Improving a C Implementation of a Feedforward Neural Network

Ed-dyb Chaymae Doha Dounia

1 Project Overview

In this project, a Multilayer Perceptron (MLP) neural network implemented in C is incrementally improved from a basic sequential implementation toward a scalable, optimized, and leak-free parallel implementation. The work combines low-level programming with High Performance Computing (HPC) tools to analyze and accelerate the training loop.

The project's main goals are:

- **Memory Management and Debugging:** identify and fix memory leaks, ensure correct allocation and deallocation of all dynamic data structures, and guarantee safe execution.
- **Performance Profiling:** analyze runtime behavior to locate hotspots using tools such as Valgrind, Callgrind, and KCachegrind.
- **Optimization:** improve computational efficiency through algorithmic changes and compiler-level optimizations.
- **OpenMP Acceleration:** parallelize key numerical routines (matrix multiplications, gradient updates, batch computations) to exploit multi-core CPUs.
- **Performance Evaluation:** compare different learning-rate schedules, activation functions, and thread counts, assessing accuracy, loss convergence, and speedup.

2 Objectives

2.1 Memory Management and Debugging

Objective Description

This part aimed to ensure that the neural network implementation is completely memory-safe and leak-free as shown in Figure 1. It involved:

- identifying and fixing memory leaks using Valgrind (memcheck) and Valgrind Massif;
- applying systematic memory allocation and deallocation for all dynamically created structures;
- improving the safety and clarity of memory handling, especially before introducing OpenMP parallelism in later objectives.

Implementation Details

In our `model.c`, we added a `SAFE_FREE` macro to encapsulate the pattern "free + set pointer to `NULL`":

```
#define SAFE_FREE(p) do { if ((p)!=NULL) { free(p); (p)=NULL; } } while (0)
```

Every dynamically allocated array has a proper deallocation in the reverse order of allocation, and all exit paths in the training routine were checked to ensure that no allocations are skipped.

Results

Valgrind Massif snapshots of the memory usage before and after the fixes are shown in Figure 1. Before the fix, memory usage grows almost linearly with time, indicating a leak. After fixing all deallocation paths and using `SAFE_FREE`, the memory footprint remains stable over the entire run, with a clean release at the end of execution.

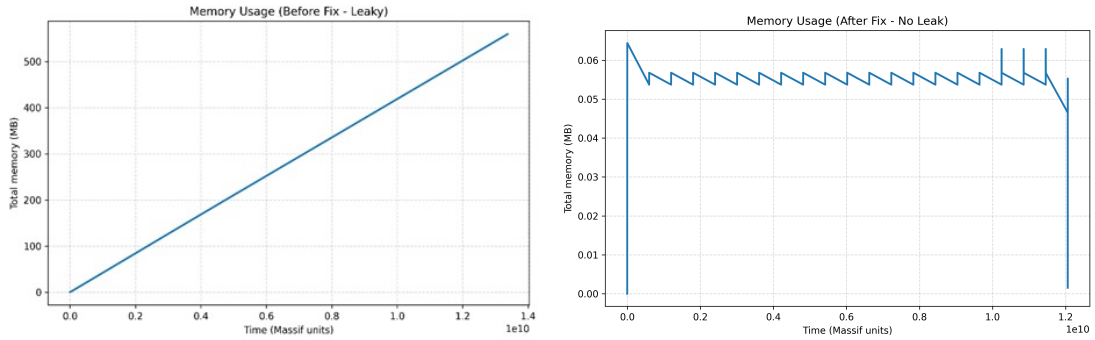

Figure 1: Memory usage of the MLP model after and before fixing memory leaks (Valgrind Massif output).

```
#ifndef SAFE_FREE
#define SAFE_FREE(p) do { if ((p)!=NULL) { free(p); (p)=NULL; } } while (0)
#endif
```

Figure 2: Implementation of the `SAFE_FREE` macro for memory deallocation.

Overall, after applying these fixes, Valgrind reports zero leaks, and the model can be trained for many epochs without unbounded memory growth.

2.2 Performance Profiling

Objective Description

This objective focuses on analyzing the execution performance of the MLP code in order to understand which parts of the code consume the most CPU cycles.

