DISTRIBUTED NEURAL NETWORK TRAINING

Bruna França Sobreira
Instituto de computação
Universidade estadual de Campinas
Campinas, SP
b217787@dac.unicamp.br

César Guedes Carneiro
Instituto de computação
Universidade estadual de Campinas
Campinas, SP
c261031@dac.unicamp.br

December 12, 2024

1 Introduction

Artificial intelligence systems are increasingly integrated into various industries, requiring powerful machine learning models to perform tasks like image classification and natural language processing, such models need large amounts of data and substantial computational resources to be trained effectively. But the training process, especially for deep neural networks, is highly resource-intensive, often taking days or even weeks on a single machine, and current solutions face scalability issues as data-sets and model sizes grow. Much effort has been made finding better ways to execute the training process, but parallelization of this process is still a non-trivial task. The goal of this project is to implement a completely parallel training of a machine learing model, in order to speed up the process by dividing the workload into multiple nodes.

This project will focus on training a neural network to classify the MNIST data-set, which contains 70,000 images of handwritten digits [1]. While MNIST is considered a relatively simple data-set, even a basic multi-layer perceptron (MLP) [2] can be challenging to parallelize due to the inherent dependencies in the forward and backward propagation stages of training. In forward propagation, each layer's outputs depend on the previous layer's activations, and during backpropagation, the gradient updates require synchronized calculations across all nodes to ensure the model converges correctly. These interdependencies make it difficult to divide the workload efficiently without introducing communication overhead or slowing down the process.

2 Background

2.1 MNIST Data-set

The MNIST data-set [1] is a classic benchmark in the field of machine learning, widely used for testing and comparing algorithms. It consists of grayscale images of handwritten digits from 0 to 9, each sized at 28x28 pixels, as shown in Figure 3a. The data-set contains 60,000 training images and 10,000 testing images, all labeled with one of 10 classes corresponding to the digits.

MNIST is highly influential in machine learning due to its simplicity and standardization [3]. Its popularity has led to countless models and techniques being benchmarked on the data-set, providing a wealth of comparative data for researchers. While simple, MNIST remains a challenging problem for certain types of models, especially when working with limited resources, due to the large amount of images sampled. It also serves as a gateway to more complex data-sets and tasks, despite its simplicity.

2.2 Neural Network Architecture

A Neural Network (NN) is a fundamental architecture in machine learning, consisting of layers of neurons connected sequentially. Each neuron, or node, performs a computation involving a weighted sum of its inputs, the addition of a bias term, and the application of a nonlinear activation function. This process allows the Neural Network to model complex, non-linear relationships in data, making it suitable for tasks where simple linear models are inadequate.

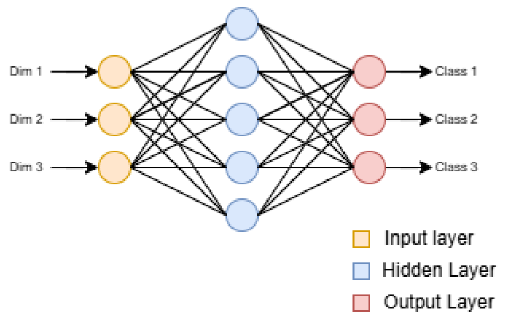


Figure 1: An Multi-layer Perceptron implementation, with one input, hidden and output layer

The minimal architecture of a Neural Network is composed of two types of layers: an input layer and an output layer. But usually more layers, called hidden layers are utilized, as showed in the Figure 1. The input layer corresponds to the features of the data-set, with the number of neurons generally equal to the dimensionality of the input data. Hidden layers, positioned between the input and output layers, enable hierarchical feature learning by extracting increasingly abstract patterns at deeper levels. The output layer generates the final predictions in a format tailored to the task, such as class probabilities for classification or continuous values for regression.

The mentioned weights are parameters associated with the connections between neurons. Each weight determines the strength and direction of the influence that a neuron's output has on the input of the subsequent neuron. The bias is another parameter added to the weighted sum before applying the activation function. It shifts the output, allowing the model to fit the data better by increasing its flexibility, even when the input values are zero. To adjust these weights and biases, training an NN involves the use of the backpropagation algorithm, a supervised learning method that computes the gradients of a predefined loss function with respect to these network's parameters. The gradients are calculated using the chain rule of calculus and are subsequently employed by optimization techniques, such as stochastic gradient descent (SGD) or its variants, to iteratively update the weights and biases. The goal of this process is to minimize the discrepancy between the predicted and actual outcomes.

