MPI solution for Fully Connected Neural Networks

Luiz F. M. Pereira
Department of Informatics
State University of Maringá
Maringá, Brazil
ra103491@uem.br

***Abstract*—The use of neural networks in the area of pattern recognition has become increasingly common, however, to be effective, these networks require a large amount of processing. With the objective of reducing the execution time of an instance of the Fully Connected Neural Network, this paper proposed two different models of paralysis, with the use of shared memory and MPI. To test the effectiveness of the models, we tried to recognize numeric characters, from zero to nine, using the MNIST database. The test case that presented the highest speedup was the one that used all images, both training and test, and four processes, being equal to 4.08. Despite the good results for four processes, the speedup for two processes are very distanced from the expected, taking almost the same time as the sequential execution.**

***Index Terms*—distributed programming, neural networks.**

I. Introduction

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition (PR) systems. PR problems are often characterized by large and redundant training sets with high dimensional inputs, which translates into large networks and long learning times. One of the main contributions of Neural Networks to PR has been to provide an alternative to design properly. Multi layer networks can learn complex mappings in high dimensional spaces without requiring complicated handcrafted feature extractors [1].

One of these neural networks is Fully Connected Neural Network (FCNN), a neural network model in which the adjacent network layers are fully connected to each other. As stated earlier, the training process in neural networks is very time consuming, a way to reduce this time is through paralysis of the process. One way to accomplish this parallelization is to use shared memory. This task can be done by using already established standards, such as MPI.

Message Passing Interface (MPI) is a standard message passing interface for distributed memory concurrent computers and networks of workstation [2]. The message passing paradigm is the most generally applicable and efficient programming model for parallel machines with distributed memory and has been used widely in parallel and distributed computing systems for some years [3]. In this paper, two solution to implement a FCNN in parallel is introduced. For this solution, the use of shared memory was chosen, which is managed by the MPI to C library, called OpenMPI. In the best scenario, it was possible to achieve a speedup equal to 4.08, using four processes and the largest input available. This paper is organized as follows: in the next section we present the basic definitions about FCNNs; then we present our proposal to carry it out in parallel. In Section IV we present the tools used and the environment where the tests were carried out. In Section V we present the results obtained and analyze them in relation to performance and the impact of MPI instructions in code.

II. Fully Connected Neural Networks

A Fully Connected Neural Network (FCNN), also called a Dense Neural Network (DNN) in data science, corresponds to the neural network model in which the adjacent network layers are fully connected to each other. Each neuron in the network is connected to each neuron in adjacent layers. To ensure that this definition is understood correctly by the reader, we present below some basic concepts related to neural networks [4].

A neuron is a basic unit of a layer in FCNN. A neuron performs the multiplication and addition of weights to the input, similarly to linear regression, and the result is passed to the activation function. An activation function is responsible for defining the output of a neuron given a set of input data, some common examples are the Linear, ReLU and Sigmoid functions [4], [5].

It is possible to divide the use of neural networks into two different moments, the training and the test. The first consists of searching for optimization parameters (weights and bias) under the given network architecture and minimizing the classification error or residuals. This process includes two parts: feed forward and back propagation. The forward feed traverses the network with the input data (as forecast parts) and then calculates the loss of data in the output layer by loss function (cost function). After obtaining data loss, it is necessary to minimize data loss by changing weights and bias, a very popular method for this is to back-propagate the loss into every layers and neurons by gradient descent[1]. The number of times this process will be performed is called an epoch.

In this type of architecture, we can classify the layers in 3 categories: input layer, hidden layer and output layer.

The input layer is generally fixed, with only one layer, and the number of nodes corresponds to the number of input

[1]https://cs231n.github.io/convolutional-networks/

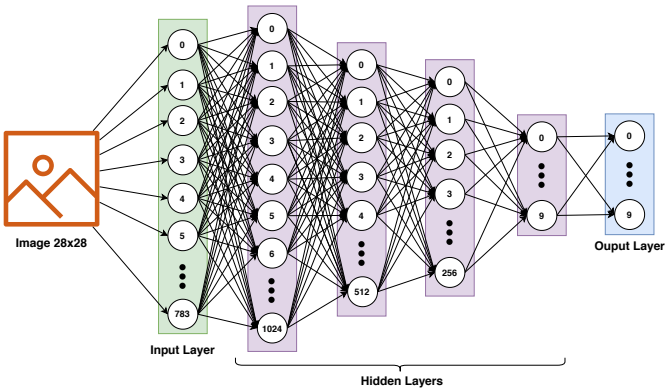

Fig. 1. FCNN with four hidden layers

features, in a scenario where images are used, the number of pixels in the image would define the number of nodes needed.

