2011-01-01

Effects Of The Usage Of Parallel Hardware Architectures In The Simulation Of Artificial Neural Networks Training Process

Carlos Beas
University of Texas at El Paso, cgbeas@miners.utep.edu

EFFECTS OF THE USAGE OF PARALLEL HARDWARE ARCHITECTURES IN THE SIMULATION OF ARTIFICIAL NEURAL NETWORKS TRAINING PROCESS

CARLOS G. BEAS

Department of Electrical and Computer Engineering

APPROVED:

Patricia A. Nava, Ph.D., Chair

Virgilio Gonzalez, Ph.D.

John Korah, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

To my parents. None of this would have been possible without your everlasting support.

EFFECTS OF THE USAGE OF PARALLEL HARDWARE ARCHITECTURES IN THE SIMULATION OF ARTIFICIAL NEURAL NETWORKS TRAINING PROCESS

By

CARLOS GUILLERMO BEAS, B.S.E.E

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

August 2011

Abstract

Long training times and non-ideal performance have been a big impediment in further continuing the use of Artificial Neural Networks for real world applications. Current research is currently focused on two areas of study that aim to address this problem. The first approach seeks to overcome large training times by devising faster learning algorithms where a set of interconnection weights for which the network produces negligible error takes a less amount of computation to find [Sun98]. The second approach aims to address the impediment by implementing existing training algorithms but on parallel hardware architectures.

While both approaches provide promising advances for future development in neural networks, it is the approach of using parallel implementations that is further considered in this study. The main advantages of focusing on this route are that it can be implemented on already existing training algorithms and, at the same time can be used as a vehicle to study improvements in error and accuracy performance of the trained network. A side byproduct of focusing on this approach is that a framework can be established to provide further speedup for future (faster and more efficient) training algorithms.

This research focuses in the parallel Backpropagation training implementation in which the processing nodes used are interconnected using the star-connected topology and arranged in a HOST-WORKERS manner, while implemented on a 40-node Beowulf cluster. Additionally, four variations of the training algorithm were evaluated across three different benchmark problems and the results were compared using the sequential version against multiple instances of the parallel implementation for an increasing number of processing elements. A decrease in average error was observed, along with an overall decrease in speed up performance as the number of processing elements was increased.

Table of Contents

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 PROBLEM STATEMENT

The process of training an Artificial Neural Network (ANN) using the Back-Error Propagation (BP) algorithm on a Serial machine may often result in a slow convergence to a given minimum error level [Mor04] when implemented on non-trivial problems which contain large amounts of training patterns. Additionally, for such problems, a Serial machine may incur in long training time periods if the optimal parameters for convergence are not initially set [Foo95]. And to make matters worse, these optimal parameters (learning rate, acceleration, number of units on hidden layer, etc.) are heuristically found through trial and error. Thus, speeding up the training process itself, is clearly a pressing need and the methods for doing so have been the subject of research in the past couple of decades [Mor04].

One clear approach to overcome these challenges may be to take advantage of the high degree of parallelism in the nature of a BP neural network in order to implement the training process on parallel or distributed hardware architecture to attempt to achieve a speedup in the computation portion of the process. A plethora of research studies have been conducted to develop such implementations of the BP training algorithm for specialized parallel machines that are built with neural network training applications in mind, such as transputers, connection machines (CM-1, CM-2 and CM-5) and hardware specialized for neurocomputing such as the DREAM machine, which is a dynamically reconfigurable extended array multiprocessor [Sun98]. Examples of studies that have successfully achieved considerable speedup using specialized hardware are Foo95, Haz04 and Yas98.

Another possibility of performing such parallel implementations is that of using a computer cluster (which is an agglomeration of personal (of-the-shelf) computers interconnected via an Ethernet

communication links) for parallelizing the BP training algorithm [Mor04, Sun98]. Traditionally, researchers have been discouraged to use this approach to parallelization to achieve speedup due to the overwhelming unbalance in the processor speed (in GHz) to communication bandwidth (in MBit) ratio, which according to [Mor04] should be at least 1:500. However, due to recent improvements in the speed of network communications for modern cluster setups, which currently provide bandwidths of 1 GBit and 10 GBit [Myr11] (a big improvement from previous standards in the range of 10 Mbit bandwidths), it has become now reasonable to consider such a platform for the parallelization of the BP training algorithm.

1.2 – PROPOSED APPROACH

Even when recent improvements in bandwidth for distributed computing networks (or clusters) have been developed, they are still not able to match the speeds achievable using specialized hardware. Thus, it follows that parallel implementations for the BP algorithm on a cluster are most beneficial when they use as little communication between computing nodes as possible [Mor04]. Thus, this study aims to explore some of the most popular Beowulf cluster implementations for the BP training process of a neural network and evaluate the positive and negative effects that are consequential of such implementations.

The study focuses on the two main learning methodologies of a neural network that is trained using the Backpropagation algorithm, namely learning by patterns and learning by epochs. A sequential version of the algorithm of each learning methodology is run for four benchmark problems. The performance of the sequential versions is then compared against their corresponding counterparts obtained using a parallel implementation on a 40-node Beowulf cluster.

Training set parallelism was chosen as the method for parallelizing the training process as it promises to minimize communication between processors and emphasize computational speed for problems with large training sets. Furthermore, two distinct methods (Winner-Takes-All and HOST-averaging) for parallel weight update were considered for each learning methodology; providing a total of four different variations of the parallel version of the training process.

1.3 – Document Organization

This thesis is organized in six chapters. Chapter 2 provides the background behind artificial neural networks, their architecture and some of the training algorithms that were historically used to derive the Backpropagation algorithm. Chapter 3 presents the background information behind parallel and concurrent computing as well as some of the most popular topologies and metrics to qualify a parallel implementation. Chapter 4 covers the use of parallel architectures for implementing the BP training algorithm as well as common techniques used for such implementations and the terminology that will be used in later chapters. Chapter 5 explains in detail the simulations performed and the results obtained. Finally, Chapter 6 provides conclusions form this study and the future work that can be undertaken to continue this research.

Chapter 2: Artificial Neural Networks

Artificial neural networks (ANNs) are mathematical models designed to act as information processing systems [Fau94] that share performance characteristics of biological neural networks [Day90]. They are assigned this name due to their similarity with biological nervous systems in the human brain, which consist of neurons that server as processing units and synapses that provide links of communications. Likewise, ANNs consists of nodes for data processing and weighted links used to interconnect the nodes and provide a means for communication [Mad07].

Artificial Neural Networks (ANNs) are commonly utilized for research purposes across multiple fields of study such as Computer Engineering, Astrophysics and Mathematics for such tasks as signal processing, financial forecasting, weather prediction or process control. ANNs are designed using the following criteria [Fau94]:

1) Information processing occurs at many simple elements called neurons.
2) Neurons communicate with each other using connection links.
3) Each connection links has a weight associated with it and whose value represents the strength of the communication link between two neurons. The weights of the connection links represent information which is used by the net to solve a given problem.
4) Neurons use an activation function, applied to its net input, to determine the corresponding output signal. Each neuron may send its activation signal to several other neurons, but it must only send one activation signal at a certain time.

The basic building block of neural network architectures is the processing unit called a "Neuron." Neurons in an Artificial Neural Network are modeled after the biological nerve cells that form part of the human brain. Although the study of the performance of Artificial Neural Networks is focused on how well a particular complex task can be implemented by an ANN, rather than how

well the ANN models the actual biological system, important parallels exist that are useful in the understanding of artificial neural networks. In particular, there are three characteristics regarding the general structure of a biological neuron that are noteworthy: the dendrites, soma and axon, as can be seen from the illustration of a biological neuron shown in figure 2.1.

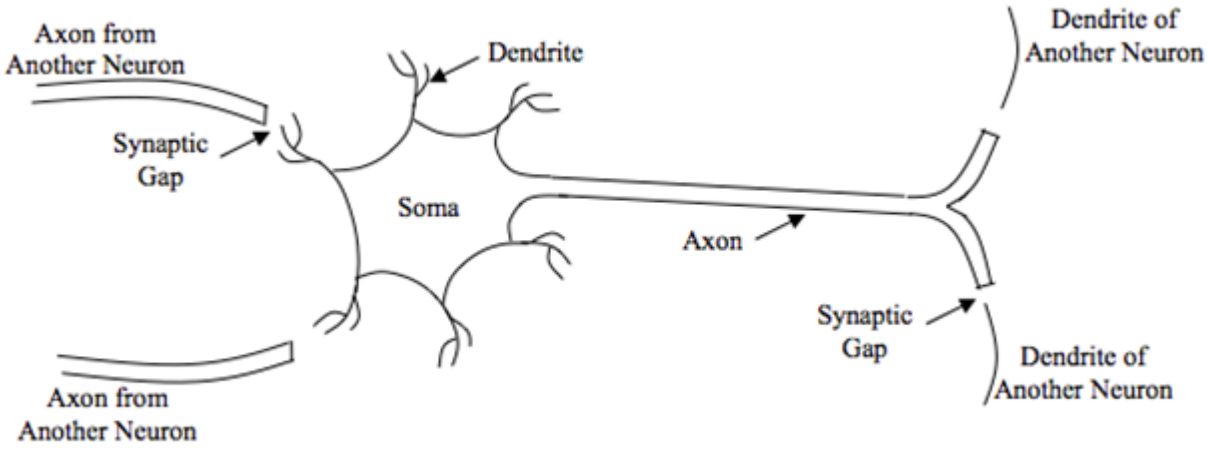


Figure 2.1: Structure of a Biological Neuron [Fau94.]

The dendrites in a biological neuron are a set of branch-like fiber structures that extend outwardly from the soma or body of the neuron and act as receptors as they sense input signals coming from the axons of nearby neurons. A neuron then accumulates the activity generated by the biochemical reactions from its dendrites interaction with other neurons and, once this activity has reached a certain threshold, the neuron produces a chemical reaction (or signal) that travels through its axon and, consequently, it is received by other neuron's dendrites. It is also noteworthy to mention that, just like their biological counterpart, artificial neurons activate their output signal just once at a time and this signal is broadcast to the multiple neurons connected to the firing artificial neuron.

The similarities between a network of biological neurons and their artificial counterparts are also found in the fact that they are both able to support fault tolerance to some extent [Fau94], that is, the ability to be able to recognize input patterns that are not exactly those with which the networks has been trained but have features that are close in similarity. An example of this characteristic is found in the fact that humans can identify the specific breed of a dog based on the visual features that the dog displays, such as size, hair color and texture and facial features even when no two dogs of a certain breed are exactly the same.

Artificial neurons (also known as *cells, nodes* or *processing units*) are interconnected to multiple other neurons by means of directed communication links, each of which has a connection strength associated to it, and which is labeled using the convention: $\boldsymbol{w_0, w_1, w_2, \ldots, w_{n-1}}$. The receiving neuron performs a weighted sum $\boldsymbol{Y_{in}}$ (as described in equation 2.1) on the input signals and with the obtained result applies a non-linear threshold function (also called an activation function) $f(Y_{in})$ to compute its output, which will be broadcast to all the nodes in the consequent stage of the network [Day90]. Furthermore, following a similar sequence of steps, a modified version of the transmitted signal is received by all the target cells. How much the received signal is modified depends on the polarity and magnitude of the connection strength (or *weight*) associated with its communication link to the firing neuron. The target cells then perform a weighted sum and apply an activation function, which may or may not be the same as the one applied in the preceding stage, and compute their output signal.

Besides input and processing neurons, a bias unit is also present in any input or hidden layers, as shown in figure 2.2.

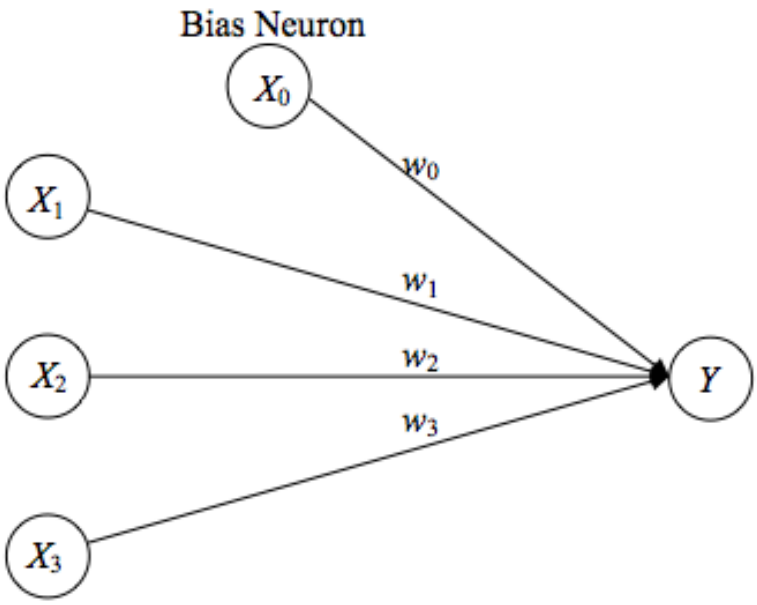


Figure 2.2 Connections and Weights of an Artificial Neuron

Considering a particular layer *j* in the network as a vector, then a bias unit in this layer can be described as adding a component $x_0 = 1$ to the vector **x,** (i.e. x = (1, x_1, …, x_i, …, x_n) [Fau94]. The weight associated with the connection of the bias unit a particular neuron is treated as any other weight and, therefore, the total (or net) input to a unit Y_j is given by:

$$\boldsymbol{y\_in_j = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3} \tag{2.1}$$

Or, in a general case:

$$\boldsymbol{y_{in_j} = b_j + \sum_{i=1}^{n} x_i w_{ij}} \tag{2.2}$$

$$y_{in_j} = \sum_{i=0}^{n} W_{ij} X_i \tag{2.3}$$

The ability of an Artificial Neural Network to learn stems from the weights associated with each link that connects two neurons to each other. The learning process of an ANN consists on modifying these weights using a training algorithm in which a series of pre-determined inputs, usually referred to as the *training data*, is fed into the net. A corresponding set of target outputs is presented to the net at the output layer, at which point, the net adjusts its weights based on the error rate obtained from the comparison of target output against the actual output of the net.

In order to achieve effective learning, the weights are adjusted to minimize the difference between the ANN's actual output and the desired output specified in the training data set for the same input. Thus, a network weight can inhibit or enhance the effect of signals communicated in between neurons [Mad07]. In other words, the more "interaction", in terms of transmitting and receiving activation signals, occurring among two particular neurons, the more the weight of the link connecting them will be fine-tuned to contribute to a correct output.

2.1 – Architectural Characteristics of Artificial Neural Networks

The topology of an ANN is one in which layers are used to group neurons that are to behave in the same manner and where the neurons within a layer are either fully interconnected or not interconnected at all. The behavior of the neurons in a layer is determined by factors such as the activation function and the pattern of weighted connections to neurons in other layers.

An artificial neural network can be classified architecturally according to the number of layers that form it. To this end, there are two main classifications of neural network architectures: the single layer and multi-layer networks.

Single-layer networks have only one layer of connection weights and the weights for one output neuron do not influence the weights for other output units. A generic example of a single layer ANN is presented in Figure 2.3, where it can be seen that the network consists of a layer of input units and a second layer of output units. Given that this is the case, one may be tempted to categorize the shown network as being multi-layered. However, the layer of input neurons in any architecture is not counted as a layer since input units perform no computation. The only purpose of the neurons located on the input layer is to receive input signals from the exterior of the network and make them available to the neurons in the following layer. Therefore, only the output layer is considered and thus the architecture is referred to as a "single layer" ANN.

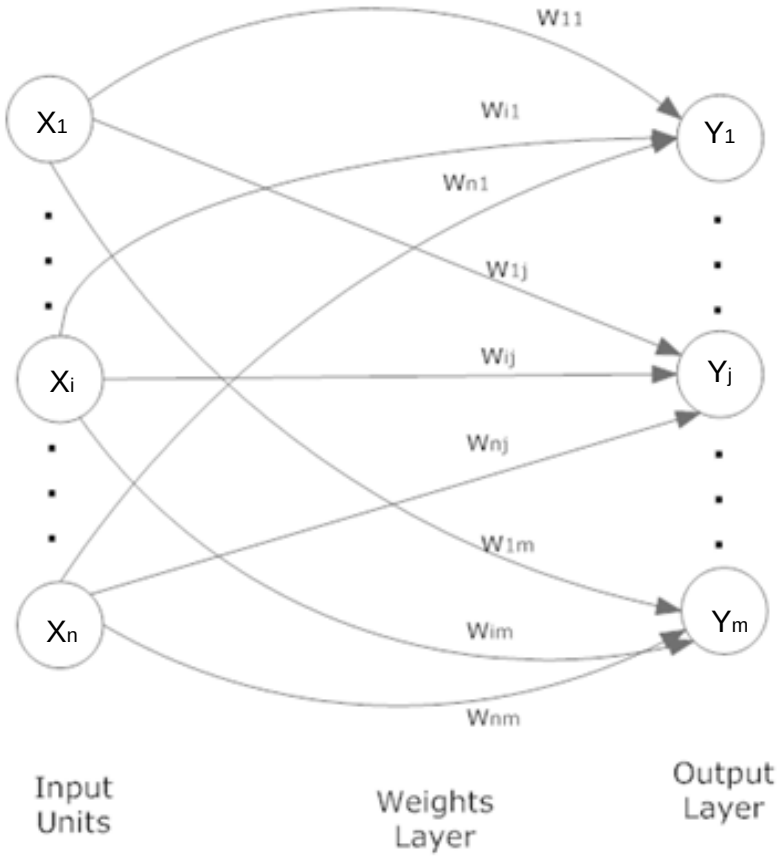


Figure 2.3 A single-layer neural net

Alternatively, a second way to determine the number of layers of a particular network is by using the number of layers of weighted interconnects links between slabs of neurons [Fau94]. Single layer networks are commonly used for pattern classification, where each output unit can be mapped to correspond to a particular category to which an input vector may or may not belong. However, this type of network has the drawback of not being complex enough for complicated mapping problems.

Multi-layer networks, on the other hand, are networks with one or more layers of nodes (that perform computation) between the input units and output units. This type of networks have a layer of weights between two adjacent levels of units and are able to solve more complicated problems than single-layer networks. A typical multilayer network is presented in Figure 2.4, where it can be seen that the network consists of a group of input units ($\boldsymbol{X_i}$) on the left and a group of output units ($\boldsymbol{Y_k}$) on the far right, just as the ones found in a single layer. However, an additional (hidden) slab of neurons ($\boldsymbol{Z_j}$) is also found in between the input and output slabs. This, in effect, causes a second layer of weighted interconnect links to be used for connecting the neurons in the hidden units and the output units, which is thus referred to as the hidden layer of weights.

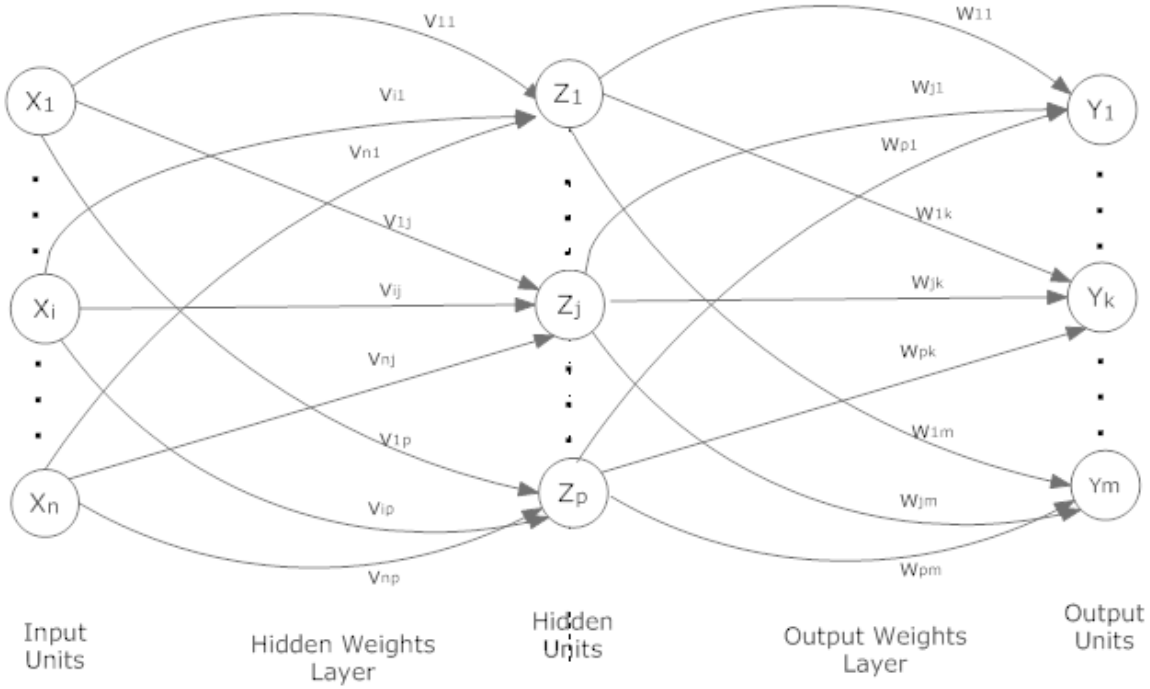


Figure 2.4 A Multi-Layer Artificial Neural Network

The above-described architecture of an artificial neural network is commonly called the "Feed-Forward" architecture, based on the fact that, when presented with a set of input values at the input units, the effects caused by the calculations performed on these values are propagated forward to the hidden layer and ultimately to the output layer. Thus, all communication occurs from the input side of the network to the output side, in the "forward" direction.

Multilayer networks are most often preferred over their single-layer counter parts, as they can be trained to solve more complicated problems. Among the benefits provided by the use of multilayer networks is the ability to modify each layer of hidden neurons to capture a particular feature in the training dataset. However, they possess the drawback that the training process may become more difficult and time-consuming as more weights must be fine-tuned to achieve effective learning in practical problems.

2.2 – THE ACTIVATION FUNCTION

As explained before, every neuron has the basic function of summing their weighted input signals and applying an activation function in order to produce an output signal [Mad07], and it is also a common practice for all the neuron in a particular layer to have the same activation function, although it is not required.

An activation function is usually an abstraction representing the rate of action potential firing in the cell [Hay99] and based on its calculated value, a given neuron can be determined to be either "ON" or "OFF," represented by the output values of 1 or 0, respectively.

A multitude of possible activation functions can be implemented for this purpose in an Artificial Neural Network. Among the possible options, the most popular are: Identity, binary step, binary sigmoid, and bipolar sigmoid functions.

The identity function, which is described by the mathematical relationship

$$f(x) = x, \text{ for all } x \tag{2.4}$$

is usually reserved for the neurons found in the input layer as their only function is usually to be a transparent buffer for input vectors to be propagated to the rest of the network. Refer to Figure 2.5 for a graphical representation of this function.

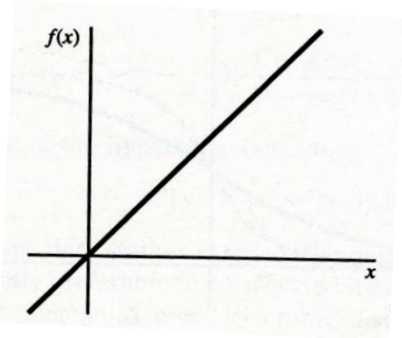


Figure 2.5 – Identity Function [Fau94]

A more useful function for general purpose neurons in an ANN is the Binary Step Function (also called Heaviside function), the reason being that it converts an input signal that is continuous in time and uses a threshold value to determine an output that is either logical "0" or "1." This relationship is further described by the following mathematical relationship:

$$f(x) = \begin{cases} 1, & if\ x \geq \theta \\ 0, & if\ x < \theta \end{cases} \tag{2.5}$$

where θ is the threshold value used to determine the output value of the function.

While the latter mentioned activation functions are applicable to implementations intended for simple neural networks, more complex activation functions are often needed for practical problems and training algorithms such as the Backpropagation algorithm. In particular, a special emphasis has been placed on the usage of the logistic sigmoid function, which can be scaled to cover any range of values but for practical purposes it is often restricted to have a range of 0 to 1, in which case it is called a Binary Sigmoid, or -1 to 1, which would be considered a Bipolar Sigmoid.

The logistic sigmoid function has found widespread use in the field of Neural Networks as it is instrumental in introducing non-linearity into the model and also because it helps to set bounds for signals to fall into a specific range of values [Mit97]. The function is expressed using the mathematical expression shown in equation 2.6 and its corresponding derivative (is shown in equation 2.7.) A graphical representation of the general shape of the signal is provided in Figure 2.6.

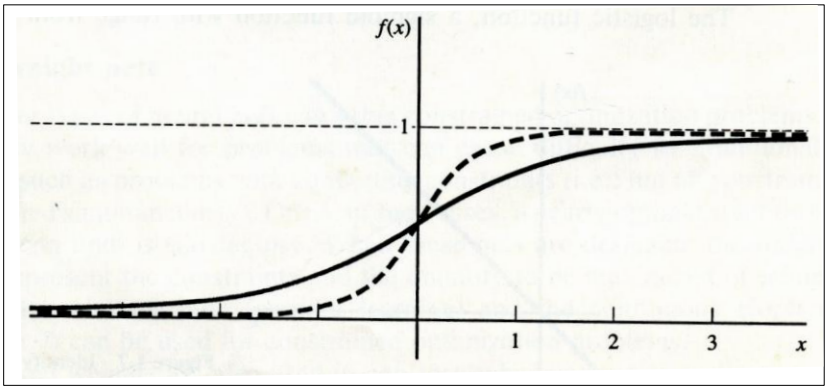


Figure 2.6 – Binary Sigmoid Function [Fau94]

$$f(x) = \frac{1}{1+\exp(-\sigma x)} \tag{2.6}$$

$$f'(x) = \sigma f(x)[1 - f(x)] \tag{2.7}$$

where σ is the steepness factor which determines how "smooth" the transition is, from a logical 0 to a 1.

