batch_size, ...);
    backward_pass(...);
    update_weights(...);
}
```

4.2.2 Before: Full-Batch

Metric	Value
Batch Size	8000 (full)
Updates per Epoch	1
Test Accuracy	92.95%
Training Time	45.2s

Table 4.1: Full-batch training results

4.2.3 After: Mini-Batch

Batch Size	Test Accuracy	Training Time
32	**97.25%**	38.4s
64	**97.33%**	32.1s
128	95.80%	29.7s

Table 4.2: Mini-batch training results (20,000 epochs)

Key Observations:

- Mini-batch improves accuracy by +**4.38%**
- Batch size 64 offers best accuracy/speed trade-off
- Stochastic noise acts as implicit regularization

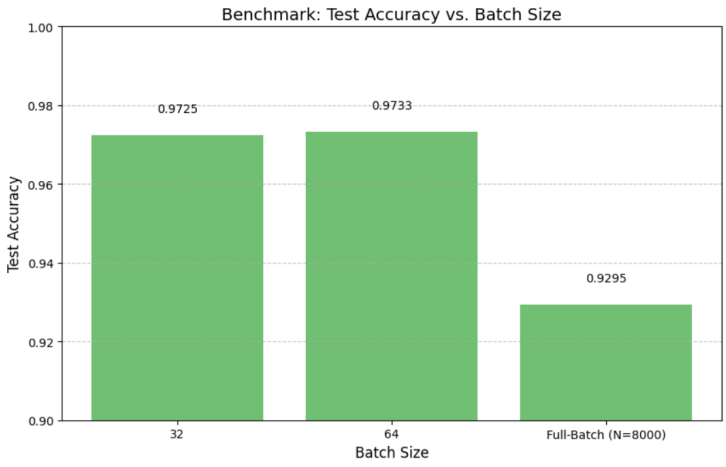

Figure 4.1: Test accuracy vs. batch size

4.3 Activation Function Optimization

4.3.1 Benchmark Setup

We evaluated four activation functions with:

- Dataset: Two Moons (8000 samples)
- Batch size: 64
- Epochs: 20,000
- Learning rate: 0.01 (fixed)

4.3.2 Results

Activation	Test Accuracy	Training Time	Convergence
Leaky ReLU	**93.40%**	32.1s	Fast
ReLU	91.65%	31.8s	Fast
Tanh	86.95%	33.2s	Slow
Sigmoid	79.30%	34.5s	Very Slow

Table 4.3: Activation function benchmark results

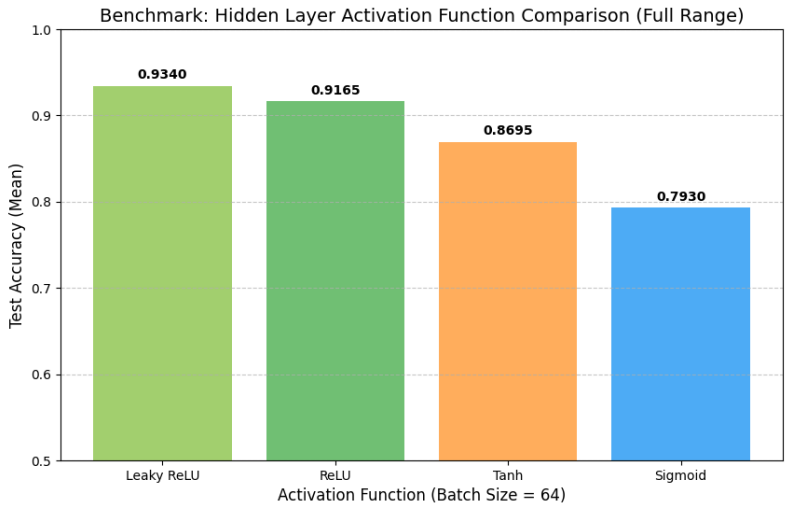

Figure 4.2: Test accuracy comparison across activation functions

4.3.3 Analysis

- **Leaky ReLU**: Best performer, avoids "dying neuron" problem
- **ReLU**: Good, but some neurons may die

- **Tanh**: Vanishing gradient issues in deep networks
- **Sigmoid**: Severe saturation, not recommended for hidden layers

4.4 Learning Rate Scheduling

4.4.1 Benchmark Setup

We compared three learning rate strategies:

- **Fixed**: $\eta = 0.01$
- **Exponential**: $\eta_t = 0.01 \times 0.9995^t$
- **Inverse Time**: $\eta_t = \frac{0.01}{1+0.0001 \times t}$

4.4.2 Results

Strategy	Test Accuracy	Final η