Neural networks are highly versatile and have been applied successfully across a variety of domains, including image recognition, natural language processing, and time-series forecasting. They are generally adapted into specialized architectures like convolutional neural networks (CNNs) or recurrent neural networks (RNNs) for tasks involving spatial or sequential data.

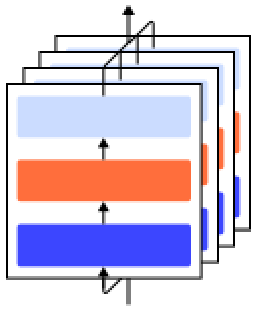
We will leverage the capabilities of this architecture to implement a machine learning model capable of solving the digit classification problem using the MNIST data-set [1]. The specific neural network implemented in this work will be detailed in the section 3.

2.3 Techniques For Training Neural Networks In Parallel

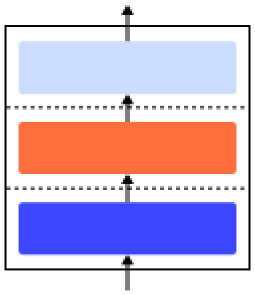
As we saw in the previous section, training a machine learning model is an iterative process. In each iteration, the model processes a batch of input data in a forward pass, computing outputs for the given examples. This is followed by a backward pass, where the contribution of each parameter to the model's overall output is calculated by determining gradients with respect to those parameters. The average gradients for the batch, along with the current parameters and any additional optimization state, are provided to an optimization algorithm, such as SGD, which updates the parameters for the next iteration. With each iteration across multiple batches of data, the model progressively improves its ability to produce accurate predictions.

Nowadays, there are various approaches to parallelizing and partitioning the training process to accelerate it. For large models, such as large language models (LLMs) [4], these techniques also make it feasible to train models that would otherwise be too large or computationally expensive to handle on a single device. Some of the key parallelism strategies include data parallelism, pipeline parallelism, tensor parallelism, and mixture of experts, among others [5, 6, 7].

Data parallelism involves copying the same parameters to multiple devices (often called "workers") and assigning different subsets of the data to each device for simultaneous processing, as shown in Figure 2a. Data parallelism alone still requires that the model fits into the memory of a single device (GPU or CPU), but allows the use of the compute power of multiple devices at the cost of storing duplicate copies of program's parameters. For each forward and backward pass, each device computes the gradients for its subset of the data independently. Once the gradients are



(a) AUC for training and validation over time.



(b) Most important features for the model.

Figure 2: Visual representation of NN train parallelizing techniques

computed, they are averaged (or summed) across all devices to ensure that each device updates its local copy of the model parameters consistently.

This approach can be effective when the data-set is large, and the cost of copying parameters to the workers is amortized due to the large batch sizes or when it is possible to train the workers with different batches. However, it is important to note that the primary challenge in data parallelism lies in the synchronization of gradients, especially when scaling to a large number of devices. This challenge is typically addressed using collective operations like All-Reduce.

Pipeline parallelism, on the other hand, focuses on breaking the model into different stages, with each stage executed on a separate device, as present in Figure 2b. In this method, the computation is divided into multiple parts (often corresponding to model layers), and each part is processed by a different device. Instead of waiting for the full model to complete its forward pass on a single device, pipeline parallelism allows multiple stages to process different mini-batches of data in parallel, with data flowing through the stages.

This approach usually consumes less memory compared to data parallelism, since each unit holds only a portion of the model's parameters. However, due to the sequential dependency between layers, a naive implementation may lead to idle times, or "bubbles," as units wait for outputs from previous units. To minimize these bubbles, we can split a batch into smaller microbatches, allowing workers to process smaller subsets of data and overlap computation with waiting times. As each worker finishes one microbatch, it starts the next, improving pipeline efficiency. With enough microbatches, workers remain active with minimal idle time, and gradients are accumulated across microbatches, with parameter updates occurring only after all microbatches have been processed.