The nodes in the output layer generally do not have an activation because it is commonly used to represent the scores of each class in the classification. For classification, the number of output nodes corresponds to the number of classes to be forecasted.

The hidden layers are diverse, both in relation to the activation function used and the number of neurons used, and are the main component of an FCNN. In general, as the complexity of the problem grows, more hidden layers are needed to capture the desired patterns [4].

To carry out the investigations proposed for this paper, the following architecture was chosen: one layer for input, one layer for output and 4 hidden layers. As the scenario chosen for tests was the classification of digits from 0 to 9, the input corresponds to an image, thus, the input layer has height x width nodes. Next we have 4 hidden layers, the first with 1024 neurons, in sequence we have the layers with 512, 256 and 10 neurons, in that order. For all hidden layers, the chosen activation function was ReLU, with the exception of the last layer used as Sigmoid as an activation function. Finally, the output layer has ten neurons, equivalent to the number of available classes. The architecture described is represented graphically in Figure 1.

Once created, the network is fed with each image and each layer performs forwarding, after that, the loss is calculated. When performing the loss calculation with the Mean Squared Error (MSE)[6], it is necessary to do back propagation and update the weights for each layer.

With the training carried out, the test begins by means of forwarding, in which, when arriving at the output layer, a forecast for that image is obtained and verified with the label. If the forecast is different from the label, the loss is increased by one unit and the entire process is restarted and repeated until the number of epochs.

III. MPI Solution

Transforming the sequential version in parallel, using distributed memory, is very simple if we use MPI. Simply distribute the set of images between the processes and add barriers after reading the image, after training and after the test is completed. Let's look at the reason for using each one.

A barrier after image reading is necessary because only the master process performs this task, so other processes must wait for him to finish. After this barrier, the image data is transmitted to all other processes, through broadcasting, so that they have a copy of all training and test images.

The second barrier, right after the training is justified by the fact that it is not possible to proceed to the test stage without having even completed all the training. The training process, in turn, is performed only by the master process, as it updates the MSE error at each failure in the forecast, which must be done sequentially.

The last barrier ensures that all processes have already carried out the training and testing for that epoch and we are ready to update the weights and biases of the network. This update is done using reduce with all copies of each layer, that is, all the weights of each layer are added and the value obtained becomes the set of weights for all layers, this same operation happens for the biases of each layer.

Unfortunately, if the objective is to achieve the same accuracy with less time, just distributing images between the processes would not be enough. This happens because by maintaining several copies of the networks, we increase the randomness factor of the network, which contributes to the increased accuracy obtained. Thus, we proposed two modifications in the architecture to obtain an accuracy similar to that obtained in the sequential, however with a reduced time. To ensure that accuracy was not impaired, empirical tests were performed for each proposed modification.

The first modification, which we will call Model 1, seen in Figure 2, removes the last two hidden layers from the sequential architecture, keeping only the layer with 1024 and the layer with 512 neurons as hidden layers. The second modification, which we will call Model 1, seen in Figure 3, removes only the last hidden layer of the sequential architecture, with 10 neurons. Only the modifications are small, the difference produced in the speedup is considerable, as can be seen in Section V.

IV. Experimental Settings

In this section, we detail the environment used for sequential and parallel execution. Following, we describe some details of the tools used for watch hardware events, for example, the number of messages sent, and measure performance, in subsection IV-B.

A. Environment and Limitations

To carry out the experiments, all codes were developed using the C language and executed on the same machine. The select machine has Arch Linux, with kernel Linux 5.4.75-1-lts, as OS; Awesome as Windows Manager(WM), without desktop environment, GCC version 10.2.0 as compiler and Open MPI in version 4.0.5 to run MPI code. The machine's hardware configurations can be seen in the Table I.

