High-Performance Simulation of Neural Networks

Timothy J. Rademacher
Lexmark International, Inc.
Lexington, KY
radm@lexmark.com

James E. Lumpp, Jr.
University of Kentucky
Lexington, KY
jel@dcs.uky.edu

Abstract—Artificial neural networks have been used for a wide range of problems in a variety of areas. The back-propagation algorithm is frequently used to train the network, but is time consuming when implemented on general-purpose computers. This paper examines methods of simulating back-propagation neural networks on parallel systems to achieve high performance. The training of artificial neural networks consists of updating the weights in several nested loops. Parallel simulation methods may be classified based on which of these loops are executed in parallel. These methods are discussed and example implementations of these methods are described.

TABLE OF CONTENTS

1. INTRODUCTION
2. THE BACK-PROPAGATION ALGORITHM
3. PARALLEL METHODS
4. CONCLUSION

1. INTRODUCTION

The human brain has many properties that make it attractive as a model for computation. It is self learning, employs massive parallelism, is fault tolerant, is adaptive, has the ability to generalize, and has low power consumption. These biological neural networks have been the inspiration for artificial neural networks (ANNs). ANNs have been successfully applied to a number of problems in a wide range of fields such as stock-market prediction, optimization problems, image recognition, and control. There was a great amount of interest created in ANNs by Rosenblatt's famous paper on the perceptron in 1958 [1]. The intensity of research into ANNs fell off when the limitations of the perceptron were pointed out by Minsky and Papert, including its inability to solve problems where multiple unconnected regions in the input space should be mapped to a single output [2]. The interest in ANNs was renewed when Rumelhart published an algorithm to train a feed-forward network with hidden nodes, i.e., a multi-layer perceptron (MLP), in 1986 [3]. The "back-propagation algorithm" back-propagates the output error through the network to adjust the weights in the hidden layers. A three-layer back-propagation network can solve a mapping problem involving any number of groupings [4]. In practice it may be desirable to use a network with more layers to decrease the size of the network or its training time.

A problem with the back-propagation algorithm is that it can require a large number of iterations to learn the training set. While ANNs can be implemented in hardware or on special-purpose neural-network computers, simulating them on general-purpose computers offers several advantages. The primary advantage is that general-purpose systems are widely available to researchers. Also, the more

This research was supported by the National Science Foundation under Grant CDA-9502645 and the Advanced Research Projects Agency under Grant DAAH04-96-1-0327.

special-purpose the system is, the more it limits flexibility in evaluating different algorithms and architectures.

The large number of iterations required tends to make simulating ANNs on general-purpose computers very slow. There are a number of ways to decrease the training time. Several improvements have been made to the rule used to adjust the weights to increase the rate of descent down the error surface or to decrease a network's likelihood of being trapped in a local minima [5]. For example, the conjugate gradient method can in some cases decrease the number of iterations by an order of magnitude [6]. Modest improvements can be made in the simulation performance by tuning the algorithm used to a specific system. For example, the algorithm can be modified to minimize cache misses and to order the calculations to keep the floating-point units (FPUs) busy [7].

Large improvements can also be made in simulation performance by executing parts of the simulation in parallel. This can be done on traditional parallel computers or on a network of workstations. This paper describes several implementations of simulations on parallel systems with an emphasis on simulating ANNs on a cluster of high-performance workstations inter-connected by a high-performance network such as an ATM network.

2. THE BACK-PROPAGATION ALGORITHM

The back-propagation algorithm presented by Rumelhart et al. is for a multi-layer feed-forward network [3]. Connections are allowed to skip a layer or layers but are not allowed to go to a previous layer or to other nodes in the same layer. Typically, fully connected networks that have connections only between adjacent layers as shown in Figure 1 are used. The input layer serves to distribute the inputs to the nodes in the first hidden layer and is therefore not counted as a layer but is referred to as layer 0 [8][9]. The notation $N_0 - N_1 - \ldots - N_L$ is used to specify a network with L layers and N_i nodes in layer i. The stimulus, or input into the network, to node i is s_i.

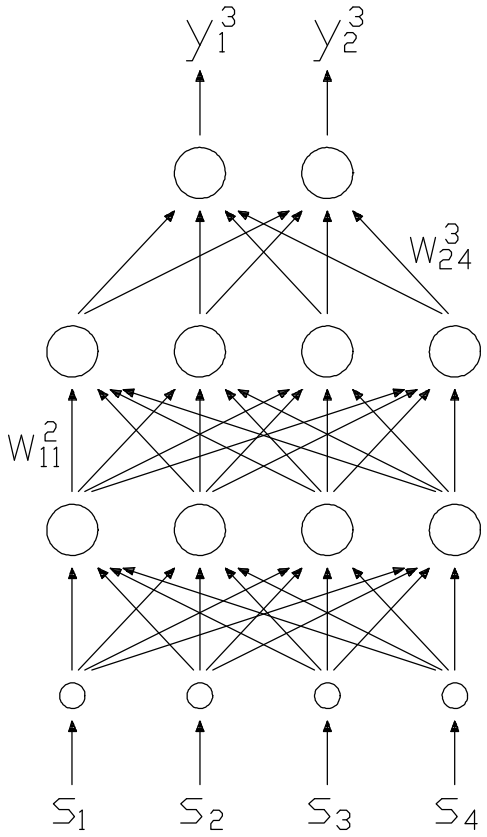

Figure 1. A fully connected three-layer feed-forward network.