Initially, we considered implementing both data parallelism and pipeline parallelism. However, we ultimately chose to implement only data parallelism, as pipeline parallelism offered limited benefits for our network. That is because the network consisted of only two layers, resulting in very small granularity, a factor that will be further discussed in Section 3.1.

2.4 OmpCluster: An OpenMP Extension for Distributed Programming

The OmpCluster is an extension of the OpenMP task-based parallel programming model, designed to enable the distribution of tasks across cluster nodes [8]. OmpCluster leverages OpenMP's [9] offloading standard to distribute annotated regions of code across the nodes of a distributed system. It hides the MPI-based data distribution and load-balancing mechanisms [10] behind OpenMP task dependencies. By complying with OpenMP, OmpCluster allows applications to use the same programming model for both intra- and inter-node parallelism, simplifying the development process and reducing maintenance complexity.

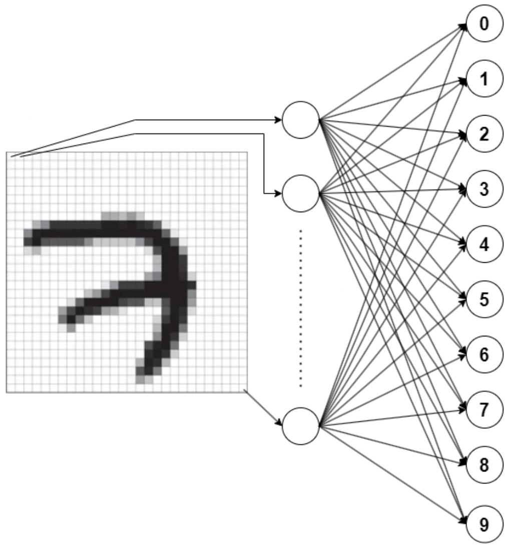
The key distinction between OmpCluster and standard OpenMP lies in how tasks are assigned and how data is managed. In OpenMP, tasks are assigned to accelerators within the same machine, while in OmpCluster, tasks are distributed across different nodes in a cluster. Additionally, while OpenMP's depend clause handles data movement between the host and accelerator on the same machine, in OmpCluster, it relies on MPI calls to transfer data between nodes in the cluster. Despite these differences, if the programmer understands that a core in OpenMP corresponds to a node in OmpCluster, the basic semantics of OpenMP still apply to OmpCluster. Therefore is important to note, that the master-worker relation between the nodes is presented in OmpCluster as a natural extension of the usually approach adopted by the standard OpenMP.

In this work, we utilized an experimental version of OmpCluster, which enables the use of GPUs and CPUs for task based inter-node parallelism. We conducted experiments using this type of device, but it is important to note that this is an experimental version, and added a considerable amount of overhead when compared to the direct 'MPI + OpenMP' implementation, as we found in our experiments.

3 Neural Network Implementation

The neural network implemented for classifying the MNIST data-set is depicted in Figure 3b and was adapted from a repository [11]. A key characteristic of this network is its notably small computational granularity, as it consists of only two layers: an input layer and an output layer, with no hidden layers. Consequently, the number of operations performed per data point is minimal. This small granularity poses challenges for effective parallelization, as memory access becomes a dominant bottleneck. Further discussion on granularity is provided in Section 3.1.

(a) Sized-normalized examples from the MNIST database. Extracted from [1].



(b) Neural network used for classification of the MNIST data-set.

Figure 3: The MNIST data-set and its neural network classifier.

The network comprises fully connected input and output layers. The weights of the connections and the biases of the output nodes are initialized randomly. These parameters are optimized using the Stochastic Gradient Descent (SGD) algorithm. During each epoch, all images in the training data-set are propagated forward through the network, and the resulting errors are accumulated. Afterward, the model is updated via backpropagation using the SGD method.

3.1 Granularity

The small granularity of both the neural network and the data-set posed a significant challenge to scalability. While some parallelization opportunities were available, they were limited in number and had a minimal impact on performance, with the primary limiting factor being the granularity.

To enhance granularity, and consequently scalability, two main approaches were identified: increasing the amount of data to be processed or increasing the number of operations applied to each data point. Increasing the number of operations would have been the ideal way to improve granularity since memory access was already a bottleneck. However, implementing this change would have required a complete redesign of the source code to transform the network into a multilayer perceptron (MLP). This transformation would not only make the problem exponentially more complex but also introduce additional communication overhead for inter-node parallelism. Therefore, we opted to increase the image size instead. This approach effectively improved granularity without adding unnecessary complexity to the problem, achieving the desired balance between scalability and simplicity.