Profiling Setup

We used Valgrind Callgrind to collect detailed profiles of function-level instruction counts, and visualized the results using KCachegrind. The code was compiled with debug symbols (and without aggressive optimization) to obtain useful source-level information.

Results

The profiles reveal that the main hotspots reside in the central numerical core of the model, particularly in dense matrix operations (forward propagation) and the gradient computations in backpropagation. This motivated the later OpenMP parallelization of these nested loops.

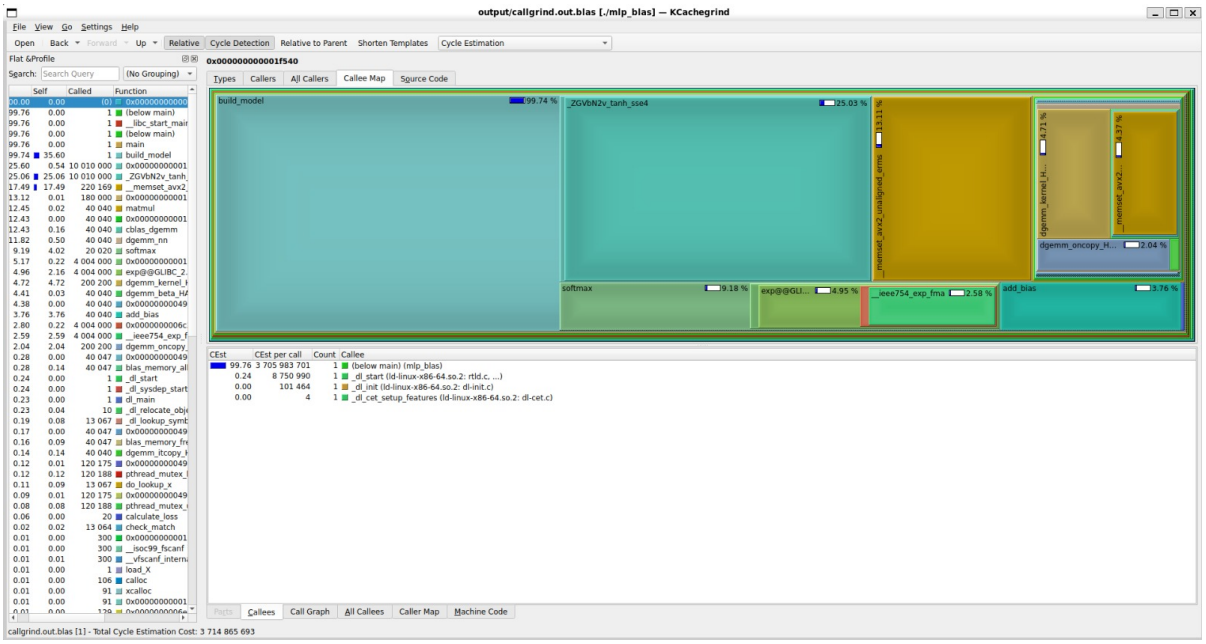

Figure 3: KCachegrind visualization of function-level execution costs.

2.3 Training Optimization

Objective Description

This objective focused on integrating Mini-Batch Gradient Descent into the training loop to balance convergence stability, noise in the gradient estimates, and computational cost.

Implementation Details

The training procedure in `build_model()` was adapted to iterate over mini-batches of size B (e.g. $B = 32$ or 128). For each batch, the forward pass (linear layers + nonlinearity + softmax) and backpropagation are computed using temporary buffers. At the end of each batch, model weights are updated with the averaged gradient.

Results

Figure 4 shows the evolution of loss and accuracy over epochs with mini-batch gradient descent. The loss decreases smoothly while accuracy increases toward high values, demonstrating that MBGD provides a good compromise between convergence behavior and computational efficiency.