Since the Backpropagation training algorithm has become the standard for practical applications of ANN's, there is considerable appeal for activation functions that are compliant with important constraints set in order for a signal to be considered useful in a Backpropagation training scheme. Namely, an activation function for a Backpropagation trained ANN should be continuous, monotonically non-decreasing, and its derivative easy to compute [Fau94]. The logistic sigmoid function, and in particular the Binary Sigmoid, are examples of functions that meet these criteria and, thus, thus are the most typical functions used.

Another signal of interest in the implementation of Backpropagation trained neural networks is a modified version of the logistic sigmoid, scaled to have upper and lower bounds 1 and -1 respectively. This signal is commonly known as the Bipolar sigmoid, given its nature to contain output values that fall in a range of positive and negative polarity. The idea behind using such an activation function for training a neural net is that a bipolar sigmoid function would tend to yield to faster learning when compared with the binary version of the logistic sigmoid function. This is true because a binary sigmoid,

when used as the activation function, blocks learning when the output of the function results in zero, whereas in the use of a bipolar function, even when the output results in a logical "false" value, learning can occur in a negative (or corrective, as opposed to assertive) direction [Fau94].

2.3 – Fundamental Training Algorithms for Artificial Neural Networks

In order for an artificial neural network to produce the desired outputs when presented with a pattern of inputs, the correct weights of the interconnection links must be first set. However, this weight-setting process does not occur instantaneously. Arriving to a set of weights that are able to produce results that meet the performance constraints set by the user, a training algorithm must be performed on the neural net in order for the ANN to gradually "learn" the desired set of weights.

Based on the architecture of an ANN and other aspects, such as the nature of the problem it is addressing, the training algorithm used to set the desired weights can fall in one of the main categories of training algorithms: Unsupervised, Supervised or Reinforcement Learning.

2.3.1 Unsupervised Learning

In contrast with supervised algorithms, un-supervised training algorithms do not use an associated target output vector value for every input vector fed into the network. Instead, networks that are trained in this manner use the provided training input vectors and group those that are similar together to specify behavior of a typical member of each group, should behave or to which group each vector belongs [Fau94]. Thus, the network effectively forms "clusters" of training input vectors to which the same output is associated, by adjusting the weights accordingly.

Alternatively, an ANN can be used for problems that may cause problems when approached with traditional techniques, due to the presence of conflicting constraints, in which a nearly optimal solution is satisfactory. In such cases, the weights are set to represent the constraints and the quantity to be maximized or minimized [Fau94].

An Example: The Hebbian Learning Rule

Following the convention for association of vector pairs (**s:t**), one of the first algorithms utilized to set the weights of an associative memory neural net is the Hebbian Rule [Fau94]. The traditional learning process of a Hebb rule-based net is built on the principle that learning occurs by modification of the synapse strength, i.e. if two interconnected neurons are both "ON" at the same time, then the weight between those neurons is increased. However, more effective learning may occur for this type of nets if the weight of the interconnection between to neurons that are "OFF" is decreased at the same time. The actual algorithm used to train a Hebb net is detailed as follows:

1. The weights are initialized to zero:

$$w_{ij} = 0 \qquad \text{For } i = 1, \dots, n;\ j = 1, \dots, m \tag{2.8}$$

2. For each training vector pair s:t:

    a. Set activations for input units to current input values:

$$x_i = s_i \quad \text{For } i = 1, \dots, n. \tag{2.9}$$

    b. Set activations for output units to current target output values:

$$y_j = t_j \quad \text{For j = 1, … , m.} \tag{2.10}$$

c. Adjust the weights accordingly

$$w_{ij}(new) = w_{ij}(old) + x_i y_j \qquad \text{For i = 1, …, n; j = 1, …, m.} \tag{2.11}$$

An important caveat to keep in mind, when implementing the Hebb rule training algorithm, is that a net will only be able to achieve convergence to a correct set of weights if the input training vectors are uncorrelated. Otherwise, if correlation exists between the training input vectors, a phenomenon known as *crosstalk* will occur, in which the response of the net when tested with one of the training vectors will include a portion of each of their target values [Fau94].

2.3.2 – Supervised Training Algorithms

A supervised training algorithm consists of presenting the neural network with a sequence of pre-determined training data, each consisting of an input pattern and its corresponding target output. The idea behind this training style is to have the input-output mapping of the network become consistent with the training data over a series of iterations. In other words, the error between the ANN's output and the training data's target output gradually reduces until it reaches an acceptable value [Yen99].

For each input pattern, the network performs a feed-forward sweep across its neurons and produces an output value which is then used by the algorithm to adjust the weights of the network accordingly.

Because of the nature of supervised algorithms, they tend to lend themselves as useful tools in addressing problems that involve mapping of inputs to outputs, pattern classification, and pattern association [Fau94]. The difference between pattern classification and pattern association is that the former classifies an input vector as belonging to a category or not; whereas the latter takes a wider range of possible outputs into consideration, i.e. the network is trained to associate a particular set of input vectors to a particular output pattern (rather than "belong" or "does not belong"). This behavior is usually referred to as *associative memory*. Furthermore, a neural network that implements associative memory may either fall into the category of being *auto-associative* or *hetero-associative*.

An auto-associative neural network is one in which the training input and target output vectors are identical to each other [Fau94]. This type of networks are commonly used for applications that require the network to *store* the vectors with which it has been trained and be able to recognize or reproduce the stored patterns when similar or noisy versions of the training patterns are introduced into the system at the input side.

In contrast, hetero-associative neural networks are those in which the values at the output side of the network are different from those present at the input side for each pattern in the training set. Consequently, this type of networks is most commonly used in problems that require the network to be able to store a set of pattern associations and use this information to classify a particular input into any of the stored categories.

For an associative memory artificial neural network – that is, one in which the end purpose is to associate or categorize the response of the network produced by a particular set of input patterns – the training process involves considering a set of input-output association vector pairs of the form **s:t**.

Where, if each target vector, **t,** is the same as the input vector, **s,** with which it is associated, then the net has an auto *associative memory*. Otherwise, if the target vector values in **t** are different from input vector values in **s**, then the net has *hetero associative* memory, as explained in the previous section. In either case, it is convenient to consider a separate set of input vectors, **x,** which is similar to the values in the dataset used during training, but are also patterns that the network has not seen before.

2.3.3 – Error Correction Learning Rules

The idea behind error correction rules is to gradually drive the output error of a given unit in the net to zero, thus, aiming to achieve system-level convergence with a "local error supervision" approach which is consequential to the fact that these rules were proposed initially as ad hoc rules for single-unit training [Moh95].

In this section, a couple of examples, which provide the background theory of such rules, are presented in order to ultimately arrive at Back Error Propagation rule which is the most widely used [Day90].

The Perceptron Learning Rule

The perceptron learning rule is a more powerful rule than many of the first rules to be implemented for ANN training given that, under suitable assumptions, its iterative procedure can be proven to converge to the correct weights [Fau94].

Historically, the original perceptron was designed with an approximate model of a human retina in mind [Day90]. It initially had three layers of neurons, which were the sensory units, associator units, and one response unit at the output. However, since only the weights from the associator units to the output unit can be adjusted, further studies have limited consideration to the single-layer portion of the net at such junctions [Fau94]. Thus, in order to visualize the implementation of the perceptron learning rule, consider the linear threshold gate shown in Figure 2.7.

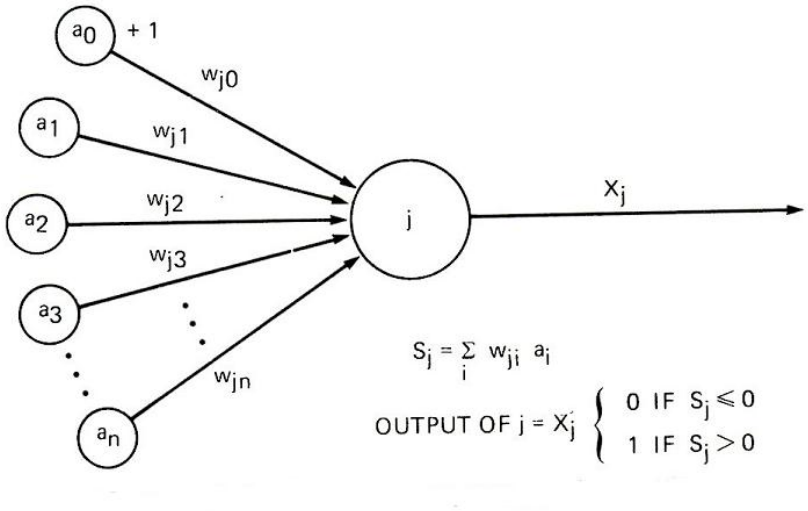


Figure 2.7 – A Perceptron Processing Unit [Day90]

The computational unit (or *perceptron*) shown in the Figure 2.7 maps an input vector **a** = $[a_0,\ a_1, \ldots a_i, a_n]^T$ to an output unit, **j,** that produces a response that may be either binary or bipolar. In order to calculate its response, the output unit of the perceptron combines the input vector **a** with the values of the weights, assigned to its interconnections with the associator units, represented by the vector **w** =$[w_0,\ w_1, \ldots w_i, w_n]^T$ by performing the weighted sum:

$$y_{in} = \sum_i a_i w_i \tag{2.12}$$

Based on the value calculated for y_{in}, the response of the output unit is determined using the following thresholding function:

$$y = \begin{cases} 1 & if\ y_{in} > \theta \\ 0 & if\ -\theta \leq y_{in} \leq \theta \\ -1 & if\ y_{in} < -\theta \end{cases} \tag{2.13}$$

The threshold, θ, of the activation function used to determine the value of y above is a fixed, non-negative value.

Once the value of y has been determined, it is compared to the desired response *t* for that particular input pattern, which would be +1 to signify belonging to a particular class or – 1 to signify not belonging. Then if an error occurred for the input pattern being considered, the weights are updated, for a particular learning rate, $\propto$, using the rule:

$$\text{If } y \neq t, \qquad w_i(new) = w_i(old) + \propto t x_i$$

$$\text{Else} \qquad w_i(new) = w_i(old)$$

Thus, in this learning rule, weights are only updated for patterns that do not produce the correct value of *y*.

Variations of the learning rule include setting the learning rate α to any non-negative constant. For example, $\propto$ could be set to $\frac{1}{|\boldsymbol{x}|}$ in order for the weight change to be a unit vector, or to $\frac{(\boldsymbol{x} * \boldsymbol{w})}{|\boldsymbol{x}|^2}$ in order to have weight changes that are just enough for the pattern **x** to be classified correctly for the current step [Fau94].

The Delta Learning Rule

In order to avoid the drawbacks and pitfalls (such as crosstalk) presented by simpler learning rules – such as the Hebb rule – other , more complex algorithms like the Delta Rule have been introduced. The Delta Rule is a gradient descent method that aims to minimize the error over all training patterns by using the produced error and adjusting the weights of the net for each pattern, one at a time as they are presented. Thus, for each weight update, the least squares error produced by the output of the network gradually moves towards a minimum error, as illustrated in Figure 2.8.

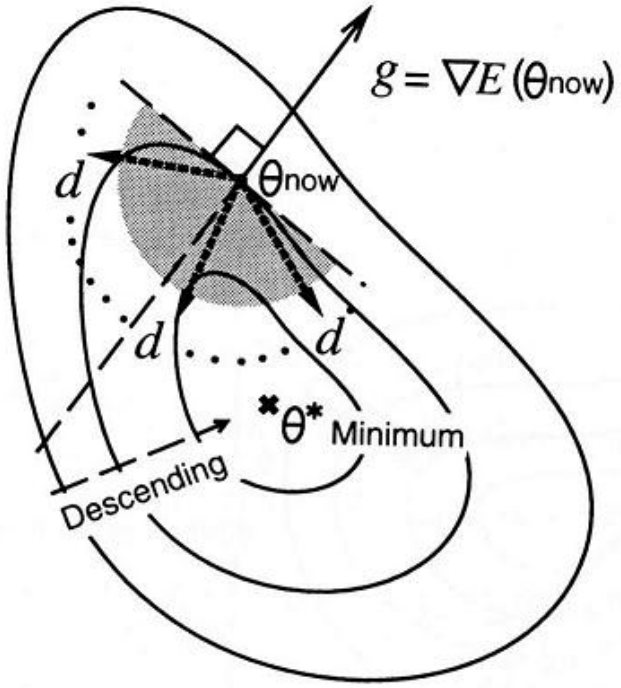


Figure 2.8 – Possible gradient descent directions towards a minimum error. Directions in the shaded area are possible descent candidates from initial position θ_{now} [Jan95.]

In order to illustrate the rule, consider that the squared error for a particular training pattern is described by

$$E = \sum_{j=1}^{m}(t_j - y_j)^2 \tag{2.14}$$

Where t_j is the target output value of output unit *j*, and y_j is the computed output for that same output unit. The value of y_j is calculated with respect to the values in an input vector $\boldsymbol{x}$ to the output unit Y as follows:

$$y_j = \sum_{i=1}^{n} x_i w_{ij} \tag{2.15}$$

And given that the gradient of the error, E, (i.e the partial derivatives of E with respect to the weights of the network) is the vector that indicates the direction of the most rapid increase in$E$. Then, the negative gradient of E $(-\frac{\partial E}{\partial w_I})$ will hence indicate the direction to the most rapid *decrease* in E.

$$\frac{\partial E}{\partial w_{IJ}} = \frac{\partial}{\partial w_{IJ}} \sum_{j=1}^{m} \left(t_j - y_j\right)^2 \tag{2.16}$$

$$= \frac{\partial}{\partial w_{IJ}} \left(t_J - y_J\right)^2 \tag{2.17}$$

Consequently, given that $y_J = f(y_{inJ})$ and

$$y_{inJ} = \sum_{i=1}^{m} x_i w_{iJ} \tag{2.18}$$

It follows that

$$\frac{\partial E}{\partial w_{IJ}} = -2(t_J - y_{inJ}) \frac{\partial y_{inJ}}{\partial w_{IJ}} \tag{2.19}$$

$$= -2(t_J - y_{inJ})x_I f'(y_{inJ}) \tag{2.20}$$

Thus, given a particular learning rate $\propto$, the general form of the delta rule is

$$\Delta w_{IJ} = \propto (t_J - y_J)x_I f'(y_{in}) \tag{2.21}$$

Thus, the delta rule uses the negative gradient of the error, E, to make adjustments to the weights in order to gradually approach a point of convergence, which happens when the value of the error of the entire system has reached or surpassed an established threshold that indicates an acceptable error rate. The delta rule for adjusting the I^{th} weight of a net with a single output is

$$\Delta w_I = \alpha(t - y_{\_in}) * x_I \tag{2.22}$$

The benefit of the Delta Rule is that it can be used for input patterns that are linearly independent but not orthogonal (correlation exists for the input vectors used for training) and, in addition, produces the least squares solution when input patterns are not linearly independent [Rum86].

2.4 – BACK-ERROR PROPAGATION LEARNING

Back-Error Propagation (commonly referred to as just Back Propagation) is a gradient descent method that minimizes the total squared error of the output values produced by a multilayer, feedforward, supervised ANN. This training method has breathed new life into the study and

applications of Artificial Neural Networks, as it has overcome multiple limitations that had previously been imposed by other, more archaic, training algorithms such as the Hebb Learning Rule (explained above), which are only found useful in applications that have linearly separable classes.

This paradigm has been found useful in a wide span of applications that range from autonomous vehicle control to medical diagnosis and it is particularly popular in applications that require pattern recognition, and this is because a neural network trained using this method has the ability to train hidden layers [Day90]. Since each hidden layer in a multi-layer ANN acts as a feature detector for a particular set of inputs, a more accurate classification of multiple patterns can be achieved by the use of the Backpropagation training, and this characteristic is where the paradigm derives its popularity.

Back-propagation networks represented a big step forward when they were first introduced, as they are not limited in the number of interconnection layers of an ANN, in contrast to other schemes such as the perceptron. The advantage of being able to have an unlimited number of layers is that a back-propagation network actually is able to achieve enhanced performance as the number of layers is increased since each layer is able to be trained to recognize a specific feature about the input by using a different activation function at each layer which all neuron in a particular layer share.

Back Propagation can be found to be simpler to implement than other paradigms that may not be as efficient, and this is because the idea at the core of this learning method is to focus on the cases when the network produces an error at the output. Once this happens the network's weights are corrected (or modified) using a learning algorithm that bears close similarity to the delta learning rule, which aims at lessening the amount of error for future input pattern responses. The goal of this learning algorithm is to eventually reach a point in which the error is ideally non-existent or at least reduced to a level that is acceptable by the user (falling below a threshold value for the desirable error of the system).

2.4.1 – The Back Propagation Network Architecture

The Backpropagation algorithm requires a neural network topology of, at least, three layers, and such topology is shown in Figure 2.8 below. As it can be seen from Figure 2.8, the first layer is called "input layer" and it consists of neurons whose only purpose is to receive external input and propagate it to the subsequent layer.

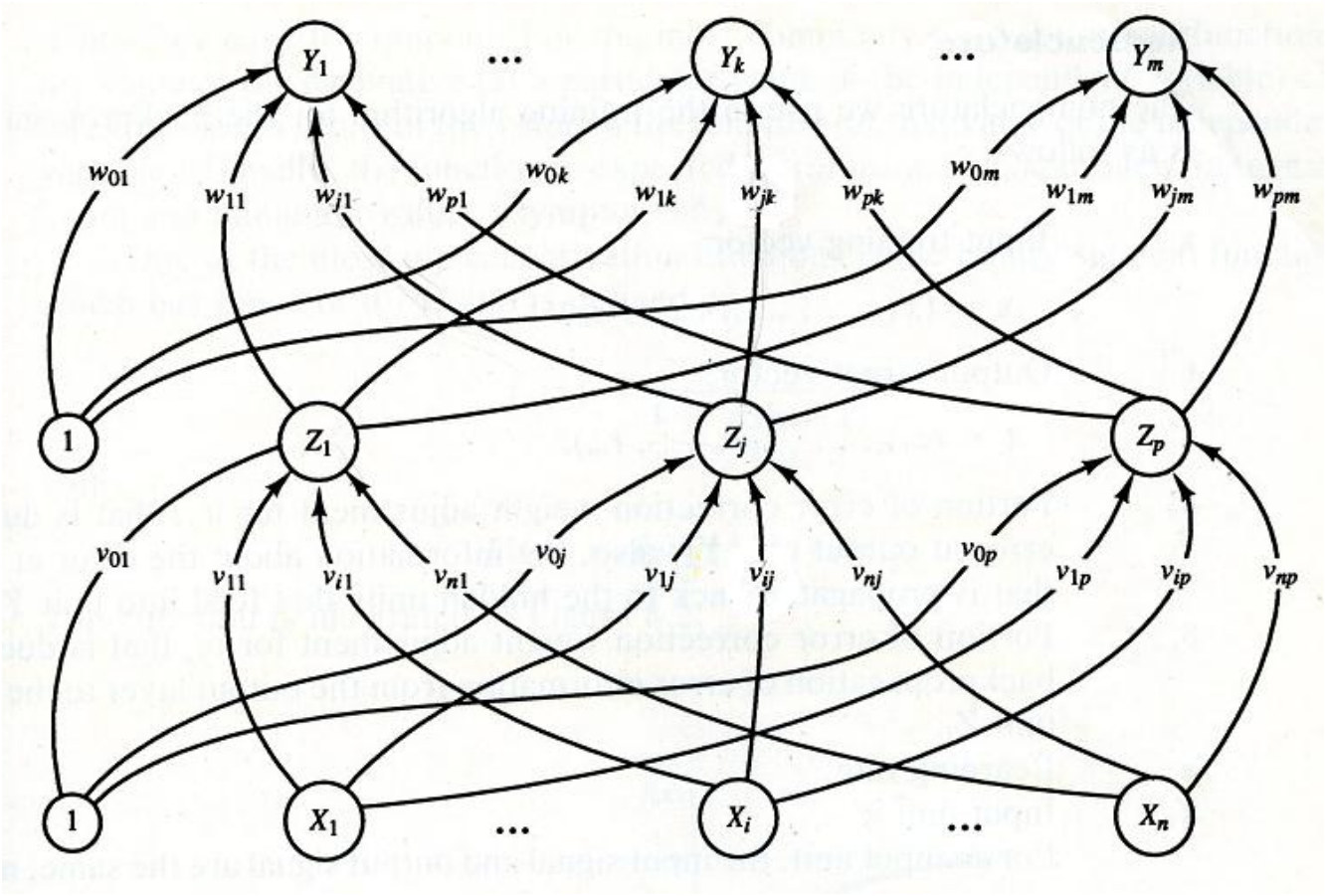


Figure 2.8 – A Backpropagation neural network with one hidden layer [Fau94]

In the case illustrated by Figure 2.8 only one middle layer is present in the network and it receives its input values from the nodes in the input layer, but its outputs are not directly reflected externally and, thus, it is called "hidden" layer. The neurons that are located in the hidden layer are fully

interconnected (as can be seen in Figure 2.8) to all the neurons located in the layers directly preceding and succeeding it, however, no connection exists between nodes in the same layer, and this is also true for all the neurons found in the input and output layers. Also, the neurons that form part of the hidden layer have the purpose of aggregating the input values received from the input layer, apply an activation function to the aggregated value and present the result as input to the neurons present in the output layer.

The last layer is called the output layer, given that the neurons located in this level have the purpose of presenting the final values produced by the network to the exterior world. The neurons in the output layer are fully interconnected to the neurons in the hidden layer directly preceding them and act as processing neurons which have the main function of receiving the values from the preceding layer, using them to perform a weighted sum and then applying activation function to the aggregate.

2.4.2 The Back Error Propagation Training Algorithm

The Back Error Propagation training algorithm consists of three main steps, namely: the feedforward and processing of the training input patterns, the back propagation phases of the resulting errors from each training pattern and the consequential weight updates derived by applying the delta learning rule on the errors obtained during back propagation.

The feedforward phase commences when a new input vector is presented to the input layer of the network, at which point, the neurons comprising the input layer assume their corresponding input values using the identity function $f(x) = x$, and subsequently the received input signal is broadcast to each of the neurons located in the hidden layer. Each hidden unit receives the input values and performs a weighted sum

$$z_{in} = \sum_i x_i w_{ji} \tag{2.23}$$

where x_i is the value being broadcasted by input unit i and w_{ji} is the weight of the interconnection link connecting unit i to unit j. Once the weighted sum z_{in} is computed, an activation function is applied to its calculated value $f(z_{in})$, where f is usually a sigmoid function of the form

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.24}$$

Thus, for the particular case of the hidden layer processing unit j, the equation for the activation function would be

$$f\left(z_{in_j}\right) = \frac{1}{1+e^{-z_{in_j}}} \tag{2.25}$$

Consequently, the calculated value z_j, obtained from the activation function, is then sent along through the output layer of interconnections to all of the output neurons, which is then used in the back propagation phase of the training algorithm.

While still focusing on the feedforward phase, it is worth mentioning the importance that the presence of a bias unit (as described in the introduction to this chapter) may have a bearing on whether a back propagation network achieves convergence or not. And if it does achieve convergence, the speed at which this occurs. As explained earlier in the chapter, each bias unit is connected to and provides a constant term in the weighted sum of the units in the next layer. By doing so, the bias unit provides an adjustable threshold effect on each unit it targets and contributes a constant term in the summation z_{in_j}.

Once the current input pattern values have been propagated entirely across the network, to the point where the neurons in the output layer have each calculated their corresponding output values y_k, it is then that the back propagation phase begins.

In the back propagation phase, the calculations begin at the output layer and proceed backwards through the network toward the input layer. Using the idea behind the delta learning rule, the Backpropagation phase compares the calculated output value y_k, for the current training pattern, to the target output value t_k and, based on the difference or error, a factor δ_k is calculated for each unit in the output layer, as shown in Figure 2.9 (c). The δ_k factor is then used to propagate the error of output unit $Y_k$back to all of the units in the previous layer. Subsequently, a similar process is followed to calculate a factor δ_j corresponding to each unit, Z_j, in the hidden layers of the network. Once the δ_j factor has been calculated for the last hidden layer, it is not necessary to propagate the error back to the input layer and the algorithm continues with the interconnection weight update phase.

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk} \tag{2.26}$$

For example, the adjustment of the weight w_{jk} (from the hidden unit Z_j to the output unit Y_k) is based on the factor δ_k and the activation z_j of the hidden unit Z_j [Fau94].