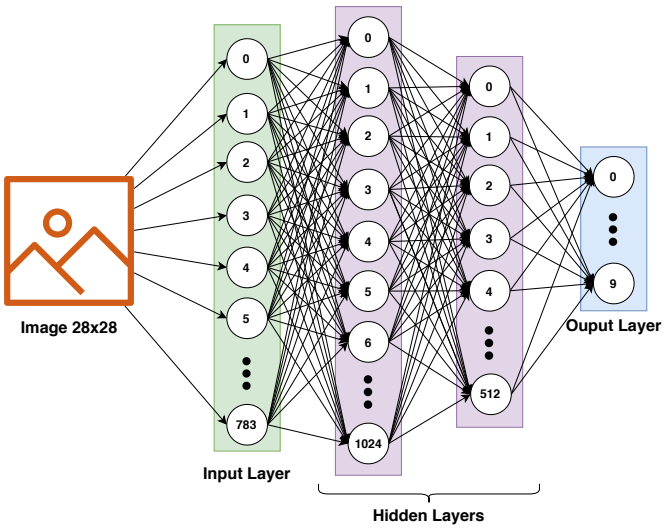

Fig. 2. FCNN with two hidden layers.

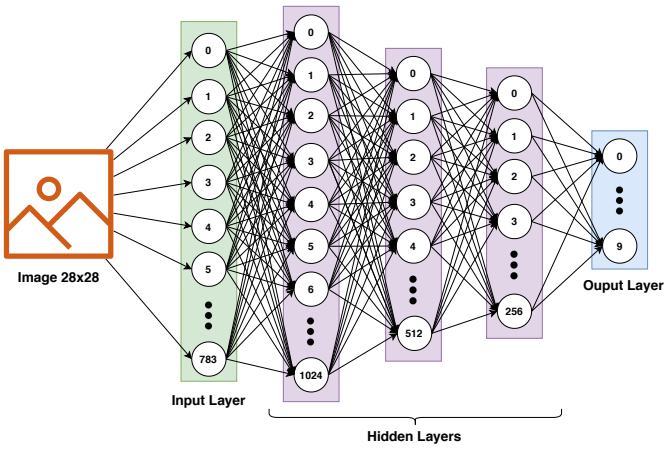

Fig. 3. FCNN with three hidden layers.

TABLE I
HARDWARE SPECIFICATIONS OF THE MACHINE RESPONSIBLE FOR CARRYING OUT THE TESTS.

Hardware	Specification
Model	Notebook Dell G3 3590
Processor	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
Physical Cores	4
Thread by core	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	8192 KB
Cache line size	64 b
RAM Memory	8 GB
Swap partition	15 GB

Considering that the machine used is a notebook, the results are expected to suffer interference in relation to performance. Since in this type of equipment the cooling system is weakened and the focus is on energy savings, the Operational System (OS) often tends to take more drastic measures in relation to the scheduling and freeze of processes.

B. Performance Analysis Tools

In order to compare the performance of the sequential algorithm with its parallel version, two tools were used. The first tools is the Perf[2] program, used to extract relevant data about the hardware and software events that occurred during the execution.The second tool is called mpiP[3] and is used to collect statistical information about MPI functions.

Perf is an event-based analysis, monitoring and debugging tool, that is, events must be predefined in order to be able to collect and process them during the analysis process. Measurable events at Perf are from different sources, are present in most components of the kernel and hardware. It is worth mentioning that the event monitoring has limitations, whether in terms of accuracy or number of events available for the chosen processor [7].

According to the project's authors, mpiP is a lightweight profile library for MPI applications and generates considerably less overhead and much less data than tracking tools, as well as merging results from all tasks into one output file.

V. Analysis and Discussion

In this section we present some information about the results obtained and what we can conclude about them. First we present an analysis of the performance through the Speedup metric, then we analyze the impact of MPI functions call in execution time.

A. MNIST Database

To carry out the tests on the models built, the MNIST database[4] was chosen and different quantities of samples were used, having 16 epochs as a standard. The MNIST database of handwritten digits has a training set with 60,000 examples and a test set with 10,000 examples. The digits were normalized in terms of size and centered on a fixed size image, 28 pixels high by 28 pixels wide. An example of the base samples can be seen in Figure 4. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

[2]https://perf.wiki.kernel.org/index.php/Main_Page
[3]https://software.llnl.gov/mpiP/
[4]http://yann.lecun.com/exdb/mnist/

Fig. 4. Example of samples from MNIST database.

B. Speedup

One of the goals of parallelizing an application is to decrease the execution time. A way to tangibly compare the performance of a sequential application with its parallel version, in relation to time, is to calculate the Speedup. The Speedup metric can be calculated by the ratio of serial execution time to parallel execution time. For an application with good scalability, the speedup should, ideally, increase at the same rate as the increase in the number of processes.