The algorithm is based on knowing what the desired output is for each training example. This is in contrast to other neural network algorithms such as the perceptron where a random output unit is activated for a given input stimulus. Usually, a sigmoidal function is used for the activation function of the nodes. One commonly used activation function is

$$y_j = 1 / 1 + e^{-x_j} \qquad (1)$$

where y_j is the output of node j [3]. The input to the activation function, x_j, is the weighted sum of the input nodes.

$$x_j = \sum_i y_i w_{ji} - u \tag{2}$$

where u is the node's bias input, y_i is the output of node i in a previous layer, and w_{ji} is the weight on the connection from node i to node j. A benefit of this activation function is that it has a convenient derivative given by

$$dy_j / dx_j = y_j (1 - y_j). \tag{3}$$

Other activation functions are sometimes used, such as [10]

$$y_j = \arctan(x_j). \tag{4}$$

The back-propagation algorithm minimizes the error between the actual and desired output for the training set,

$$E = \tfrac{1}{2} \sum_p \sum_j (y_{j,p} - d_{j,p})^2 \tag{5}$$

where p is the case in the training set and j is the node in the output layer. The idea is to change the weights so that E is minimized by taking small steps on the error surface towards lower error. The simplest form of weight adjustment is

$$\Delta w = -\eta \, \partial E / \partial w \tag{6}$$

where η is the learning rate and is between 0 and 1. The weight adjustment can be improved by adding a momentum term, α , $0 < \alpha < 1$:

$$\Delta w(t) = -\varepsilon \, \partial E / \partial w(t) + \alpha \Delta w(t-1). \tag{7}$$

For each cycle through the training data, t is incremented by 1. The addition of the momentum term improves the ability of the algorithm to jump past local minima. This weight adjustment equation is known as the "generalized delta rule." This rule is used to adjust the weights to both the output layer and the hidden layers. The error must be back-propagated from the output layer to lower layers in order for the weight changes to be computed in the hidden layers.

The application of all inputs in the training set and the corresponding adjusting of the weights in the network is called an "epoch." The weights may be adjusted either after each training pattern or once per epoch. Adjusting the weights after each pattern has the advantage that $\partial E/\partial w$ does not need to be accumulated over all of the patterns; however, doing so can change the learning behavior of the network. In some cases, the learning rate is higher with the weights adjusted after each pattern and in others it is higher adjusting the weights only once for all patterns. Each weight-update cycle consists of the following five steps.

1. Feed-forward the input
2. Compute the output error
3. Back-propagate the error
4. Compute the weight deltas
5. Update the weights

To adjust the weights after each training pattern, steps 1 through 5 are repeated for each pattern. To adjust the weights once per epoch, steps 1 through 3 are repeated for each training pattern and then steps 4 and 5 are performed.

Neural network performance is often measured in millions of connection updates per second (MCUPS), connections per second (CPS), or interconnections per second (IPS). There are several common benchmark networks that are used as performance tests. NETtalk is an ANN that generates phonemes to pronounce English text [11]. A two-layer network in a 203-80-26 configuration was used although other numbers of nodes have been used in the hidden layer.

3. PARALLEL METHODS

The performance of ANN simulations can be improved by executing parts of the simulation in parallel. The typical ANN simulation consists of adjusting the weights in a number of nested loops as follows:

For each training session
For each training example
For each layer
For each node in the layer
For each weight to the node
For all bits in the weight

Each of these operations can be done in parallel, either individually or in combination [12]. This section examines some methods that have been used to simulate ANNs on parallel systems and classifies them using the above framework.

The amount of parallelism that can be exploited at the different layers varies. Training-pattern parallelism is limited by the typically large number of examples in the training set. Layer parallelism is limited by the number of layers in the network and is usually around three. The amount of parallelism can be doubled by also pipelining the error back-propagation [12]. A large number of nodes may be used in the network depending on the problem being solved. The number of weights in a fully connected network is

$$\sum_{l=1}^{L} N_l N_{l-1} . \tag{8}$$

Bit-parallelism is already used by most simulations as each math operation is done in parallel in the processor or FPU.

Training Session Parallelism

In training session parallelism, multiple epochs are run each on its own processing element (PE) simultaneously. The networks can be started with different random initial weights or each network can run with a different learning rate. Using different learning rates may allow a training session on one PE to escape a local minimum that the network would otherwise settle in. This method also allows different networks to be run on each PE.

