Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.8178

Research Paper

Learning Parallel Computing with Multilayer Perceptron Neural Networks

Mateus Henrique Medeiros Diniz [Pontifícia Universidade Católica de Minas Gerais | *mateushmdiniz@gmail.com*]
Henrique Cota de Freitas [Pontifícia Universidade Católica de Minas Gerais | *cota@pucminas.br*]

Computer Architecture and Parallel Processing Team (CArT), Department of Computer Science, Pontifícia Universidade Católica de Minas Gerais, R. Dom José Gaspar, 500, Coração Eucarístico, Belo Horizonte, MG, 30535-901, Brazil.

Abstract. As multi-core architectures become the standard for computational efficiency, parallel programming has evolved into a primary avenue for optimization and resource management. However, an integrated design-and-evaluation method is often difficult for beginners to envision, leading to incorrect intuition about the relationship between theory and practice. This article proposes an implementation-driven methodological approach using a Multilayer Perceptron (MLP) to bridge this gap, comprising three main components: (1) establishing a sequential baseline, (2) applying parallelization and data containerization strategies to the MLP, and (3) evaluating through key metrics such as speedup and hardware efficiency. By addressing data dependencies and synchronization challenges inherent to backpropagation, the proposed trajectory provides a framework for comprehending data integrity in parallel systems through an exemplified parallel implementation. Our approach led to successful optimization of the feedforward and backpropagation routines, resulting in a speedup, e.g., of 4 times the baseline execution time using 8 hardware threads. Furthermore, the results indicate that outer-loop nesting and data containerization should be considered for managing structures with high data dependencies.

Keywords: Parallel Programming, Performance Evaluation, Multilayer Perceptron Neural Network, Parallelism Education

Received: 14 May 2026 • **Accepted:** 17 July 2026 • **Published:** 07 August 2026

1 Introduction

Many computational solutions nowadays require or could benefit from performance enhancements in code, such as those in artificial intelligence, finance, and scientific-domain applications. The demand for speed, alongside new multi-core processing unit architectures, invokes concepts from the high-performance computing (HPC) domain, such as parallelism, which require strategies and methods for teaching and learning parallel programming [de Freitas, 2012; Vasconcelos *et al*., 2019; Conte *et al*., 2020].

Several approaches to studying parallel programming [Lara Soares *et al*., 2019; Carneiro Neto *et al*., 2022] involve identifying potential parallelization strategies for an implementation that works as intended sequentially and determining which concepts best fit parallel architectures to maximize performance. To familiarize oneself with parallelism, working in a domain that offers clear and effective opportunities for parallelism is a forgiving yet powerful exercise.

Artificial intelligence is a domain rich in opportunities for parallelism. Specifically, in the context of Multilayer Perceptrons (MLPs) [Indrakumari *et al*., 2021], the independence of neuron weight calculations within a single layer, for example, enables significant acceleration through simultaneous execution on high-performance architectures. It is easily identifiable and affects processing speed, making it a promising candidate for approaches that use practical and case-based learning.

Although there are many studies on parallel computing education, we observe a lack of work associated with artificial intelligence that provides a practical incentive and a student-friendly approach that systematizes concepts into a design and evaluation method. Usually, when algorithms are not isolated excerpts with specific conditions (e.g., loops, scheduling, and critical sections), common educational practice focuses on small algorithms such as sorting, summing, and matrix multiplication. Given the inherent limitations of each, programming techniques and practices can be presented without criticism of the best approaches or programming choices. A practice specifically oriented toward algorithms, with more options for design and evaluation, could solve this issue.

Therefore, choosing an algorithm-based learning method aligned with the high demand for artificial intelligence can help students understand the need for parallelism. Furthermore, as parallelism is fundamentally coupled to hardware, it demands a grasp of low-level intricacies that remain hidden within purely theoretical frameworks. This way, our problem statement is fundamentally grounded in the lack of an approach, supported by a simple artificial intelligence algorithm, for studying parallel computing concepts from design to evaluation. Thus, the research question is: Does an artificial intelligence algorithm present parallelization opportunities that can be critically analyzed in terms of design and evaluation?

Given these points, our goal is to develop an approach that connects knowledge of OpenMP-based parallel programming with an Artificial Neural Network (ANN), specifically an MLP. We recognize that the transition from abstract rule-following to intuitive systems engineering is inherently subjective. For this reason, our proposed approach does not offer a universal formula but a guided trajectory. The main contribution of our work is an implementation-based methodological approach (design and evaluation) that integrates theoretical neural networks with practical parallelism techniques via the OpenMP API and is applied to a sequential MLP algorithm.

In terms of educational aspects, our method focuses on

analyzing MLP algorithms for design and evaluation across different parallelism approaches. This paper presents analyses of Feedforward, Backpropagation, and Mini-batch Gradient Descent to explore possibilities for parallelization, giving students a critical analysis of programming choices. In the scope of this paper, we highlight the importance of execution time, speedup, and efficiency as performance metrics, and accuracy as a baseline for predictive quality. We understand that these metrics represent a subset of the alternatives for evaluating any program, but we believe they also introduce the fundamentals for any student beginning research on the application of parallelism to artificial intelligence. The target of our work is an analysis of a design and evaluation approach, which includes computational performance for learning parallelism. The scope of this work does not include an analysis of student performance, as this must be addressed within the framework of an educational program involving the students.

This paper is organized as follows: Section 2 describes the background concepts, Section 3 presents related work, Section 4 details our methodological approach, Section 5 presents our parallelism design, Section 6 presents the algorithm performance evaluation results, and finally, Section 7 concludes the paper.

2 Background

This section describes the fundamental concepts necessary to understand certain decisions regarding our parallel MLP algorithm's design and evaluation. At the end of this section, we describe important cause-and-effect correlations that are important for deeper studies in parallel computing education.

2.1 Multilayer Perceptron

An ANN is an artificial intelligence model philosophically similar to the biological brain, composed of neurons and connections. One of the most basic types of an ANN is the feedforward neural network, in which data enter the input layer and propagate through the connections to the output layer [Islam *et al.*, 2019].

The feedforward relies on neuron activations, weights, and biases. The activation is a value held by a neuron, whereas the weights are values assigned to the connections between neurons. The biases are extra inputs connected to the neurons, each with its own weight [Rosenblatt, 1958].

The general formula that allows data to travel between layers is as follows (**Equation 1**):

$$y_j = f\left(\left[\sum_i y_i \cdot w_{ji}\right] + b_j\right) \tag{1}$$

where y_j is the activation value of the j^{th} neuron of the current layer, y_i is the activation value of the i^{th} neuron of the previous layer, w_{ji} is the weight of the connection between the current neuron j and the previous neuron i, b_j is the bias related to the current neuron, and f is the activation function.

The perceptron is one of the most famous neural networks. Historically, it started very simply: a static network with no hidden layers. That was until backpropagation was proposed. With this algorithm, it became possible to adjust the weights to minimize error. It was also when the hidden "units" were first mentioned. These contributed to the development of new versions of the perceptron with intermediate layers, commonly referred to as MLPs [Macukow, 2016; Rumelhart *et al.*, 1986]. **Figure 1** depicts a simple MLP example.