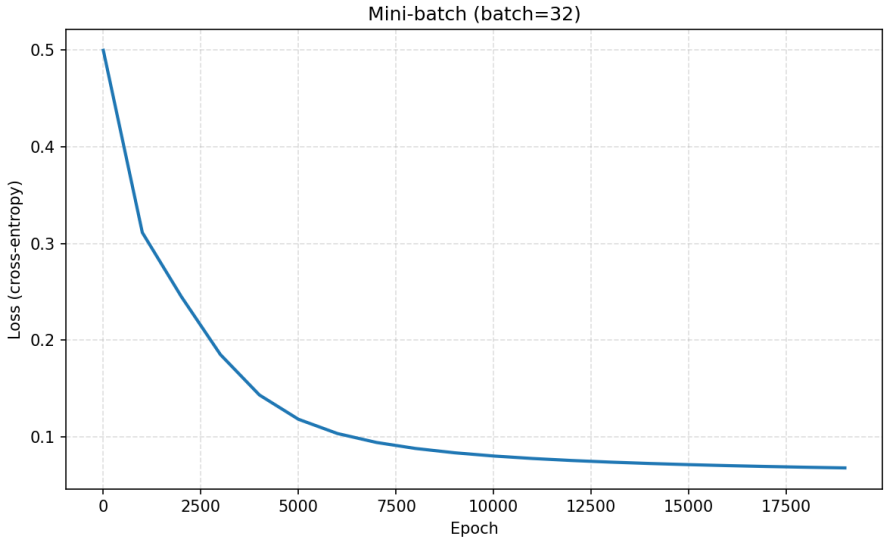

Figure 4: Training curve using Mini-Batch Gradient Descent (loss and accuracy vs. epochs).

2.4 Dynamic Learning Rate (Annealing Schedule)

Objective Description

Here we evaluate how different learning-rate (LR) schedules affect the convergence speed, final loss, and accuracy of the MLP trained with mini-batches.

Implementation Details

We implemented a small LR scheduler, called at each update step. Four main schedules were used: constant, time-based, exponential decay, and a step-based schedule. Implementation details are summarized in Table 1.

Table 1: Implementation details of the dynamic learning-rate (LR) scheduler used during training.

Schedule	Formula	Key parameters	Behavior
Constant	$\eta_t = \eta_0$	$\eta_0 = 0.01$	Fixed step size
Time-based	$\eta_t = \frac{\eta_0}{1+kt}$	$k = 10^{-4}$	Gradual decay with t
Exponential	$\eta_t = \eta_0 e^{-kt}$	$k = 9 \cdot 10^{-6}$	Faster decay over time
Step-based	$\eta_t = \eta_0\, \gamma^{\lfloor t/s \rfloor}$	step size s, factor γ (e.g. $s = 4000$, $\gamma = 0.5$)	Piecewise-constant LR with sudden drops

Results

Our results demonstrate the impact of different learning-rate schedules on convergence dynamics. The constant schedule achieves the lowest final loss and the most stable training, while the time-based decay offers a robust alternative with slightly slower convergence. The step-based schedule behaves in between the time-based and exponential schemes. The exponential schedule decays the LR too aggressively for this configuration, leading to early saturation of the loss and accuracy.