Training-session parallelism was used to simulate an ANN on a Coral68K, a binary-tree

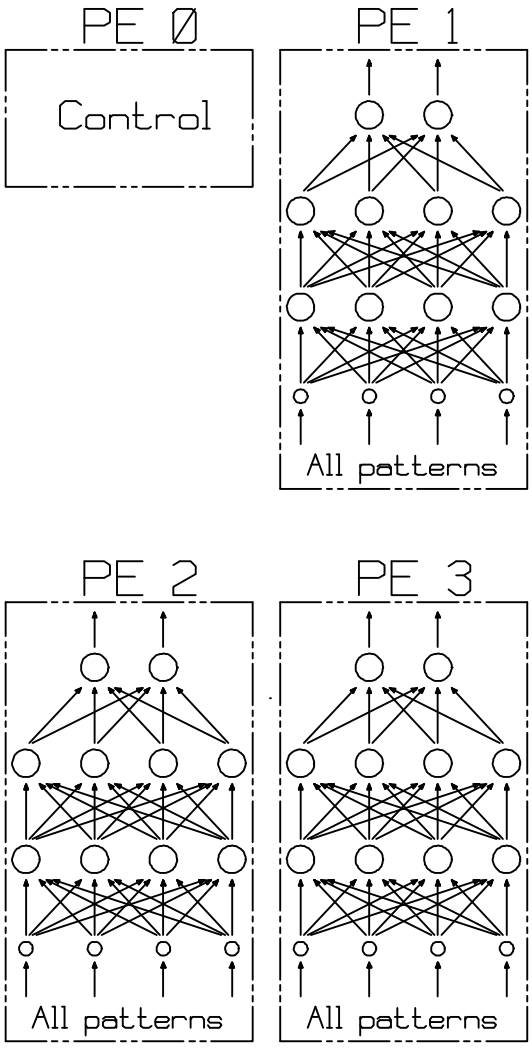

Figure 2. Training-session parallelism with three computational PEs and one control PE.

MIMD computer [13]. Given a computer with P PEs, this method uses P-1 PEs as slaves and one PE as a master. A system with P=4 is shown in Figure 2. Each slave processes a copy of the entire network. The master serves as a "critic" to determine which slave's network has made the most progress. The algorithm operates as follows. The master broadcasts to all slaves the initial weights for the network and a learning rate $\bar{\eta}(t)$. Slave i uses $\bar{\eta}(t)$ to calculate its unique $\eta_i(t)$ using an equation based on the slave's identification number. $\eta_i(t)$ varies across the slaves from 0.2 $\bar{\eta}(t)$ to 1.8 $\bar{\eta}(t)$. Each slave trains its copy of the network, updating the weights using

$$\Delta w_i(t) = -\eta_i(t)\frac{\partial E(t)}{\partial w_i(t)} + \alpha_i(t)\Delta w_i(t-1) \qquad (9)$$

where t is the epoch number and α_i is the momentum term unique to each slave. After training for a few epochs, the slave transmits its error summation for the entire training set to the master. The master determines which slave has obtained the best performance, broadcasts the weights of the winning slave to all slaves, and assigns to $\bar{\eta}(t)$ for the next epoch the value of $\eta_i(t-1)$ from the winning slave. This process then repeats until the training is complete.

This algorithm was tested on the Coral68K using a handwritten-character recognition problem with a 64-30-5 network. The initial learning rate was set to 0.6. It took a single PE 1344 training epochs to obtain an output error of less than 1% on the data in the training set. Using 63 PEs (62 slaves), achieving the same error level in only 85 epochs.

The network comparison and weight update may be done after each epoch or after a number of epochs [13]. Let A be the number of epochs executed between weight updates. For the critic to compare the networks, each of the P-1 slave processors transmits its error on the training set. The master processor transmits a message to the winning slave for it to broadcast its network weights. The total communications per epoch is

$$\frac{P + \sum_l N_l N_{l-1}}{A} \qquad (10)$$

values.

Training Example Parallelism

The number of training examples is usually large compared to the number of nodes in the network being trained. If the number of training patterns is smaller than the number of nodes then the network can "memorize" the training data rather than "learning" the underlying concept in the data. In this case, the network is likely to perform poorly on data outside the training set. A large number of training examples improves the network's ability to generalize. NETtalk, for example, has 309 nodes and its dictionary training set contains 19,075 words for a total of 141,924 training patterns [14].

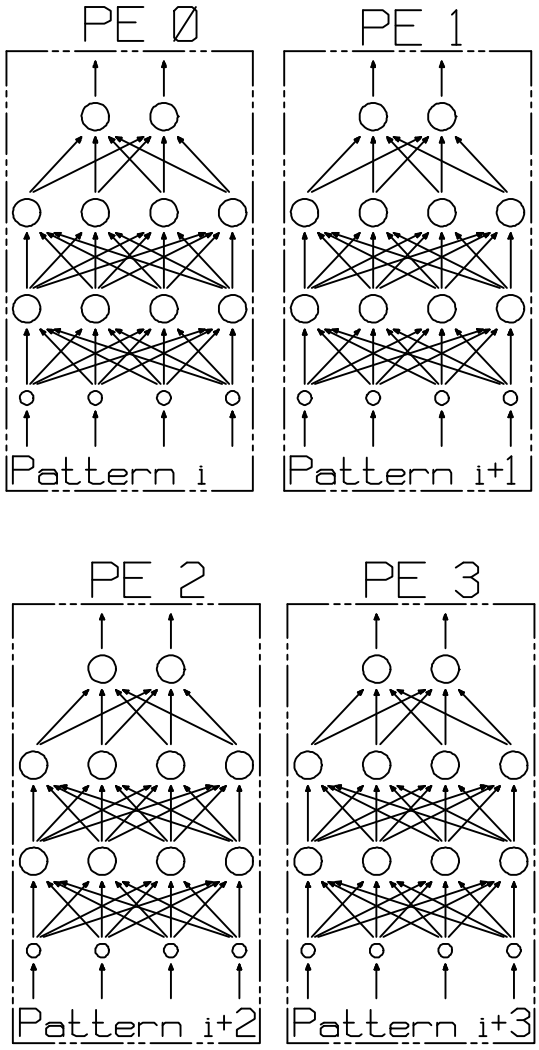

Figure 3. Training-example parallelism with four PEs.