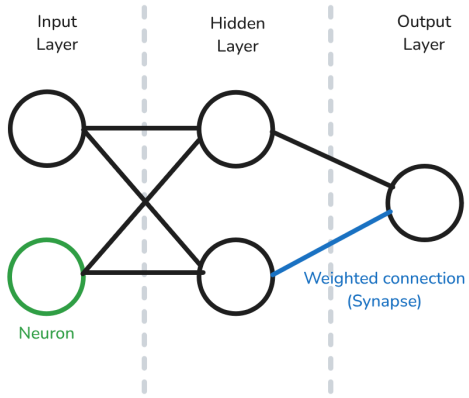

Figure 1. Multilayer Perceptron example

The intrinsic structure of multiple, variable-length hidden layers is the primary reason MLPs were selected as our parallel programming case study. While an MLP can become computationally complex simply by adding a new layer or slightly increasing a layer's size, the independence of both neuron calculations and training samples in Mini-Batch Stochastic Gradient Descent forms a clear path to parallelizing such an algorithm. Furthermore, the strict dependencies between layers in an MLP naturally promote the discussion of data integrity and synchronization under parallel implementations.

2.2 Parallelism

Parallelism is the process of subdividing a large task into multiple smaller instances, computing them simultaneously across different processing units, and then merging the results into the solution [Kumar, 2017]. With its increased accessibility [Ayguade *et al.*, 1999], it has become possible to enjoy the benefits on a personal computer, allowing most to put it into practice.

One could parallelize software applications. This is known as software-level parallelism, facilitated by high-level extensions to existing programming languages [Ayguade *et al.*, 1999]. Already well-explored, the software layer offers several techniques to transform what was once thought, planned, and executed sequentially into a parallelized task, partitioned across process models [Kumar *et al.*, 2006].

By descending the abstraction layer, instruction-level parallelism is achieved. The hardware itself exploits parallelism through specialized instructions to either achieve the same goals as software-level parallelism or to parallelize task execution. Although less flexible and affected by hardware limitations, there are still opportunities for improvement [Chadwick *et al.*, 2025].

Although promising, parallelization is not always more performant than a sequential approach. Some nuances in thread and shared-resource management can become significant bottlenecks that nullify parallelization efforts or lead to incorrect behavior, invalidating the solution [Pawar *et al.*, 2014].

To ensure the correct application of parallelization tech-

niques, it is necessary to analyze the context and data usage. Pawar *et al*. [2014] presented an approach that applied mathematical concepts to formulate a more effective strategy. These concepts involve, e.g., analyzing the commutativity and associativity of the problem. Such characteristics are important for proving reliability.

To evaluate our parallel implementation, this study focuses on performance and scalability [Machado *et al*., 2026], using two main metrics: speedup (S_p) and efficiency (η). The speedup shows the relative speed gain ($S_p > 1$) or loss ($S_p < 1$) compared to the base speed and can be calculated as follows (**Equation 2**):

$$S_p = \frac{t_{base}}{t} \tag{2}$$

where t_{base} is the base time, such as the original algorithm execution time, and t is the current execution time, such as the optimized algorithm execution time.

The efficiency measures the hardware's processing capabilities usage. Sub-utilization ($\eta < 1$) indicates the presence of bottlenecks, such as sequential fragments in code, that limit how much the program can utilize hardware resources, in contrast with an ideal efficiency ($\eta = 1$), which means the resources are used to their limits without distractions. There is also a super-linear efficiency ($\eta > 1$), where factors not strictly related to the processor, e.g., cache locality, push the implementation beyond the processor's theoretical capabilities. It can be calculated as follows (**Equation 3**):

$$\eta = \frac{S_p}{C} \tag{3}$$

where S_p is the speedup, and C is the number of available cores.

Furthermore, the state of sub-utilization of resources is predicted by Amdahl's law [Amdahl, 1967; Hill and Marty, 2008], as described in **Equation 4**: sequential parts (f_{seq}) of the algorithm effectively limit the potential for parallelization, asymptotically limiting speedup and yielding diminishing returns.

$$S_p = \frac{1}{f_{seq} + \frac{1-f_{seq}}{C}} \tag{4}$$

For parallel scaling, these metrics will be used in the context of strong scaling. In strong scaling, the problem size remains constant while the number of cores varies. It contrasts with weak scaling [Gustafson, 1988], in which the problem size scales with the number of cores. Strong scaling is primarily used for fine-tuning in CPU-bound applications, i.e., adjusting the environment parameters until a satisfactory result is obtained, whereas weak scaling is preferred for improving memory scalability in memory-bound applications. In general, more cores scale better with memory due to CPU affinity: neighboring cores might share the same cache levels, depending on the architecture.

2.2.1 Design and Evaluation Remarks

As previously described at the beginning of this section, cause-and-effect correlation can lead us to a more complex scenario, adding more research fields for a better comprehension. Computer architecture and, consequently, hardware characteristics commonly affect the algorithm's performance [Magalhães *et al*., 2024]. For a given sequential algorithm with high cache misses, when its parallel code executes on two cores, cache misses can be reduced because the workload now has more cache memory at the same level to distribute data, resulting in higher cache hits. This behavior can generate a super-speedup [Che and Nguyen, 2014] that exceeds the limits defined by Amdahl's law, which applies to the number of processing cores and parallel parts of the code. In other words, for this example, the super-speedup is due to a stronger influence of more cache memories rather than to a high number of parallel parts of the code. To identify possible hardware characteristics, such as cache misses, page faults, and others, hardware counters present in the processor architecture can provide information for a good cause-and-effect correlation.

According to the concepts of strong and weak scalability, a parallel code maintains speedup and efficiency as more processing cores are added, with or without an increase in problem size. Again, hardware counters, along with other strategies for monitoring program performance, can provide the programmer with a pool of data to analyze and explore all possible processing cores for each thread. This research field presents CPU and memory affinity [Ribeiro *et al*., 2012] as important concepts for better exploration of thread-to-core mapping [Bindi *et al*., 2026]. For instance, in a multicore architecture, shared caches can be leveraged to increase cache hits through thread pinning (i.e., affinity), which maps threads to cores that share the same cache. Thread mapping can optimize resource utilization by preventing thread migration from the operating system and scaling the parallel code to achieve the highest speedup and efficiency across many processing cores. This behavior can improve the scalability of parallel code, enabling it to run on more cores and process more data efficiently.

These remarks highlight important cause-and-effect correlation concepts that can open the mind to a broad range of learning opportunities. This paper describes part of them and suggests future work in more advanced parallel computing education.

3 Related Work

The related papers in this section were found using the following search criteria: (1) IEEE Xplore, ACM Digital Library, and SBC-OpenLib as digital libraries and Google Scholar as search engine; (2) "parallelism", "parallel programming", "high performance computing", "neural networks", "teaching" and "multilayer perceptron" as the main keywords, used in different combinations; (3) the period from 2024 to 2026, with the time range extended as necessary.