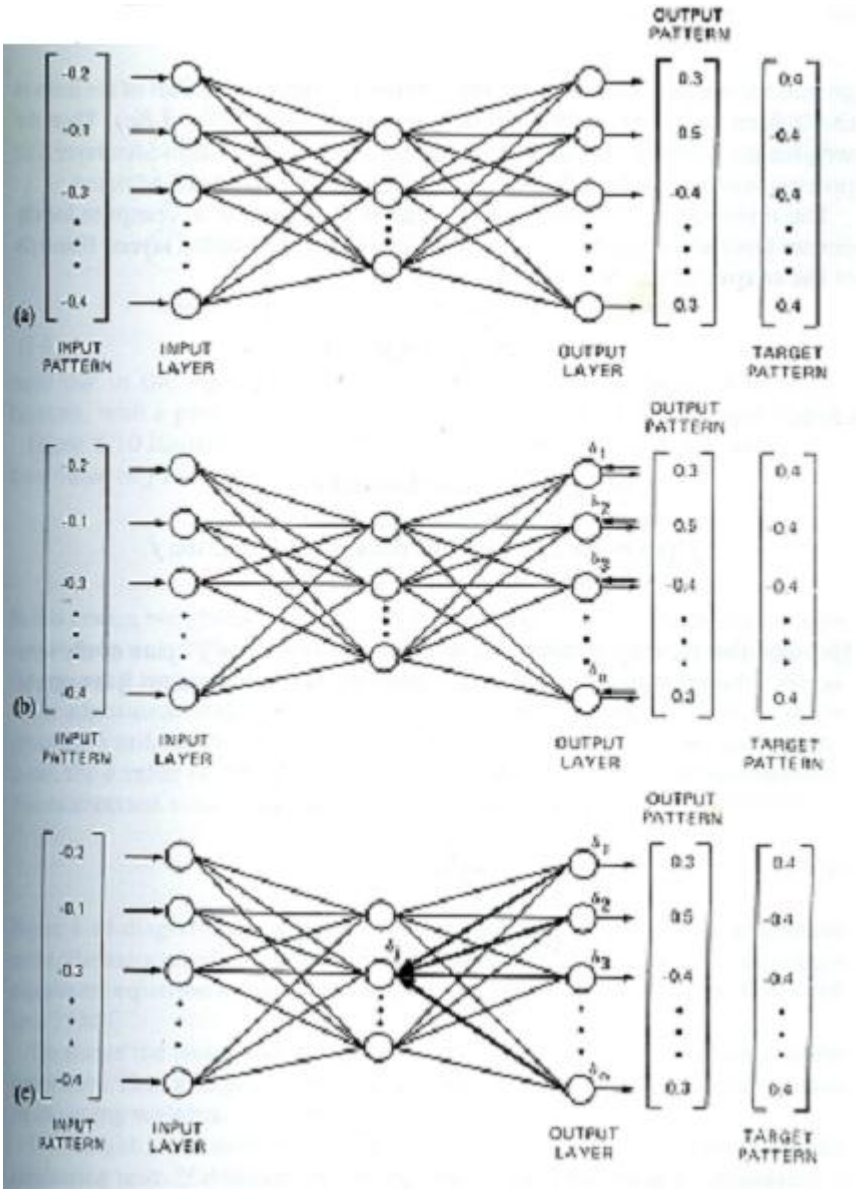


Figure 2.9 – Illustration of the Steps of the Backpropagation training [Day90]

2.5 – REINFORCEMENT LEARNING

Reinforcement learning occurs when a learning algorithm follows a process of trial and error designed to maximize the expected value of a criterion function otherwise known as a reinforcement signal. Such an algorithm would use positive or negative reinforcement signals to modify the learning parameters of a net accordingly. This type of learning is based on the idea that if an action is followed

by an "improvement" in the state of affairs, then the tendency to produce that action is strengthened i.e., positive reinforcement occurs. Otherwise, if an action is followed by an adverse response, the tendency of the system to produce that action is weakened.

2.6 – AN EXAMPLE APPLICATION: THE XOR PROBLEM

In order to further visualize the theory behind multilayered, feedforward neural networks and discern some of the benefits they provide, it is useful to consider one of the simplest pattern recognition problems in neural network literature, the XOR (exclusive-OR) problem. The task in this problem is to be able to train a neural net to implement the Boolean logic operation $F = AB' + A'B$, which corresponds to the function table shown in Table 2.1. As can be observed from the table of the XOR function, the output of the net is expected to be activated (or have a value of 1) whenever the two input units have values that differ from each other, otherwise the output is to remain de-activated (or have a value of 0), thus the problem is said to be of binary nature. If the same problem was to be considered using bipolar logic, all the instances of 0 in the truth would be changed to -1.

Table 2.1 XOR Problem Truth Table

X0	X1	Y
0	0	0
0	1	1
1	0	1
1	1	0

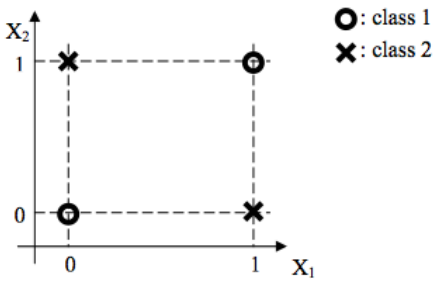


Figure 2.10 The XOR Problem Class Representation [Jan97]

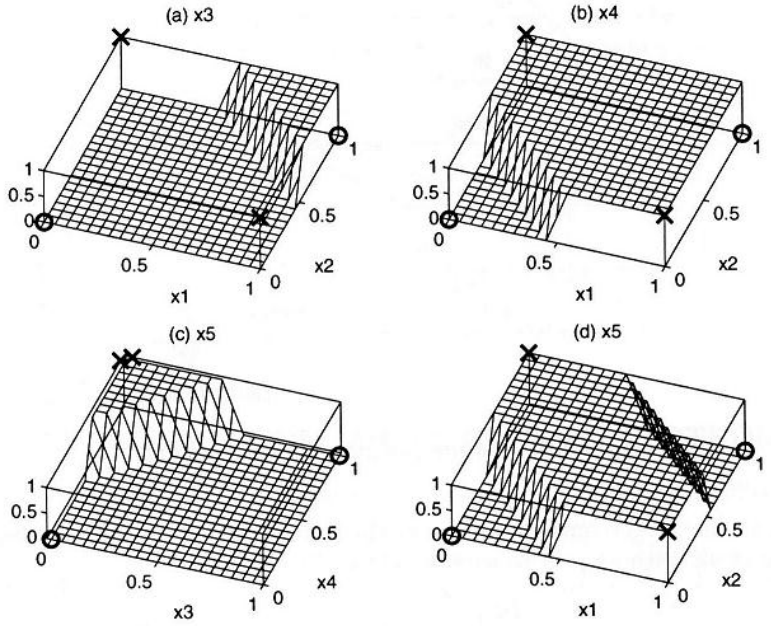


Figure 2.11 – Decision Surface of the XOR benchmark during training [Jan95.]

It is also worth noting that this problem is not linearly separable as can be seen in the plot shown in Figure 2.10. In other words, a single-layer perceptron would not be able to construct a straight line to partition the two-dimensional input space into two regions, each containing only data points of the same

class. It is, however, possible to solve the problem with the two –layer perceptron illustrated in Figure 2.12, in which the connection weights and threshold values are indicated.

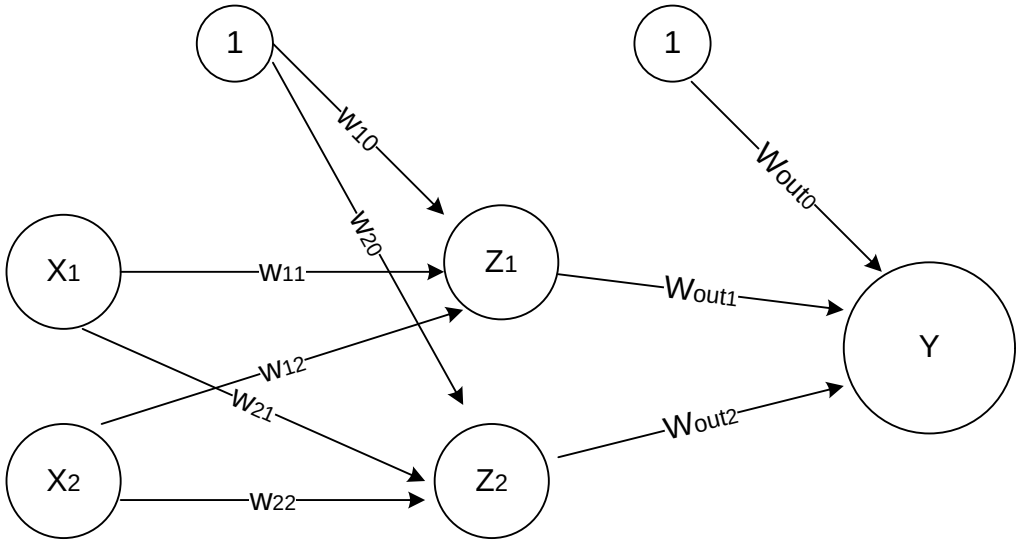


Figure 2.21 – A two-layer perceptron used for the two-input XOR problem

In summary, by considering this simple example, it is clear to see how multilayer perceptrons are more powerful than their single-layer counterparts since they are able to solve nonlinearly separable problems, and can actually be applied to practical problems.

Chapter 3 – Parallel and Distributed Computing

3.1 – INTRODUCTION

In today's scientific world, the usage of extraordinarily powerful computers has become the norm to address the intense numerical computation needs imposed by significant scientific problems. This is true particularly in the fields of astrophysics, computational fluid dynamics, global weather modeling and Protein folding, to name a few examples.

In order to satisfy the demand for more computational power, the microprocessor industry is continuously struggling to produce new architectures that are able to increase the peak floating point operation execution rate (floating point operations per second, or FLOPS). In order to do so, tremendous advances have been made in the past decades in the area of VLSI circuit design and manufacturing that have translated into an increase in clock rates from about 40 MHz (e.g., a MIPS R3000, circa 1988) to over 2.0 GHz (e.g., a Pentium 4, circa 2002) [Gra03].

These advances have been made possible through the development of architectural as well as technological improvements. In particular, the implementation of innovative ideas – such as cache memory, instruction pipelining, interleaved memory and multiple functional units – have contributed quite significantly to the improvement in performance of traditional single-processor computers, and since their implementation, further improvement in performance has become increasingly difficult since the speed of electronics is limited by the speed of light [Qui94]. Thus, to achieve the extremely high speeds demanded by contemporary science, computer architectures must now incorporate parallelism at the highest level of the system.

3.2 – PARALLEL COMPUTING TERMINOLOGY

Common examples of this widespread use are multiprocessing – a method used to achieve concurrency at the program level – and instruction pipelining – a method used to achieve concurrency at the instruction level, in which an instruction is divided into sub-tasks and treated as stages of an assembly line. However, while it might be the case that, at any particular level, modern computers have some degree of parallelism, it is not desirable to call every modern computer a parallel computer. Hence, it is important to establish a set of definitions that may help to put things in perspective when comparing parallel and concurrent systems.

When talking about *Parallel Processing,* what is meant is the process in which concurrent manipulation of data elements belonging to one of more processes solving a single problem is emphasized by information processing. Consequently, a *parallel computer* is a multiple processor computer that is capable of performing parallel processing.

Also, in order to categorize the performance of a particular parallel computer when compared to another, reliable metrics that provide meaningful information about the qualitative differences of a system are needed. One of such metrics is the *throughput* which is defined as the number of results a particular system is able to produce per unit time [Qui94]. One of the most common ways to increase the throughput of a particular system is through the use of instruction concurrency.

Another metric that is often used to qualify the effectiveness of a parallel implementation is the *speedup.* The speedup achieved by a parallel algorithm running on p processors is the ratio between the time needed for the most efficient sequential algorithm (running on a single processor) and the time taken by the same computer when implementing a parallel version of the algorithm using p processors. And once the speedup has been calculated for a particular parallel implementation, the *efficiency* of such implementation can then be determined by dividing the speedup by the number of processors p.

3.3 – PARALLEL COMPUTER ARCHITECTURES

A parallel computer may be designed with an architectural point of view that is focused on a particular application. For example, parallel computers that are designed to perform graphics rendering may have an architecture that is heavily invested in vector matrix multiplication and may be very different from the architecture used for a processor that is to be used for an application that requires a large amount of scientific computations, which must rely on an architecture that focuses on large vector computation.

This is why, in order to put things in perspective, it is convenient to classify the dichotomy of parallel architectures based on their logical organization. To this end, the best classification scheme found in literature is Flynn's Taxonomy, which is based on the idea that any particular computer architecture consists of two main concepts: an instruction stream and a data stream. An instruction stream is a sequence of instructions performed by a computer, while a data stream is a sequence of data manipulated by an instruction stream [Qui94]. Flynn uses these two concepts and the idea that a particular architecture may implement, or not, a level of multiplicity of instruction streams, data streams, or both. Thus, four classes of computers result from the given multiplicity combinations of instruction and data streams:

The first of the four classes considered is the *Single Instruction stream, Single Data stream* (SISD), where even though computers that fall into this category may have multiple functional units that could potentially execute multiple tasks at once, these functional units are still managed by a single control unit. This is the category in which most serial computers would fall.

The second category is the *Single Instruction stream, Multiple Data stream* (SIMD) which is similar to SIMD in the fact that a computer in this category executes a single stream of instructions; However, it contains multiple arithmetic processing units, each capable of fetching and manipulating its own data.

Hence, at any given time, a single operation is in the same state of execution on multiple processing units, but each one handles a different set of data and produces output data independently of the other processing units. Computers that are designed to work on applications that require well-structured computations on parallel data structures such as memory arrays are well suited examples for the SIMD model. Hence, machines such as The Connection Machine CM-200 would fall into this category.

Multiple Instruction stream, Single Data stream (MISD) is the third class of computers in Flynn's taxonomy. Computers that fall in this category have architectures that contain multiple functional units and each one may perform a different operation on the same data as the rest. Machines in this category are rare to find given that this architecture approach is only useful for a reduced number of applications, such as fault-tolerant computing or on systolic arrays [Qui94], where data is fed from one processing unit to the next in a pipeline fashion.

Lastly, the fourth class of computers is Multiple Instruction stream, Multiple Data stream (MIMD). The architecture style of the computers found in this category achieves parallelism by having different processors executing different instruction on different sets of data at any particular time. MIMD can then be thought of as multiple-CPU computers designed for parallel processing and designed to allow efficient interactions among their CPU's.

3.3.1 – Sequential vs. Parallel Physical Organization

In order to fully understand the benefits of parallel computation, it is useful to consider the differences that exist in terms of how their architectures affect the models that are considered for developing a particular algorithm for each type of machine. At the same time, in order to compare and contrast both approaches, it is useful to think of each in general terms and keeping a high-level point of

view. This is why it is useful to think of the hypothetical random access machine (or RAM) and its parallel counterpart, the parallel random access machine (PRAM).

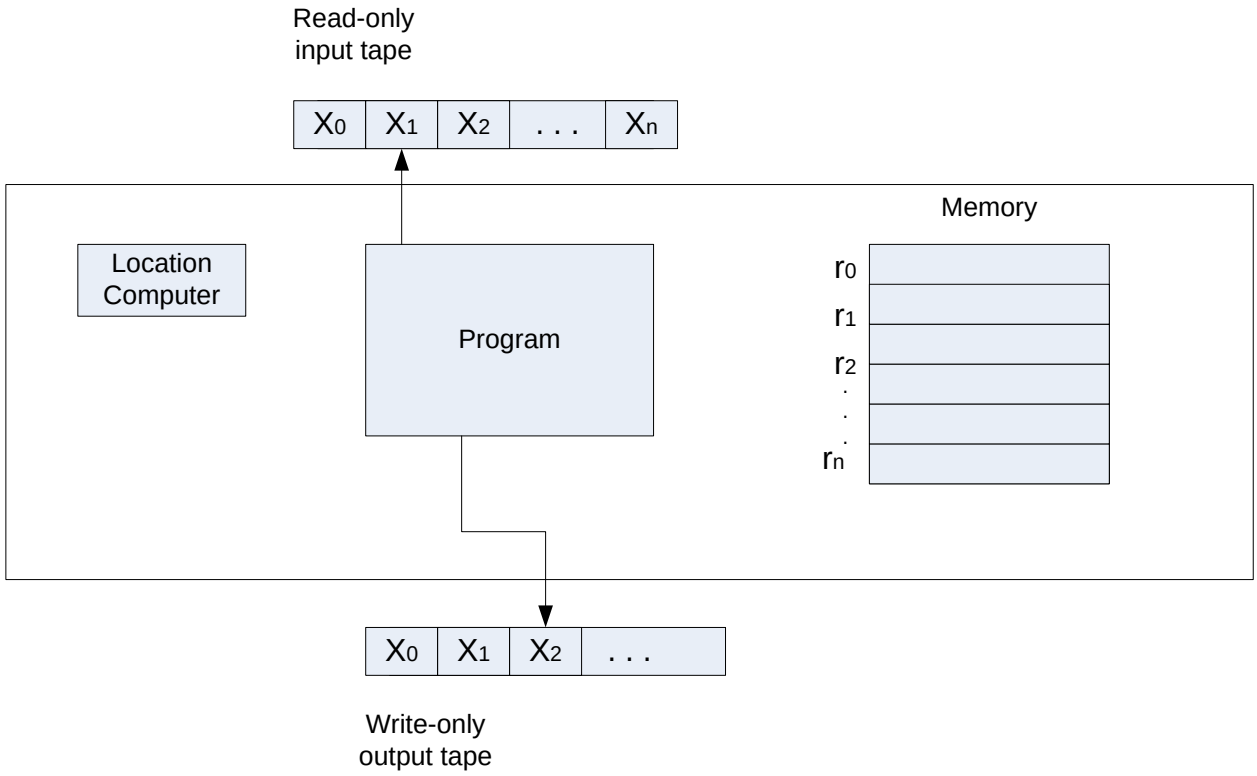


Figure 3.1 RAM Model for serial computation [Qui94.]

The RAM is a model of one-address computer and consists of a memory, a read-only input tape, a write only output tape, and a program which is not stored in memory and cannot be modified [Qui94]. The input tape contains a sequence of integers, and every time that a value is read, the input head advances one square. In the same manner, the output head advances after every time an output value is written to the output tape. In the RAM model, memory consists of an unbounded sequence of registers, which follow the nomenclature$r_0,\ r_1, r_2, \ldots, r_n$, where each register holds a single integer and it is able to perform computations on the integer stored in it.

Given the particular constraints placed upon the RAM model, the *worst-case time complexity* of a program implemented in this model is the function $f(n)$, which is the maximum time taken by the program to execute over all inputs of size n. Consequently, the *expected time complexity* of this model is the average of the execution times over all inputs of size n [Qui94]. A couple of important constraints are set to qualify the performance limit of a particular algorithm implemented in a RAM machine are the *uniform cost criterion,* which establishes that each RAM instruction requires one time unit to execute and every register requires one unit of space, and the *logarithmic cost criterion*, which takes into account that an actual word of memory has a limited storage capacity.

A PRAM, on the other hand, is an extension of the serial model of computation which consists of p processors and a global memory of unbounded size that is uniformly accessible to all processors [Gra03]. Due to the physical organization of a PRAM and the fact that concurrent access to memory locations is allowed, multiple categories of PRAM's exist depending on how simultaneous memory accesses are handled. In particular, four classes are highlighted in which PRAMs can be sub-divided: Exclusive-Read Exclusive-Write (EREW), Concurrent-Read Exclusive-Write (CREW), Exclusive-Read Concurrent-Write (ERCW), Concurrent-Read Concurrent-Write (CRCW). The nature of each one of these four classes of PRAM's is self-descriptive. It is important to notice, however, that the weakest model of the four is the Exclusive-Read Exclusive-Write PRAM since it affords the minimum concurrency in memory access, whereas, the most powerful would be Concurrent-Read Concurrent-Write, as it allows the highest degree of concurrency for memory access.

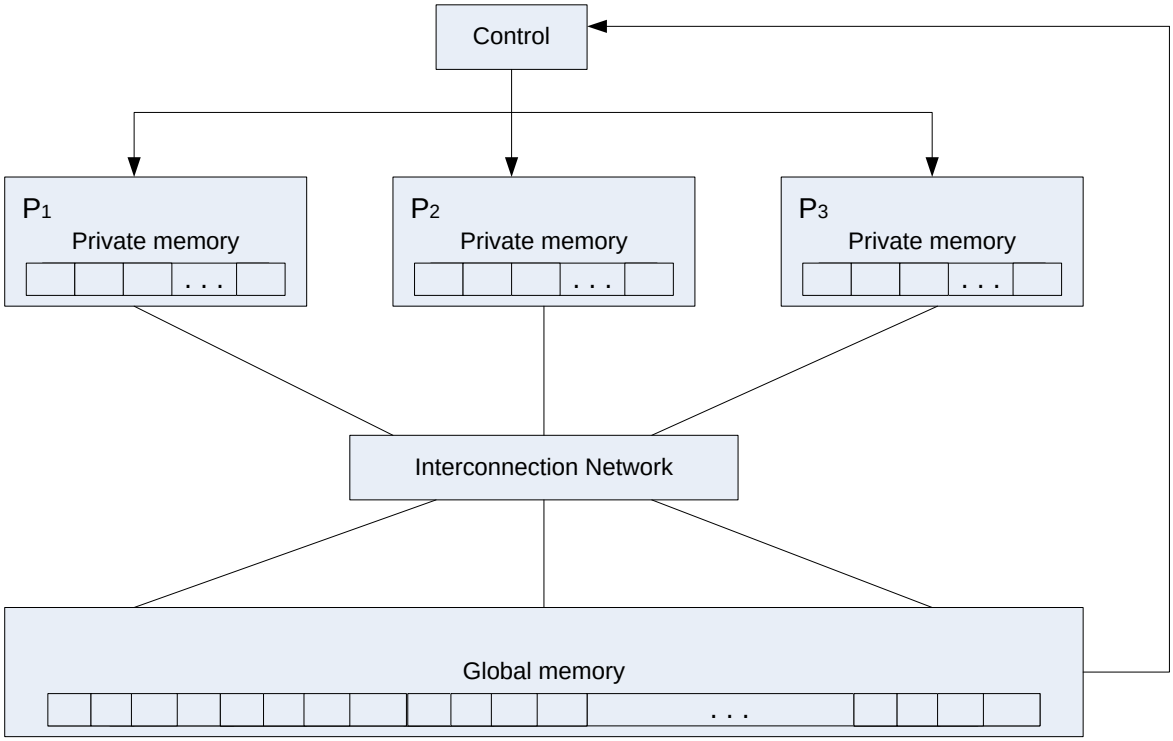


Figure 3.2 PRAM Architecture Model [Qui94]

Allowing concurrent write access to memory, however, presents a significant constraint on the integrity of the contents of memory and requires comprehensive arbitration. In order to address this constraint, several protocols are used to resolve concurrent writes to memory, with the most popular consisting of usage algorithms as follows [Gra03]:

- Common data values, in which all the processors attempting a concurrent write into the same global address must be writing identically the same value.
- Arbitrary, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- Prioritization, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority is the only one to succeed on the write operation.
- Summation, where the sum of all the quantities is written.

Thus, evaluating all possible combinations of these protocols with the four models of concurrency (for read and write operations), the CRCW model using the priority protocol to handle concurrent writes is found to be the strongest [Qui94]. However, in considering the PRAM model as the ideal architecture for parallel computation, it is important to also consider some of the practical difficulties associated with realizing this model. Consider for example, the implementation of an EREW PRAM as a shared-memory computer with p processors and a global memory of m words. Consequently in such a model, each one of the p processors in the network can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. Then to ensure such connectivity, the total number of switches must be $\theta(mp)$ which, for practical applications, constructing such a switching network would be very expensive, and thus, it is clear that PRAM models of computation, though ideal, are impossible to realize in practice [Gra03].

3.3.2 Multiprocessor Network Topologies

Given that the ideal architectural model for a parallel implementation – The PRAM – is nearly impossible to realize, other, more feasible, parallel architectures need to be considered for practical applications. One way to address this need is to consider *interconnection networks*, which provide the mechanisms for data transfer between multiple processing nodes and between these nodes and the memory modules they access. Typical interconnection networks are built using links and switches, where a link corresponds to physical media such as a set of wires or fibers capable of carrying information [Gra03].

Thus, interconnection networks can be classified as either *static* or *dynamic*. Static networks consist of point-to-point communication links among processing nodes. Whereas, dynamic networks are built using switches and communication links by using the switches to dynamically connect

communication links to one another in order to establish communication paths among processing nodes and memory banks.