Although we want Speedup to be always linear, this is not always the reality. In line with Amdahl's law[8], we have to, the speedup of a parallel program is limited by the time needed to compute the sequential fraction of that same program, this implies that in these types of problems the ideal Speedup will hardly be reached.

To guarantee the effectiveness of the parallelism with MPI, tests with four different input sizes were carried out for both proposed modifications. Due to the limitations in the amount of physical cores of the test machine, only four, the inputs were tested only for two, three and four processes. We will define the entries as follows: 1000 images for training and 300 images for testing; 9000 images for training and 3000 images for testing; 30000 images for training and 6000 images for testing; finally, 60000 images for training and 10000 images for tests will be called *small*, *medium*, *large* and *extra large*, respectively.

First, it is important to note that the time used to calculate the speedup werethe overall average time of ten runs, ie, they include the reading time of the images and labels. As expected, as we increase the amount of hidden layers the speedup drops, as it is possible to verify by comparing Figures 5 and 6. It is worth remembering that, although Model 1 has a shorter execution time, Model 2 is the one who tends to present greater accuracy. Thus, the choice of which model to use will depend on the programmer's need.

It is also possible to observe that the performance for two processes had the result much below the ideal for both models. Several may be the reasons why this has occurred, we can cite as an example the amount of time required to perform the broadcast and reduce the data, however, to discover and overcome all obstacles that prevent the model to approach the ideal result, it would be necessary to run other performance analysis tools that would provide a broader amount of data on execution.

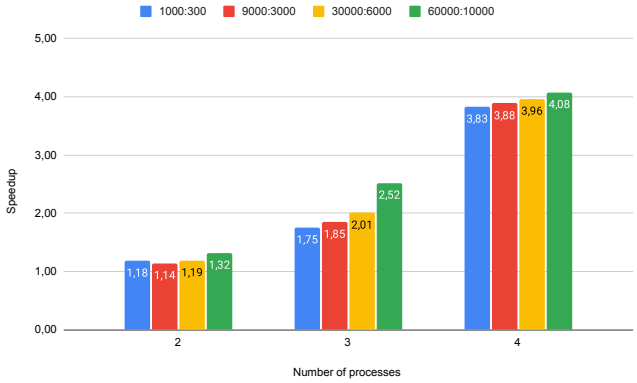

Fig. 5. Model 1 speedup for 2, 3 and 4 processes.

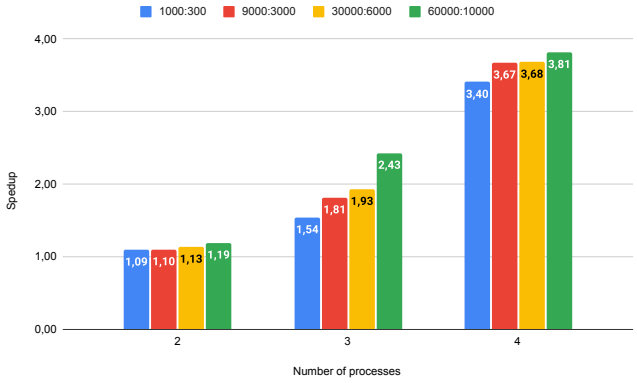

Fig. 6. Model 2 speedup for 2, 3 and 4 processes.

Despite the performance far from ideal for two and three processes, for four processes the speedup obtained was satisfactory, given that the problem is not embarassingly parallel, even reaching the linear speedup for the extra large entry. This result can only be obtained thanks to the modifications made to the sequential model, because, as stated before, when the amount of hidden layers increases, the speedup tends to decrease.

C. MPI impact

In exchange for a parallel execution, there are costs in relation to sending and receiving data, barriers, etc, which must be paid when using MPI. In the proposed models of FCNN, as the number of processes increases, the time spent with functions of the MPI library increases. This behavior is justified by the fact that each process must have a copy of all the training and test images, that is, the more processes, the greater the amount of messages that are trafficked.