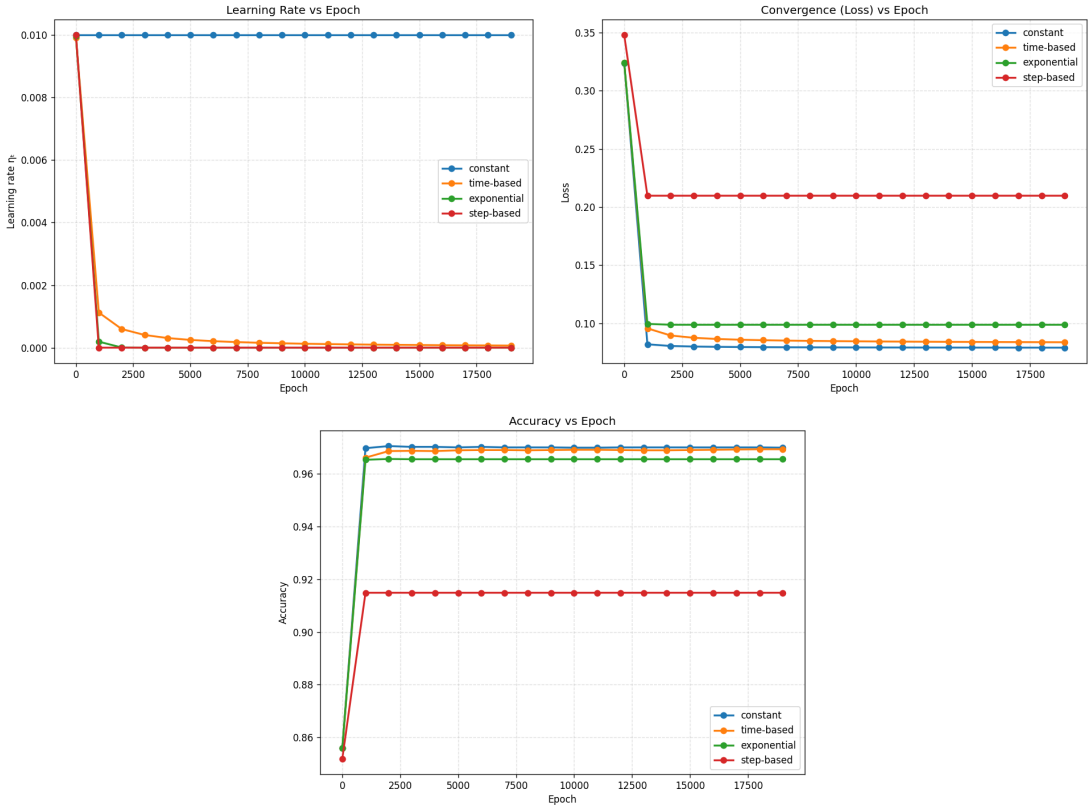

Figure 5: Comparative impact of learning-rate schedules on convergence, learning dynamics, and accuracy.

2.5 Activation Functions

Objective Description

The goal of this objective is to analyze the influence of different activation functions on learning dynamics and final performance, since the choice of nonlinearity directly affects gradient flow and stability.

Implementation Details

To enable activation testing, we implemented modular activation and gradient routines controlled by a global `activation_type` selector. This allows easy switching between Tanh, ReLU, Sigmoid, and Leaky ReLU while keeping the rest of the architecture fixed. Code snippets for the activation and gradient computations are shown in Figure 6.

```c
static inline double act_fn(double x, int type) {
    switch (type) {
        case 1:  return (x > 0.0) ? x : 0.0;
        case 2:  return 1.0 / (1.0 + exp(-x));
        case 3:  return (x > 0.0) ? x : leaky_alpha * x;
        default: return tanh(x);
    }
}

static inline double act_grad_from_a(double a, double x, int type) {
    switch (type) {
        case 1:  return (x > 0.0) ? 1.0 : 0.0;
        case 2:  return a * (1.0 - a);
        case 3:  return (x > 0.0) ? 1.0 : leaky_alpha;
        default: return 1.0 - a*a;
    }
}

static const char* act_name(void){
    switch (activation_type){
        case 1: return "relu";
        case 2: return "sigmoid";
        case 3: return "leaky";
        default: return "tanh";
    }
}
```

Figure 6: Code snippets implementing activation and gradient computation for different functions.

Results

The results show loss vs. epoch for each activation function. Tanh and Leaky ReLU provide the fastest and most stable convergence, with final accuracies around 0.95–0.96. ReLU performs similarly but can be slightly more sensitive to initialization. Sigmoid saturates early, leading to vanishing gradients and a lower final accuracy (around 0.81), confirming its limitations for deeper networks in this setting.