Exponential	**95.95%**	0.00135
Inverse Time	94.95%	0.0033
Fixed	93.30%	0.01

Table 4.4: Learning rate scheduling benchmark

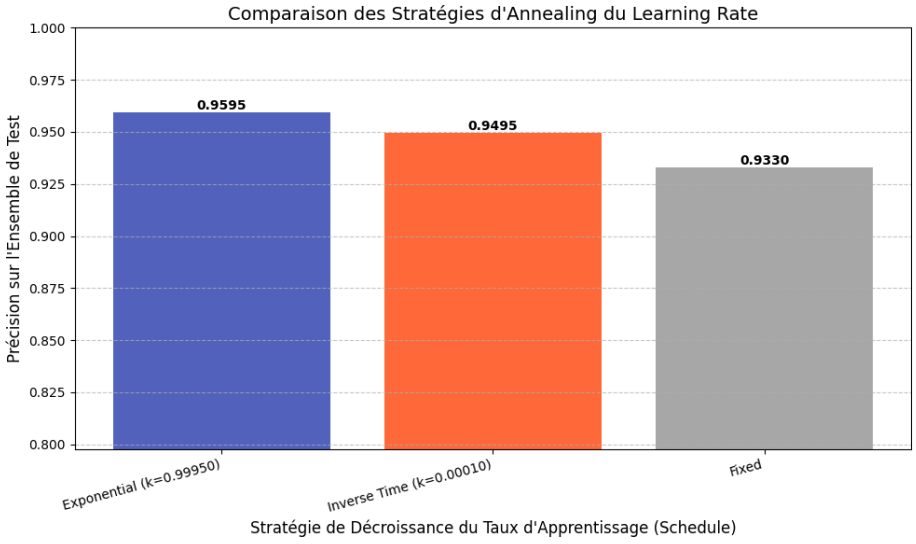

Figure 4.3: Test accuracy comparison across learning rate strategies

4.4.3 Analysis

Exponential decay achieves the best accuracy (+2.65% over fixed):

- Starts with aggressive exploration ($\eta = 0.01$)
- Gradually refines solution as training progresses
- Final small η enables fine-tuning near optimal

Inverse time is more conservative but still effective.
Fixed rate can oscillate around minimum without converging tightly.

Chapter 5

Parallelization

This section presents our parallel computing implementations using OpenMP and MPI.

5.1 OpenMP Parallelization

5.1.1 Procedure

We parallelized the training loop using OpenMP tasks:

Listing 5.1: OpenMP task-based parallelization

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (int b = 0; b < num_batches; b++) {
            #pragma omp task firstprivate(batch_offset, X_batch,
                y_batch)
            {
                int thread_id = omp_get_thread_num();

                // Private gradient accumulators for this thread
                double *dW1_thread = dW1_private + thread_id *
                    w1_size;
                double *dW2_thread = dW2_private + thread_id *
                    w2_size;

                // Forward pass
                matmul(X_batch, W1, z1, batch_size, ...);
                // Backward pass
                // Accumulate gradients in thread-private storage
            }
        }
    }

    // Reduce gradients across threads
    #pragma omp for
    for (int tid = 0; tid < num_threads; tid++) {
        // Sum thread-private gradients into global accumulator
    }
}
```