3.2 Neural Network Profiling

The profiling results summarized in Table 1 highlight the distribution of execution time across various functions in the neural network pipeline implementation. The analysis reveals that the `neural_network_hypothesis` and `neural_network_gradient_update` functions dominate the runtime, accounting for 56.53% and 43.18% of the total execution time, respectively. This result is expected, as these two functions perform the core operations of a neural network. Specifically, `neural_network_hypothesis` executes the feedforward pass, propagating the input data through the network layers to produce predictions. In turn, `neural_network_gradient_update` updates the gradients based on the network's output and the loss function, enabling the optimization of weights during training. These two functions are called for each image in the training data-set across 100 epochs, resulting in a total of 6 million calls. Together, they consume over 99% of the total runtime, making them the primary candidates for parallelization.

In contrast, several other functions, such as `get_images`, `get_labels`, and `mnist_get_data-set`, have minimal contributions to the overall runtime. These functions are called infrequently and execute quickly, indicating that data retrieval and data-set management are not significant bottlenecks. This observation suggests that parallelization efforts should prioritize computationally intensive functions rather than data access routines.

% Time	Cumaltive (S)	Self (S)	Calls	s/call (Self)	s/call (Total)	Name
56.53	120.92	120.92	6010000	0.00	0.00	neural_network_hypothesis
43.18	213.29	92.37	6000000	0.00	0.00	neural_network_gradient_update
0.27	213.88	0.58	6010000	0.00	0.00	neural_network_softmax
0.01	213.90	0.02	100	0.00	2.14	neural_network_training_step
0.01	213.91	0.02	-	-	-	_init
0.00	213.92	0.01	1	0.01	0.01	neural_network_random_weights
0.00	213.92	0.00	12	0.00	0.00	map_uint32
0.00	213.92	0.00	2	0.00	0.00	get_images
0.00	213.92	0.00	2	0.00	0.00	get_labels
0.00	213.92	0.00	2	0.00	0.00	mnist_free_data-set
0.00	213.92	0.00	2	0.00	0.00	mnist_get_data-set
0.00	213.92	0.00	1	0.00	0.20	calculate_accuracy

Table 1: Profiling Results for Neural Network serial implementation obtained using Gprof

It is interesting to note that the `neural_network_training_step` function serves as a wrapper for training one epoch. Consequently, it is called only 100 times during the training process. Despite this, its average time per call (`s/call`) is 2.14 seconds, which we also can call as total time spent in one epoch. Figure 4 provides a clearer understanding of how this function is a part as wrapper of the critical path in the machine learning pipeline. Our goal in parallelizing the training of the neural network is to reduce the overall time spent in the training process. As a collateral effect, we expect to observe a reduction in the execution time of the `neural_network_training_step` function calls as well.

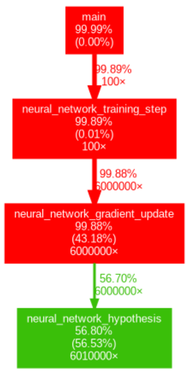


Figure 4: Function call graph of the critical path of the Neural Network pipeline

4 Parallel Neural Network

4.1 Parallelization with OpenMP

The most basic parallelization performed in the algorithm was the parallelization of for loops using the OpenMP directives. We used this strategy to achieve intra-node parallelism, allowing multiple iterations of the loops to be executed concurrently on different CPU cores. Specifically, regardless of the distributed framework adopted, we parallelized the loop shown in Listing 1 using OpenMP, since it is a critical loop in our program, responsible for iterating the gradient update among the images for each epoch. As a result, this loop calls the `neural_network_gradient_update` function for each image in our data-set, which, in the case of the serial program, means performing this function call 60,000 times.

```
for (int i = 0; i < data-set->size; i++) {
    total_loss += neural_network_gradient_update(&data-set->images[i], network, &gradient, data-set->
     labels[i]);
}
```

Listing 1: Serial loop for gradient update calculation

Therefore, we parallelized the gradient update step using the OpenMP directive `#pragma omp parallel for`. This directive ensures that the loop is divided among multiple threads, with each thread computing a portion of the total loss in parallel. The `reduction(+:local_loss)` clause ensures that the partial results from each thread are correctly combined into the total loss. Is important to point, that the called function changed the gradient structure parameters, but only aggregating on it, so we didn't have to implement any specific synchronizations related to this structure. This parallelization improves performance by allowing the iterations to be distributed across multiple CPU cores.