In the context of parallel programming and heterogeneous computing, Abdurhaman *et al*. [2024] examined the effectiveness of HPC instruction through hands-on activities, drawing on their experiences from ministering summer workshops for high school students. By exposing them to HPC topics through practical examples rather than abstract theory, the overall understanding and feedback remained satisfactory. Their results showed that working in practical activities to tackle advanced concepts makes the content more digestible, even for those without a background in the subject.

Most algorithms have simpler, slower versions for learning and complex, faster versions for real-world problems. However, Strazdins [2025] proposed a different parallelization algorithm that is both easier for students to understand than traditional methods and more performant. Their initiative to think beyond already-consolidated techniques demonstrated that there remains space for innovation, whether in education or applications.

Vargas-Pérez [2024] showed a concern regarding learning performance metrics for parallel solutions in conjunction with implementation exercises. Measuring this data is a great way to develop critical thinking and understanding. The method yielded promising results regarding the impact of such exercises on students and served as an incentive to explore the topic further.

To mitigate the negative effects of the fast-paced nature of Computer Science courses on learning, Iudean [2024] investigates the use of storytelling techniques in teaching HPC courses. The narration-driven approach successfully created connections between meaningful stories and parallelism concepts, supporting students in developing their knowledge in a less formalistic manner.

Given the relevance of parallel computing for Graphics Processing Units (GPUs), Lupo *et al*. [2012] adopted a collaborative approach to optimize Ray Tracing algorithms. By uniting two undergraduate classrooms (one specialized in graphics processing and the other in parallel solutions), the study integrated distinct domains into a unified hands-on example. This strategy successfully consolidated theoretical studies into practical skills, demonstrating that complex concepts are best internalized when anchored in high-impact applications.

Together, these works demonstrate the importance of practical experiences in the learning journey of parallel programming students. They show how connecting explicit knowledge to palpable problems enhances learning, increasing students' conceptual mastery throughout their academic experience. However, these related works do not focus on learning parallel programming in the context of artificial intelligence, guided from design to performance evaluation. We understand that a well-applied approach, as we propose in this paper, can help students to understand design decisions and their implications on performance.

Moreover, our work addresses the following gaps left by the related work:

1. **Methodological transition from baseline to complex parallelism**: details of the expository transition from sequential code to managing the high data dependencies inherent to routines like backpropagation.
2. **Handling data integrity in nested structures**: an analysis of how specific parallelism behaviors and data containerization strategies resolve the synchronization challenges unique to the proposed case study.
3. **Connection between theory and practice**: demonstrating exactly how structural techniques apply to the computational demands of the MLP, bridging the gap between parallel concepts and the actual implementation.
4. **Evaluation of hardware utilization**: presentation and reasoning of common HPC metrics to evaluate how the MLP scales across different thread counts and parallelism approaches.

Table 1 summarizes our work and related work based on the approach used to address the problem, the target devices for implementation, the languages used, and the algorithms employed. The diversity of approaches underlines the subjectivity of learning. As there is no universal formula for mastering high-performance computing, the field benefits from new perspectives. While many existing materials focus on throughput, our work addresses the challenge of data integrity and synchronization. By employing an algorithm with strong data dependencies, such as the MLP, we provide a guided trajectory that forces the HPC learner (undergraduate or graduate students) to understand the design and evaluation steps to achieve high-performance computing.

4 Methodology

The experimental approach aims to isolate the performance impacts of specific parallelization strategies on a standard machine learning task. By keeping close to the hardware, the relationship between code, OpenMP directives, and hardware resource usage becomes clear. The following subsections detail the environment and tools used to ensure reproducible and accurate benchmarks. The main elements of our approach are described in this section and can be highlighted as materials (hardware and tools), benchmarking procedures, MPL parallelization, and performance evaluation. These elements comprise our design- and evaluation-based learning proposal, which is aligned with the background, parallelization approaches, and results sections.

4.1 Hardware & Tools

All tests were performed on the same machine equipped with an 11th-generation Intel Core i5 processor. A topological map of this processor is shown in **Figure 2**, which depicts four cores, 8-thread support, and cache information (private L1 and L2 caches and a shared L3 cache).

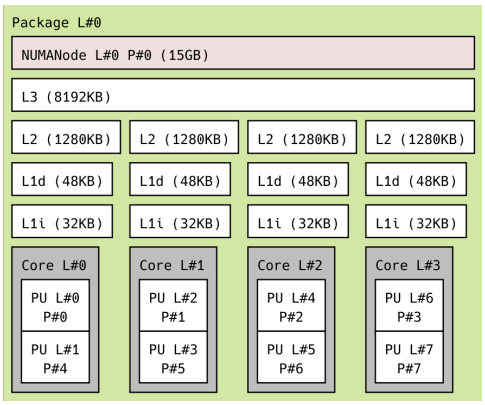

Figure 2. 11th Gen Intel Core i5 topological map

The implementation used for running the tests was written in C. In addition to the OpenMP API and standard libraries, no other tools were used to run the MLP model. The metrics were assembled into statistical visualizations using Python.

OpenMP is a user-directed parallelization API for C, C++, and Fortran. To be user-directed means that the pro-

Table 1. Comparative analysis of methodologies in parallel computing education.

Work	Target Audience	Approach	Device	Language	Algorithm
Abdurhaman *et al*. [2024]	High School	Hands-on Workshops	Edge devices/GPU	Python/C++	TSP, Mat. Mul.
Strazdins [2025]	Undergraduates	Tiled-based approach	CPU/GPU	C	LU, Mat. Mul.
Vargas-Pérez [2024]	Undergraduates	Performance metrics	CPU	C++	Histogram
Iudean [2024]	Undergraduates	Storytelling	-	-	-
Lupo *et al*. [2012]	Undergraduates	Collaborative work	GPU	CUDA	Ray Tracing
This work	**HPC Students**	**Implementation-Driven**	**CPU**	**C**	**Neural Network**

grammer explicitly defines instructions for parallelizing routines. Specifically, in C, this is primarily accomplished by using directives [OpenMP Architecture Review Board, 2025].

The API provides greater control over the parallelized code by design, an important characteristic given the scope of the problem, as transparency enables deeper analysis of the behavior of different parallelization strategies. In user-directed implementations, the responsibility for handling parallelism is shifted to the programmer, making them error-prone and highly modifiable, which is exactly what is needed.

4.2 Benchmarking Procedures

A benchmarking routine was performed for each of the presented approaches, as well as for any necessary proofs. Some routines were also performed for specific approaches while varying OpenMP environment variables. The benchmarks measured the time required to compute the entire routine and the model's accuracy.

Furthermore, two datasets were used to train the model. The first is a simple XOR test, generated exclusively for this study, with two inputs and one output: a non-linear problem that defined the boundary between the very first neural network model and the MLP.