Note, however, that, in relation to the total time, these costs are decreasing in relation to the size of the input grows, that is, for the same amount of processes, the greater the size of the input, the less expressive the percentage of time spent with

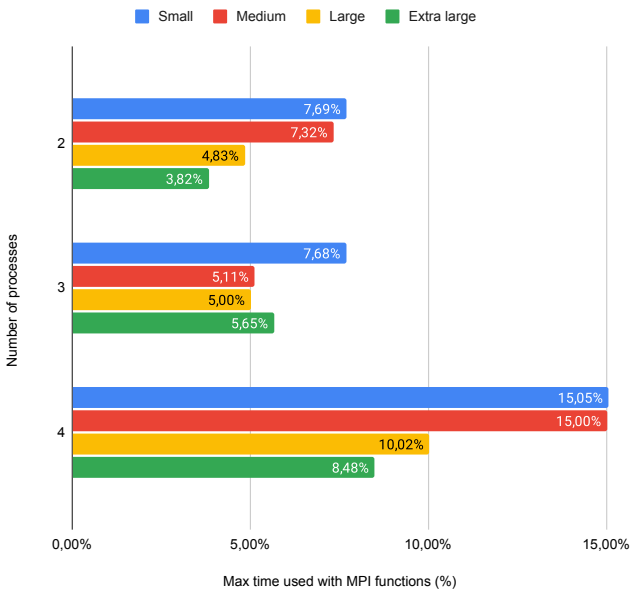

Fig. 7. Max time used with MPI functions for Model 1.

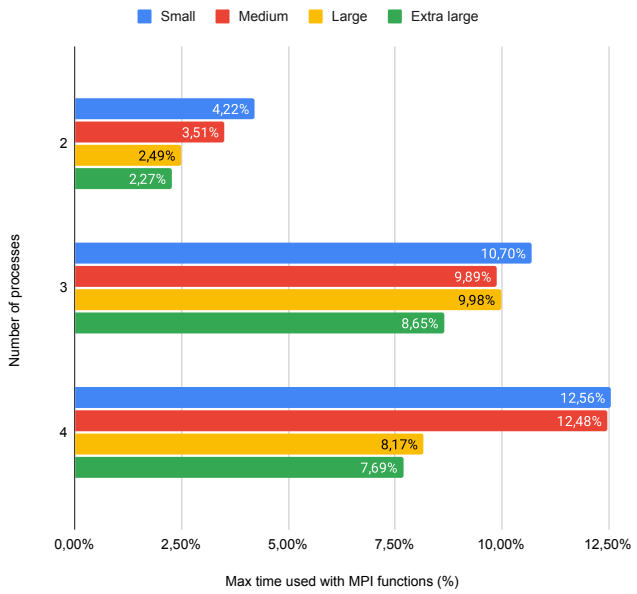

Fig. 8. Max time used with MPI functions for Model 2.

intrinsic functions of MPI. This behavior can be clearly seen in Figure 7, which shows the time, in percentage, spent only with the operations of the MPI library for Model 1.

The same behavior can be observed for Model 2 in Figure 8. Note, however, that the percentage of time spent with the same operations in Model 2 is lower compared to the previous one, for two and four processes, this fact is probably linked to the addition of a layer in relation to Model 1, which implies a greater amount of messages to be trafficked, since the number of copies to be made available also increases.

VI. Concluding Remarks

FCNNs are highly generic models and can be used for pattern recognition, optimization, data generation tasks, etc. In this paper, we compare three different implementations of this neural network model, one of which is the sequential model and the other two, models that use shared memory through MPI. The maximum speedup obtained was 4.08, achieved in tests with four processes and using 60,000 training images and 10,000 test images. Unfortunately, the results for two processes were far from expected. As future work, investigations will be carried out to find out what caused the model's low performance for lower process values. In addition, it is intended to implement new variations to the model in order to achieve greater accuracy with less execution time.

References

[1] Y. LeCun and Y. Bengio, "Pattern recognition and neural networks," *The Handbook of Brain Theory and Neural Networks. MIT Press, Cambridge, MA*, 1995.

[2] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993, pp. 878–883.

[3] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming environments for massively parallel distributed systems*. Springer, 1994, pp. 213–218.

[4] I. Kononenko and M. Kukar, *Machine learning and data mining*. Horwood Publishing, 2007.

[5] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, M. Hasan, B. C. Van Essen, A. A. Awwal, and V. K. Asari, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, 2019.

[6] E. L. Lehmann and G. Casella, *Theory of point estimation*. Springer Science & Business Media, 2006.

[7] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013, p. 5.

[8] B. Schmidt, J. Gonzalez-Dominguez, C. Hundt, and M. Schlarb, *Parallel programming: concepts and practice*. Morgan Kaufmann, 2017.