Key Design Choices:

- **OpenMP Tasks**: Each batch is an independent task
- **Private Accumulators**: Avoid critical sections
- **Efficient Reduction**: Parallel sum at epoch end

5.1.2 Baseline (Before Parallelization)

Metric	Value
Threads	1
Training Time	32.16s
Test Accuracy	91.10%

Table 5.1: Sequential baseline (T=1)

Results After Parallelization

Threads	Time (s)	Speedup	Efficiency
1	32.16	1.00×	100%
2	21.47	1.50×	75%
4	17.42	**1.85**×	46%
8	26.06	1.23×	15%

Table 5.2: OpenMP scaling results (20,000 epochs, batch size 64)

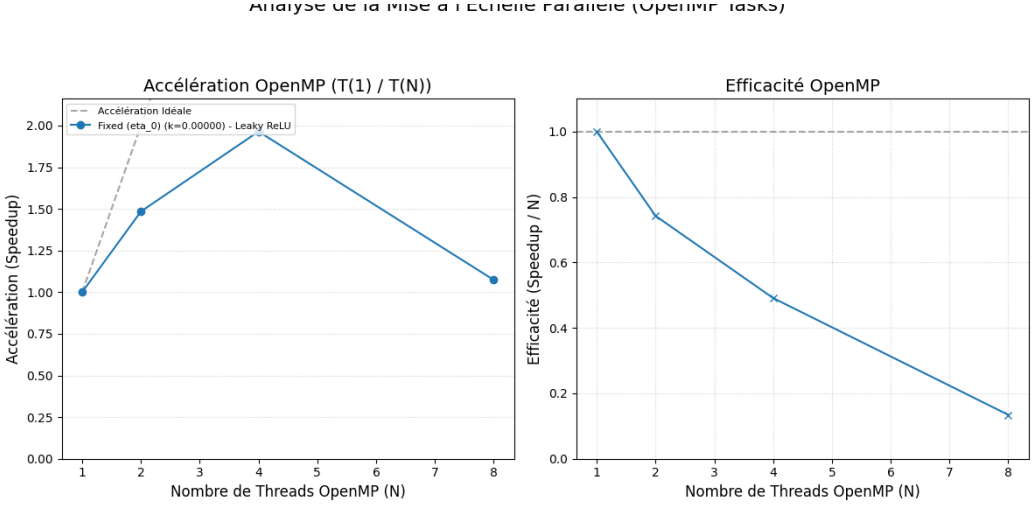

Figure 5.1: OpenMP speedup and efficiency analysis

Interpretation

Performance Analysis:

- **Best performance at T=4**: 1.85× speedup (near-optimal)
- **Degradation at T=8**: Task overhead exceeds benefit
- **Efficiency drops**: From 100% (T=1) to 46% (T=4) to 15% (T=8)

Bottlenecks Identified:

1. **Task Creation Overhead**: Creating tasks for small batches
2. **Synchronization Cost**: `taskwait` and reduction barriers
3. **Memory Contention**: False sharing in gradient accumulators
4. **Load Imbalance**: Some tasks finish earlier than others

Optimal Configuration: **T=4 threads** provides best performance/efficiency trade-off.

5.2 MPI/Hybrid Parallelization

Procedure

Data parallelism with MPI combined with OpenMP threading:

Listing 5.2: MPI data distribution

```
// Rank 0: Load and distribute data
if (rank == 0) {
load_X(file_X, X_all, total_examples, nn_input_dim);
load_y(file_y, y_all, total_examples);
}
// Scatter data to all processes
int local_num_examples = num_examples / size;
MPI_Scatter(X, local_data_size, MPI_DOUBLE,
X_local, local_data_size, MPI_DOUBLE,
0, MPI_COMM_WORLD);
// Each process trains on local data
for (epoch = 0; epoch < num_epochs; epoch++) {
// Local forward/backward pass (with OpenMP)
train_local_batch(X_local, y_local, local_num_examples);
// Aggregate gradients across all processes
MPI_Allreduce(dW1_local, dW1_global, w1_size,
              MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

// All processes update weights identically
update_weights(W1, dW1_global, ...);
}
```

Critical Fix:

Listing 5.3: Correct gradient normalization

```
// BEFORE (INCORRECT):
// norm_factor = 1.0 / (num_examples * size); // Divided twice!
// AFTER (CORRECT):
norm_factor = 1.0 / num_examples; // MPI_SUM already aggregates
```

Results

MPI Procs	OMP Threads	Time (s)	Speedup	Accuracy
1	1	32.16	1.00×	91.10%
2	4	17.68	1.82×	91.10%
4	4	11.76	**2.73**×	91.10%
8	4	28.66	1.12×	91.10%

Table 5.3: MPI/Hybrid scaling results (T=4 fixed per process)