The second dataset is the Modified National Institute of Standards and Technology (MNIST) dataset, comprising 28x28-pixel grayscale images of handwritten digits and their corresponding numeric labels from zero to nine. Since there are 784 pixels per image to work with, the complexity of the neural network increases rapidly. **Figure 3** shows an example of an image with its respective label.

The XOR dataset, being a simple binary dataset with two inputs and one output, was used to debug and validate the correctness of the algorithm and parallelization implementations. The MNIST dataset, being an image dataset with 784 inputs in grayscale format and 10 possible outputs, was used for the scaling tests presented in this article, as it is complex and requires more computational time, making it easier to observe changes in the measurements.

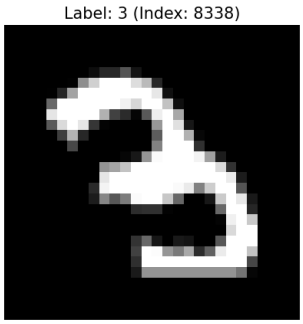

Figure 3. MNIST image and label example

4.3 Multilayer Perceptron Parallelization

The MLP algorithm is implemented using Stochastic Gradient Descent with Mini-batch optimization for weight updates. Furthermore, the MLP exposes a set of hyperparameters that allow configuration of the number of neurons and layers, batch size, number of epochs, and learning rate. Exceptions were made for the weight and bias initialization, which were random, and for the activation function, which was a fixed Sigmoid function as described in **Equation 5**:

$$a = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

where a is the activation result matrix, and z is the multiplication of the current layer weight matrix and the previous layer activation matrix.

Two main parallelization strategies compose our methodological approach, described below:

- **Intra-Layer Parallelism**: consists of parallelizing the portions of the code that handle neurons, weights, and biases, namely the feedforward and backpropagation algorithms;
- **Inter-Sample Parallelism**: consists of parallelizing the processing of independent training samples with a mini-batch. This strategy distributes the workload of multiple samples across threads.

4.3.1 *Extra Optmization*

Before parallelization, the algorithm underwent thorough optimization to mitigate issues beyond the scope of parallel programming.

- **Pre-calculated Derivatives**: The backpropagation algorithm needs the derivative of the sigmoidal function $\sigma'(z)$ for weight adjustment calculations. In a common implementation of a feedforward algorithm, the z parameter is discarded after the activation is stored. To avoid repeated calculations, an additional step was added to compute and store the derivative in advance, provided that the z parameter is known.
- **Flat Matrices**: All matrices were flattened in row-major order to favor spatial locality. Since the algorithm often iterates over these matrices in a single loop to perform neuron activation calculations, organizing the data contiguously in memory increases the likelihood of finding the next value in the loaded cache line, thereby increasing cache hits.

4.4 Performance Evaluation

The execution time was measured using C's time library, which provides precise clock time snapshots to nanosecond precision. To calculate the time required for a training routine to finish, it computes the difference between a snapshot taken immediately before execution and one taken immediately after. The goal was to remove all extra setup and cleaning time from the benchmarking results.

The method used to evaluate the algorithms and underlying optimization measures the time required to complete the `fit()` routine. This routine executes the mini-batch optimization algorithm on the provided neural network using a training dataset with its corresponding ground-truth labels.

To ensure environmental consistency on a mobile workstation, several precautions were taken to mitigate external interference. The device remained connected to a constant-power source throughout all measurements to disable power-saving frequency scaling and ensure that the processor operated in high-performance mode. Furthermore, all unnecessary general-purpose user applications were terminated.

Strong scaling was chosen for our methodology due to the workload's inherent CPU-bound nature. Evaluating the metrics across different core counts while keeping the problem size fixed ensures fairness in the benchmarking process. This approach allows for direct comparative conclusions without introducing more variables related to workload scaling or context switching. Conversely, weak scaling would be more appropriate if the primary objective were to investigate memory interactions or data distribution capabilities at scale.

To mitigate statistical noise and account for potential OS-level background activity, each benchmark was executed 10 times, as more iterations would not contribute to meaningful precision changes, and the total execution time was recorded for both the parallel and sequential code versions. The mean and the margin of error for the 95% confidence interval of the computational performance metrics are presented in Section 6.

5 Parallelization Approaches

This section details the application of the design method, in which parallelization decisions constitute the design of a parallel algorithm. These decisions are based on the algorithm and the programmer's knowledge, and changes to the sequential version can alter the potential parallel solutions.

We describe two approaches (intra-layer and inter-sample parallelisms), analyses, and techniques to parallelize three MLP algorithms, i.e., feedforward, backpropagation, and mini-batch gradient descent. Additional optimizations related to instruction-level parallelization and thread scheduling were implemented to improve performance.

5.1 Intra-Layer Parallelism

Algorithm 1 details the state of the intra-layer parallelism algorithm before the first parallelization attempt. In this code, each for loop is a potential source of parallelism. However, a key observation is that to compute any given layer l_i, the activations from the previous layer l_{i-1} are required. This creates a strict data dependency: $l_0 \leftarrow l_1 \leftarrow \cdots \leftarrow l_{n-1}$. If one layer depends on the previous layer, the former must wait for the latter to complete its computation. Therefore, attempting to parallelize the loop in **line 2** will predominantly fail due to data races, as one layer being processed by a thread accessing data from its predecessor, which is being processed by a different thread, can lead to unpredictable behavior.

With the first loop discarded, there is still a chance for parallelism to work with the inner loops. The loop in **line 5** iterates over the rows of the *i-th* layer and computes the neuron activations. The operations here are already well contained. All of them operate in relation to the `j` index and depend on data from a previous layer, which is immutable at this point. Given this analysis based solely on data dependency and availability, it is possible to parallelize with the following directive: `#pragma omp parallel for`. This directive automatically makes the `j` index private, a requirement for threads to use it safely, since a contained copy is created for each thread.

```
1:  procedure FEEDFORWARD(network, neurons, derivatives)
2:      for i ← 1 to network.layerCount do
3:          rows ← network.layerSizes[i]
4:          cols ← network.layerSizes[i − 1]
5:          for j ← 0 to rows do
6:              z ← network.biases[i − 1][j]
7:              for k ← 0 to cols do
8:                  z ← z + neurons[i − 1][k] · network.weights[i − 1][j · cols + k]
9:              activation ← σ(z)
10:             derivatives[i − 1][j] ← sigmoid′(activation)
11:             neurons[i][j] ← activation
```

Algorithm 1: Sequential Feedforward Algorithm

However, it is not possible to draw the same conclusion for the loop in **line 7**. The `z` variable is incremented by the product of the previous *k-th* neuron activation and the *k-th* weight of the connection between the former and the current *j-th* neuron. For the most basic parallel implementation, this leads to a race condition in the `z` variable.

The first option is to tag the arithmetic operation as critical with `#pragma omp critical`. This way, only one thread can change the `z` variable at a given time, forcing all other threads to wait. It is quickly noticeable, though, that this does not solve the problem at all. If threads must wait to perform the block's most important operation, the new

implementation not only does not differ from the sequential implementation but also may be slower, as there is no speedup to compensate for the thread-management bottleneck.