4.2 Parallelization with Open MP + MPI

Another parallelization explored was the combination of both OpenMP and MPI (Message Passing Interface) to achieve both intra-node and inter-node parallelization. By utilizing OpenMP for parallelization within a single node and MPI for communication and parallel execution across multiple nodes, we were able to significantly improve the performance of the neural network training process. This hybrid approach allowed for efficient utilization of available computational resources, enabling parallel execution of multiple tasks both within individual nodes and across a distributed system, providing a whole new level of scalability.

Using a data parallelism strategy, we partition the original data-set into subsets distributed across multiple nodes. Each MPI process is responsible for processing its assigned subset of the data-set. Within each node, OpenMP is employed, as explained in the previous section, to parallelize the gradient computation step. This enables the simultaneous computation of gradients and loss values for multiple images, improving efficiency.

Unlike the intra-node approach, where synchronization is relatively straightforward, a distributed environment demands careful coordination to maintain consistency in network parameters and gradients across all nodes. In our implementation, we use `MPI_Reduce` to aggregate gradients and loss values from all processes at the root node. The root node updates the network parameters using these aggregated gradients and then broadcasts the updated parameters back to all nodes using `MPI_Bcast`. This ensures that all nodes operate with consistent and up-to-date model parameters. However, this approach can be considered naive, as it relies solely on a single node for parameter updates, introducing a potential single point of failure. This design choice may limit the scalability and fault tolerance of the system, as any failure in the root node would disrupt the entire training process. To solve this is necessary think in a different approach to update the neural network.

4.3 Parallelization with OmpCluster

Another toolset we explored was OmpCluster (OMPC), notable for its ability to provide inter-node task-based parallelism. This tool stands out for its ease of implementation compared to other frameworks, simplifying the development process. It allowed us to approach inter-node parallelism in a manner similar to how we handled intra-node parallelism, focusing our efforts on optimizing the loop shown in Listing 1.

As with the 'MPI + OpenMP' implementation, the dataset was partitioned into subsets and distributed across all working nodes. This was achieved using the `#pragma omp target enter data` command prior to initiating the training process. The key advantage of this approach was a significant reduction in communication costs per epoch, as the images did not need to be transmitted during every iteration.

However, the primary drawback of using OmpCluster was the incompatibility of our base algorithm's data structures with the inter-node communication methods employed by this tool. Consequently, substantial modifications were required to adapt the algorithm's logic, making the process highly time-consuming. This challenge was further agravated by the limited availability of online information on OMPC's functionality.

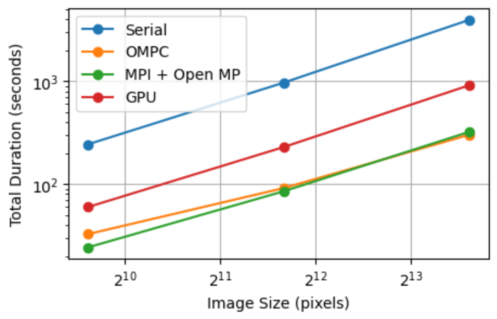
4.4 Parellelization with GPU

The final approach we explored for parallelizing network training involved using GPUs. One significant advantage of OmpCluster is its seamless integration and scalability when working with clusters that include GPUs. Unlike frameworks such as CUDA, which can be challenging to use, OmpCluster simplifies the process by eliminating the need for specialized frameworks. With OMPC, the same script could be executed on parallel CPUs or GPUs, though not simultaneously as of the time of this project. When using GPUs, it is important to note that the base image for OMPC differs from the one used for CPUs. Both images will be provided in the GitHub project repository.