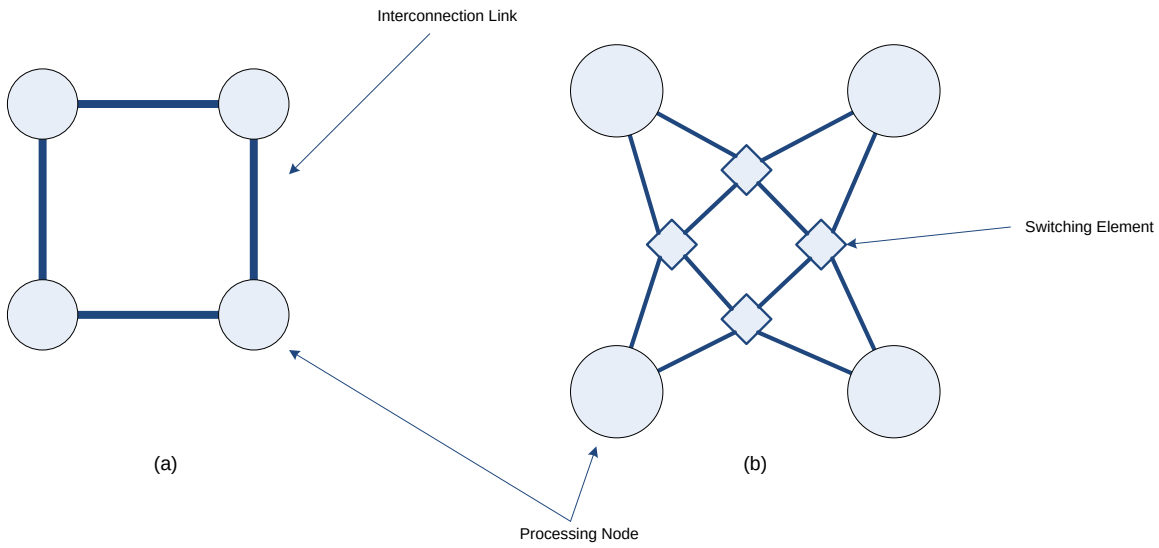


Figure 3.3 (a) A Static Network; and (b) A Dynamic Network

Several topologies exist which implement interconnection networks that are either static or dynamic. There is not a single "pure" topology that is always the best choice to tackle all practical problems as each one presents its own set of tradeoffs regarding scalability and performance. In practice, rather, a combination or modification of two or more topologies for a particular application is often found to produce the best results. In order to determine what tradeoffs must be made when considering one topology against another for a particular application, an evaluation is made to understand the effectiveness in implementing efficient parallel algorithms on real hardware. The criteria used in this evaluation process consist of four qualifying metrics which are: diameter, bisection, number of edges per node, and maximum edge length.

The *diameter* of a network is the largest distance between two nodes in the network. In practice, a the lowest diameter for a particular network is always preferred, because the diameter puts a lower bound on the complexity of parallel algorithms requiring communication between arbitrary pairs of nodes [Qui94].

The *bisection* width of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves. Usually, a large bisection width is desired given the fact that the implementation of algorithms that require large amounts of data to be passed around in the network, the size of the data set divided by the bisection width sets a lower bound on the complexity of the parallel algorithm.

Furthermore, it is generally better when the *number of edges per node* in a network is kept constant as the size of the network increases or decreases (in terms of number of nodes). This is because the processor organization is easily scalable when a larger number of nodes is used. Also, it is preferable, for scalability reasons, if the *nodes and edges of the network* can be laid out in three-dimensional space so that the maximum edge length is a constant independent of the network size [Qui94].

The Star-Connected Network

Among static topologies, one of the easiest to implement is the *star-connected network* topology, which is presented in Figure 3.4. In this topology, one processor acts as the central processor, and every other processor has a communication link connecting it to this processor. Communication between any pair of processors is routed through the central processor and, thus, the central processor becomes the bottle-neck of communication in the star-connected network. A natural counterpart of the star-

connected network that gets around the bottle-neck problem is the *fully-connected network*, shown in Figure 3.5.

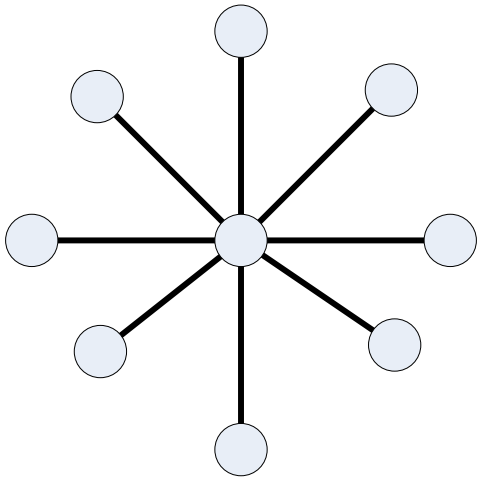


Figure 3.4 A Star Connected Network

The Fully Connected Network

The fully connected network, as its name suggests, consists of multiple processing nodes and each one of them has a direct communication link to every other node in the network. By not having to depend on a central node to communicate from one node to another, this topology has the advantage of providing the ability of one node to directly send a message to another in only one communication step. However, this network has the particular disadvantage of becoming very expensive to implement when considering networks that are appropriately sized for practical applications as it requires a large number of communication links since for every processing node n in the network there are $\frac{n(n-1)}{2}$ communication links.

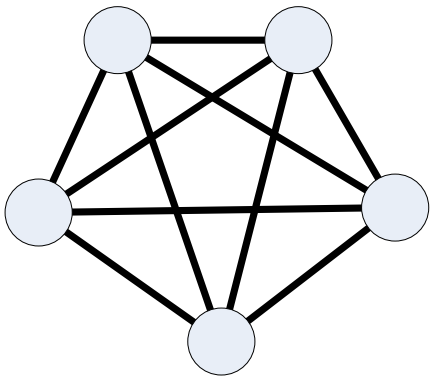


Figure 3.5 A Fully Connected Network

Linear Arrays and Mesh Networks

In order to move away from the problem of having a large number of communication links (which ultimately become overly expensive to implement), sparser networks are typically the weapon of choice to build parallel architectures of processing nodes. Perhaps the most sparse of all topologies in this category is the linear array (shown in Figure 3.6), which is a static network in which each node (but those found at the front and back ends) have a direct link connecting them to their right or left neighbor.

Figure 3.6 A Linear Array of Four Nodes

Additionally, taking a linear array and adding a wraparound connection link between the nodes at the extreme ends creates what is called a 1-D torus (Figure 3.7), in which each node has two neighbors.

Using this topology, one can expect the diameter to be in the order of $\left\lfloor \frac{p}{2} \right\rfloor$, a bisection of 2 (since any partition cuts across only two communication links), and have a cost of *p-1* links.

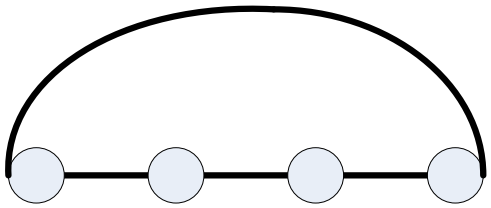


Figure 3.7. A 1-Dimensional Torus of Four Nodes

Furthermore, the linear array may be used as the basis for creating other, more complex and multi-dimensional, topologies. One example of how this is done, is the case in which a linear array (like the one shown in Figure 3.3), consisting of *n* nodes, is used as one of n rows in a n x n 2-Dimensional mesh as shown in Figure 3.8. In a mesh network, the nodes are arranged into a q-dimensional lattice and communication is only allowed between neighboring nodes, such that all nodes, with the exception of those in the periphery, communicate with $2q$ other nodes. Also, just like the linear array had the variant of becoming a torus, so does the mesh with the inclusion of a wraparound connection between processors on the edge of the mesh.

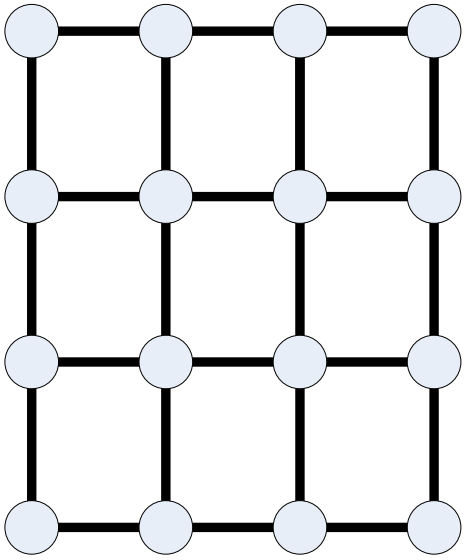


Figure 3.8 A 2-Dimensional Mesh

Thus, a two-dimensional mesh like the one shown in Figure 3.8 is a natural extension of the linear array shown in Figure 3.6 to two dimensions. Where each dimension has $\sqrt{p}$ nodes with each node identified by an ordered pair (*i*,*j*), and every node, with the exception of those found at the edges of the mesh, is connected to four other nodes whose indices differ by one in any dimension. Thus, a 2-dimensional *p*-node mesh without wraparound connections is will have a bisection of $\sqrt{p}$, a diameter of $2(p-1)$, a maximum number of edges per node of 2d, and a constant maximum edge length (for 2-D and 3-D meshes) [Qui94].

Moreover, the family of topologies associated with the linear array does not stop at the 2-D mesh, at the other extreme of the spectrum lays a topology which is often zealously studied and considered in the academic environment, called *The Hypercube*. A hypercube, or cube-connected network, consists of 2^k nodes forming a *k*-dimensional hypercube, in which each node is labeled 0, 1, 2, ..., $2^k - 1$; and two nodes are considered adjacent to each other if their binary labels differ in exactly one bit position.

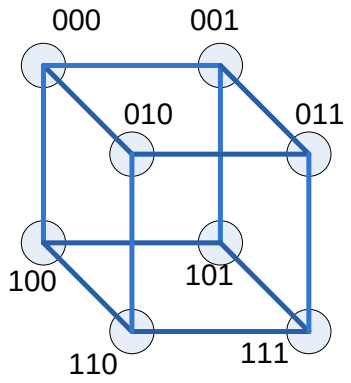


Figure 3.9 A 3-Dimensional Hypercube With Corresponding Labels

For example, the three-dimensional hypercube shown in Figure 3.9 has eight nodes with labels 000, 001, 010, …, 111. This numbering scheme has the useful property that the minimum distance between two nodes is given by the number of bits that are different in the two labels. This property becomes particularly useful when developing parallel algorithms to be implemented on hypercube architecture, which might not necessarily be available on other parallel architectures. Also, this numbering scheme applies very well to cases in which a hyper cube is extended from dimension *k* to *k+1* or vice versa, as it is illustrated below in Figure 3.10, where a three-dimensional hypercube is extended to a four-dimensional one by connecting corresponding nodes of two three-dimensional hypercubes. Thus, it follows that a d-dimensional hypercube can be expected to have a bisection of $\frac{p}{2}$, a diameter of *log p* (since two node binary labels can differ in at most *log p* positions), a number of edges per node log p, as well, and the length of the longest edge in this type of network can be expected to increase as the number of nodes in the network increase.

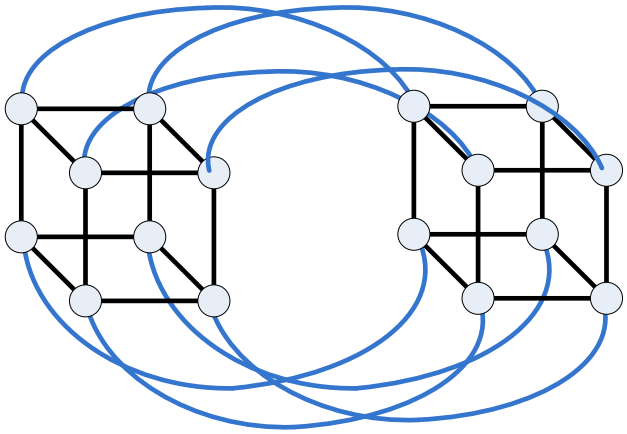


Figure 3.10 4-Dimensional Hypercube

Dynamically Connected Networks

Dynamically connected networks are also referred to as *indirect networks*. These networks differ from their static counterparts in the use of switches to provide connections between the nodes of a network. A single switch in an interconnection network consists of input and output ports, and the total number of ports on a switch determines the *degree* of the switch. These devices provide the minimal functionality of mapping their set of input ports to its corresponding set of output ports, but some may also provide support for internal buffering, routing, and multi-cast.

In a dynamic or indirect network, an interface is required in order to provide connectivity between the nodes in the network. This interface is needed as it is in charge of providing the means for packetizing data, computing routing information, buffering incoming and outgoing data for matching speeds of network and processing elements, as well as error checking [Gra03].

Crossbar Networks

Among the most popular dynamic networks used in parallel architectures is the *crossbar*. A crossbar network employs a grid of switches or switching nodes to connect *p* processors to *b* memory banks. In a crossbar networks the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks, making it an exemplar of non-blocking networks. As shown in Figure 3.11, in a crossbar network, the switches are arranged in matrix form and the total number of switching nodes required to implement this architecture is *p* x *b*. Hence it is apparent that while a crossbar can be scalable in terms of performance, as the number of processors of memory banks increases, so does the cost of implementing larger sizes of crossbar networks which becomes the main disadvantage to using this type of network.

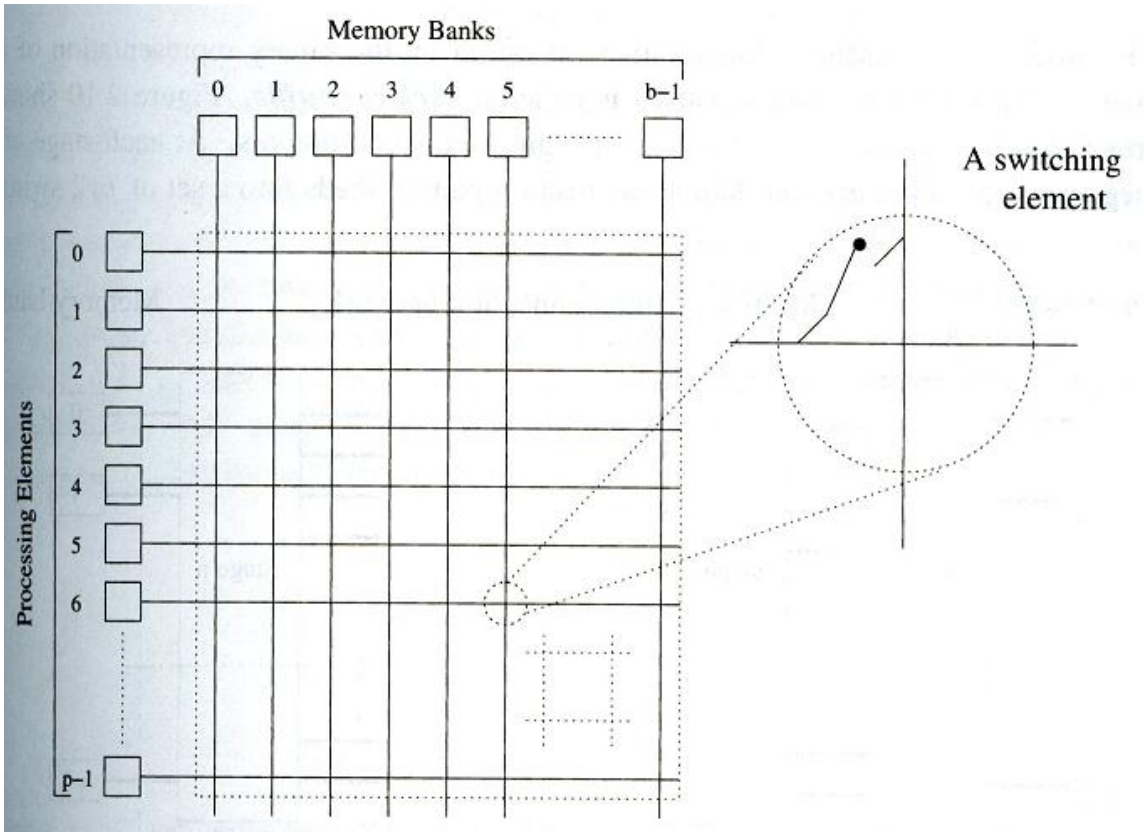


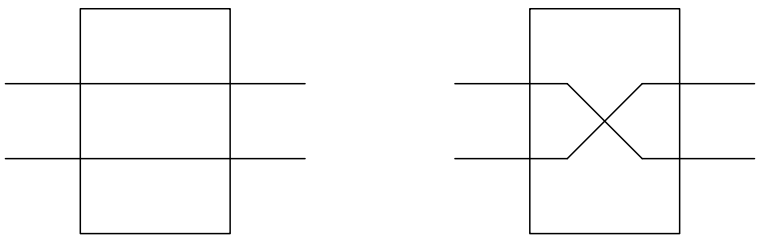
Figure 3.11- A cross-bar network with p processing elements and b memory banks [Gra03]

The Omega Network

When the crossbar has the disadvantage of becoming overly expensive to implement as it scales up, an alternative is to consider the use of multi-stage interconnected networks. In particular, a commonly used multistage connection network is the *omega network*. This network consists of log p stages, where p is the number of processing nodes and also the number of memory banks. Each stage of the network consists of an interconnection pattern that connects p inputs and p outputs; where a communication link exists between input i and output j if the following is true [Gra03]:

$$j = \begin{cases} 2i, & 0 \leq i \leq \frac{p}{2} - 1 \\ 2i + 1 - p, & \frac{p}{2} \leq i \leq p - 1 \end{cases} \tag{3.1}$$

Furthermore, at each stage of the omega network, a perfect shuffle interconnection pattern feeds into a set of $\frac{p}{2}$ switching nodes, where each switch is in one of two connection modes: pass-through or crossover (as shown in Figure 3.12).



(a) (b)

Figure 3.12 Illustration of (a) Pass-through and (b) crossover connection modes [Gra03]

In the pass-through connection mode the inputs are sent straight through to the outputs; whereas, in the crossover mode, the inputs to the switching node are crossed over and then sent out to the outputs. An example of an 8 x 8 omega network is shown in *Figure 3.13* below.

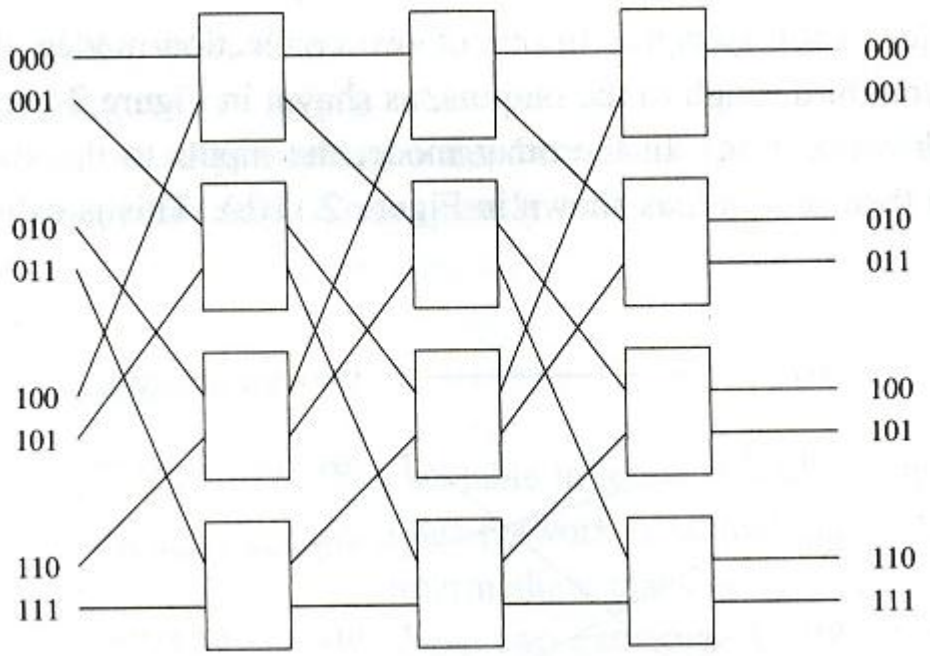


Figure 3.13 - An 8 x 8 Omega Network [Gra03]

In order to route data to and from the processors in an omega network, a routing scheme must be used. Let *s* be the binary representation of a processor that needs to write some data into memory bank *t*. The data traverses the link to the first switching node. Then, if the most significant bits of the s and t are the same, then the data is routed in pass-through fashion. However, if the bits are different, then the data is routed through in cross-over mode. Such scheme is repeated at each stage using the next most significant bit until the data has reached its destination, traversing *log p* stages while using *log p* bits in the binary representations of *s* and *t* [Gra03].

Thus, in an omega network, access to a memory bank by a processor may prevent access to another memory bank by another processor, which makes the omega network a *blocking* network.

3.4 – COMMUNICATION COST IN PARALLEL IMPLEMENTATIONS

Communication among processing nodes in parallel architecture during the execution of a program is one of the factors that can contribute a significant amount of overhead and decrease the performance of such system. Furthermore, message passing is one of the most common methods used to provide an avenue for communication among processing nodes. What constitutes a message may vary from system to system but, in general, it consists of a header, the data to be sent or received, a trailer and a corresponding error correction code.

When sending a message from one processing node to another, the time taken to prepare the message for transmission as well as the time taken for the message to traverse from its source to its destination must be considred. In particular, there are three parameters that are used to describe the total time taken to transmit a message: Startup time (*ts*), per-hop time (*th*) and the per-word transfer time (*tw*).

The *startup time* is accounted for only once for every single message transfer and it is the time incurred at both the sending and receiving ends in preparing the message for transmission and running the routing algorithm (if necessary). The *per-hop time* (also called node latency) is the time it takes the header of a message to travel from one node to the next one in its path to its destination. Lastly, the per-word transfer time is the time it takes each word of the message to travel through a communication link.

If the link has a channel bandwidth of *w* words per second, then each word takes $t_w = \frac{1}{r}$ to traverse the link.

By keeping these three parameters that contribute to communication latency in mind, measures can be taken to attempt to minimize the effect of each one of the parameters. Such measures may include: Aggregating small messages into a single large message in order to avoid incurring multiple startup latency, minimizing the volume of data communicated as much as possible in order to reduce the overhead paid in terms of per-word transfer time, or minimizing the number of hops a message must make prior to arriving to its destination. However, the per-hop time may most times be negligible given that it is usually dominated by either the startup time (when multiple small messages are sent) or the per-word time (when large messages are sent). Thus, the per-hop time can be ignored, without lack of accuracy, only for cases in which it is expected to be dominated by the other two latencies (i.e. for uncongested and well-mapped networks), yielding a simpler communication cost model:

$$t_{comm} = t_s + t_w m. \tag{3.2}$$

Chapter 4: Parallel Hardware Processing for Simulating Artificial Neural Networks Training

4.1 – Performance Metrics

When analyzing or comparing the speedup of a neural network simulation performed on a parallel architecture to its sequential counterpart or other parallel implementations, the performance of such implementation must be characterized in terms of throughput and latency [Gam96]. To this end, performance must be characterized in clearly-defined absolute metrics, as opposed to the utilization of theoretical FLOPS. This is why; two metrics are the most commonly used in the literature: Million Connection Updates Per Second (MCUPS) and the Connections Per Second (or CPS) [Haz04]. The Connection Updates per Second accounts for the number of weights updated per second; whereas, Connections per Second describe the number of weights multiplication in the forward pass per second. However, both of these metrics can sometimes be misleading as they are sensitive to multiple factors, such as the internal architecture of the machine on which training is performed or memory limitations [Sun98], this is why some other authors in literature (such as [Cro94]) prefer to present the elapsed time during the entire training process as well.

4.2 – Parallel Implementations of Backpropagation Training

Out of the many artificial neural networks training algorithms, the Backpropagation (BP) training algorithm has found wide popularity in recent literature of this area of study [Tor98]. In particular,

parallel implementations of BP training are of special interest given the big need for speed up and the potential for parallelization they provide; and therefore, a detailed overview of these implementations is covered in this section. Before a detailed explanation of these implementations is presented, however, it is convenient to review the basic terminology used in the study of parallel BP implementations.

- **Training Set**: Each application or problem that is to be implemented on an artificial neural network has, at its core, a set of training patterns each consisting of an input vector and its corresponding target output vector. Heuristically, a good (or well-behaved) training set must be well-balanced in the sense that it must contain a number of patterns that fairly represent each class in the problem (using the example of a classification implementation). Additionally a well-behaved training set must also be properly "shuffled" when presented to the network, such that all training patterns for a particular class are not presented all at once and never presented again.
- **Network Size**: A Backpropagation trained network usually consists of N_i input units, N_h hidden units, and N_O output units; Thus, the notation $N_i \; X \; N_h \; X \; N_o$ is used as a shorthand to describe the network.
- **Epoch (iteration)**: Denotes a single presentation of the entire training set.
- **Weight Update Approaches**: Depending on how far along in the algorithm the weights are updated, a ANN can be trained using one of three approaches:
    - *Learning by pattern (lbp)*: In this approach, the weights are updated after each training pattern is presented.
    - *Learning by block (lbb)*: Weights are updated after a subset of the training pattern had been presented.
    - *Learning by epoch (lbe)*: Weights are updated after all patterns have been presented (after every epoch).

4.3 – PARALLEL BACKPROPAGATION TRAINING APPROACHES

Neural networks, by nature, possess a high degree of parallelism which stems from both: the architecture of the network and the way in which the network is trained. Thus, four different styles of parallelism can be identified as possible avenues for implementation: *Training session parallelism*, *training set parallelism*, *pipelining* and *node parallelism*.

Training set parallelism (also called *Data Parallelization*) consists on storing an entire copy of the network in all the Processing Elements (PE's) and performing Back Propagation training using different slices of the entire data set to train each PE as shown in Figure 4.1. After every session, the resulting weight updates from all the individual PE's are collected and an average is calculated by a Master node. It may also be the case that from all the resulting weights collected, the one with the lowest error achieved is chosen by the master node. The master node then broadcasts the calculated average, or chosen set of weights, to the rest of the PE's, which use the broadcast set of weights as a starting point for the next training session. This cycle is repeated until convergence to a desirable or acceptable error is reached. The advantages of this approach are best noticed when using a learn-by-block or learn-by-epoch weight update approach.