Another option is to use OpenMP's reduction clause in **line 7** as follows: `#pragma omp parallel for reduction( + : z )`. This approach creates a copy of the variable for each thread, which is equivalent to privatizing it. Once the loop is finished, the partial results are summed into the final result. Since the sum operation has the commutative property, the order in which the partial results are combined does not matter, as any sequence leads to the same result [Kumar, 2017].

Finally, the last option is to descend to the instruction level to exploit Single Instruction, Multiple Data (SIMD) while keeping the reduction clause. Instead of using threads to perform multiple sums at once, a single specialized instruction with a fixed vector length is used to carry out many of the same operations, which is especially convenient for operations within loops [Huang *et al*., 2024]. The directive declaration would be: `#pragma omp simd reduction( + : z )`.

Preliminary analysis indicates that there are many valid strategies for parallelization. It can be said that the parallelization in **line 5** is the preferred approach. However, this does not preclude optimizations in the batch layer, even though SIMD could still be used to parallelize at a lower level while thread resources are consumed by higher-level algorithms. For this reason, parallelizing the backpropagation algorithm becomes straightforward. Algorithm 2 shows the implementation.

procedure BACKPROPAGATION(network, target, workspace, ∇W, ∇B)
  $L \leftarrow$ *network.layerCount* $- 1$
  rows $\leftarrow$ *network.layerSizes*$[L]$, *cols* $\leftarrow$ *network.layerSizes*$[L-1]$
  for $j \leftarrow 0$ *to rows* **do**
    $\delta \leftarrow$ *workspace.dActZ*$[L-1][j] \cdot 2($*workspace.neurons*$[L][j] -$ *target*$[j])$
    workspace.partials$[L-1][j] \leftarrow \delta$
    $\nabla B[L-1][j] \leftarrow \nabla B[L-1][j] + \delta$
    for $k \leftarrow 0$ *to cols* **do**
      $\nabla W[L-1][j,k] \leftarrow \nabla W[L-1][j,k] +$ *workspace.neurons*$[L-1][k] \cdot \delta$
  for $i \leftarrow L-1$ *down to* 1 **do**
    for $j \leftarrow 0$ *to network.layerSizes*$[i]$ **do**
      for $k \leftarrow 0$ *to network.layerSizes[i+1]* **do**
        $p \leftarrow p +$ *weights*$[i][k,j] \cdot$ *workspace.partials*$[i][k]$
      $\delta \leftarrow$ *workspace.dActZ*$[i-1][j] \cdot p$
      workspace.partials$[i-1][j] \leftarrow \delta$
      $\nabla B[i-1][j] \leftarrow \nabla B[i-1][j] + \delta$
      for $k \leftarrow 0$ *to network.layerSizes*$[i-1]$ **do**
        $\nabla W[i-1][j,k] \leftarrow \nabla W[i-1][j,k] +$ *workspace.neurons*$[i-1][k] \cdot \delta$

Algorithm 2: Sequential Backpropagation Algorithm

There is a specialized loop block starting at **line 4** for the last layer, as the calculations differ from those in the other layers. In the first part of the algorithm, only the mentioned loop can be parallelized. For the second part, however, the outermost loop in **line 10** cannot be parallelized due to the same data dependency problem discussed previously. The loop in **line 11**, however, can run in parallel, and SIMD fits perfectly in **line 12**.

5.2 Inter-Sample Parallelism

Moving beyond weight and bias manipulations, the next approach is to parallelize the mini-batch split. To better illustrate, Algorithm 3 describes the sequential implementation.

The loop in **line 3** is not parallelizable because the algorithm requires the weights and biases to be adjusted from previous iterations. Stochastic Gradient Descent uses small step sizes throughout the learning curve, and unstable parameters can compromise the algorithm in unpredictable ways.

Initially, parallelizing a single pass through feedforward and backpropagation is not possible without resorting to critical sections or private variables, since the network matrices are used extensively. Making all these variables and data private is theoretically the correct approach if the complexity of privatizing non-trivial data structures is disregarded.

1: **procedure** FIT(network, dataset)
2: *workspace* $\leftarrow$ InitializeWorkspace()
3: **for** $e \leftarrow 0$ *to network.epochs* **do**
4: **for** *start* $\leftarrow 0$ *to dataset.size step network.miniBatchSize* **do**
5: *end* $\leftarrow \min($*start* $+$ *network.miniBatchSize*, *dataset.size*$)$
6: $\nabla W, \nabla B \leftarrow$ ZeroMatrices()
7: **for** $i \leftarrow$ *start to end* **do**
8: *workspace.neurons*$[0] \leftarrow$ *dataset.inputs*$[i]$
9: FEEDFORWARD(network, *workspace*)
10: BACKPROPAGATION(*network*, *dataset.target*$[i]$, *workspace*, $\nabla W, \nabla B$)
11: $\eta \leftarrow$ *network.learningRate*$/($*end* $-$ *start*$)$
12: **for** $L \leftarrow 0$ *to network.layerCount* $- 1$ **do**
13: **for** $j,k \in$ *Weights*$_L$ **do**
14: *network.weights*$[L][j,k] \leftarrow$ *network.weights*$[L][j,k] - \eta \cdot \nabla W[L][j,k]$
15: **for** $j \in$ *Biases*$_L$ **do**
16: *network.biases*$[L][j] \leftarrow$ *network.biases*$[L][j] - \eta \cdot \nabla B[L][j]$

Algorithm 3: Sequential Mini-batch Gradient Descent

The idea of the "workspace" appears in this scenario. Instead of relying on the compiler to generate code for complex data privatization, which it cannot do, one can manually replicate the structures. In an array of workspaces, each thread has its own unique workspace. The workspace contains what is needed for the algorithms to run. Since the data are hidden inside protected containers, there is no risk of race conditions.

The only issue that could be predicted in advance is memory resources. Having multiple copies of the same neural network requires a lot of space. In addition, indexing the correct workspace for a given thread is straightforward using `omp_get_thread_num()`, which returns a unique ID

for the current thread. This ID can be used for indexing the workspace.

6 Results

The results described in this section focus on understanding the impact of parallelization on computational performance, highlighting key aspects necessary for learning parallel programming with MLP algorithms. It is important to note that the scope of this paper does not cover student performance evaluation, which must be conducted with a proper syllabus and approved by an ethics committee.

Now that all possible approaches discussed in Section 5 have been accounted for, it is time to run both the parallelized and sequential MLP algorithms and compare their performance. From this point onward, all tests will measure a complete learning session using the full MNIST dataset. The model will undergo a "Fit" routine with a 10-epoch training session, where the full 600000 training samples from MNIST are traversed in each epoch. The result analysis is based on performance metrics using means and confidence intervals as described in Section 4.4.

To rule out distractions that do not contribute to speeding up the model as a whole, **Figure 4** provides evidence of the bottleneck discussed for Algorithm 1. Each approach was tested by running eight threads. It is worth noting that the critical strategy sample size had to be reduced by a factor of 12 to complete it promptly. **Figure 4** indeed proves the previously stated point. Critical sections are unreliable for the project's scope. In contrast, the **line 5** strategy performs astonishingly better than the **line 7** and SIMD strategies, even surpassing the expected 8-factor speedup from a generic parallel optimization using eight threads, despite the considerable instability compared to the others.