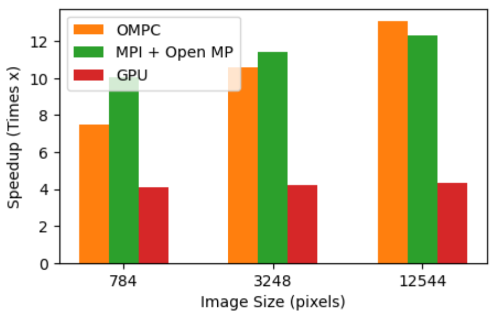
5 Experiments and Results

Our experiments were conducted on the Sorgan Cluster, which consists of 10 nodes, 4 of which are equipped with Infiniband and GPUs. To minimize communication overhead, we restricted our experiments to the nodes with Infiniband, limiting our analysis to parallelism across a maximum of 4 concurrent nodes.

One limitation we encountered while working with OmpCluster was its Head-Worker architecture, which required dedicating one node to act as the head. This reduced the number of available worker nodes by one. While this issue might have been able to be addressed if we had a deeper understanding of OMPC's operations, it constrained our experiments. Additionally, two of the GPUs in the cluster were non-functional during our testing, leaving only two GPU-equipped nodes available. As one of these nodes had to serve as the head when using OMPC with GPUs, we were restricted to running GPU experiments with only a single worker node.



(a) Total Duration vs. Image Size.



(b) Speedup relative to Serial Execution.

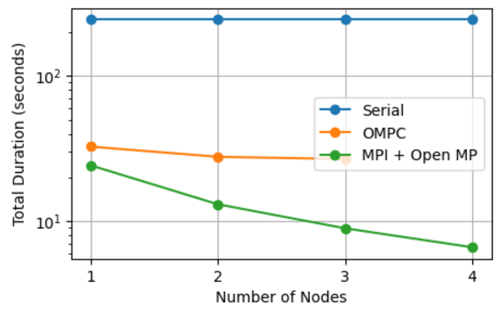
Figure 5: Parallel training performances for 1 Node.

The experiments conducted using a single node, as shown in Figure 5, confirmed our expectation that the training process would benefit significantly from parallelization. The speedups achieved with OMPC and MPI + OpenMP, both at about 10x for all image sizes, were exclusively due to intra-node parallelism, as neither MPI nor OMPC directly contributed to the speedup, only OpenMP did. For smaller image sizes, the additional overhead introduced by OmpCluster's Head-Worker architecture resulted in lower speedups compared to the MPI + OpenMP implementation. However, for larger image sizes, OMPC exhibited a higher speedup, likely due to random variations in execution and the reduced significance of overhead relative to the program's execution time.

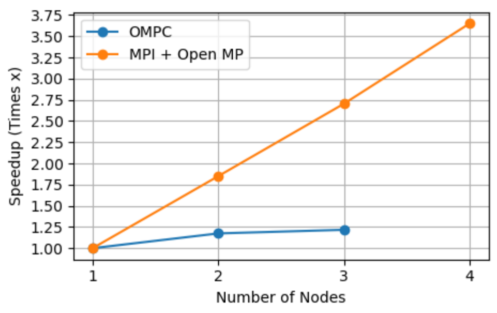
The poor performance observed when using a GPU was unexpected during the project's initial stages. It was anticipated that the GPU would significantly enhance the performance of the training step; however, these expectations were not met. This outcome is likely due to the specific algorithm being unsuitable for GPU execution. Additionally, numerous other factors could have contributed to the poor results, compounded by the researchers' limited familiarity with both GPU code optimization and OMPC functionality. Nonetheless, the execution in the GPU still had a speedup in relation to the Serial execution in a scale of 4x.

The experiments conducted with multiple nodes, shown in Figures 6, 7, and 8, highlight the scalability of each implementation. By comparing the performance of each implementation with varying numbers of devices available for execution, we can assess their scalability. The results clearly show that the MPI + OpenMP implementation is the most scalable, achieving near-linear speedup as the node count increases, as depicted in Figures 6b, 7b, and 8b.

In contrast, the OMPC implementation exhibits poor scalability. While there is some improvement with additional nodes, the reduction in the total duration of the algorithm is minimal. Two primary factors likely contribute to this outcome. First, we had to modify the algorithm's structure to make it compatible with OMPC, which introduced

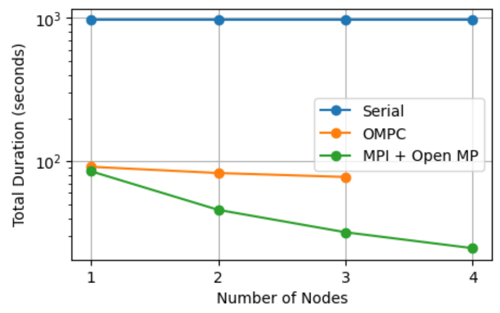


(a) Total Duration vs. Node Count.