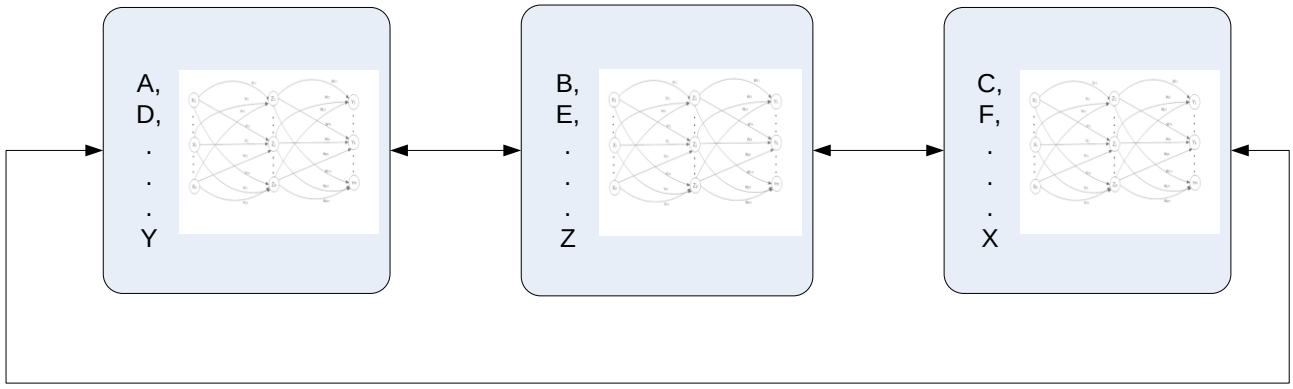


Figure 4. 1 Training set parallelism for an application that requires learning the English alphabet [Sun98].

Training session parallelism is similar to training set parallelism in the sense that every processing element has an exact copy of the network stored in memory. However, it varies in that, rather than slicing the dataset and feeding each PE with a different portion of the same dataset, all PE's are trained using the same dataset but with different initial conditions (i.e. Different initial weight matrices).

Pipeline parallelism consists on performing either different stages of the Backpropagation training process (such as the feedforward and Backpropagation phases) on different PE's or computing the different weight layers on different PE's, as shown in Figure 4.2. In this scheme, for example, the output values may be computed by a set of PE's while another set is processing the next input vector and waiting from the "output" PE's to communicate error values from the previous iteration. One of the limitations posed by the use of this approach is that it requires a delayed weight update scheme (in other words, learning by epoch or learning by batch is required).

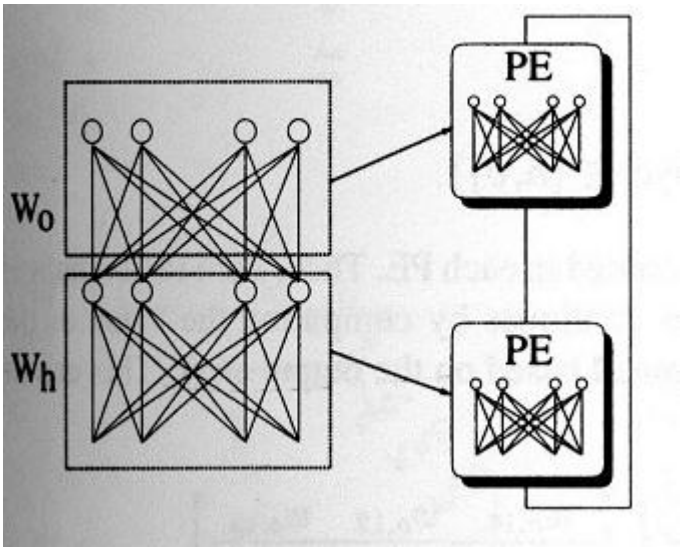


Figure 4. 2 – Weight layers pipelining parallelism [Sun98.]

Taking the case in which the architecture of the network is considered to implement a parallel training algorithm (*Node or Neuron Parallelism*), the approach is to evenly distribute the neurons of the network so as to map them accordingly to each processor in the cluster, such that (ideally) each neuron of the neural net is mapped to a single processor in the cluster in a one-to-one basis. In most cases, however, when the number of neurons in the network surpasses that of the available processes in the cluster, the neurons are divided as evenly as possible and distributed among the processes in the cluster; such that two or more neurons may be mapped to a single process.

One of the most common ways to perform neuron parallelism is by using a technique called *vertical slicing*, where all incoming weights to one hidden and one output neuron are mapped to each PE. That is, each PE stores all the incoming weights to the neuron assigned to that PE [Sun98], as shown in Figure 4.3. In this fashion, the cluster "acts" as a physical implementation of the network itself and the computations necessary to carry the feedforward propagation of input patterns and the adjusting of the interconnection weights are performed by each process accordingly, resembling the operation of the actual neural network training.

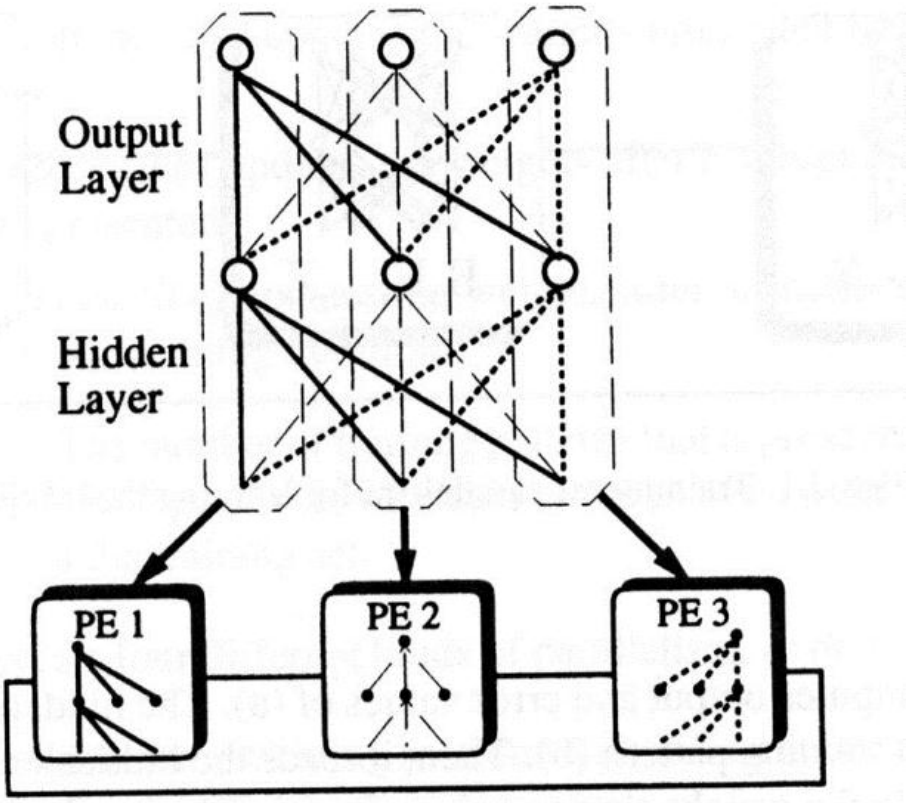


Figure 4. 3.- Neuron parallelism of a 3-layered artificial neural network [Sun98.]

From these three, fundamentally different approaches, it is the first one – which is referred to as training set parallelism [Sun98], or data slicing [Haz04] – that is further considered in this study as it allows parallelization of the training process with relatively minor modifications to the sequential (one processing element) implementation. At the same time it provides opportunity for investigation on the improvement of accuracy and performance of the trained ANN by applying multiple variants of the weight update algorithm that can only be achieved in a parallel implementation.

Chapter 5 - Approach and Results Evaluation

5.1 – INTRODUCTION

As explained in Chapter 4 ("Parallel and Concurrent Computing") of this paper, multiple approaches have been considered for going about performing a parallel implementation of Backpropagation training. Among the most popular of these, the approach where the inherent parallelization of an ANN is exploited to map the multiple layers, links, and neurons onto massively parallel architectures in order to parallelize the network itself onto a physical parallel machine or cluster. At the same time a second major, and strategically different, approach has been considered as well in which an entire instance of the network is kept at multiple processing elements (PE's) and each one trains their network using different portions (or sub-blocks) of the entire training dataset in parallel. This research focuses on the latter approach and takes a look at the effects of parallelization on the improvement in average error performance, speed-up of the training process, efficiency, and accuracy of the trained network.

This chapter presents a detailed explanation of the approach taken by this study to evaluate a parallel implementation of the Backpropagation training algorithm. The implementation considered consisted in a training set parallelism approach [Haz04, Sun98, Hwa99] which was combined with the two different learning methodologies (learn by pattern and learn by epoch), each one using two different algorithms for weight update. A total of 4 different implementations were simulated using four datasets belonging to different benchmark problems. Furthermore, each implementation was evaluated by varying the number of processing elements, learning rate, acceleration and number of neurons in the hidden layer.

5.2 - PARALLEL IMPLEMENTATION FOR THE SIMULATION OF BP TRAINING

The present study focuses in the parallel BP training implementation in which the processing nodes used are interconnected using the star-connected topology (explained in Chapter 4: Parallel And Concurrent Computing), where one of the Processing Elements (PE) is designated the role of HOST and the rest of the nodes are designated the role of WORKERS. Such an arrangement is illustrated in Figure 5.1 below, where the HOST is placed by itself on top and $n - 1$ WORKER nodes (where *n* is the total number of nodes) have a direct link of communication with the HOST, but do not communicate directly with each other.

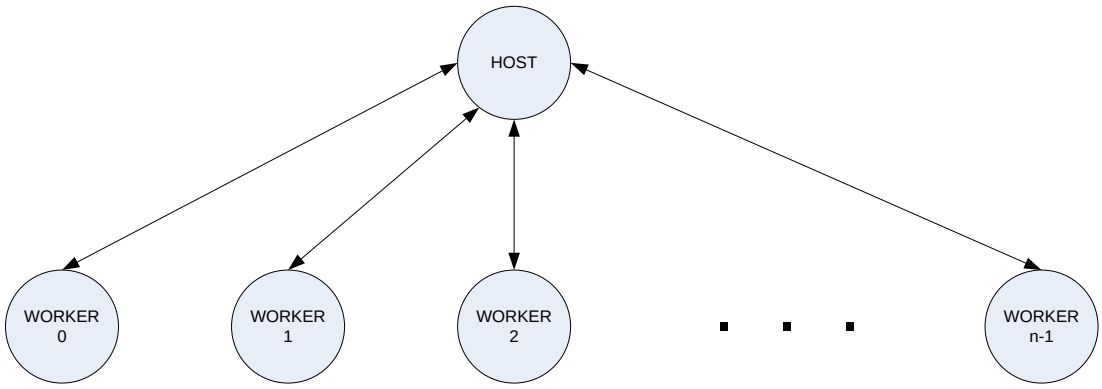


Figure 5.1.- Star-connected topology with a HOST-WORKER arrangement using *n* nodes

This parallel topology was used to simulate the training of four benchmark problems, namely: The XOR problem, a classification scheme for binary numbers, The Wine Recognition Dataset [Fra10] and the Yeast Classification Dataset [Fra10, Mad07]. The selection of these benchmark problems was done such as to have a gradual increase in length and complexity of training dataset as well as the topology of the neural network itself. The XOR problem was explained in detail in Chapter 2 (Neural Networks); therefore, there is more emphasis on explaining the other three datasets in this chapter.

5.2.1. - The Classification Scheme of Binary-Coded Hexadecimal Numbers.

The second benchmark evaluated in this study is a classification scheme for binary –coded hexadecimal numbers. Each pattern in the dataset is the 4-bit binary representation of a decimal number (0 through 9) and the output (or class) is an integer representing the number of "1's" present in this binary representation, and such relationship is further illustrated in Table 5.1. This benchmark dataset was considered for this study as it provides a "smooth" and simple transition from the XOR problem, given that it consists of only 16 training patterns and 5 classes, as well as introducing the need for having multiple neurons in the output layer of the network to enable effective classification.

Table 5.1. – Classification Scheme of Binary-Coded Hexadecimal Numbers.

Hexadecimal	Binary	Class
0	0 0 0 0	1
1	0 0 0 1	2
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	2
5	0 1 0 1	3
6	0 1 1 0	3
7	0 1 1 1	4
8	1 0 0 0	2
9	1 0 0 1	3
A	1 0 1 0	3
B	1 0 1 1	4
C	1 1 0 0	3
D	1 1 0 1	4
E	1 1 1 0	4
F	1 1 1 1	5

5.2.2 - The Wine Recognition Dataset

The Wine Recognition Dataset is a classification problem that consists of the results from chemical analysis of wines grown in the same region of Italy but derived from three different cultivars. The analysis is aimed at determining the quantities of thirteen different constituents found in the wines originating from each of the cultivars. The output of the corresponding neural network is a numeric representation of each one of the three different types of wine and it is represented using a 4-bit binary number. Therefore, the topology consists of at least 13 units at the input layer, 13 hidden units and 4 units at the output layer, or 13-13-4. This benchmark problem was chosen as, it is a "well-behaved" (in terms of having well-posed class structures) classification problem and requires the processing of 178 training (refer to table 5.2 for further explanation of the distribution of the training patterns across the three classes).

Table 5.2 – Classification Scheme for Wine Recognition Dataset

Class	Number of Instances
1	59
2	71
3	48

5.2.3. – The Yeast Classification Dataset

The Yeast Classification Dataset is the fourth benchmark dataset that was used to test and measure training time and performance of a neural network trained using the parallel implementation considered by this study. It was chosen as a means for examining a training dataset which is

substantially larger in size and consists of a classification scheme of more classes than previously considered. The entire data set consists of 1484 training patterns, each containing 9 attributes (8 floating point predictive values and 1 name) as input values and one alphanumeric output representing one of ten possible classes (refer to table 5.2 below for a list of the alphanumeric representation of each class and their number of instances.)

Table 5.3 – Classification scheme of the Yeast Classification Dataset

Class	Name	Number of Instances
1	CYT (cytosolic or cytoskeletal)	463
2	NUC (nuclear)	429
3	MIT (mitochondrial)	244
4	ME3 (membrane protein, no N-terminal signal)	163
5	ME2 (membrane protein, uncleaved signal)	51
6	ME1 (membrane protein, cleaved signal)	44
7	EXC (extracellular)	37
8	VAC (vacuolar)	30
9	POX (peroxisomal)	20
10	ERL (endoplasmic reticulum lumen)	5

5.3 – Computation and Communication Model

In the implementation used by this study, the HOST node is used primarily for data distribution, communication control and arbitration of the training process; whereas the WOKER nodes are used purely for processing of the training data using the Back Error Propagation algorithm. To this end, and using the aforementioned arrangement, the HOST node partitions the training dataset as evenly as possible and packages it into sub-blocks of training patterns which are then sent to each WORKER node. This is done such that (ideally) every WORKER node receives the same number of training

patterns. Given the particular case, however, in which the training dataset is not evenly divisible by the number of nodes, the remaining patterns left after dividing the dataset by $n - 1$ are then set aside and assigned to the last WORKER node. Thus, at the end of the data distribution process, each WORKER node has received its particular sub-block of training data as illustrated in Figure 5.2 below.

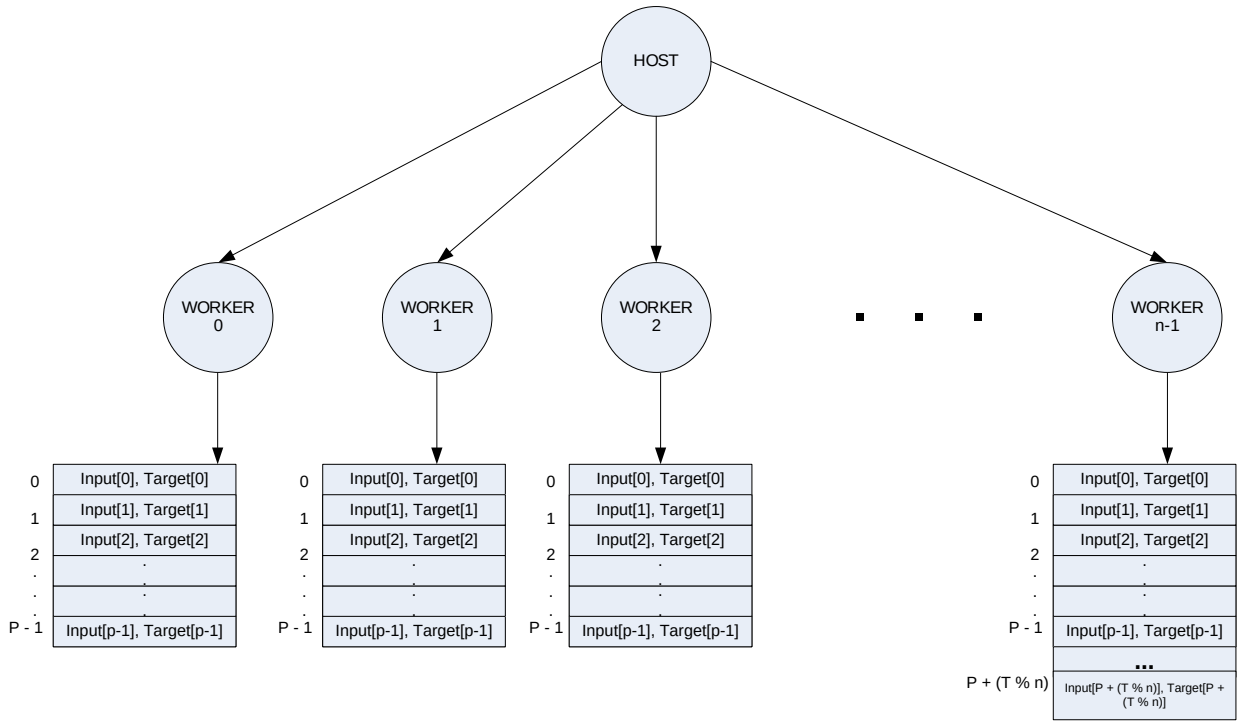


Figure 5.2.- Illustration of the data distribution in parallel implementation of BP training, where P is the partition and T the total number of training patterns.

At the completion of the training data distribution process the parallel training of the network begins at each node; where one of three different styles of training is used to update the weights. Namely the trained weights of ANN can be attained using either the *learn-by-pattern* (lbp) or *learn by epoch* (lbe) approaches. Using any of the formerly mentioned training styles, the training data is processed (passed through feedforward and Backpropagation phase) and the weights of the network are modified to reflect the response of the network to such training values after every pattern or epoch (respectively).

In any case, once a weight update must be performed, one of two possible algorithms for weight update is used. The first one is termed by author as the *host directed* weight update algorithm, in which each one of the WORKER nodes in the topology sends its calculated matrix of errors (containing the errors found at the output and hidden layers) to the HOST node. Once received by the HOST, it then computes an average of all the matrices received and sends the averaged matrix to each WORKER where it can then be used to calculate the new weights of the network. The following pseudo-code algorithm shows how the training set parallel BP is implemented using the HOST directed weight update method.

HOST	WORKER, $i = 1, 2, \ldots, n$.
`Send_block` i, `to WORKER` i	`receive_block` i
`Broadcast_weights,` w	`receive_weights` w
`For L epochs`	`For L epochs`
`For WORKER` $i = 1, \ldots, p$	`calculate error matrix for batch` i
`Receive_Error_Matrix` i	`Send_Error__Matrix` i
$E \leftarrow \frac{\sum_{i=0}^{n-1} Ei}{n-1}$	`Receive_Error_Matrix`
`Broadcast_Error_Matrix`	`Calculate_new_weights`
`End for` i	`End for L`
`End for L`	

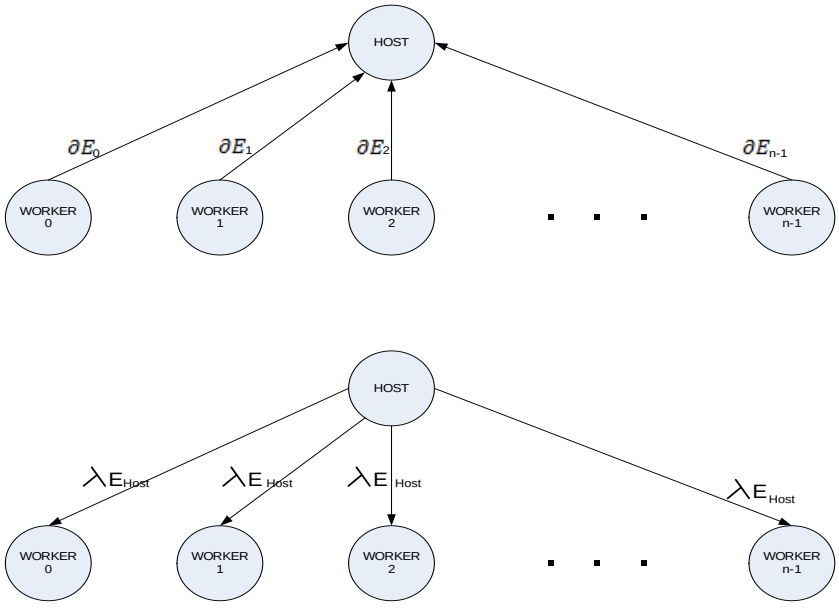


Figure 5.3.- Host Directed weight update algorithm

Alternatively, a second weight update algorithm, termed *winner takes all (WTA),* may be used where, instead of sending the error matrix, each WORKER sends its calculated system-error to the HOST, as shown in Figure 5.4 (a). Once it is received, the HOST then determines which one of the WORKER nodes achieved the lowest system-error – In the case illustrated in Figure 5.4, the WORKER with the lowest System-Error is WORKER 1 - and requests the weight matrix from that WORKER as shown in Figure5.2(b) to then broadcast this matrix to the rest of the WORKER nodes (Figure 5.4 (c)). Lastly, the WORKER nodes use the broadcasted set of weights as the starting point for the next iteration in the training process. The pseudo-code algorithm for this weight update mechanism is detailed as follows:

HOST	WORKER, $i = 1, 2, \ldots, n$.
Send_block i, to WORKER i	receive_block i
For L epochs	For L epochs
Broadcast_weights, w	receive_weights w
For WORKER $i = 1, \ldots, p$	Calculate_System_Error for block i
Receive_System_Error i	Send_System_Error p
Select_Lowest_ERROR	Receive_Request_Status
Request_Weights_From_Winner	if WINNER { Send_Weights_To_HOST }
End for i	End for L
Receive_Weights	
End for L	

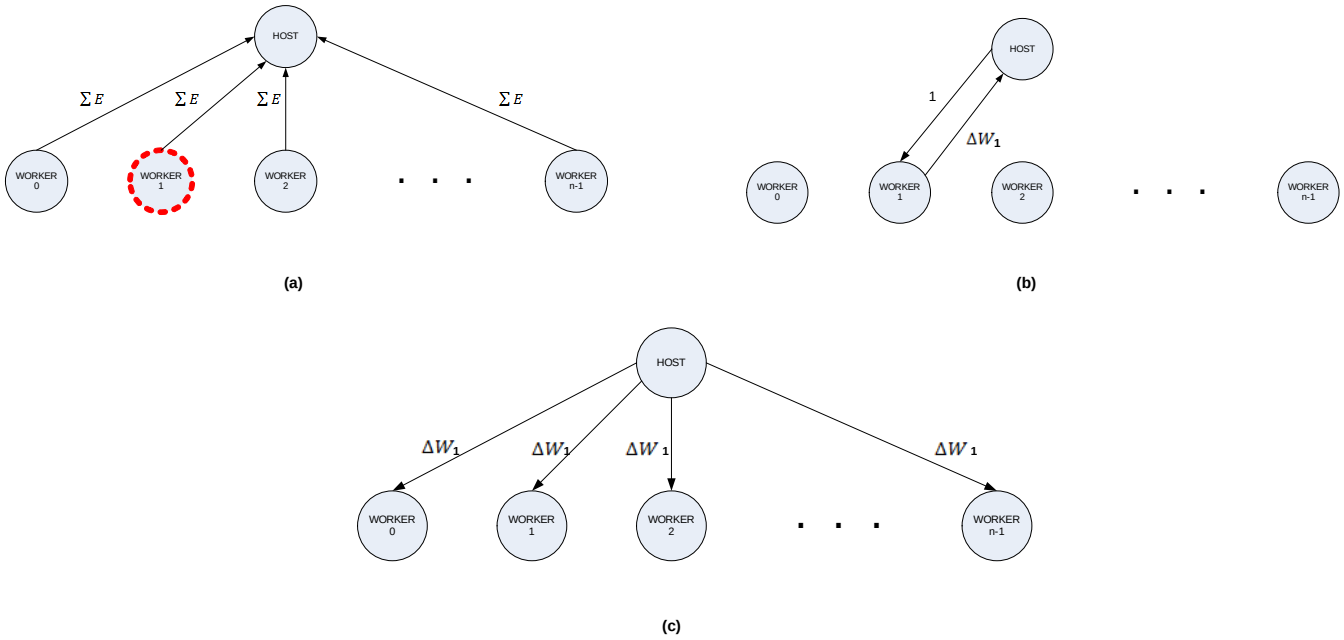


Figure 5.4.- Winner- takes-all weight update algorithm.

After every iteration of the training process is completed, the HOST node aggregates and calculates the average of the system-errors produced by each WORKER. It then uses this average to determine whether or not the error goal has been met and the training process must be stopped. In such case, the HOST sends a termination signal to each WORKER indicating that no further communication will be allowed and to execute a termination procedure. Otherwise, the HOST and WORKER nodes in the topology cycle through the aforementioned steps until either the error goal is reached or the iteration limit has been reached.