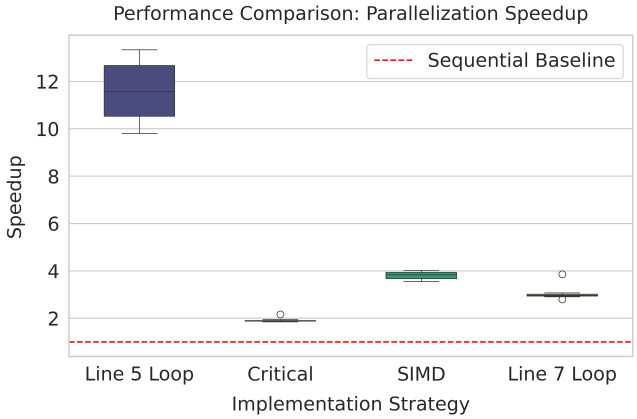

Figure 4. Speedup comparison between strategies

Akinsola *et al.* [2025] proposed a hyperparameter set for a Feedforward Deep Neural Network (FFDNN) that is highly accurate for the MNIST dataset. With a model comprising three hidden layers with 512, 256, and 128 neurons, respectively, the parallel strategies will truly demonstrate their capabilities under stress. However, networks of this size are highly computationally intensive, making results increasingly slower to obtain. Since the goal is to evaluate parallel performance rather than neural network accuracy, the first hidden layer was discarded, thereby removing 30% of the total neurons and 60% of the total weights. This results in a balance between case-specific and test hyperparameters. **Figure 5** shows the average and maximum time taken to complete the Fit routine for each of the relevant strategies as follows:

- **Sequential**: no parallelism at all
- **Intra-layer**: parallelism at the feedforward and backpropagation algorithms level with OpenMP's `#pragma omp parallel for`
- **Inter-sample**: parallelism at the mini-batch algorithm level with OpenMP's `#pragma omp parallel for`
- **Inter-sample with SIMD**: Inter-sample with SIMD optimization in the feedforward and backpropagation algorithm level

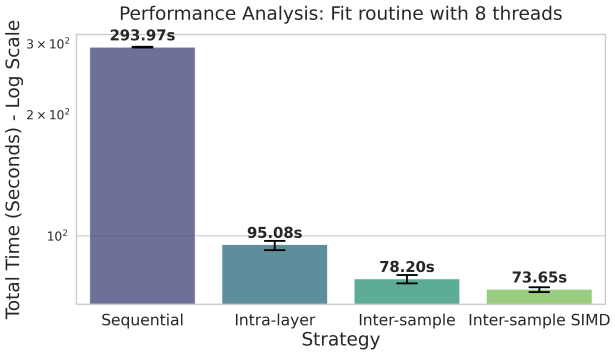

Figure 5. Fit routine with 8 threads

The difference between the parallelized and sequential MLP models is clear. The Intra-layer strategy achieved a speedup of approximately 3, whereas Inter-sample with SIMD achieved a speedup of approximately 4. To confirm that the model remained deterministic during optimization with different strategies, **Figure 6** shows the model's accuracy after each iteration of the Fit routine.

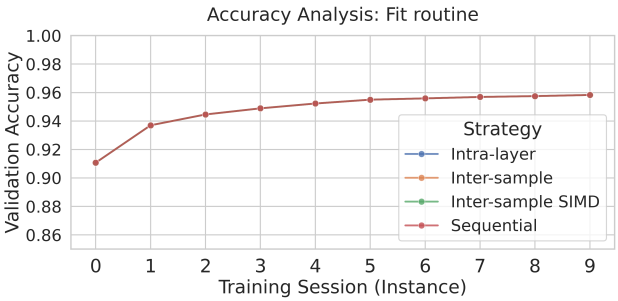

Figure 6. Model accuracy across training iterations. Note that the curves for all strategies overlap perfectly, validating the mathematical determinism of the parallel implementations.

The fact that only one line is visible already proves that all strategies behaved the exact same way under the same initial conditions, hyperparameters, and random number generation seed. The accuracy followed a logarithmic-shaped curve with a tendency to reach 96%. Although it is fairly accurate on the MNIST dataset, further studies on the MLP model itself could be conducted to assess its capabilities.

It is also worth noting that the results in **Figure 4** and **Figure 5** complement each other. The analysis made solely on feedforward optimization hinted that parallelizing the outermost loops yields better results. This can be explained logically, as inner loops require less computational power than outer loops. Threads are particularly useful when there is sufficient work to offset the overhead of managing them.

Now that the best strategy is known, there is still space to discuss parallelization parameters. An important question is whether reducing the number of threads speeds up execution. **Figure 7** displays the average and maximum time spent executing the Fit routine.

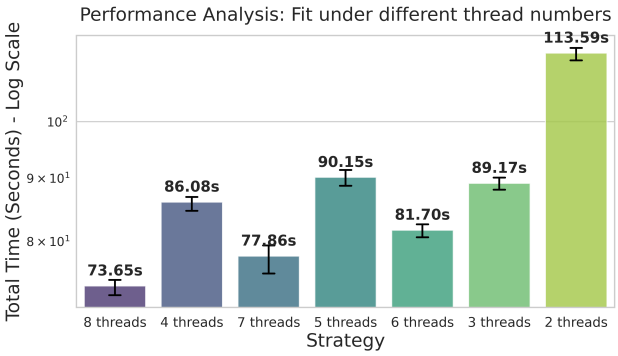

Figure 7. Fit routine performance with different thread numbers

In this case, increasing the number of threads is the appropriate choice to achieve speedup. An interesting observation was that performance worsened with five threads. This is mostly due to the fact that our parallel region is a Fork-Join model. With five threads, there will generally be a core with two active threads, while the remaining cores will have just one. This is a known problem called load imbalance, in which some cores are underutilized while others are heavily loaded, leading to significant performance degradation [Ma *et al.*, 2024].

Figure 8 presents these results from a different perspective using the speedup metric. In an ideal scenario, the speedup should scale linearly with the number of threads. For instance, eight threads should, in principle, result in a program running eight times faster, corresponding to an ideal speedup. However, according to Amdahl's Law, the sequential fraction of the code, which cannot be parallelized across processing cores, imposes a theoretical ceiling on the achievable speedup.