In training-example parallelism, the training patterns are divided among the PEs and the network is trained with multiple patterns in parallel as shown in Figure 3. Training-example parallelism has been used frequently in ANN simulations and is applicable to SIMD

and MIMD computers. The number of PEs used can be as large as the number of training examples; therefore, large speedups are possible. One limitation of this method is that because the output error is calculated for all patterns in parallel the network weights can only be adjusted after applying the entire training set, as opposed to adjusting the weights after applying each pattern in the training set.

Training-example parallelism has been used for large problems with up to 10^6 training examples on a Thinking Machines Corporation CM-2 [15]. A machine with 64000 processors achieved 1.3 billion IPS by using a single processor for each input pattern and storing the weights of the network in the virtual memory of a front-end machine on the CM-2. Due to the low efficiency of the weight-adjustment algorithms on a SIMD computer, the weight adjustment is done in the front-end computer using its higher processing speed in parallel with the operation of the SIMD array.

The patterns in the training set are equally divided among the PEs. The partial sums of the output error are carried out in parallel without any inter-processor communications. Then the error values are summed over all patterns, requiring a global summing operation. If there is one PE available for each training example then the summing operation is the only calculation whose time is dependent upon the size of the training set. The CM-2 has an optimized global summing operation that operates in $O(\log(n))$ time for n processors. Therefore, this approach can achieve nearly linear speedup as the number of processors and examples increase.

Each processor stores one input pattern, the temporary values for the outputs of the hidden nodes, and the network outputs. Therefore, each processor requires enough local storage for the input pattern and the outputs of two adjacent layers. The size of local memory on typical tightly-coupled massively-parallel systems limits the size of possible problems. Larger problems can be handled by using multiple adjacent processors in parallel to process one pattern in the training set.

Given that P, the number of PEs, is less than or equal to T, the number of examples in the training set, then each PE is processing one or more input patterns. Each PE must broadcast its weight deltas to the other PEs for each epoch. The communications per epoch for training-example parallelism is then

$$P\sum_{l} N_l N_{l-1}. \tag{11}$$

Layer Parallelism

In layer parallelism, multiple patterns are processed by the network in a pipelined fashion. While the first layer is processing the network inputs for pattern p, the second layer processes the first layer's output for pattern p-1, and so on. At the end of each computation, the output of one layer becomes the input to the next layer. The pipelining can also be done during the back-propagation. After the error has been back-propagated, the weight adjustments for each layer can be done in parallel. The parallelism is limited by the number of layers in the network. A three-layer network using layer parallelism is shown in Figure 4. In this example, the weights going into a given layer and its nodes are contained in the same PE. PE 0 contains the training patterns.

For one training pattern, the PE for layer l transmits its N_l activation values to the PE for layer l+1 in the feed-forward phase. For the error back-propagation, the PE for layer l transmits $N_{l\text{-}1}$ error values to layer l-1. The communications in the forward and back-propagation phases are equal. These values are

communicated for each of T patterns per epoch. The communications per epoch are

$$2T\sum_{l=1}^{L-1} N_l \,. \tag{12}$$

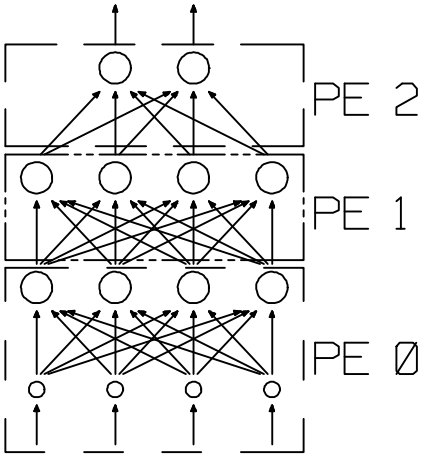

Figure 4. A three-layer network using layer parallelism.

Layer parallelism is seldom used independently of other forms of parallelism. The degree of parallelism possible is limited by the number of layers in the network, frequently two or three. Pipelining both the feed-forward and back-propagation phases doubles the amount of parallelism, but increases the communications because the weight deltas must be transmitted to the feed-forward processors. For most networks, the number of nodes per layer is much larger than the number of layers. In this case, node parallelism offers a greater amount of parallelism. If T is large relative to the average N_l then the communications for training-example parallelism will be lower.