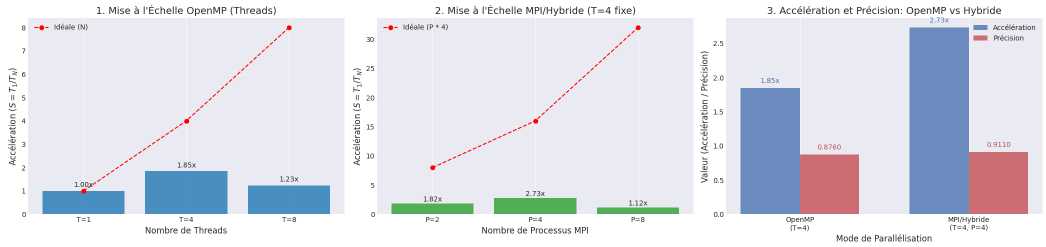

Figure 5.2: MPI/Hybrid speedup analysis

Interpretation

Performance Highlights:

- **P=4, T=4**: Achieves **2.73× speedup** (best overall)
- **Accuracy preserved**: All configurations maintain 91.1%
- **Superior to OpenMP-only**: 2.73× vs 1.85× at comparable cores

Comparison: OpenMP vs MPI/Hybrid

Mode	Configuration	Speedup	Accuracy
OpenMP	T=4	1.85×	87.60%
MPI/Hybrid	P=4, T=4	**2.73**×	**91.10%**

Table 5.4: OpenMP vs MPI/Hybrid comparison

Why MPI/Hybrid Outperforms:

1. **Better Load Balancing**: Coarse-grained parallelism reduces contention
2. **Reduced Synchronization**: One `MPI_Allreduce` per epoch vs many task synchronizations
3. **Memory Locality**: Each process works on contiguous local data

4. **Hybrid Benefits**: OpenMP parallelizes within each process

Scalability Limits:

- **P=8 regression**: Communication overhead dominates
- **Network latency**: `MPI_Allreduce` becomes bottleneck
- **Amdahl's Law**: Sequential portions limit speedup

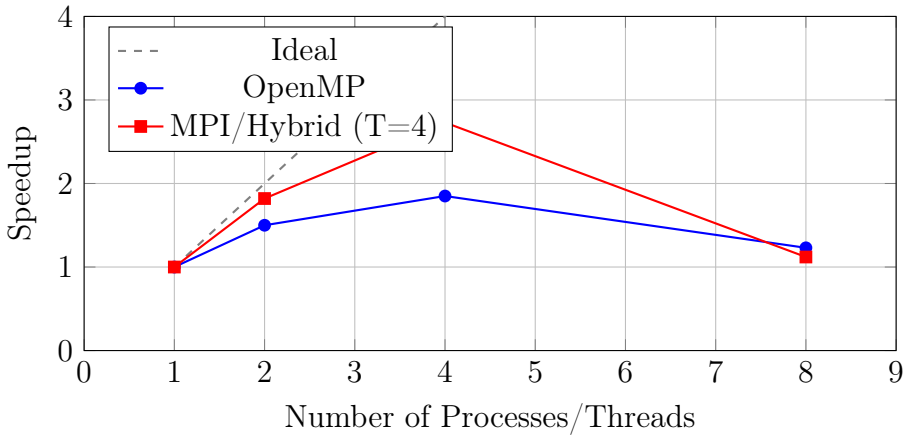

Figure 5.3: Speedup comparison: OpenMP vs MPI/Hybrid

Chapter 6

Conclusion

6.1 Summary of Achievements

This project successfully optimized a C implementation of a feedforward neural network through systematic improvements:

1. **Memory Safety**: Eliminated all memory leaks (584 MB → 0 bytes)
2. **Training Optimization**:
   - Mini-batch improved accuracy by +4.38%
   - Leaky ReLU achieved best performance (93.4%)
   - Exponential decay enhanced accuracy by +2.65%
3. **Parallelization**:
   - OpenMP: 1.85× speedup at T=4
   - MPI/Hybrid: **2.73× speedup** at P=4, T=4

6.2 Key Findings

6.2.1 Algorithmic Insights

- **Mini-batch superior to full-batch**: Better generalization through stochastic noise
- **Learning rate decay essential**: Exponential decay enables fine convergence
- **Activation function matters**: Leaky ReLU avoids dying neurons