5.4 - HOST DIRECTED VS. WINNER-TAKES-ALL METHODOLOGIES

The motivation behind the implementation of the Winner-Takes-All (WTA) algorithm comes from the observation that having more than one processing element working to train a network for a particular problem may be instrumental to the goal of finding a global minimum error point in the system, which is a heuristic methodology in the process of training a neural network in general [Hwa99, Sun98.]

In the suggested parallel implementation, each WORKER has only a subset of the entire training dataset and it trains its own instance of the network according to its subset of the training patterns such that the system error, internal to each WORKER, moves toward finding the minimum error produced by its instance of the network for its sub-block of training data. At the same time, the HOST node aims to consolidate all of the internal (or local) minimum errors achieved by the WORKER nodes and arbitrate the training process, at a system-level, such that the system error advances in the direction of achieving a *global* minimum for the entire training set.

Whereas the HOST directed weight update algorithm seeks to have the HOST commanding the entire system in the direction of global minimum by taking the direction dictated by the average of the individual local errors of each WORKER, the WTA weight update algorithm gives the HOST the role of a referee that compares the performance of each WORKER node and decides which one will take the lead of moving the entire system to global convergence by sharing its weights to the entire system.

Furthermore, since all WORKER nodes hold a diversified sample of the training data – which has been shuffled by the HOST prior to distribution to the WORKER nodes – and which can contains samples from all classes to train, then it could be expected that the sum of least square error computed by any of the WORKER nodes be an optimal solution for the entire training process, as [Hwa99] explains in his own study.

Thus, at any time when a weight update is needed (either by epoch or by pattern), each WORKER node is said to compete with the rest to contribute to reaching the goal of finding an optimal solution to the problem for which the network is being trained. At the same time, the WTA weight update methodology promises a speedup over the HOST directed method as the communication overhead among the HOST and WORKERS is reduced.

This is because, in the first phase of the parallel weight update algorithm of the WTA method, the WORKERS only communicate the value of their calculated least squares error to the HOST as opposed to the error matrix – which can become quite large as the number of units in the hidden layer increases – and this may particularly become a significant burden on the training time as the number of processing elements in the topology increases. In addition, using the WTA method, the HOST only requests the weight matrix from a single WORKER, as opposed to having to collect weight matrices from all the WORKERS and then average them before re-distributing them. Therefore, the WTA weight update is expected to have an advantage over the HOST directed method in regards to average training time, as it reduces the communication overhead.

5.5 - METRICS USED FOR RESULTS EVALUATION

In order to be able to implement either of the training algorithms mentioned above, the adoption of a message passing communication model must be considered. To this end, this study made use of the Message Passing Interface (MPI) library of the C programming language, which provides the necessary functions for the sending, receiving, broadcasting and gathering (among other functions) of network-level messages among the Processing Elements comprising the topology.

Using the Send and Receive mechanisms provided by MPI, the broadcasting of the weight matrix function of the HOST can be lumped together with the receiving of this same matrix at the WORKERS side into one task and a communication model can be set up in term of the message size being communicated as well as the number of processors encompassing the topology [Haz04]. Therefore, considering a general communication model – which depends on the number of processors, the type of topology, and the message size – can be derived using the concepts explained in chapter 3. Such a model for this implementation is presented in equation 5.1

$$T_{comm}(n, W) = t_{setup} + t_{pack}(W) + t_{comm}(n, W) \quad (5.1)$$

Where t_{setup} is defined as the time needed by a processing element (such as the HOST) to initialize its data structures and prepare for communication (i.e. in the case of the HOST, this would be the time required to initialize its training data structures from an external source and establish the partition size that will be used to construct the blocks that will be communicated to the WORKERS), t_{pack} is the time needed to prepare the communication buffer with the data that will be communicated (in the case of the HOST, this is the time taken to load the communication buffer with each partition of the training data block sent to each WORKER), t_{comm} is the time taken by one byte to travel from one processing node to another and W represents the total number of weights in a particular neural network.

Additionally, taking into consideration that the training process also consists of periods of computation that occur in between communication steps, the definition of a computational model is required in order to be able to make a more accurate prediction of the time it will take to train a neural network for a particular benchmark problem.

In a single-processor (or Sequential) implementation of the BP training algorithm, the total number of instructions executed can be divided into two extrapolated groups: those that are performed once every pattern (referred to as t_{pat}) and those that are performed once every epoch (referred to as

t_{epoch}, both of which are functions of the number of weights W in the network. Thus, the computation model for a single processor implementation is the one described in equation 5.2.

$$T_{comp}(1, W) = P t_{pat}(W) + t_{epoch}(W) \tag{5.2}$$

where P is the total number of training patterns for the benchmark problem to be trained.

The computation and communication models, described above, can be combined in order to assess a predicted performance, in terms of training time and MCUPS, for the implementation evaluated in this study. Using equations 5.1 and 5.2 a total predicted time can be obtained as follows [Haz04]:

$$T_{pred}(P, n, W) = T_{comp}(n, W) + T_{comm}(n, W)$$

$$= \frac{P}{n} t_{pat} W + t_{epoch} W + t_{setup} + t_{pack} W + t_{comm} W * n \tag{5.3}$$

Furthermore, another commonly used metric used to judge the performance of a parallel BP training implementation is the observed *Speed Up* of the time taken by the implementation to train a particular neural net using a parallel implementation when compared with the sequential counterpart [Haz04, Sun98]. Such a measurement can be calculated by using the above equations describing the training time in order to evaluate the ratio of $T(P, 1, W)$, the sequential training time (i.e. using only one processing element,) over $T(P, n, W)$, the training time when using *n* processing elements, as described by equation 5.4.

$$S(n) = \frac{T(P,1,W)}{T(P,n,W)} \tag{5.4}$$

Moreover, normalizing the speed up with respect to the number of processors produces a second performance measure denominated as *Efficiency* by [Haz04] which is described by equation 5.5. An optimal parallel architecture will ideally produce a unit-valued efficiency regardless of the number of processing elements [Haz04].

$$E(n) = \frac{S(n)}{n} \tag{5.5}$$

Both of these metrics were utilized to qualify the performance of the parallel implementation used in this study and the results of such metrics are presented below.

5.6 - MEASUREMENT PROCESS AND RESULTS

The parallel BP training simulations were implemented on a 40-node Beowulf computer cluster that consists of one front-end node and 20 rack-mounted computers that act as the computing nodes. Each computing node has two 3.06 GHz Pentium 4 XEON CPU's (for a total of 40-CPU's), 4 GB of memory (2 GB per CPU) and 80 GB of hard disk [Dcl05].

Simulations were performed by using the XOR, BCHD and WINE benchmarks using both, the WTA and HOST Directed, weight update methodologies. Measurements for the average least square error, accuracy and training time were gathered for a set of learning rate and acceleration parameter combinations which spanned from an initial value of 1 and had gradual increments up to the value of 9 for both parameters in order to produce a total of 81 readings for the aforementioned measurements.

The training process was implemented using a neural network topology of $N_i \times N_i \times N_o$, where i is the number of input units and o is the number of output units (refer to table 5.1 for a list of input and output unit values for each benchmark) in the input and output layers of the network, and where these values were determined based on the number of input parameters and number of classes to be trained, respectively. Moreover, the binary sigmoid function was chosen as the activation function for the neurons in the network and, therefore, each dataset was normalized (as a heuristic measure) to fit this activation function. Training was performed using a limit of 3000 epochs for the XOR and BCHD benchmark problems, 100 epochs for the Wine Names Classification problem and a least squares error goal of 2%.

Table 5.4 – Number of units present at the input and output layer for each benchmark considered.

Benchmark	No. of Input Units	No. of Output Units
XOR	2	1
BCHD	4	4
WINE	11	3

Once the results from the 81 measurement readings were obtained, an average of each metric was calculated and compared with those achieved for the same algorithm implementation of each benchmark using different number of processing units.

A noteworthy caveat to keep in mind is that the points of comparison that were considered for each benchmark is limited by the number of training patterns available in the corresponding training set as well as the number of WORKER nodes in the topology (i.e. The XOR benchmark has an upper bound limit of 4 WORKERS as it consists of 4 training patterns). This is because once the number of WORKERS is equal to the number of training patterns then each worker will get exactly one training pattern, based on the following equation.

$$Partition = \left\lfloor \frac{n}{P-1} \right\rfloor \tag{5.6}$$

Where n is the total number of training patterns for a particular benchmark problem and p is the number of processing elements in a particular parallel computing topology. Beyond this point (when the number of WORKERS has surpassed the number of training patterns), the training set cannot be further divided and there will be WORKERS that are not assigned with any training patterns. Hence, increasing the number of WORKERS after this point has no effect in the performance of the system or the speedup of the training process.

For each benchmark considered in this study, the above mentioned process was applied using both the learn-by-pattern and learn-by-epoch training methodologies individually while, using them in combination with the WTA and HOST weight update algorithms. Thus, a total of four training methodologies were evaluated for each benchmark problem, namely: WTA By Patterns, HOST-directed By Patterns, WTA By Epochs and HOST directed By Epochs. The results from each training methodology were compared to those obtained from the other three as well as with the sequential version of the BP training algorithm for the achieved average error, average speed up (against sequential algorithm), efficiency and the highest accuracy obtained.

5.6.1 – Achieved Average Error on Parallel Implementation

The results obtained for the measurements of the average least squares error across all of the benchmarks considered for this study are hereby presented. Each of the four training methods described in the previous section by varying the number of processing elements (PE's) used for the parallel implementation and maintaining the rest of the training parameters relatively equal (i.e. Error Goal, Epoch Limit, learning rate and acceleration) for each benchmark used.

Table 5.5 presents the average error obtained using the sequential version of the algorithm (i.e. only one processing element). The average value presented was calculated using the resulting error obtained from each variation of learning rate and acceleration (a total of 81 as described in the previous section) training variables; an example of these results can be found in appendix A. From table 5.5, it can be observed that, for every benchmark used, the best average error achieved occurs when the network is trained by updating the weights matrix in a by-patterns fashion. This is expected since the network's weights are better fine-tuned to produce a correct result each time the network is presented with a new training pattern.

Table 5.5 – Measured values for the average error obtained for the sequential BP training algorithm.

Sequential Average Error		
	By patterns	By Epochs
XOR	0.02015494	0.031046
BCHD	0.019936802	0.042779
WINE	0.020840086	0.162372099

The values presented in table 5.5 can then be used as the points of comparison with which to qualify the performance of the parallel implementation for achieving or not a good average error. The point of this comparison is to see whether or not implementing the training process on a parallel architecture can lead to improved learning performance.

Hence, Table 5.6 presents the obtained measurements for the average error achieved when the neural network was trained for the XOR benchmark using both, the learn-by-epoch (lbe) and learn-by-pattern (lbp), approaches in combination with the two weight update methodologies (HOST-directed and WTA). Alternatively, this same data is presented in Figure 5.5 in a graphical representation that illustrates the gradual change in average error as the number of PE's is increased from 2 to 5 and it is compared with the corresponding average error values achieved using the sequential algorithm.

Table 5.6 – Measurements obtained for average error for the XOR benchmark across a varying number of processing elements and using the learn-by-epochs training method.

No. of PE's	XOR - Average Error			
	Learn-by-pattern		Learn-by-epoch	
	HOST	WTA	HOST	WTA
2	0.166415963	0.123771802	0.283903691	0.271667765
3	0.020038049	0.016438185	0.012767494	0.019248741
4	0.123862309	0.062382099	0.012186469	0.121895519
5	0.019541679	0.005011667	0.01141863	0.01141863

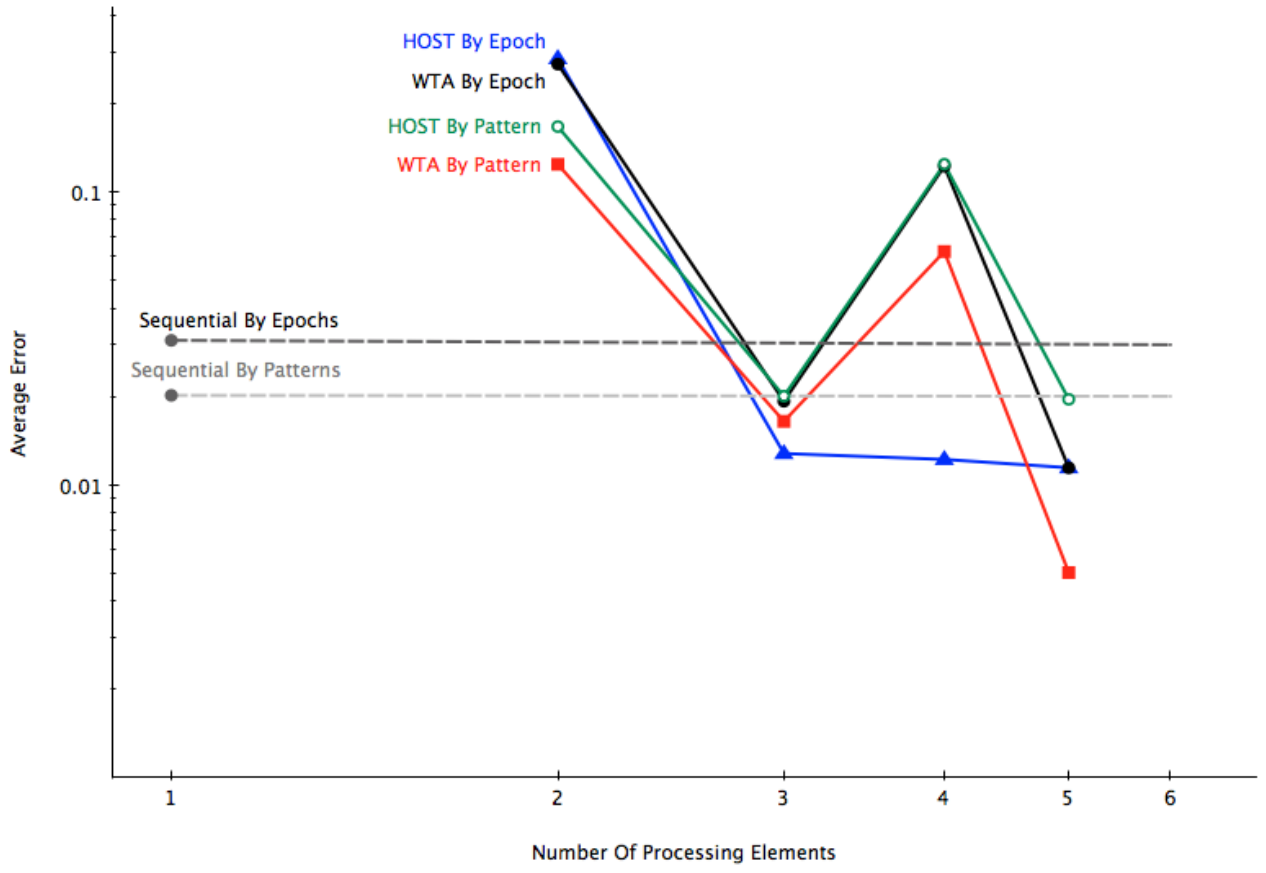


Figure 5.5. – Average Error achieved for the XOR benchmark across a varying number of processing elements and using the learn-by-epochs training method. (WTA stands for Winner Takes All, HOST for Host directed weight update.)

From Figure 5.5 it can be seen that all four of the training methods are able to take advantage of the parallel implementation to achieve better average error performance than their sequential counterpart with the exception of the HOST By Patterns method. An important trend to notice is that, as the number of processing elements is increased, the average error performance seems to follow an overall trend to perform better (i.e. achieve lower average error) than the sequential training algorithm; with the one training method that seems to outpace the rest being the WTA By Patterns. Furthermore, by examining Figure 5.5 it can be observed clearly that the particular case in which four PE's are used for the parallel implementation is the exception to the trend. This is believed to be caused by the imbalance in the distribution of training patterns as, out of the three workers in the implementation, the last worker will get assigned more training patterns than the other two and introduce a bias for the direction of the network in the error surface. The opposite is expected (and found to be true) of cases in which the training set is evenly divided among all the processors (i. using 3 and 5 PE's).

Similarly, the average error values obtained for the Binary-Coded Hexadecimal (BCHD) problem were collected. For this benchmark problem the number of training patterns in the training dataset consists of 16 different training patterns, thus, the number of PE's used for the parallel implementation was varied from 3 to 10 in order to vary the partition size of the training sub-blocks given to each the processors as dictated by the upper bound of equation 5.6. The obtained measurements for average error on the BCHD problem are presented in table 5.6 and shown plotted in a graphical representation in Figure 5.6.

Table 5.6 – Measurements obtained for average error for the BCHD benchmark across a varying number of processing elements and using the learn-by-epochs training method.

No. of PE's	BCHD - Average Error			
	Learn-by-pattern		Learn-by-epoch	
	HOST	WTA	HOST	WTA
3	0.292488569	0.810003	0.66492608	0.09829529
4	0.194959868	0.587108222	0.5856335	0.07018598
5	0.210823667	0.693112716	0.5412246	0.594490543
7	0.126778556	0.142593444	0.167011605	0.13335021
8	0.093525595	0.128878395	0.125901139	0.126647815
9	0.019967082	0.107764901	0.019394432	0.073119691
10	0.109533945	0.064666012	0.075031705	0.102244753

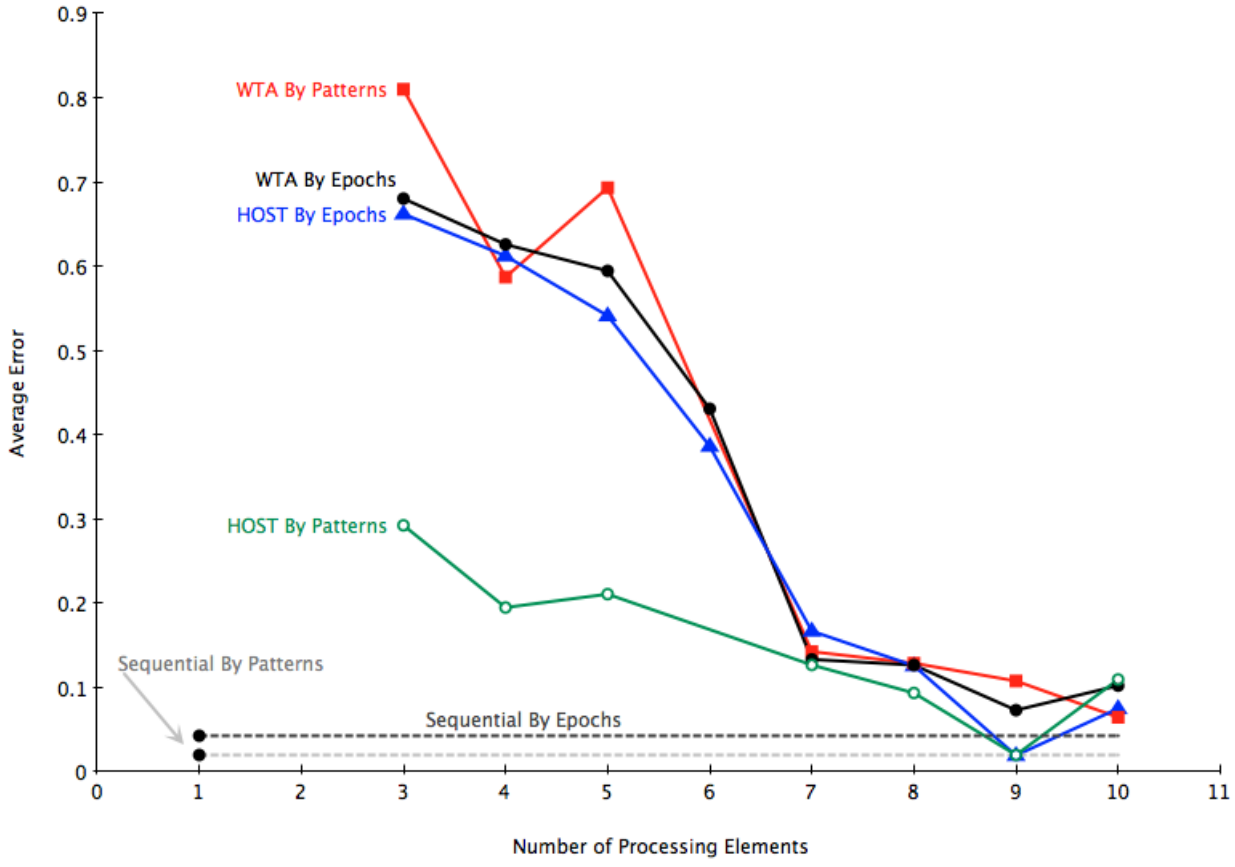


Figure 5.6 – Average Error achieved for the Binary Coded Hexadecimal (BCHD) benchmark across a varying number of processing elements and using the learn-by-epochs training method. (WTA stands for Winner Takes All.)

From Figure 5.6 it can be observed that the trend to increase error performance that was noticed in the average error measurements for the XOR problem is again manifested (and re-affirmed) in those obtained for the BCHD problem. The best average error performance achieved for this problem was found using the HOST By Patterns parallel training methodology with 9 processing elements. It can also be perceived from Figure 5.6 that this particular choice of the number of PE's produces the best average error performance for all parallel training methodologies, with the exception of WTA By Patterns. As mentioned for the XOR benchmark, this is believed to be caused by the fact that the training set is most evenly distributed for this particular choice in number of PE's used for the implementation. It is also noteworthy to mention that the WTA By Patterns was again able to achieve the best performance at the very last point of comparison (using 10 PE's).

Lastly, the Wine Names Recognition (WINE) benchmark problem was studied. In this problem, the number of training patterns greatly exceeds the number of available computing nodes in the distributed computing cluster that was used for the study.

Similarly as with the previous datasets, the average error for the Wine Recognition Dataset was calculated by using the reported error values for the 81 different variations of the learning rate and acceleration parameters. Furthermore, the average for these 81 measurements was obtained for 10 different points of comparison where the number of processing elements was gradually increased from 2 to 38 as can be seen in Table 5.7 (below).

The data acquisition process for this benchmark differed from the previous two since the Epochs limit was reduced to 100. This adversely affected the performance observed for the average error measurements in comparison to the other two benchmarks used. However, these adverse effects were counter acted by the fact that the training set for the WINE problem has a larger number of redundant training patterns, which help the network to learn faster.

The values presented in Table 5.7 have been plotted in a graph that tracks the changes in average error as the number of processing elements is increased and it is illustrated in Figure 5.7 below. From Figure 5.7, it can be seen that the average error decreasing trend observed in the previous two problems is present yet again for the WINE problem, with the WTA by Patterns method achieving the best performance in average error when using the maximum number of processing elements (i.e. 38 PE's).

Table 5.7 – Measurements obtained for average error for the WINE benchmark across a varying number of processing elements and using the learn-by-epochs training method.

No. of PE's	WINE - Average Error			
	Learn-by-pattern		Learn-by-epoch	
	HOST	WTA	HOST	WTA
2	0.260720092	5.129032258	0.50570887	0.59948558
6	1.127640676	0.590538926	0.49122199	0.545571543
10	2.009282727	0.489637309	0.46437961	0.485710575
14	3.082322208	0.547740025	0.4769627	0.506377531
18	3.770850714	0.666629222	0.44603395	0.515399037
22	5.046687727	0.62148137	0.42259058	0.436661889
26	4.799327658	0.626116519	0.41290513	0.464632111
30	0.405370909	0.489923716	0.37579846	0.387099272
34	0.420942235	0.493536432	0.32442803	0.341590012
38	8.4245678	0.23632784	0.33180106	0.288117235

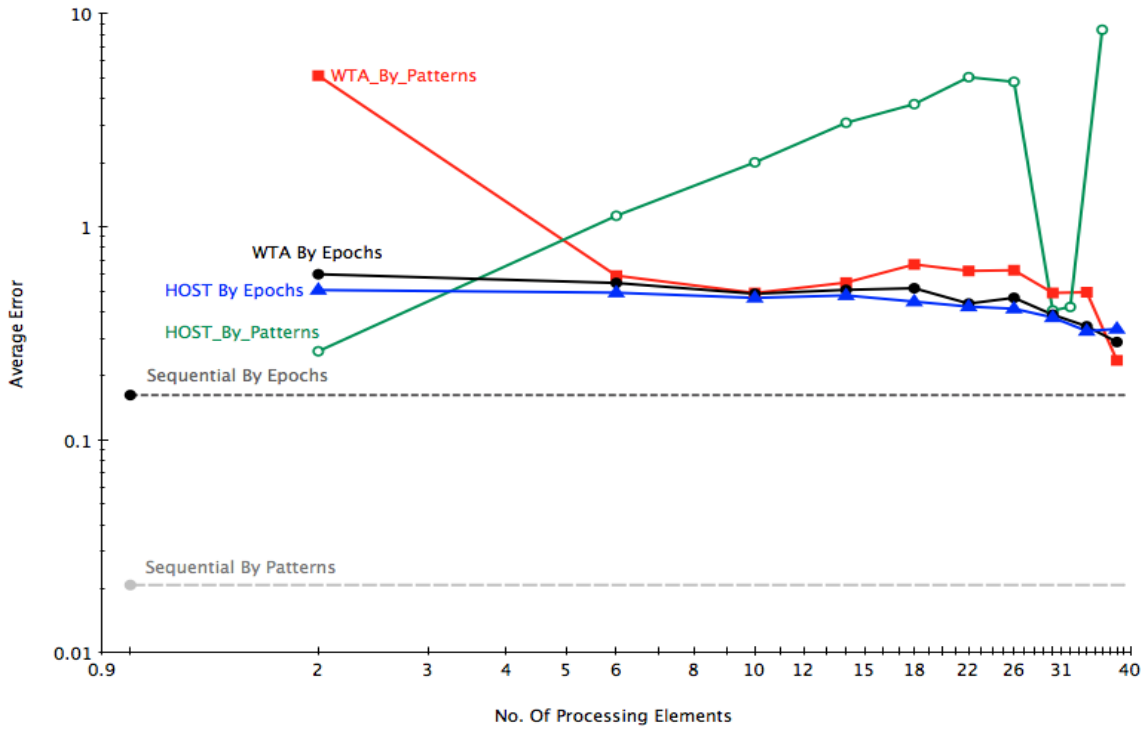


Figure 5.7 – Average Error achieved for the Wine Names Classification benchmark across a varying number of processing elements and using all four training methods.