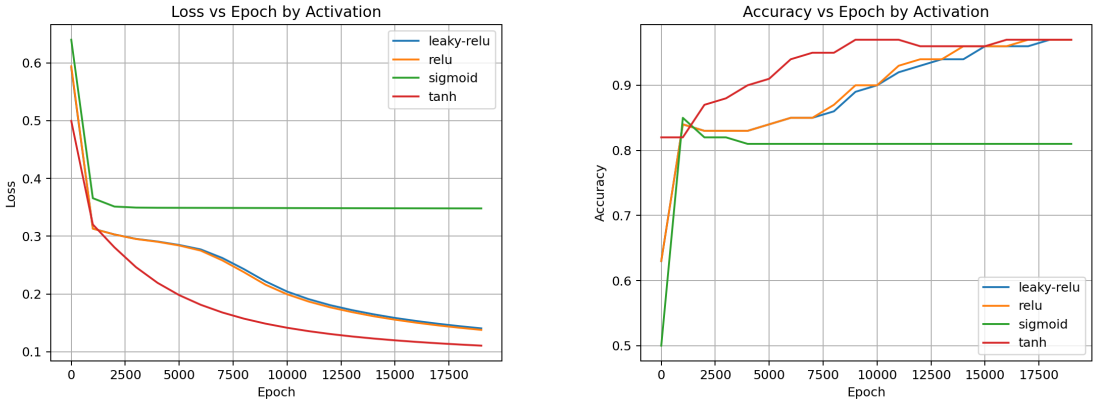

Figure 7: Performance comparison of Sigmoid, Tanh, ReLU, and Leaky ReLU activations on loss and accuracy.

2.6 Parallelization

Objective Description

The final objective is to accelerate the MLP training using OpenMP on a shared-memory node. The sub-goals are:

- parallelize nested loops in matrix multiplications and gradient calculations;
- explore the use of OpenMP tasks for independent batch computations;
- measure performance scaling with different thread counts.

Implementation Details

We first introduced `#pragma omp parallel for` directives around the outer loops of the most expensive kernels: forward matrix multiplications, gradient computations for W_1, W_2, and the bias terms. This "NO-TASKS" version uses mini-batch SGD and parallelizes only the inner numerical loops.

Then we implemented a task-based mode, where each mini-batch is processed as an independent OpenMP task. A single `omp parallel` region with a `single` construct creates one task per batch. Each task computes forward and backward passes and writes to thread-local gradient buffers, which are reduced at the end of the epoch into global gradients. The environment variable `OMP_TASK_MIN_BS` controls the minimum batch size required to spawn a task, avoiding overhead for tiny batches.

Results

All OpenMP experiments are performed on Toubkal with 10 000 training samples, hidden dimension 128, 2000 passes, batch size 128, and time-based LR. Table 2 reports the measured training times and speedups for both the NO-TASKS and TASKS implementations.

Table 2: OpenMP scaling results: runtime and speedup for NO-TASKS (loop parallelization) and TASKS (batch-parallel tasks) modes.

Threads	Time NO-TASKS [s]	Speedup NO-TASKS	Time TASKS [s]	Speedup TASKS
1	119.20	1.00	127.21	1.00
2	72.14	1.65	63.88	1.99
4	52.94	2.25	32.84	3.87
8	43.09	2.77	17.94	7.09
16	72.42	1.65	11.29	11.27

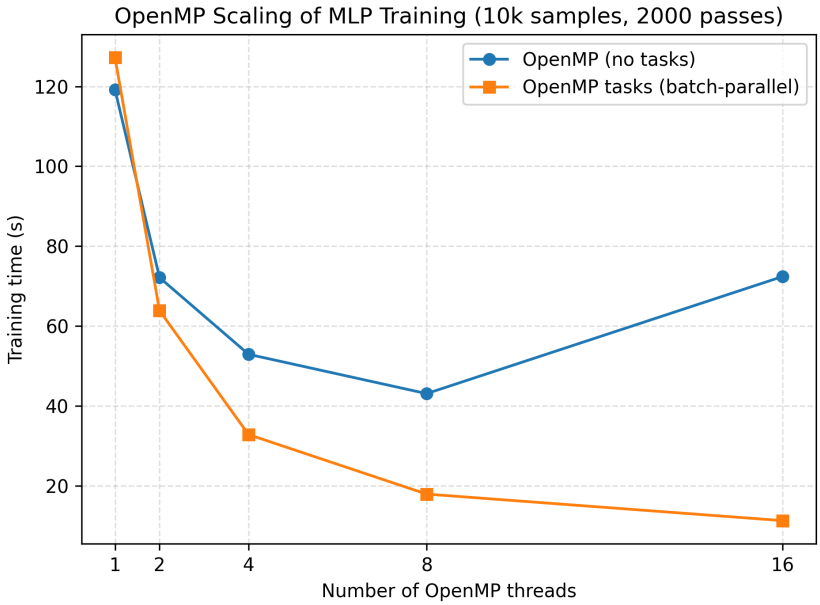

Figure 8: OpenMP execution time vs. number of threads for the loop-based (NO-TASKS) and task-based (TASKS) implementations.