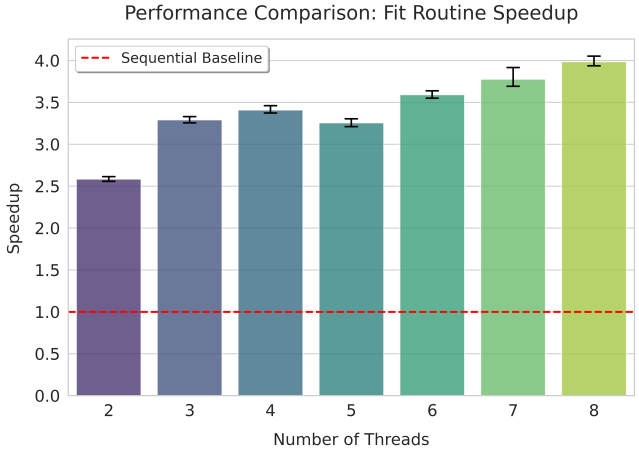

Figure 8. Speedup with different numbers of threads

Therefore, although the eight-thread configuration achieved the lowest total execution time, it did not provide a proportional speedup. In contrast, executions with two and three threads exhibited superlinear speedups, performing slightly better than the theoretical linear limit. This phenomenon is often attributed to hardware intricacies, since the overall workload is divided among more cache memories, which reduces cache misses and, consequently, the parallel execution time. This behavior is not present in the sequential version, since it runs in a single thread and all the workload is processed in a single cache at each level (L1 and L2). It should suggest an unfair comparison, but parallelism demands more resources. For this reason, we need high speedup and efficiency to justify the cost. As an example, if we remove the effect of cache misses, the sequential execution time would decrease more than the parallel one. As a consequence, the sequential fraction of a parallel code, according to Amdahl's Law, would impose a limit on the achievable ideal speedup.

To quantify this limitation, by algebraically manipulating Amdahl's Law (**Equation 6**), we can estimate this fraction (f_{seq}) based on the measured speedup (S_p) and the number of threads (C) using an adaptation of the original equation that accounts for physical cores instead of threads. Applying our 8-thread speedup of approximately 4, we find that about 14.3% of the MLP routine runs sequentially. This estimated fraction demonstrates that, despite the cache benefits observed at lower core counts, the inherent sequential constraints alongside potential parallelization overhead lead to performance degradation at the 8-thread scale.

$$f_{seq} = \frac{C - S_p}{S_p(C-1)} \tag{6}$$

Figure 9 initially contradicts the perception that adding threads gradually reduces efficiency. In general, efficiency tends to decrease as the number of threads and cores increases at the same rate, because the target to be exploited is the number of cores used. However, when the number of threads is higher, i.e., due to hardware multithreading support, efficiency can increase, at least within the experimental limits. The observed curve is approximately a downward-opening parabola, indicating that adding threads from 6 to 8 increases efficiency at some point.

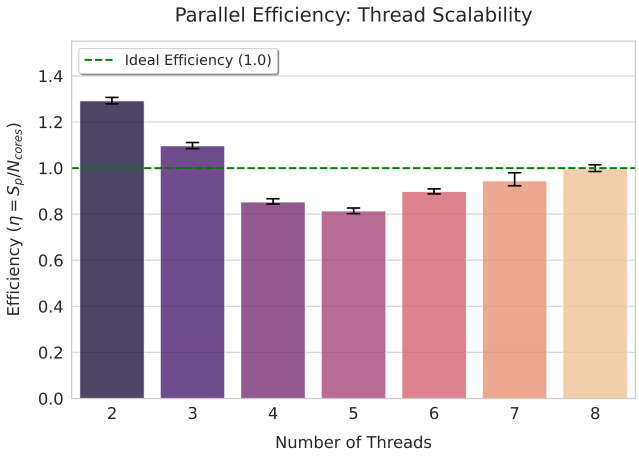

Figure 9. Efficiency with different numbers of threads

According to Che and Nguyen [2014], Amdahl's scaling model cannot predict speedup for multithreading multicore processors. While Amdahl's Law is mathematically sound for the number of processing cores and parallel parts of algorithms, it omits architectural factors that significantly influence performance, such as shared memories. Our results show that the five-thread speedup was lower than the four-thread speedup, which is why the efficiency chart seems like a parabola. In this case, the workload distribution across five threads is unbalanced, placing greater pressure on the same L1 and L2 cache hierarchy than on the others. Although 6 and 7 threads exhibit the same unbalancing characteristic, they tend to have a lower impact because more threads share more cache memories, and, for each cache line, better data locality can be achieved, resulting in more cache hits for shared variables. To quantify this analysis, a more extensive learning experiment based on hardware counter measurements is planned for future work.

7 Conclusion

This work successfully applied neural networks as a case study for parallel computing. The basic parallelization approaches were identified and tested to understand how parallelism operates and why some strategies are more effective than others. All changes respected data dependencies, yielding results identical to a sequential MLP algorithm.

According to the experiment results, maximizing both the workload each thread computes and the number of threads, while leveraging SIMD, yields the greatest speedup. However, pushing hardware resources to their architectural limits yields diminishing returns in parallel efficiency. Although this conclusion may seem broad, it is important to note that an algorithm can exhibit different performance when compiled with different compilers and/or executed on different architectures.

It is expected that this work can support HPC students, as additional literature, by guiding them throughout their studies in parallel computing, complementing their knowledge with applications and metrics that link theoretical parallelization to real-world solutions and foster critical thinking.

7.1 Limitations and Future Work

This paper presents a scope limited by its approach, which focuses on a specific MLP algorithm, computer architecture, and programmer knowledge. It is important to note that computational performance depends not only on the programming techniques based on parallel libraries but mainly on how a programmer uses them. Moreover, different architectures offer the possibility to achieve higher or lower performance. For instance, more cache memory can boost the sequential algorithm, reducing the speedup. Finally, a different algorithm usually presents other parts for exploration of parallelism and, again, changes the potential speedup. For this reason, the art of programming is a continuous effort to understand all available hardware and software variables and use them to achieve the best performance. As previously discussed in the Introduction section, this paper does not provide a universal solution but rather a well-organized approach (from design to evaluation) for an ANN algorithm, serving as an additional guideline for future learners in parallel programming.

As future work, we propose conducting a deeper study using other compilers and architectures, as well as other OpenMP strategies (e.g., scheduling) and the effects of thread pinning on performance. It is also possible to conduct an empirical study by observing the efficiency during parallel execution through hardware performance counters. Furthermore, we intend to evaluate heterogeneous architectures by integrating graphics processing units (GPUs) for parallel computing. Beyond this study of classical architectures, we propose delving further into parallel programming education, with a focus on quantum computing [Markidis, 2024]. Finally, we intend to evaluate student performance using multiple syllabi, approved by the university ethics committee, to improve students' overall achievement.

Declarations

Funding

The National Council for Scientific and Technological Development (CNPq; grant numbers 311697/2022-4 and 402837/2024-0), the Research Support Foundation of the State of Minas Gerais (FAPEMIG; grant number APQ-05058-23), and the Pontifical Catholic University of Minas Gerais (PUC Minas).

Authors' Contributions

Mateus Diniz contributed to the conception and development of this work and is the main contributor and writer of this manuscript. Henrique Freitas supervised and reviewed the final manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The code used for this work is publicly available at https://github.com/cart-pucminas/parallel-ML.