5.6.2 –Average Training Time and Speed up

Measurements for average training speedup were obtained in much the same manner as those for the average error (described in section 5.6.1). The measured values, in seconds, for the average training time that were obtained for the sequential version of the training algorithm on each benchmark problem are presented in Table 5.8. From this table, It is noteworthy to point out that, as expected, the average training time when updating the weights matrix by epochs leads to a better performance (lower training time) than that obtained when the weights are updated after every pattern is presented.

Table 5.8 – Measured values, in seconds, for the average training time obtained for the sequential BP training algorithm.

Sequential Average Training Time (Sec.)		
	By Patterns	By Epochs
XOR	0.029385	0.00213456
BCHD	0.62041	0.272716
WINE	0.762963	0.5037037

Similarly, the training time required to train each of the three networks was measured for each of the 81 learning rate and acceleration combinations and its average was calculated. These values were then used, in conjunction with equation 5.4, to calculate the speed up achieved when compared against the corresponding average training time of the sequential implementation. The calculated average speed up values for the XOR problem for all four training methodologies are presented in Table 5.9 and the plotted graph representation of the table is presented in Figure 5.8.

Table 5.9 – Measurements obtained for the average speed up achieved for the XOR benchmark across a varying number of processing elements and using all four different training methods.

No. of PE's	XOR - Average Speed up			
	Learn-by-pattern		Learn-by-epoch	
	HOST	WTA	HOST	WTA
2	0.31048918	0.275014607	0.004369767	0.004313591
3	0.00271388	3.289237377	1.850504099	0.277930934
4	0.00021051	0.558140524	4.690594395	0.007629636
5	0.00835243	4.334816515	4.290748461	3.2901372

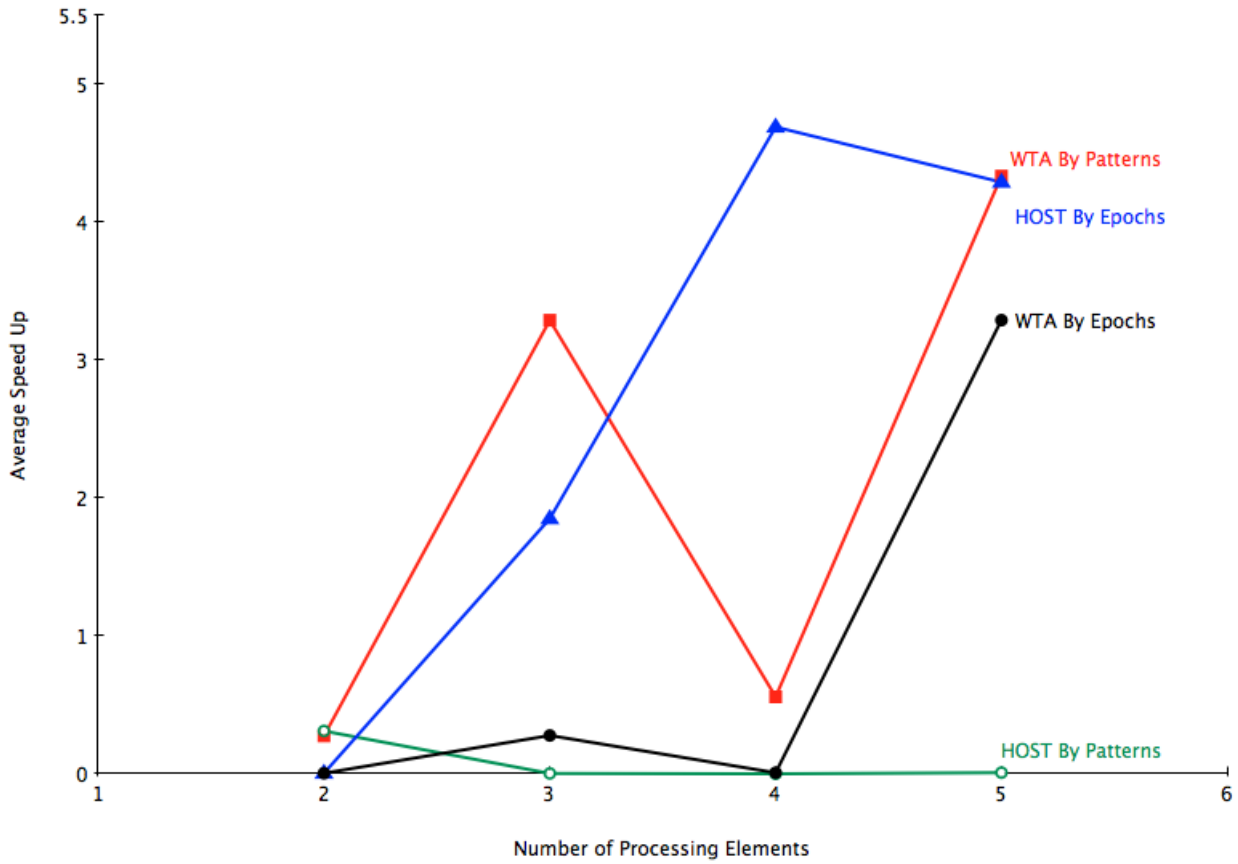


Figure 5.8 – Speed up achieved for the XOR benchmark across a varying number of processing elements and using all four training methods. (WTA stands for Winner Takes All, HOST for Host directed weight update.)

From the plotted graph present in Figure 5.8 it can be observed that the WTA algorithm achieves a superior performance than the HOST weight update method in cases where the number of worker nodes is such that the training set is better evenly divided and no worker has a bigger training set sub-block than the rest. For the particular case of the XOR problem, this occurs when the number of processing elements is 3 and 5 (i.e. when training set sub-block sizes are 2 and 1, respectively). In both of these two cases and when considering only the training by patterns scenario, it is the WTA training method that achieves the best speed up when compared to the sequential by patterns training time. However, when training by epochs was used, the HOST-directed weight update method achieved a better speed up.

Using the calculated values for speed up and applying them to equation 5.5, the efficiency of each variation in the number of processing elements can be obtained and evaluated. Hence, table 5.10 presents the calculated efficiency values when training for the XOR benchmark problem, and their corresponding plotted graph representation is shown in Figure 5.11

Table 5.10 – Measurements obtained for the average efficiency for the XOR benchmark across a varying number of processing elements.

No. of PE's	XOR - Average Efficiency			
	Learn-by-pattern		Learn-by-epoch	
	HOST	WTA	HOST	WTA
2	0.155245	0.137507	0.00218488	0.002157
3	0.000905	1.096412	0.6168347	0.092644
4	5.26E-05	0.139535	1.1726486	0.001907
5	0.00167	0.866963	0.85814969	0.658027

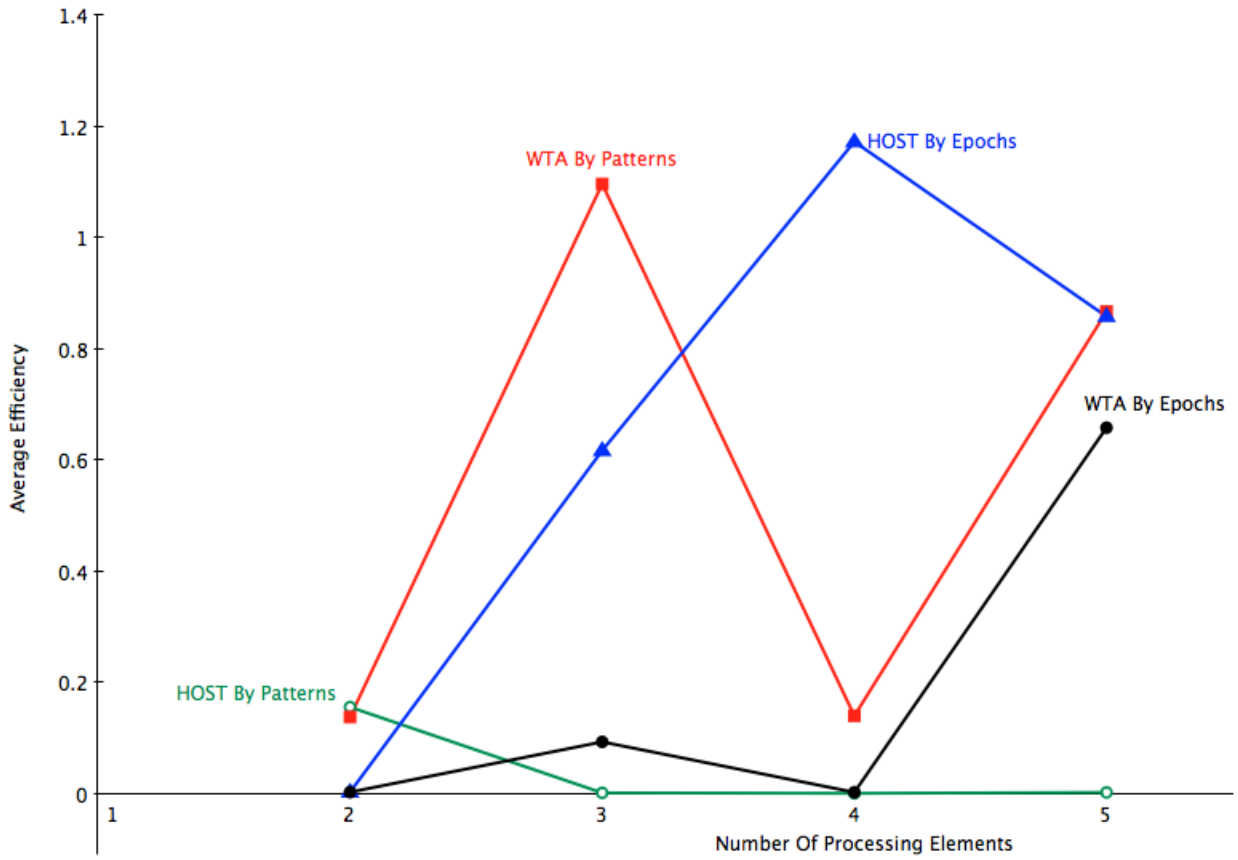


Figure 5.9 – Efficiency achieved for the XOR benchmark across a varying number of processing elements and using all four training methods. (WTA stands for Winner Takes All, HOST for Host directed weight update.)

In a similar manner as was done with the XOR benchmark problem, the values for the average speed up in training time for the BCHD problem were calculated for each of the four training methods, they are presented in Table 5.11 along with their plotted graph representation found in Figure 5.10.

Table 5.11 – Measurements obtained for the average training time achieved for the BCHD benchmark across a varying number of processing elements.

Number of PE's	BCHD - Average Speed Up			
	Learn-by-pattern		Learn-by-epoch	
	WTA	HOST	WTA	HOST
3	2.786935	0.421852	2.746524	0.038474
4	4.165309	0.04942	1.606055	0.010905
5	5.259534	0.018512	5.125554	0.006886
7	3.102763	0.008994	3.67161	0.002243
8	3.958955	0.006923	3.735291	0.002119
9	0.685496	0.108301	8.206896	0.099549
10	0.592541	0.010918	2.198324	0.007771

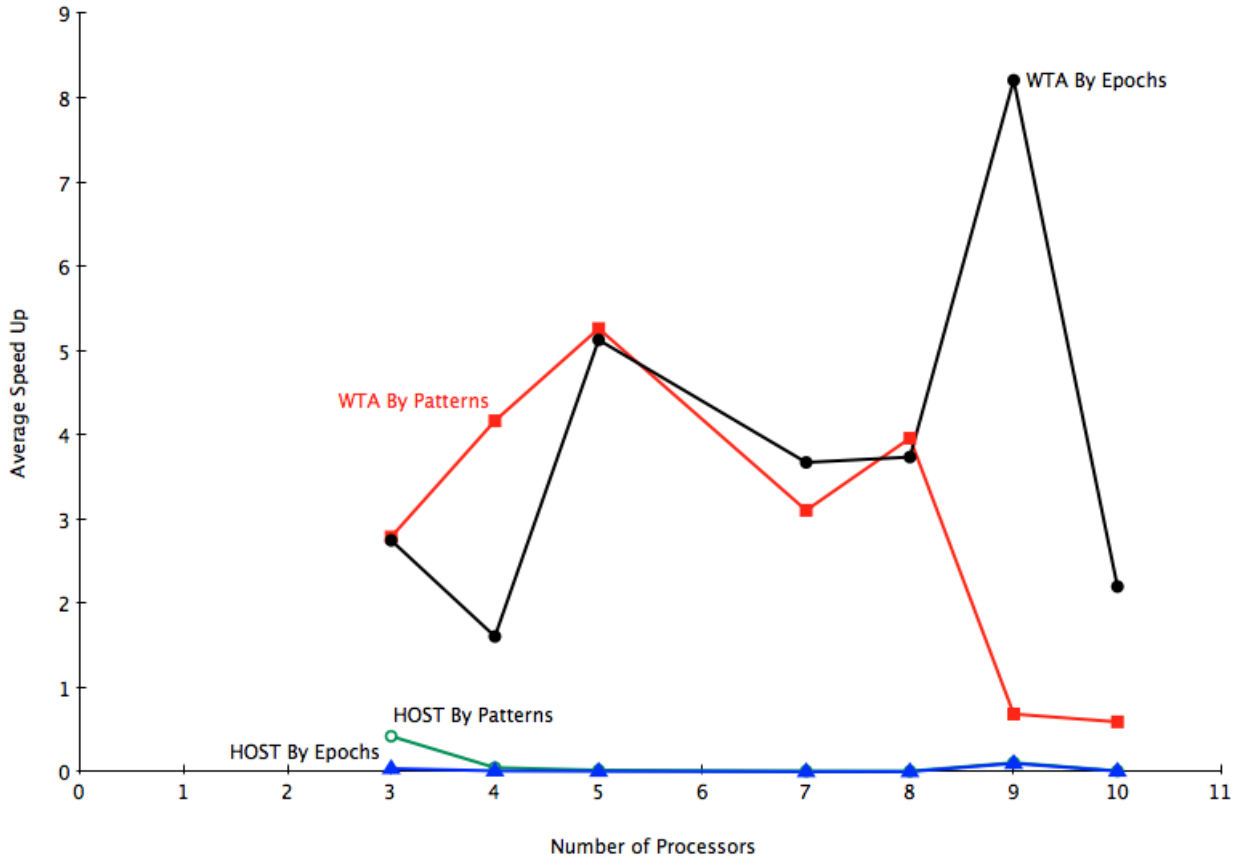


Figure 5.10 – Average Speed up achieved for the Binary Coded Hexadecimal (BCHD) benchmark across a varying number of processing elements and using all four training methods. (WTA stands for Winner Takes All).

For the BCHD benchmark problem, which contains a slightly larger training set than the XOR problem, it was found that for both training methodologies (training by patterns and by epochs) the WTA weight update method resulted in superior speed up than its HOST-directed counterpart. In particular, the WTA training by epochs achieved a speed up of 5.125554 when utilizing 5 PE's and the highest speed up of 8.206896 when the number of processing elements was increased to 9. Both of these instances occur when the training set is evenly partitioned (using equation 5.5) into sub-blocks of 4 and 2 training patterns for all workers, respectively.

As mentioned, a speed up of 5.125554 was achieved for the case in which 5 PE's were utilized for the parallel training implementation. Such an improvement in training time is attributed to memory allocation mechanisms internal to the processing elements themselves and that are inherent to their hierarchical organization of memory, such as cache line replacement [Fly95]. Given that the training dataset of the BCHD problem consists of a relatively small number of training patterns, and the nodes of the Beowulf cluster used in this study consist of dual-core processors, it is possible that the cache available to both cores becomes shared and incurs into faster communication among two cores of the same processor, which in turn, leads to greater speed up. Consequently, this to leads to an efficiency value that is greater than one as reported in Table 5.11.

From Figure 5.9, it is also observable that the HOST-directed weight update by patterns learning method is found to be the worst in terms of speed up, as it displays a tendency to remain at zero speed up regardless of any variation in the number of processing elements utilized in the parallel implementation. This confirms the author's expectation of the HOST-directed algorithm to display the worst performance of all four methods used but, in particular, this behavior is further manifested when the weight update is performed in a by patterns fashion as it incurs in more communication overhead when worker nodes communicate with the host node for weight updating. In effect, this trend

translates into very low efficiency for the HOST By Patterns training methodology (presented in Table 5.12).

Table 5.12 – Measurements obtained for the average efficiency for the BCHD benchmark across a varying number of processing elements.

No. of PE's	BCHD - Average Efficiency			
	Learn-by-pattern		Learn-by-epoch	
	WTA	HOST	WTA	HOST
3	0.928978	0.140617	0.91550808	0.012825
4	1.041327	0.012355	0.40151382	0.002726
5	1.051907	0.003702	1.02511081	0.001377
7	0.443252	0.001285	0.52451573	0.00032
8	0.494869	0.000865	0.46691139	0.000265
9	0.076166	0.012033	0.91187728	0.011061
10	0.059254	0.001092	0.21983241	0.000777

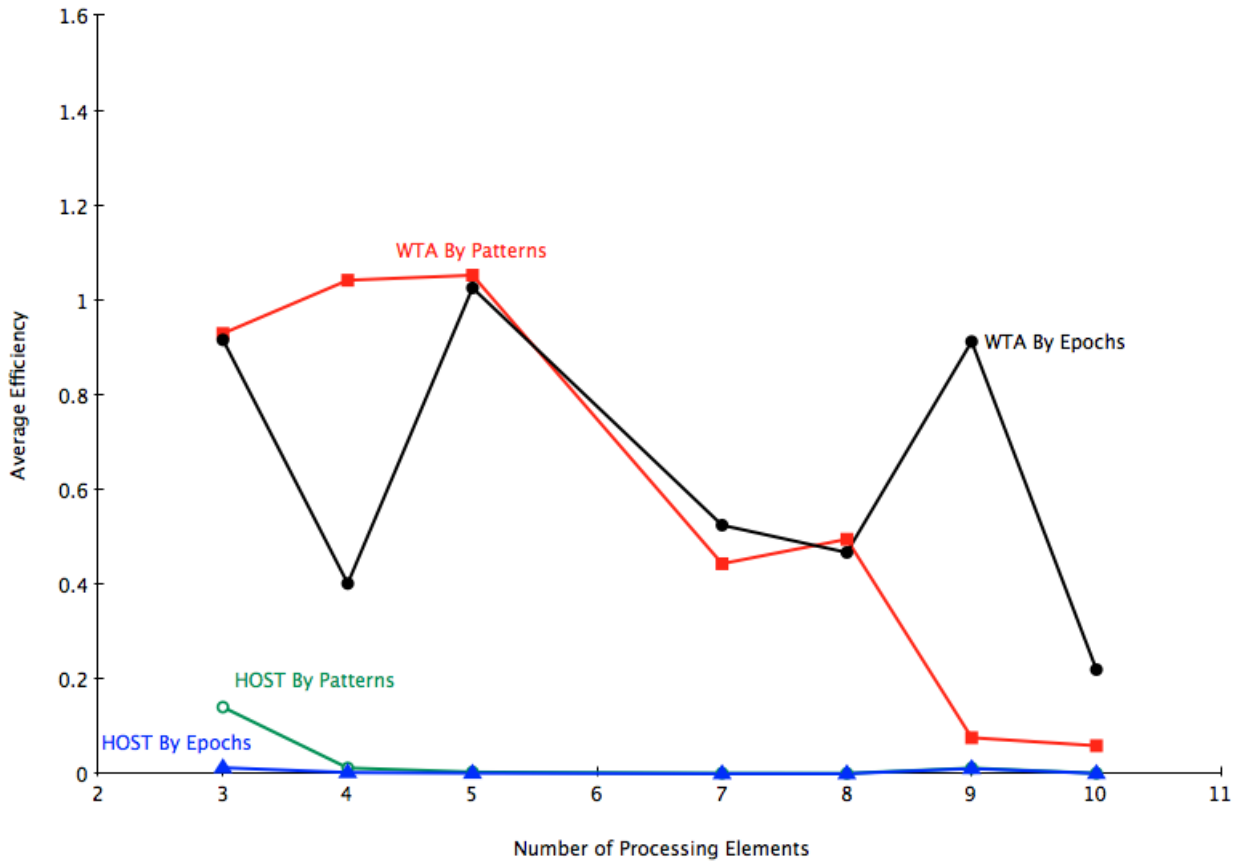


Figure 5.11 – Efficiency achieved for the BCHD benchmark across a varying number of processing elements and using all four training methods. (WTA stands for Winner Takes All, HOST for Host directed weight update.)

Measurements for training time were collected as well when training the Wine Names Classification neural network. The collected measurements were then used to calculate speed-up achieved relative to the training time taken to train the network using only one processor. The average speed up vales achieved for each of the four different training methods and across a varying number of processing elements is reported in Table 5.13, and its plotted graph representation is presented in Figure 5.12.

Table 5.13 – Measurements obtained for the average speed up achieved for the WINE benchmark across a varying number of processing elements and using all four training methods.

No. of PE's	WINE – Average Speed Up			
	Learn-by-pattern		Learn-by-epoch	
	WTA	HOST	WTA	HOST
6	1.934846941	0.676601137	**5.373155793**	4.348469821
10	2.225772814	0.379719067	4.929988499	1.877788078
14	2.263473518	0.24752862	4.220448071	0.95219107
18	2.021099362	0.202331787	3.559373624	0.470524364
22	1.96688335	0.151180934	3.154382678	0.547007877
26	1.637655259	0.158972885	2.695060094	0.394042749
30	1.493791772	1.882135461	2.249476542	0.295291215
34	1.28268536	1.812512265	1.835529205	0.204631189
38	1.374988116	0.090564048	1.541098782	0.158416402

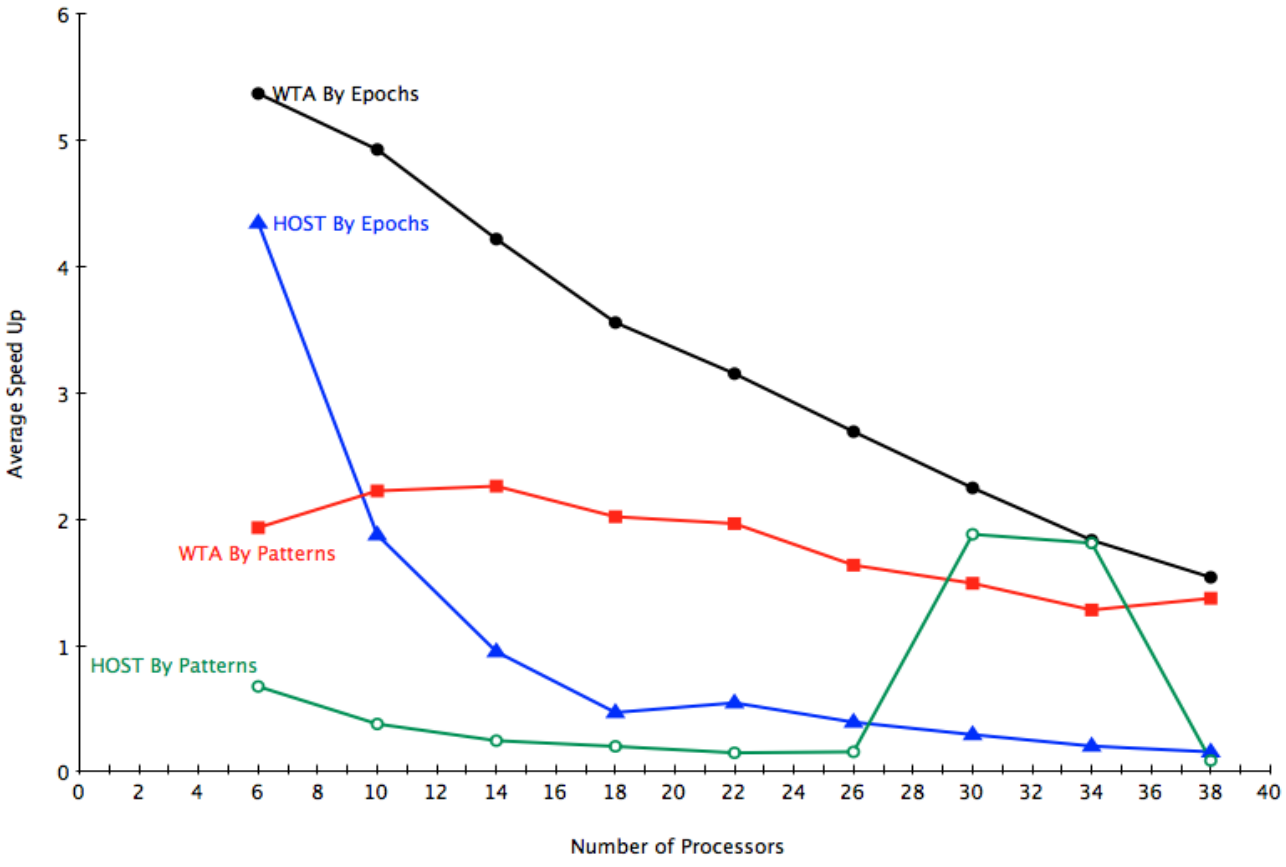


Figure 12.- Average speed up achieved for the Wine Names Classification benchmark across a varying number of processing elements and using all four training methods.