Node Parallelism

Node parallelism exploits the parallelism in the number of nodes in each layer. If there are at least as many PEs as there are nodes in a layer, as in Figure 5, then each PE calculates the output of one node. Otherwise, the nodes are divided among the available PEs. When a PE has computed the output of a node, it may use a multicast operation to send the value to all other processors. When all outputs have been computed and broadcast, each PE will contain the output for all nodes and may proceed to calculate the output for its node in the next layer.

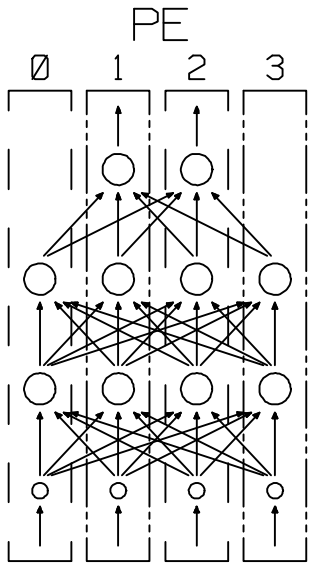

Figure 5. Example of node parallelism.

Node parallelism was used on an array of 20-MHz T805 transputers connected in a message-passing ring topology [16]. Each processor contains the entire set of training patterns and the nodes in each layer are divided equally among the PEs. The processors compute the activation values for one layer in parallel. Each processor must send its activation values to every other processor. An all-to-all broadcast with an array of P processors requires P-1 transfers as one link of a processor can be used for receiving at the same time its other link is used for transmitting. The system was tested with 1 to 10 PEs on the Encoder, Num18x12, and NETtalk benchmarks. A 189-N_2-27, N_2={9, 27, 45, 63, 81, 99, 117}, network was used for NETtalk. The network was modified so that the number of nodes in each layer was a multiple of 1, 3, and 9, the number of processors used on the problem. No absolute

performance measures were given, but the expected performance was calculated and compared to the experimental performance. The calculated performance matched the actual performance within ±5% when there was a sufficient number of nodes per processor. The experimental performance degraded as the number of nodes per processor became small due to overhead that was not part of the calculation.

After the PEs calculate the activation values for layer l, $l < L$, each PE transmits its activation values to every other PE for inputs to layer l+1. All N_l values are transmitted for each layer up to the output layer. The back-propagation communications are the same. These values are communicated for each of T patterns per epoch. The communications per epoch are

$$2T\sum_{l=1}^{L-1} N_l \,, \tag{13}$$

the same as in layer parallelism. Node parallelism has the potential for large levels of parallelism. In general, it is suited for more tightly-coupled architectures because of the number of communications that occur.

Weight Parallelism

Another approach is to compute the products of the weights with the activation values or back-propagated error in parallel. Depending on the implementation, the sum of the products may also be done in parallel. The Simple Checkerboard (SC) scheme is a weight-parallelism algorithm for hypercubes only with an even-power-of-two number of nodes [17]. The Rectangular Checkerboard (RC), a generalization of SC, can use a rectangular processor grid and has performance at least as good as SC in general and better performance on non-uniform networks, i.e., $N_i \neq N_j$ for $i \neq j$. RC was implemented on a Thinking Machines Corporation CM-5 and used some special features of the CM-5 but is generally applicable to hypercubes or rectangular mesh systems [18]. Various configurations of a symmetric non-uniform two-layer network were tested on a 512 node CM-5 with $N_0 = N_2 = \{512, 8192, 32768, 131072\}$ and $N_1 = \{32, 128\}$. With the 131072-128-131072 network, a performance of 350 MCUPS was achieved.

RC uses a P processor grid P_H processors high by P_W processors wide. Let node_i^l be node i in layer l and P[r,c] be the processor in row r and column c of the grid. The nodes and weights of the network are mapped to the processors as follows:

even l:

$$\text{node}_i^l \Rightarrow \text{P}[0,\ i \bmod P_W] \tag{14}$$

$$w_{ij}^l \Rightarrow \text{P}[j \bmod P_H,\ i \bmod P_W] \tag{15}$$

odd l:

$$\text{node}_i^l \Rightarrow \text{P}[i \bmod P_W,\ 0] \tag{16}$$

$$w_{ij}^l \Rightarrow \text{P}[i \bmod P_H,\ j \bmod P_W] \tag{17}$$

The nodes for even layers are contained in processor row 0 and the nodes for odd layers are contained in processor column 0. The activation values for an odd layer l are computed as follows. Each processor in row 0 transmits the activation values for the nodes in layer l-1 to the other processors in its column using a single-source broadcast. Each processor calculates the partial sums for the weights that it contains. The partial sums are transmitted on each row using a single-source-accumulate operation to the processors in column 0. The processors in column 0, that contain the nodes in layer l, calculate output of the sigmoidal function on the sums. The operation for an even layer l, $l > 0$, is similar. The output of the nodes in layer l-1 is broadcast across the rows. The partial sums are transmitted using the single-sink-accumulate operation to the processors in row 0 where the activation values for layer l are computed.