In NO-TASKS mode, loop parallelization yields modest speedups up to 8 threads (2.77×) before performance degrades at 16 threads, indicating memory and overhead limitations. In contrast, the TASKS mode scales much better: runtime decreases from about 127 s at 1 thread to 11.3 s at 16 threads, corresponding to an 11.3× speedup and roughly 70% parallel efficiency. Task granularity tuning via `OMP_TASK_MIN_BS` slightly reduces overhead for small batches but does not change the overall conclusion that batch-level tasks significantly improve scalability compared to loop-only parallelism.

2.7 MPI Data Parallelism

Objective Description

In the last part of the project we extend the MLP to a distributed-memory setting using MPI. The goals are:

- run the model in a *data-parallel* way on several MPI processes;
- let each process train on a different subset of the mini-batches;
- synchronize weight updates with the collective operation `MPI_Allreduce`;
- measure speedup and parallel efficiency when increasing the number of processes.

Implementation

After loading the dataset, the N training samples are split evenly across the P MPI ranks. Rank p only sees its local chunk $(X^{(p)}, y^{(p)})$ and runs the usual mini-batch SGD loop on that data. For each epoch and mini-batch it computes local gradients $\nabla W_1^{(p)}, \nabla b_1^{(p)}, \nabla W_2^{(p)}, \nabla b_2^{(p)}$.