From Figure 5.12, and given the fact that the WINE problem is a considerably larger problem with respect to the others used in this study, it becomes clear that the parallel implementation under study has a tendency to produce better speed up performance when the number of processing elements is relatively small.

Just as with the previously evaluated problems, the HOST-directed weight update method proves to have the worst performance in speed up when compared to its corresponding counterpart for training by patterns and by epochs. Not only does it have the lowest speed up for cases where the number of PE's is small, but it also tends to quickly decrease and approach zero speed up as the number of PE's is increased.

Lastly, it is observed that the WTA By Epochs training method achieves the overall best speed up performance for any number of processing elements, although it too has a tendency to decrease and seems to converge at a speed up of 2 along with its by-patterns version. This trend is best appreciated when taking into consideration the calculated efficiency values which are presented in Table 5.14 and Figure 5.13, which show a clear trend for the efficiency to decrease as the number of processing elements is increased.

Table 5.14 – Measurements obtained for the average efficiency for the BCHD benchmark across a varying number of processing elements.

No. of PE's	WINE - Average Efficiency			
	Learn-by-pattern		Learn-by-epoch	
	WTA	HOST	WTA	HOST
6	0.322474	0.112767	0.89552597	0.724745
10	0.222577	0.037972	0.49299885	0.187779
14	0.161677	0.017681	0.30146058	0.068014
18	0.112283	0.011241	0.19774298	0.02614
22	0.089404	0.006872	0.14338103	0.024864
26	0.062987	0.006114	0.10365616	0.015155
30	0.049793	0.062738	0.07498255	0.009843
34	0.037726	0.053309	0.05398615	0.006019
38	0.036184	0.002383	0.04055523	0.004169

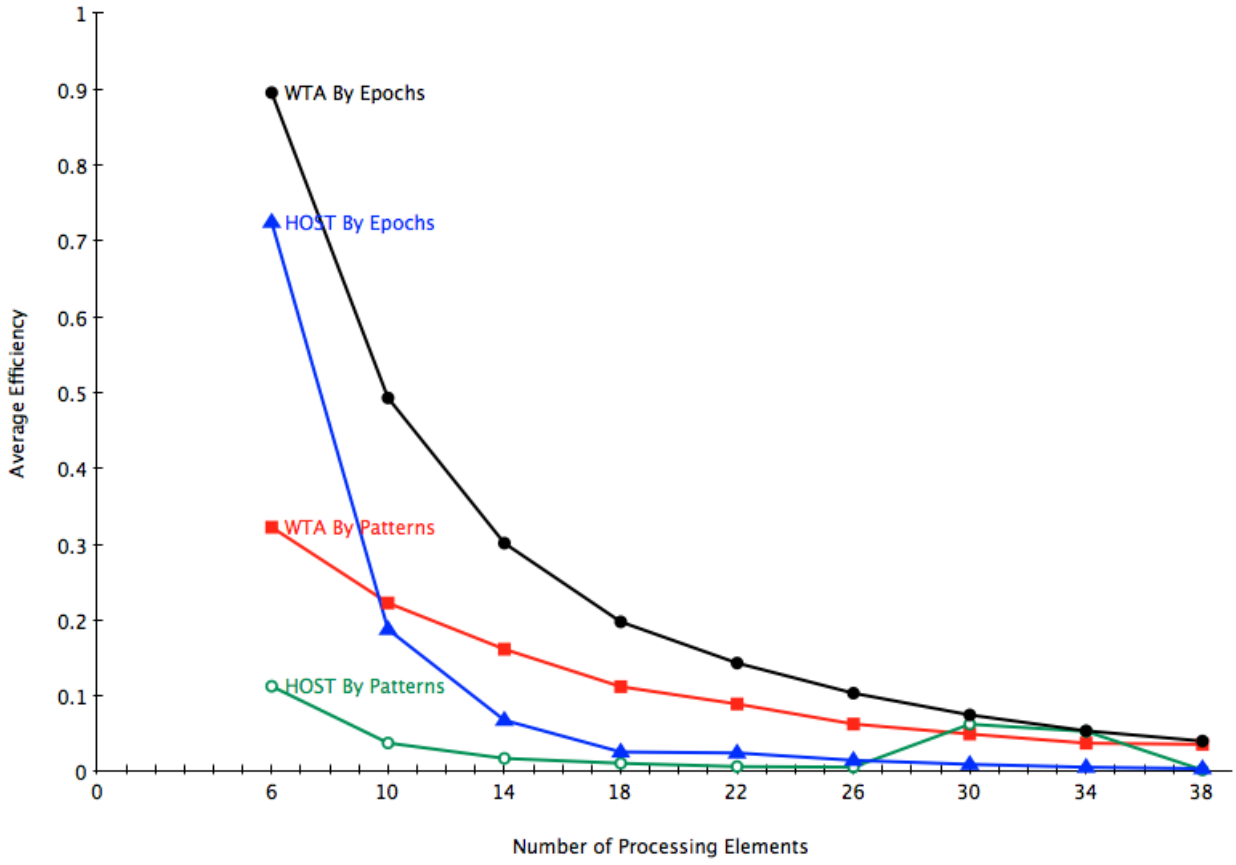


Figure 5.13 – Efficiency achieved for the WINE benchmark problemacross a varying number of processing elements and using all four training methods. (WTA stands for Winner Takes All, HOST for Host directed weight update.)

5.6.4 – Accuracy

Even when a neural network is able to reach the desired error goal, it may be the case that the accuracy of its output values when tested using patterns that the network has not previously seen is not 100% correct. This was found to be the case during the testing of the parallel implementation evaluated in this study, with the exception of the XOR benchmark problem. This is true in part because the study was not concerned with finding the optimal parameters to achieve a perfect classification as it has been mentioned before in this chapter. The XOR benchmark, in particular, is the exception because it was known by the author beforehand that the optimal number of hidden layer neurons was 2 and that was the number of hidden layer units used for this study from the outset, and even in such case, the sequential version of the algorithm was only able to achieve 75% accuracy (i.e. it was only able to learn 3 out 4 training patterns,) as it is shown in Table 5.15.

However, to the surprise of the author, it was found that, through the use of the parallel implementation (and in particular that which uses the WTA By Patterns learning method,) it was possible to successfully train the network to learn all four patterns and achieve 100% accuracy (as shown in Table 5.14).

Unfortunately, the same cannot be said of the two other benchmark problems used. In fact, a slight decrease in accuracy was observed from the use of the parallel implementation for the BCHD and the WINE benchmarks. This effect is attributed (by the author), and as described by [Nav98] to the fact that the training datasets of the BCHD and WINE benchmarks tend to favor certain classes and neglect other as can be seen in Tables 5.1 and 5.2, where it is shown that certain classes in the data set are significantly better represented than the rest (specially for the WINE benchmark).

Table 5.15 – Comparison of the Average Accuracy achieved using the Winner Takes All weight update algorithm and learning-by-epoch methodology.

Benchmark	Number of Patterns	Accuracy (Sequential)	Best Accuracy (Parallel)	Training Method	Number of PE's
XOR	4	75%	100%	WTA By Patterns	2
CLASS	16	87.50%	68.75%	WTA By Patterns	3
WINE	132	36.17%	34.78%	All but HOST By Epochs	N/A

Additionally, this adverse effect can also be attributed to the way in which weight update is performed by the Winner Takes All algorithm. This is because the number of times that each PE contributed to the weight update mechanism was tracked for the entire simulation; and it was found that, consistently, it was only a small number of processors that contributed to the weight updating process while the majority of the other processing elements had no contribution at all. An example of such report in which this effect can be clearly seen is provided in Appendix A for the reader's consideration.

Chapter 6 – Conclusions and Future Enhancements

6.1 – Summary of Results

The aim of this research is to study the benefits, drawbacks and possible avenues for improvement of the parallel implementation approach for training a neural network in which training set parallelism is used. In particular, the case in which the process of training a neural network for a given classification problem, using the Backpropagation algorithm, is parallelized by partitioning the dataset of a given benchmark problem into sub-blocks which are then sent to multiple processors – interconnected in a star topology architecture; and where each processing element has an entire instance of the neural network that is trained using only the training patterns present in the sub-block assigned to the corresponding processing element.

The study focused on three benchmark problems, namely the XOR, BCHD, Wine Names Recognition. Additionally, two different methods for parallel weight update – HOST-directed and Winner-Takes-All (WTA) – were implemented in combination with two different training methods – Learn by Pattern and Learn by Epoch. Thus, for each benchmark problem considered in this study, a neural network was trained using the following four combinations: Learn-by-pattern with HOST-directed weight update, Learn-by-pattern using WTA, Learn-by-epoch with HOST-directed, and Learn-by-epoch using WTA.

It can be seen in Figures 5.5, 5.6 and 5.7 that, even though the average error produced by all four approaches used appear to produce an overall decrease in average error, it is the WTA By Patterns parallel training methodology that seems to take the most advantage of an increase in the number of processing elements in the implementation.

Furthermore, when taking the amount of speed up (when compared with the sequential version of the training algorithm) into consideration, it was observed from Figures 5.8 and 5.10 that those

parallel training approaches based on the WTA weight update method display an overall trend to produce beneficial speed up when the number of processing elements is such that the training set is evenly divided across all worker nodes in the topology. Whereas, those approaches based on the HOST-directed weight update method display an overall trend to produce zero speed-up as the number of processing elements is increased.

Additionally, an overall decrease in accuracy was observed for all four parallel training approaches considered in the study; with the only exception to the rule being the XOR benchmark problem, which actually displayed an increase in accuracy to 100%.

6.2 – CONCLUSIONS

Through evaluation of the obtained results, presented in section 5 of this document, the following has been found about the studied parallel implementation of the Artificial Neural Networks Backpropagation Training algorithm:

1. The parallel Backpropagation training implementation considered by this study was found to produce beneficial effects in terms of the average error as the number of processing elements was increased. A clear overall decrease in the average error was observed for all training methods (with the exception of the HOST By Patterns for the WINE benchmark) while all training parameters were kept constant and the only variation was the number of processing elements used for parallel training. And this is attributed to the manner in which the weight update algorithm is organized in which more processors contribute to approaching a global minimum error (in the error surface) and the training moves faster to that point than when only one processor for training the network.
2. Of the four different parallel training methods considered in this study, it is the WTA-by-Patterns that has been found to outperform the rest of the training methods with respect to the average error

achieved when the highest number of PE's was used, and across all benchmarks evaluated. This trend is expected to continue being true for benchmarks that consist of larger training data sets.

3. It has been found that speed up, when comparing against the training time incurred by the sequential version of the training algorithm, achieves its highest performance when the number of processing elements utilized for the parallel implementation is such that the training set is evenly distributed across all PE's and no processing element has more training patterns than the rest.
4. For benchmarks that consist of a relatively larger number of training patterns, the parallel implementation was found to display a behavior of decreasing speed up. Particularly for the WINE benchmark problem, the average speed up for number of processing elements of 6 or less was found to be close to linear when using the WTA-by-epochs training methodology. Furthermore, this same training method, the average speed up resulted in a steady decrease to converge at a speed up of 2 when the maximum number of processing elements was utilized.
5. A decrease in accuracy was experienced for benchmark problems that do not have a balanced representation of all of its classes and tend to favor one class particularly more than the rest. For the one benchmark that was evaluated using optimal training parameters and a fair representation of its classes, an increase in accuracy was observed and it could serve as point of focus for future research to duplicate this environment on other benchmark problems to investigate if this is a trend inherent to the parallel implementation or exclusive to the benchmark used.
6. Lastly, through examination of the results obtained as the number of processing elements was increased, it was found that the algorithm used for updating the weights of the network during training greatly influences the performance of the parallel implementation in average error and speed up. Consequently, making use of algorithms that are based on techniques that aim to avoid the adverse effects of communication overhead while , at the same time, being conscious of heuristic measures adopted for artificial neural networks training (such as the Winner-Takes-All algorithm) are more likely to result in superior performance and could be the subject for future research moving forward.

6.3 – FUTURE ENHANCEMENTS

The study made use of three benchmark problems that can be organized as being small and medium sized (in contrast with most practically significant problems), thus, in order to provide a more accurate and realistic measure of its potential, it would be convenient to evaluate problems that consist of training datasets in the range of the more than one thousand training patterns. For this task, the Yeast Classification problem provided by [Fra10] is suggested to be a good candidate for evaluation.

Additionally, given the observation that the Winner-Takes-All weight update algorithm has been found to have superior performance than the HOST-directed version of the parallel implementation, further improvements to the WTA algorithm are in order to explore additional gains in performance and counteract the weaknesses of this algorithm in the present study. Namely, modifications to the current WTA algorithm could be made to enforce a more strict policy for encouraging all worker nodes to contribute to weight update in a more balanced manner in order to achieve greater improvements in average error.

Lastly, In order to determine if the presented parallel implementation of the Backpropagation training algorithm is able to produce substantial gains in accuracy, it is imperative to find the optimal training parameters for each of the benchmarks used and re-evaluate the performance of the parallel training approach. It has been suggested by [Nav98] that, in order to find an appropriate number of hidden-layer neurons, the number of neurons in this layer should be incremented by multiples of the number of classes of the problem at hand.

References

[Cro94] Crowl, L. A., "How to measure, present and compare parallel performance", IEEE Parallel & Distributed Technology, vol. 2, no. 1, pp. 9-25, 1994.

[Day90] Dayhoff, Judith E., Neural Network Architectures: An Introduction, *Van Nostrand Reinhold*, 1990.

[Dcl05] Distributed Computing Lab at UTEP, "Cluster Design" http://ece.utep.edu/research/webdcl/ClusterDesign.htm.

[Fau94] Faucett, L., Fundamentals of Neural Networks: Architectures, Algorithms and Applications, 1st Ed., *Prentice Hall, Inc.,* 1994.

[Fly95] Flynn, Michael J., Computer Architecture: Pipelined and Parallel Processor Design, 1st Ed., Jones and Bartlett Publishers, Inc., 1995.

[Foo94] Foo Shou King; Saratchandran, P.; Sundararajan, N.; , "Optimal distribution of patterns in a heterogeneous array of transputers for backpropagation networks," Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on , vol.6, no., pp.3950-3955 vol.6, 27 Jun- 2 Jul 1994

[Foo95] Foo, Shou King, "Comparison of parallel and serial implementation of feedforward neural networks," J. Microcomputer Applications, vol. 17, pp. 83-94, 1995.

[Foo95] Foo, S., P. Saratchandran, and N. Sundararajan, "Analysis of Training Set Parallelism for Backpropagation Neural Networks," Proceedings of the International Journal of Neural Systems, 6(1), Vol. 61-78, March 1995.

[Foo97] Foo, S., P. Saratchandran, and N. Sundararajan, "Parallel Implementation of Backpropagation Neural Network on a Heterogeneous RING Processor Topology", Proceedings of the IEEE International Conference on Neural Networks, 1997, vol. 2, pp. 937 - 942

[Fra10] Frank, A. & Asuncion, A. (2010). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[Gam96] Games, R.A., "Panel: "Benchmarking for Real-Time High-Performance Computing", "Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems", vol., no., pp.197-199, 15-16 Apr 1996

[Gra03] Grama, Ananth, Introduction to Parallel Computing, 2nd Ed., *Pearson Addison Wesley*, 2003

[Haz04] Abbas H.M., "Performance of the Alex AVX-2 MIMD architecture in learning the NetTalk database" (2004) IEEE Transactions on Neural Networks, 15 (2), pp. 505-514.

[Hay99] Haykin, Simon, Neural Networks: A Comprehensive Foundation , 2nd Ed., Prentice Hall, Inc., 1999.

[Hwa99] Hwang, Doosung, "A Parallel Backpropagation Learning Algorithm for Urban Traffic Congestion Measurement", Intelligent Engineering Systems Through Artificial Neural Networks, v 9, p 75-80, 1999.

[Jan97] Jang J. –S. R., Sun C. –T., Mizutani E., "Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence," Prentice Hall, 1997.

[Kum96] Kumar, Mohan J., "Mapping of Artificial Neural Networks onto Message Passing Systems", December 1996, IEEE Transactions on Systems, Man, And Cybernetics-Part B: Cybernetics, vol. 26, no. 6.

[Mad07] Maduko, E, Development And Testing Of A Neuro-Fuzzy Classification System For IOS Data In Asthmatic Children, M.S. Thesis, The University of Texas at El Paso, December 2007.

[Mor04] Morchen, F.; , "Analysis of speedup as function of block size and cluster size for parallel feed-forward neural networks on a Beowulf cluster," Neural Networks, IEEE Transactions on , vol.15, no.2, pp.515-527, March 2004

[Mit97] Mitchel, Tom, Machine Learning, 1st Ed., McGraw Hill Inc., 1997.

[Moh95] Hassoun, Mohammad H., Fundmentals of Artificial Neural Networks, 1st Ed., *Massachusetts Institute of Technology Press,* 1995

[Myr11] Myrnet Network Products, Myricom, Inc. http://www.myri.com

[Nav98] Nava, Patricia A., "Implementation of Neuro-Fuzzy Systems Through Interval Mathematics", Proceedings of the 1998 IEEE International Joint Conference on the Science and Technology of Intelligent Systems, pp. 365 – 369, 1998.

[Nov01] Novokhodko, A., and Valentine, S. "A parallel implementation of the batch backpropagation training of neural networks", Proceedings of the International Joint Conference on Neural Networks, v 3, p 1783-1786, 2001

[Qui94] Quinn, Michael J., Parallel Computing: Theory and Practice, 2nd Ed., McGraw Hill, Inc., 1994.

[Rum89] Rumelhart, D.E., Mclelland, J. L., and the PDP Research Group: "Parallel Distributed Processing: Explorations in the microstructure of cognition, Vol 1: Foundations," MIT Press, Cambridge Massachusetts, 1986.

[Sun98] Sundararajan, N. Parallel Architectures for Artificial Neural Networks: Paradigms and Implementations, 1st Ed., IEEE Computer Society, Inc., 1998.

[Tor98] Torresen, Jim and Tomita, Shinji. “A Review of Parallel Implementations of Backpropagation Neural Networks.” Chapter 2 in the book by N. Sundararajan and P. Saratchandran (editors): Parallel Architectures for Artificial Neural Networks, IEEE CS Press, 1998. ISBN 0-8186-8399-6.

[Yen99] Yen J., Langari R., “Fuzzy Logic: Intelligence, Control and Information,” Prentice Hall, New Jersey, 1999.

Appendix A: Sample Output File for Data Collection

LR	ACC	Error	Accuracy	Epoch	Time
1	1	0.019775	0.500000	35	0.002608
1	2	0.019649	0.500000	36	0.002450
1	3	0.019902	0.500000	34	0.002243
1	4	0.019607	0.500000	36	0.002475
1	5	0.019616	0.500000	35	0.002362
1	6	0.019504	0.500000	35	0.002341
1	7	0.019947	0.500000	34	0.002266
1	8	0.019566	0.500000	35	0.002412
1	9	0.019471	0.500000	36	0.002171
2	1	0.019859	0.500000	12	0.000835
2	2	0.019773	0.500000	11	0.000808
2	3	0.019560	0.500000	12	0.000844
2	4	0.019254	0.500000	12	0.000821
2	5	0.018688	0.500000	12	0.000791
2	6	0.019627	0.500000	11	0.000783
2	7	0.019053	0.500000	12	0.000875
2	8	0.019323	0.500000	12	0.000823
2	9	0.018777	0.500000	12	0.000858
3	1	0.016987	0.500000	4	0.000439
3	2	0.016623	0.500000	3	0.000431
3	3	0.018591	0.500000	3	0.000443
3	4	0.019547	0.500000	3	0.000439
3	5	0.018494	0.500000	3	0.000386
3	6	0.017656	0.500000	4	0.000435
3	7	0.019113	0.500000	3	0.000403
3	8	0.016132	0.500000	4	0.000410
3	9	0.018119	0.500000	3	0.000430
4	1	0.016134	0.500000	2	0.000518
4	2	0.019094	0.500000	2	0.000330
4	3	0.009645	0.500000	3	0.000392
4	4	0.010915	0.500000	3	0.000487
4	5	0.009480	0.500000	3	0.000424
4	6	0.009954	0.500000	3	0.000385
4	7	0.012016	0.500000	3	0.000395
4	8	0.010129	0.500000	3	0.000383
4	9	0.011870	0.500000	3	0.000395
5	1	0.006904	0.500000	2	0.000317
5	2	0.009404	0.500000	2	0.000353
5	3	0.014854	0.500000	2	0.000353
5	4	0.010084	0.500000	2	0.000378
5	5	0.010916	0.500000	2	0.000350
5	6	0.008269	0.500000	2	0.000356

5	7	0.019106	0.500000	2	0.000367
5	8	0.006408	0.500000	2	0.000405
5	9	0.009494	0.500000	2	0.000393
6	1	0.009990	0.500000	2	0.000389
6	2	0.010199	0.500000	2	0.000387
6	3	0.008025	0.500000	2	0.000302
6	4	0.007102	0.500000	2	0.000387
6	5	0.005618	0.500000	2	0.000371
6	6	0.005617	0.500000	2	0.000326
6	7	0.008887	0.500000	2	0.000351
6	8	0.010118	0.500000	2	0.000393
6	9	0.005574	0.500000	2	0.000329
7	1	0.004480	0.500000	2	0.000371
7	2	0.007731	0.500000	2	0.000327
7	3	0.005911	0.500000	2	0.000395
7	4	0.006556	0.500000	2	0.000309
7	5	0.005160	0.500000	2	0.000343
7	6	0.005158	0.500000	2	0.000430
7	7	0.006413	0.500000	2	0.000390
7	8	0.006765	0.500000	2	0.000432
7	9	0.006436	0.500000	2	0.000386
8	1	0.003323	0.500000	2	0.000329
8	2	0.003035	0.500000	2	0.000311
8	3	0.003591	0.500000	2	0.000315
8	4	0.003909	0.500000	2	0.000342
8	5	0.004882	0.500000	2	0.000391
8	6	0.002228	0.500000	2	0.000342
8	7	0.005034	0.500000	2	0.000370
8	8	0.005061	0.500000	2	0.000321
8	9	0.003526	0.500000	2	0.000436
9	1	0.003338	0.500000	2	0.000318
9	2	0.003726	0.500000	2	0.000380
9	3	0.004081	0.500000	2	0.000371
9	4	0.002732	0.500000	2	0.000404
9	5	0.002784	0.500000	2	0.000333
9	6	0.004552	0.500000	2	0.000392
9	7	0.003855	0.500000	2	0.000326
9	8	0.018435	0.500000	1	0.000264
9	9	0.003502	0.500000	2	0.000425

Processor Contributions to Weight Updates:

Pocessor	Contribution
0	0
1	80
2	51
3	54
4	462

Appendix B: Sample of Input File (BCHD Benchmark)

```
1
0.0;0.0;0.0;0.0;1
0.0;0.0;0.0;1.0;2
0.0;0.0;1.0;0.0;2
0.0;0.0;1.0;1.0;3
0.0;1.0;0.0;0.0;2
0.0;1.0;0.0;1.0;3
0.0;1.0;1.0;0.0;3
0.0;1.0;1.0;1.0;4
1.0;0.0;0.0;0.0;2
1.0;0.0;0.0;1.0;3
1.0;0.0;1.0;0.0;3
1.0;0.0;1.0;1.0;4
1.0;1.0;0.0;0.0;3
1.0;1.0;0.0;1.0;4
1.0;1.0;1.0;0.0;4
1.0;1.0;1.0;1.0;5
```

Curriculum Vitae

Carlos Guillermo Beas was born on July 09, 1985 in Guadalajara, Jalisco, Mexico to Carlos and Blanca Beas. He graduated from the University of Texas at El Paso with a bachelor's Degree in Electrical and Computer Engineering in December of 2008. As an undergraduate, he interned with Texas Instruments Inc. as a Product Engineer and with IBM as a Verification Engineer. He was in charge of the software design and integration of a new bio-informatics instrument developed as part of his senior project. He was also a member of the Distributed Computing Lab, the Society of Hispanic Professional Engineers, Tau Beta Pi (UTEP Chapter) and president of the Institute of Electrical and Electronic Engineers (UTEP Student Chapter).

In spring 2009, he entered the Graduate School program at the University of Texas at El Paso, where he worked in the Department of Electrical and Computer Engineering as a teaching assistant and researched on parallel implementation of neural networks training under the guidance of Dr. Patricia Nava.

While pursuing a master's degree, he worked as an Engineering Intern with Lockheed Martin Corp. during the summer 2009 and summer 2010. He earned his Master's of Science degree in Computer Engineering in summer 2011. And he has accepted a job offer with Intel Corporation where he will be working as a Product Reliability Engineer.

Permanent address: 1811 East Apache Boulevard
Tempe, AZ 85281

This thesis was typed by Carlos Guillermo Beas.