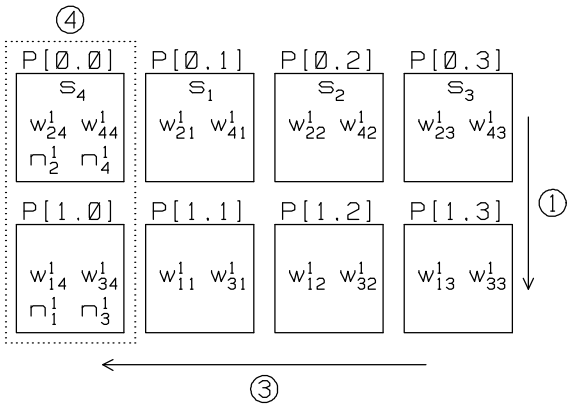

Figure 6. RC operation for layer 1.

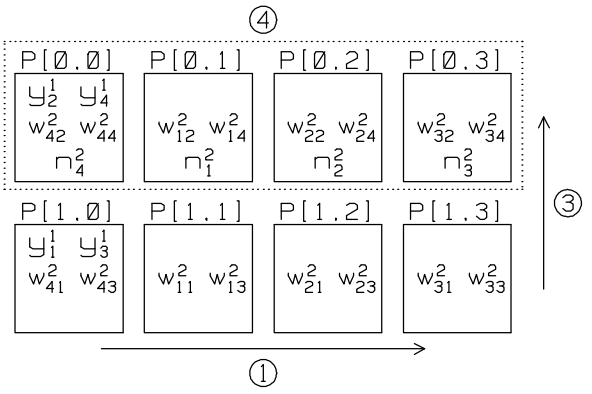

Figure 7. RC operation for layer 2.

As an example, consider a 2 by 4 processor grid. Figure 6 shows how layer 1 of the network in Figure 1 is mapped to the processors. The input stimulus s_i for all training patterns is contained in P[0,*i*] and the nodes of layer 1 are contained in column 0. The mapping for layer 2 is shown in Figure **7**. The outputs of the nodes in layer 1, y_i^1, are contained in the processor were they were computed. For either layer, the following operations take place.

1. Broadcast the input values for the layer.
2. Each processor computes the weight-input product and computes the partial sums for each destination node.
3. Send the partial sums to the node processor using a single-sink-accumulate operation.
4. Compute the activation value for each node.

The performance of RC is dependent on the existence of a single-sink-accumulate operation. On a system without this operation, weight parallelism could work as follows. The weights into each node are equally divided among the PEs. The PEs compute the partial sums of the weight-input products for each node *i* in the layer. Each PE then broadcasts the partial sums for all nodes in the layer and computes the activation values from the partial sums. Assuming $P \le N_l$, this requires P broadcasts of N_l values for each of the L layers. This process is repeated for each training pattern and for the error back-propagation. The number of values communicated per epoch is

$$2TP\sum_{l=1}^{L} N_l \,. \tag{18}$$

Bit Parallelism

All of the implementations discussed above employ bit parallelism; that is, the bits of the weights and inputs are acted on in parallel in the processor or FPU. In the feed-forward phase, a weight is multiplied by the output of a node in a lower layer. The processor or FPU inputs the bits of each term in parallel and computes the product operating on all bits in parallel. This is opposed to a bit-serial multiplication where the terms are input into the multiplier one bit at a time and the product calculated as the bits are input. For an ANN implemented in hardware, it may be desirable to use bit-serial PEs. Doing so may reduce the size of each PE and allow a greater number of PEs in a given area, increasing the degree of parallelism at a higher level. A systolic array of bit-serial PEs has been designed to implement a multi-layer perceptron (MLP) [8]. The advantages of the bit-serial design are an improved area versus speed tradeoff and reduced pin count. This is the first VLSI architecture for MLPs that uses the same

processors, memory, and configuration for both training and recall.

The architecture of the bit-serial MLP is comprised of a control generator, a state memory, a number of processor blocks, and a number of activation blocks. A processor block contains multiply-and-accumulate (MAC) operators for both the error and feed-forward phases and an operator to adjust the weight. An activation block computes the activation value for the node using a piece-wise linear approximation to the sigmoidal function. The control generator generates the signals that control the flow of events and signals when a forward and backward pass for a training example is complete.

A 2-2-1 MLP was used with the XOR problem for the first design. The system was designed and simulated in the FIRST (Fast Implementation of Real-time Signal Transforms) silicon compiler developed at the University of Edinburgh. The design used a 16-bit fixed-point word length for the weights and activation values. Double words were used to prevent overflow in the MAC operations. The design comprised 4 processor units and 2 activation units and required 384 clock cycles to complete one forward and backward pass. The learning rate was set at 0.5 and the momentum parameter was set to 0.95. The design was simulated in FIRST and compared to the results for the PDP implementation using 32-bit floating-point operations [19]. The FIRST simulation produced results similar to and actually learned at a slightly faster rate than the PDP implementation.

One of the processor units was constructed in a 1.2 μm two-level CMOS process. The processor operated successfully at 25 MHz. A 203-60-26 MLP was designed to for the NETtalk benchmark. Based on the constructed processor unit, the NETtalk design would require a total silicon area of 502 mm^2 in nine 240-pin packages. The speed of the system was estimated at 88.0 MCUPS and is comparable in size and speed to other parallel implementations in hardware.