References

Abdurhaman, A., Singh, A., Hossain, A., and Ahmed, K. (2024). A hands-on approach to teaching parallel and heterogeneous computing. In *2024 IEEE 31st International Conference on High Performance Computing,*

Data and Analytics Workshop (HiPCW), pages 9–16. DOI: 10.1109/HiPCW63042.2024.00012.

Akinsola, J. E. T., Olatunbosun, M. A., Olaniyi, I. M., Adeagbo, M. A., Olajubu, E. A., and Aderounmu, G. A. (2025). Application of artificial intelligence on mnist dataset for handwritten digit classification for evaluation of deep learning models. *Acadlore Transactions on AI and Machine Learning*. DOI: https://doi.org/10.56578/ataiml040305.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1465482.1465560.

Ayguade, E., Martorell, X., Labarta, J., Gonzalez, M., and Navarro, N. (1999). Exploiting multiple levels of parallelism in openmp: a case study. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 172–180. DOI: 10.1109/ICPP.1999.797402.

Bindi, L., D'Amico, S., Mencagli, G., and Torquati, M. (2026). Enabling pinning strategies for stream processing applications on multicores. *International Journal of Parallel Programming*, 54(2). DOI: 10.1007/s10766-026-00813-x.

Carneiro Neto, J. A., Alves Neto, A. J., and Moreno, E. D. (2022). A systematic review on teaching parallel programming. In *Proceedings of the 11th Euro American Conference on Telematics and Information Systems*, EATIS '22, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3544538.3544659.

Chadwick, A. W., Erdős, M., Bora, U., Bhosale, A., Lytton, B., Guo, Y., Cooper, R., Gabrielli, G., and Jones, T. M. (2025). The future of instruction-level parallelism (ilp). In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 350–352. DOI: 10.1109/ISPASS64960.2025.00040.

Che, H. and Nguyen, M. (2014). Amdahl's law for multithreaded multicore processors. *Journal of Parallel and Distributed Computing*, 74(10):3056–3069. DOI: https://doi.org/10.1016/j.jpdc.2014.06.012.

Conte, D. J., de Souza, P. S. L., Martins, G., and Bruschi, S. M. (2020). Teaching parallel programming for beginners in computer science. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. DOI: 10.1109/FIE44824.2020.9274155.

de Freitas, H. C. (2012). Introducing parallel programming to traditional undergraduate courses. In *2012 Frontiers in Education Conference Proceedings*, pages 1–6. DOI: 10.1109/FIE.2012.6462263.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533. DOI: 10.1145/42411.42415.

Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38. DOI: 10.1109/MC.2008.209.

Huang, K.-T., Lin, T.-Y., Cheng, P.-W., and Chen, P.-S. (2024). Enhancing tvm vta simulator performance through simd vectorization. In *2024 10th International Conference on Applied System Innovation (ICASI)*, pages 418–420. DOI: 10.1109/ICASI60819.2024.10547748.

Indrakumari, R., Poongodi, T., and Singh, K. (2021). *Introduction to Deep Learning*, pages 1–22. Springer International Publishing, Cham. DOI: 10.1007/978-3-030-66519-7_1.

Islam, M., Chen, G., and Jin, S. (2019). An overview of neural network. *American Journal of Neural Networks and Applications*, 5:05. DOI: 10.11648/j.ajnna.20190501.12.

Iudean, B. (2024). Experience report on teaching parallel and distributed programming through storytelling. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 1–9. DOI: 10.1109/SANER-C62648.2024.00019.

Kumar, K., Pappu, A., Kumar, K., and Sanyal, S. (2006). Hybrid approach for parallelization of sequential code with function level and block level parallelization. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 161–166. DOI: 10.1109/PARELEC.2006.44.

Kumar, S. A. (2017). Achieving software parallelism through alternative process models. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–4. DOI: 10.1109/ICISC.2017.8068596.

Lara Soares, F. A., Neri Nobre, C., and Cota de Freitas, H. (2019). Parallel programming in computing undergraduate courses: a systematic mapping of the literature. *IEEE Latin America Transactions*, 17(08):1371–1381. DOI: 10.1109/TLA.2019.8932371.

Lupo, C., Wood, Z. J., and Victorino, C. (2012). Cross teaching parallelism and ray tracing: a project-based approach to teaching applied parallel computing. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, page 523–528, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2157136.2157288.

Ma, Y., Chen, X., Guo, H., Li, F., and Liu, X. (2024). Multilevel load balancing algorithm for domestic heterogeneous manycore architecture. In *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 1931–1936. DOI: 10.1109/ISPA63168.2024.00263.

Machado, P. R. D. S., Souza, M. A., and Freitas, H. C. D. (2026). Evaluation of thread scalability and compiler optimization in parallel job execution using unity's job system. *IEEE Access*, 14:5012–5025. DOI: 10.1109/ACCESS.2026.3651527.

Macukow, B. (2016). Neural Networks – State of Art, Brief History, Basic Models and Architecture. In *Computer Information Systems and Industrial Management*, pages 3–14, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-45378-1_1.

Magalhães, J., Gonçalves, A., Nobre, C., and Freitas, H. (2024). Avaliação de desempenho e escalabilidade do algoritmo de otimização de colônia de formigas em c++ e python. In *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 97–108, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/sscad.2024.244375.

Markidis, S. (2024). What is quantum parallelism, anyhow? In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12. DOI: 10.23919/ISC.2024.10528926.

OpenMP Architecture Review Board (2025). Technical report 14: Public comment draft of openmp api version 6.1. Technical Report TR14, OpenMP Architecture Review Board. Available at: `https://www.openmp.org/wp-content/uploads/openmp-TR14.pdf`.

Pawar, P., Mehta, P., Boggarapu, N., and Grange, L. (2014). Enhanced automated data dependency analysis for functionally correct parallel code. In *Proceedings of the 2014 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'14)*, pages 253–259. WorldComp. Available at: `https://worldcomp-proceedings.com/proc/p2014/PDP2864.pdf`.

Ribeiro, C. P., Castro, M., Marangozova-Martin, V., Mehaut, J.-F., Freitas, H. C., and Martins, C. A. P. S. (2012). Evaluating cpu and memory affinity for numerical scientific multithreaded benchmarks on multi-cores. *IADIS International Journal on Computer Science and Information Systems*, 7:79–93. Available at: `https://www.iadisportal.org/ijcsis/papers/2012140106.pdf`.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408. DOI: 10.1037/h0042519.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536. DOI: 10.1038/323533a0.

Strazdins, P. E. (2025). A simple tiled approach to teaching parallel computing. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 385–394. DOI: 10.1109/IPDPSW66978.2025.00064.

Vargas-Pérez, S. (2024). Teaching performance metrics in parallel computing courses. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 385–390. DOI: 10.1109/IPDPSW63119.2024.00086.

Vasconcelos, L. B. A., Soares, F. A. L., Penna, P. H. M. M., Machado, M. V., Góes, L. F. W., Martins, C. A. P. S., and Freitas, H. C. (2019). Teaching parallel programming to freshmen in an undergraduate computer science program. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. DOI: 10.1109/FIE43999.2019.9028566.