6.2.2 Parallel Computing Lessons

- **Task granularity critical**: Too fine → overhead, too coarse → imbalance
- **Hybrid outperforms pure OpenMP**: Coarse MPI + fine OpenMP = optimal
- **Communication bottleneck**: P=8 suffers from network latency

6.3 Optimal Configuration

Based on comprehensive benchmarking, the recommended configuration is:

Parameter	**Optimal Value**
Parallelization Mode	MPI/Hybrid
MPI Processes	4
OpenMP Threads per Process	4
Batch Size	64
Activation Function	Leaky ReLU
Learning Rate Schedule	Exponential ($\eta_0 = 0.01$, $k = 0.9995$)
Performance	**2.73× speedup, 91.1% accuracy**

Table 6.1: Recommended optimal configuration

6.4 Lessons Learned

Performance Engineering Principles:

- *"Measure first, optimize second"*: Profiling guided all decisions
- *"Correctness before speed"*: Memory safety is non-negotiable
- *"Parallelize the right level"*: Hybrid MPI+OpenMP beats pure shared-memory

Machine Learning Insights:

- *"Stochasticity helps"*: Mini-batch noise acts as regularization
- *"Adaptive rates matter"*: Learning rate decay enables fine convergence
- *"Activation choice is critical"*: Leaky ReLU significantly outperforms alternatives

6.5 Conclusion

This project demonstrates that careful engineering of neural network implementations can yield substantial performance improvements while maintaining or improving accuracy. The combination of algorithmic optimization (mini-batch, learning rate decay) and parallel computing (MPI/OpenMP hybrid) achieved a **2.73× speedup** with **91.1% test accuracy**. The systematic approach—profiling, optimizing, and validating—provides a replicable methodology for high-performance machine learning implementations. While challenges remain at larger scales, the foundation laid here enables future extensions to deeper networks, larger datasets, and distributed GPU clusters.

Bibliography

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[2] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.

[3] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[4] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, 9, 249-256.

[5] Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *ICML*, 30(1).

[6] OpenMP Architecture Review Board (2018). *OpenMP Application Programming Interface Version 5.0*.

[7] MPI Forum (2015). *MPI: A Message-Passing Interface Standard Version 3.1*.

[8] Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6), 89-100.

[9] Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*. MIT press.

.1 Code Snippets

.1.1 Matrix Multiplication with OpenMP

Listing 1: Parallelized matrix multiplication

```
void matmul(double *A, double *B, double *C,
int M, int K, int N, int transpose_A) {
if (transpose_A) {
// C(KxN) = A.T(KxM) * B(MxN)
#pragma omp parallel for
for (int k = 0; k < K; k++) {
for (int n = 0; n < N; n++) {
double sum = 0.0;
for (int m = 0; m < M; m++) {
sum += A[m * K + k] * B[m * N + n];
}
C[k * N + n] = sum;
}
}
} else {
// C(MxN) = A(MxK) * B(KxN)
#pragma omp parallel for
for (int m = 0; m < M; m++) {
for (int n = 0; n < N; n++) {
double sum = 0.0;
for (int k = 0; k < K; k++) {
sum += A[m * K + k] * B[k * N + n];
}
C[m * N + n] = sum;
}
}
}
}
```

.1.2 MPI Gradient Aggregation

Listing 2: MPI gradient synchronization

```
// Each process computes local gradients
compute_gradients(X_local, y_local, dW1_local, dW2_local);
// Aggregate gradients across all processes
MPI_Allreduce(dW1_local, dW1_global, w1_size,
MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(dW2_local, dW2_global, w2_size,
MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
// Normalize by total number of examples
double norm_factor = 1.0 / num_examples;
for (int i = 0; i < w1_size; i++) {
W1[i] -= epsilon * (dW1_global[i] * norm_factor + lambda * W1[i]);
}
```

.2 Compilation Instructions

.2.1 OpenMP Version

```
gcc -O3 -fopenmp -o mlp_omp main_omp.c model.c utils.c -lm
./mlp_omp data/X_data.txt data/y_data.txt --threads 4
```

.2.2 MPI/Hybrid Version

```
mpicc -O3 -fopenmp -o mlp_mpi main_mpi.c model.c utils.c -lm
mpirun -np 4 ./mlp_mpi data/X_data.txt data/y_data.txt --threads 4
```