Bit-serial hardware implementations of neural networks allow a larger number of PEs to be implemented in a given amount of silicon and package size. The performance of bit-serial ANNs can be similar to bit-parallel systems. However, on most computers, bit-parallel hardware is already present in the processor and FPU. Therefore, bit parallelism is used for high-performance simulations on general-purpose computers.

Hybrid Approaches

There are many ways that these parallel techniques can be used together. This section considers an implementation using pattern and node parallelism on a high-performance network of workstations. A cluster of loosely-coupled workstations has several advantages over multicomputers. The workstations have greater availability, are economically attractive, have familiar tools, and non-homogeneous architectures can be used together through systems such as the Parallel Virtual Machine (PVM) [20]. Distributed Matrix Back-Propagation (DMBP) is an algorithm for a network of LAN-connected workstations that uses training-example and node parallelism [21]. DMBP uses an efficient matrix representation of the network called the matrix back-propagation (MBP) algorithm [7]. MBP increases performance by tuning the back-propagation algorithm to RISC workstations. MBP performs calculations on large two-dimensional arrays in sub-blocks to minimize cache misses and uses loop unrolling to calculate four connection updates per loop iteration to improve FPU performance.

DMBP initially determines how the back-propagation calculations are to be distributed between a fixed number of processors. The

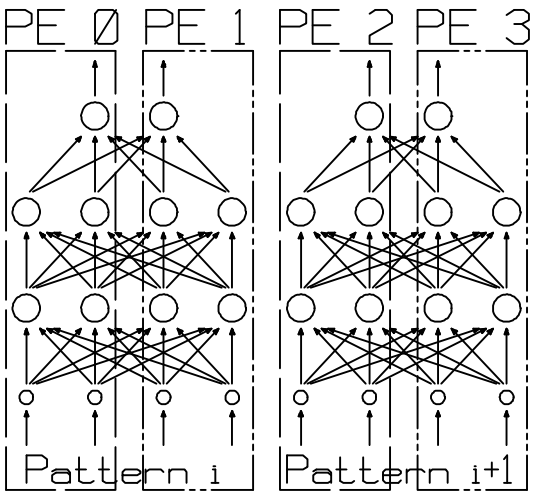

Figure 8. DMBP with two network partitions and two pattern partitions.

algorithm uses both the data transfer rate between processors and the processing power of the workstations to optimize the distribution. The calculations are partitioned by both pattern and by network. Let D_p be the number of pattern divisions and D_n be the number of network divisions. The N_p patterns are distributed by giving each processor N_p / D_p patterns. The neural network is divided into sub-networks with each piece of size N_l / D_n going to one processor. Given that $D_nD_p \leq P$ where P is the number of PEs available, the values of D_n and D_p are determined to minimize the sum of the computation and communication time for one application of the patterns and adjustment of the weights. Figure 8 shows a 4-4-4-2 network running on four PEs with D_n =2 and D_p =2.

DMBP was implemented on a 3.5 Mbit/s Ethernet network consisting of an IBM RISC6000/550, an HP 9000/750, and an HP 9000/720. A 500-500-1 network was used to solve the 500-bit parity problem. The performance measured in MCUPS increased as the number of patterns increased and surpassed that of the single-processor version with greater than 1000 patterns. A performance of 8.1 MCUPS on an image compression problem was achieved, a speed-up of 2.2 over the single-processor implementation.

Summary

Generally, parallelism of the outer-most loops is appropriate for loosely-coupled architectures and the lower levels of parallelism are appropriate for tightly-coupled architectures. The frequency of the communications increases with lower levels of parallelism. For training-session parallelism, communications only need be performed once every epoch or even several epochs, but when this happens, every weight in the network needs to be broadcast. The result is a small number of large messages. At the lowest level with weight-bit parallelism, a number of communications take place each time a weight is used. Therefore, the higher levels of parallelism are appropriate when there is a large communications latency, such as in distributed systems connected on a network. Paralleling the inner-most loops is appropriate when the communications latency is minimal as when there is a dedicated bus connecting the parallel entities.

Paral-lelism	Values	Messages	Average Size (bytes)
session	18,434	9	16,386
pattern	147,408	8	147,408
layer	30,087,888	576,696	424
node	30,087,888	4,541,568	53
weight	701,672 k	4,541,568	1236

Table 1. Comparison of communications for one epoch of the NETtalk network using different forms of parallelism.

Consider the NETtalk benchmark running on 8 PEs using 64-bit floating-point values for the inputs, outputs, and weights. Table 1 compares the communications for NETtalk using different forms of parallelism. The values shown are for one-weight adjustment per

epoch. It is assumed that only simple communications are available. Therefore, the weight parallelism values are for the operation described for distributed systems, not for a system that has multiple-source-accumulate communications as described in the CM-5 implementation. In systems where the communications start-up time is high, training-session and training-pattern parallelism are the best option due to their low number messages sent per epoch. Weight parallelism requires the highest number of values and messages so it is best suited to a tightly-coupled architecture with low communications start-up time and parallel communications paths.