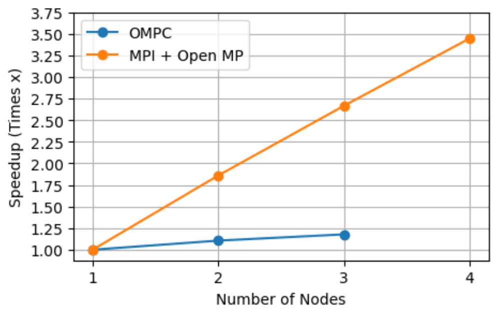


(b) Speedup relative to Single Node Execution.

Figure 6: Parallel training performances for Image Size of 784.

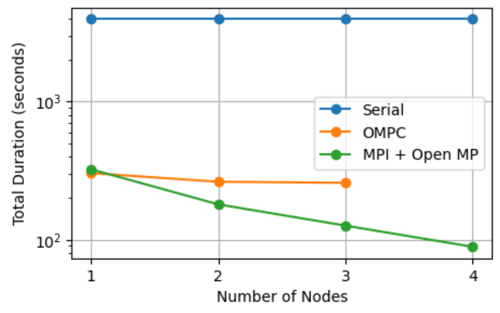


(a) Total Duration vs. Node Count.

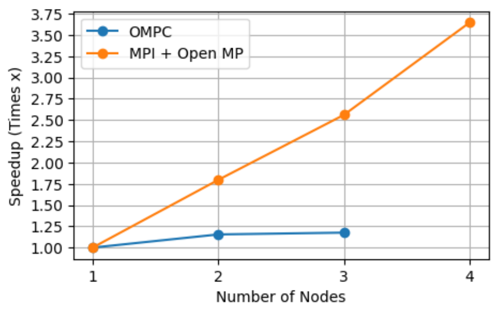


(b) Speedup relative to Single Node Execution.

Figure 7: Parallel training performances for Image Size of 3248.



(a) Total Duration vs. Node Count.



(b) Speedup relative to Single Node Execution.

Figure 8: Parallel training performances for Image Size of 12544.

overhead that negatively impacted performance. Despite testing several variations of the base algorithm, none met our expectations. Second, the researchers' lack of familiarity with OMPC likely resulted in suboptimal application of the tool, introducing unnecessary overhead and further hindering scalability.

6 Conclusion

In this work, we investigated various approaches to parallelizing the training of a neural network for the MNIST dataset, including OpenMP, MPI + OpenMP, OmpCluster, and GPU-based parallelism. Each approach demonstrated its strengths and limitations, influenced by the specific characteristics of the neural network and the infrastructure used. Our results showed that MPI + OpenMP provided the best scalability, achieving near-linear speedups as the node count increased. In contrast, OmpCluster struggled with scalability due to the overhead introduced by its Head-Worker architecture and the modifications required for compatibility with the tool. GPU-based parallelism yielded unexpectedly poor results, likely due to algorithmic incompatibility and limited familiarity with GPU optimization and OMPC functionality.

Through these experiments, we identified critical factors affecting scalability, such as granularity, communication overhead, and algorithmic design. While the implemented network was relatively simple, the insights gained are applicable to more complex models and larger datasets.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] F. Baiardi, R. Mussardo, R. Serra, and G. Valastro. Parallel implementation of a multi-layer perceptron. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 161–166, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Shiva Kumar Pentyala, Zhichao Wang, Bin Bi, Kiran Ramnath, Xiang-Bo Mao, Regunathan Radhakrishnan, Sitaram Asur, Na, and Cheng. Paft: A parallel training paradigm for effective llm fine-tuning, 2024.

[5] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.

[6] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[7] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[8] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Pedro Rosso Gustavo Leite, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. The OpenMP Cluster Programming Model. *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)*, 2022.

[9] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[10] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[11] Andrew Carter. Mnist neural network in c. `https://github.com/AndrewCarterUK/mnist-neural-network-plain-c`, 2018.