Before updating the weights, all ranks call `MPI_Allreduce` with `MPI_SUM` to sum their gradients:

```
MPI_Allreduce(grad_local, grad_global, G, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

The global gradient is then divided by P and used to update the parameters. Because every rank applies the same update, all models stay synchronized. The same code also supports a hybrid MPI+OpenMP mode, where each rank uses a few OpenMP threads inside the local forward and backward passes.

Results

We first performed a strong-scaling test with a fixed problem size (10 000 samples, hidden size 128, batch size 32, 2000 epochs) and varied the number of MPI ranks P. Table 3 and Figure 9 summarize the results.

Table 3: MPI strong scaling: runtime and speedup for data-parallel training.

MPI ranks P	Time [s]	Speedup $S(P)$
1	48.56	1.00
2	27.02	1.80
4	19.77	2.45
8	21.46	2.26

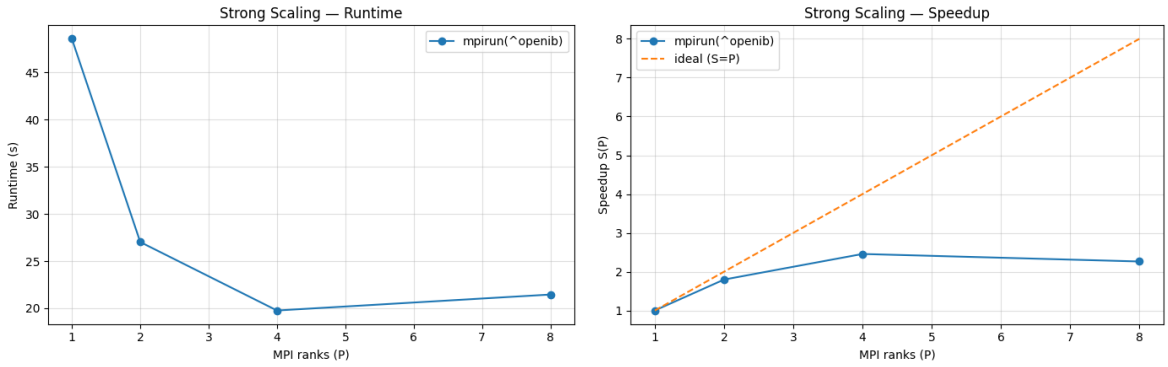

Figure 9: MPI strong scaling: runtime (left) and speedup with ideal line $S(P) = P$ (right).

The model scales well up to $P = 4$ (speedup $\approx 2.5\times$) with reasonable efficiency. At $P = 8$ the runtime slightly increases and speedup drops, showing that collective communication (`MPI_Allreduce`) and smaller local workloads start to dominate.

To explore hybrid configurations, we fixed the total number of cores to 16 and changed the split between MPI ranks P and OpenMP threads per rank T ($P \times T = 16$). As shown in Figure 10, the configuration ($P = 8, T = 2$) is fastest, while a single process with $T = 16$ threads is slowest. This suggests that combining several MPI processes with a small number of threads per process is more effective than a purely OpenMP setup.

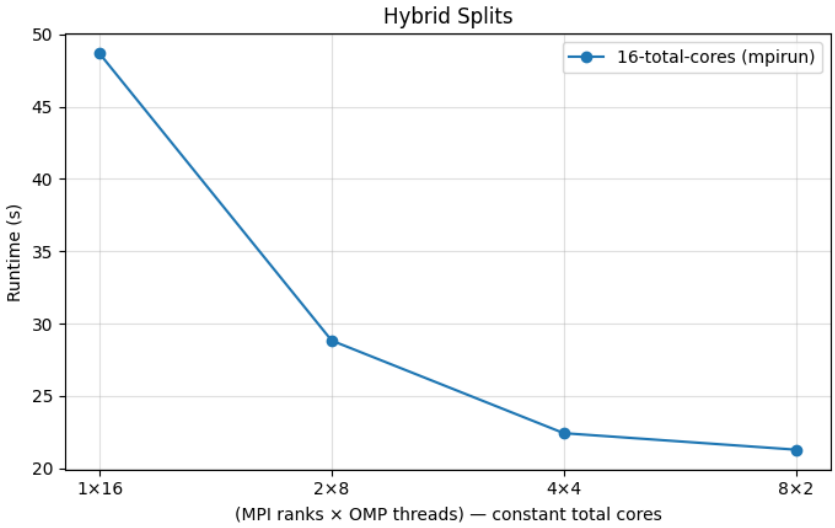

Figure 10: Hybrid MPI+OpenMP: runtime for different (P, T) splits with 16 total cores.

Finally, we varied the mini-batch size for a fixed configuration ($P = 4, T = 2$). The curve in Figure 11 shows a clear minimum around batch size 256: smaller batches lead to too many synchronizations, while very large batches make each iteration heavier. This confirms that the batch size is an important tuning knob for performance in data-parallel training.

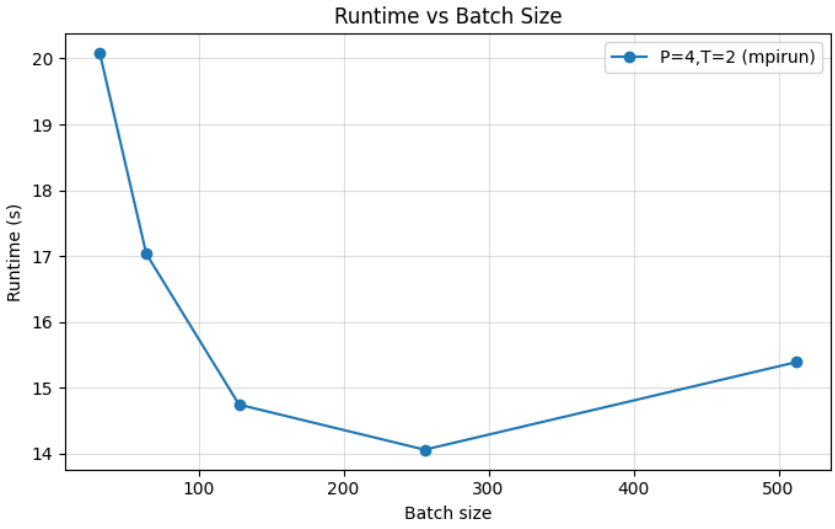

Figure 11: MPI+OpenMP (P=4,T=2): runtime as a function of batch size.

3 Conclusion

We started from a basic sequential C MLP and turned it into a safe, fast, and scalable training code. Valgrind removed memory bugs, profiling exposed dense linear algebra as the main bottleneck, and switching from full-batch GD to mini-batch GD improved convergence and flexibility. Learning-rate schedules, activation choices, and careful optimisation further improved accuracy and stability.

OpenMP loop parallelism gives moderate gains, while OpenMP task parallelism provides much better scaling within a node. Adding MPI data parallelism reduces runtime even further and enables hybrid MPI+OpenMP execution. Overall, this project shows how classical HPC

techniques can greatly enhance the performance and scalability of a simple neural network implemented in C.