4. CONCLUSION

Presently, simulation time can prohibit the use of an ANN as a solution to some problems. In addition, larger networks will be needed for more difficult problems in the future, further increasing the simulation time. These large networks will make parallelism a necessary part of simulations. This paper has reviewed previous approaches used to parallelize the back-propagation algorithm using training-session, training-example, layer, node, weight, and bit parallelism. Clusters of high-performance workstations offer an attractive alternative to tightly-coupled parallel machines and specialized hardware. DMBP was described which is suitable for simulating ANNs on clusters of workstations on an ATM network. Finally, it was shown that the higher levels of parallelism are most appropriate for such systems because of the lower-frequency of inter-processor communications required.

REFERENCES

[1] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review* Vol. 65, pp. 386-408, 1958.

[2] Marvin Minsky and Seymour Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, pp. 533-536, October 9, 1986.

[4] Ralf Östermark, "A Flexible Multicomputer Algorithm for Artificial Neural Networks," *Neural Networks*, Vol. 9, No. 1, pp. 169-178, 1996.

[5] Ramana Vitthal, P. Sunthar, and Ch. Durgaprasada Rao, "The Generalized Proportional-Intergral-Derivative (PID) Gradient Descent Back Propagation Algorithm," *Neural Networks*, Vol. 8, No. 4, pp. 563-569, 1995.

[6] E. M. Johansson, F. U. Dowla, and F. U. Goodman, "Backpropagation Learning for Multilayer Feed-forward Neural Networks Using the Conjugate Gradient Method," *International Journal of Neural Systems*, Vol. 2, No. 4, pp. 291-301, 1992.

[7] Davide Anguita, *MBP - Matrix Back Propagation v1.1: An efficient implementation of the BP algorithm*, the user's manual for the MBP software package, November 1993.

[8] P. Murtagh, A. C. Tsoi, and N. Bergmann, "Bit-serial systolic array implementation of a multilayer perceptron," *IEE Proceedings-E*, Vol. 140, No. 5, pp. 277-288, September 1993.

[9] Anil K. Jain and Jianchang Mao, "Artificial Neural Networks: A Tutorial," *Computer*, pp. 31-44, March 1996.

[10] George Cybenko, "Theme Editor's Introduction: Neural Networks in Computational Science and Engineering," *IEEE Computational Science & Engineering*, pp. 36-42, Spring 1996.

[11] Terrence J. Sejnowski and Charles R. Rosenberg, "Parallel Networks that Learn to Pronounce English Text," *Complex Systems*, Vol. 1, pp. 145-168, 1987.

[12] Tomas Nordström and Bertil Svensson, "Using and designing massively parallel computers for artificial neural networks," *Journal of Parallel and Distributed Computing*, Vol. 14, No. 3, pp. 260-285, 1992.

[13] Julio Tanomaru, Omichi Shigeo, and Akio Azuma, "General Purpose MIMD Computers and Neural Networks: Three Case Studies," *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 4587-4592, 1995.

[14] Matt White and Scott E. Fahlmann, "CMU Neural Network Benchmark Database," *a collection of information and common benchmarks for ANN, anonymous ftp from pt.cs.cmu.edu in directory /afs/cs/project/connect/bench*, 1994.

[15] Robert M. Farber, "Efficiently Modeling Neural Networks on Massively Parallel Computers", NASA workshop on parallel computing, November 1991.

[16] R. Arularasan, P. Saratchandran, and N. Sundararajan, "An Analysis of Network Parallelism of Backpropagation Neural Networks on a Transputer Array," *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 4597-4602, 1995.

[17] Vipin Kumar, Shashi Shekhar, and Minesh B. Amin, "A Scalable Parallel Formulation of the Backpropagation Algorithm for Hypercubes and Related Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 10, pp. 1073-1089, Oct. 1994.

[18] Minesh B. Amin and Shashi Shekhar, "Customizing Parallel Formulations of Backpropagation Learning Algorithm to Neural Network Architectures: A Summary of Results," *Tools for Artificial Intelligence*, Nov. 1994.

[19] James L. McClelland and David E. Rumelhart, "Training hidden units: the generalized delta rule," in James L. McClelland and David E. Rumelhart, *Explorations in parallel distributed processing: a handbook of models programs, and exercises*. MIT Press, USA, pp. 121-159, 1988.

[20] A. Geist and V. Sunderam, "Network-based concurrent computing on the PVM system," Concurrency: Practice and Experience, Vol. 4, No. 4, June 1992.

[21] D. Anguita, A. Da Canal, W. Da Canal, A. Falcone, A. M. Scapolla, "On the Distributed Implementation of the Back-Propagation," *International Conference on Artificial Neural Networks*, 1994.

Timothy J. Rademacher is an engineer at Lexmark International, Inc. and a graduate student in the Department of Electrical Engineering at the University of Kentucky. He received his BS degree in Electrical Engineering from Rose-Hulman Institute of Technology. His research interests include parallel processing, neural networks, and artificial intelligence.

James E. Lumpp, Jr. is an assistant professor in the Department of Electrical Engineering at the University of Kentucky. He received BS and MS degrees from Purdue University, and PhD from the University of Iowa in Electrical and Computer Engineering.