Correspondence

Efficient Mapping of Backpropagation Algorithm onto a Network of Workstations

V. Sudhakar and C. Siva Ram Murthy

***Abstract*— In this paper, we present an efficient technique for mapping a backpropagation (BP) learning algorithm for multilayered neural networks onto a network of workstations (NOW's). We adopt a vertical partitioning scheme, where each layer in the neural network is divided into p disjoint partitions, and map each partition onto an independent workstation in a network of p workstations. We present a fully distributed version of the BP algorithm and also its speedup analysis. We compare the performance of our algorithm with a recent work involving the vertical partitioning approach for mapping the BP algorithm onto a distributed memory multiprocessor. Our results on SUN 3/50 NOW's show that we are able to achieve better speedups by using only two communication sets and also by avoiding some redundancy in the weights computation for one training cycle of the algorithm.**

***Index Terms*— Backpropagation algorithm, distributed memory multiprocessors, multilayered neural networks, network of workstations, network partitioning, pattern partitioning, performance analysis.**

I. Introduction

Artificial Neural Networks (ANN's) have recently been shown to be powerful computational models that can effectively address complex pattern classification and pattern recognition problems. Due to their adaptive, self-organizing, fault-tolerant, and nonlinear features, ANN's are emerging as an attractive technology for potential artificial intelligence [4], [7], [30], [31], [36], [39] and signal processing applications [9], [12], [13], [19], [26], [28], [33], [41]. Since they require large computational resources, the usual approach adopted for studying ANN's is to simulate them on conventional uniprocessor high speed computers. However, these simulations are limited by the speed and storage capacity of the computer. Parallel implementation of ANN's offers a natural way out of this problem for meeting both the speed and storage requirements. Parallel implementation models take advantage of the several parallel computational structures inherent in ANN's to achieve high processing rates. As a result, ANN's have been implemented on several commercially available multiprocessor platforms, such as the Connection Machine [1], [3], Warp [27], MPP [13], and BBN butterfly [5], and also on several different architectures, such as systolic arrays [17], [18], hypercubes [15], [41], [23], reduced mesh of trees [19], reconfigurable array processor [32], and SIMD arrays [33].

One class of ANN's that has been widely studied is the multilayered feedforward neural networks. Typically, the network consists of a set of input nodes that constitute the *input layer*, one or more sets of intermediate nodes that constitute the *hidden layers*, and a set of output nodes the constitute the *output layer*. The input signal propagates through the network in the forward direction, on a layer-by-layer basis. The backpropagation (BP) algorithm [16], [29] is one of the most popular supervised (represented by a set of input-output examples) training algorithms for multilayered feedforward neural networks. It has been used for a large number of practical applications such as speech processing [29], [30], sonar/radar target detection [9], [10], [28], control [22], [24], and medical imaging [25]. The BP algorithm is unfortunately computationally intensive. As a result of this, there have been several investigations into developing parallel formulations of this algorithm for a diverse range of parallel architectures, such as linear arrays, meshes, and hypercubes. However, due to the high cost of these machines, computing on a network of workstations (NOW's) is proving to be an economical alternative for a number of scientific and engineering applications. The viability of (network) computing on a NOW's has been established for many large scientific and engineering applications [34]. It has been shown that, for a small number of processors, computing on a NOW's (RS/6000) for these applications is quite competitive with hypercube multiprocessors [34]. The most important feature of this type of (network) computing is that it enables the use of existing resources. Furthermore, these resources can be shared with other applications that require them.

Some recent simulations of neural networks on distributed-memory message-passing multiprocessors (DMM's) have been reported in the literature in [6] and [20]. They consider the mapping of generic neural network models and models employing BP algorithms for learning, respectively, onto transputer-based DMM's. Their approach is based on systolic algorithms, which do not exploit the architectural features of DMM's. In [8] and [35], they also consider DMM's for neural network simulations. However, they assume that the network can be partitioned into groups of neurons such that the connectivity between the neurons within a group is much higher than the overall network connectivity, an assumption which does not suitably model fully connected neural networks. In [37] and [38], they study the mapping of a BP algorithm onto transputer-based DMM's. In this paper, we discuss the design and implementation issues in mapping a BP algorithm onto a NOW's. We also study the performance of our formulation on an Ethernet network of Sun 3/50 workstations.

The paper is organized as follows. In Section II, we introduce the BP algorithm and discuss the issues involved in parallelizing the algorithm. In Section III, we describe our distributed algorithm and analyze its time complexity. In Section IV, we present our performance results and compare them with a recently proposed BP algorithm implemented on DMM's [38]. Finally, in Section V we make some concluding remarks.

II. The Backpropagation Algorithm

The BP algorithm is a supervised training algorithm for multilayered feedforward neural networks. The training data consists of many pairs of input/output training patterns. The trained neural network is then used later in the retrieval phase to process real test patterns and yield classification results. We have employed the BP algorithm for training a fully connected multilayered neural network. The critical issue in executing this BP algorithm on a distributed system of NOW's is determining how to map the neural network onto the processors of the distributed system to minimize execution time. In this section, we will introduce the fully connected multilayered neural network, describe the sequential BP algorithm for its training, and

Manuscript received March 18, 1995; revised May 18, 1996, April 9, 1997, and April 10, 1998.

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India.

Publisher Item Identifier S 1083-4419(98)09256-5.

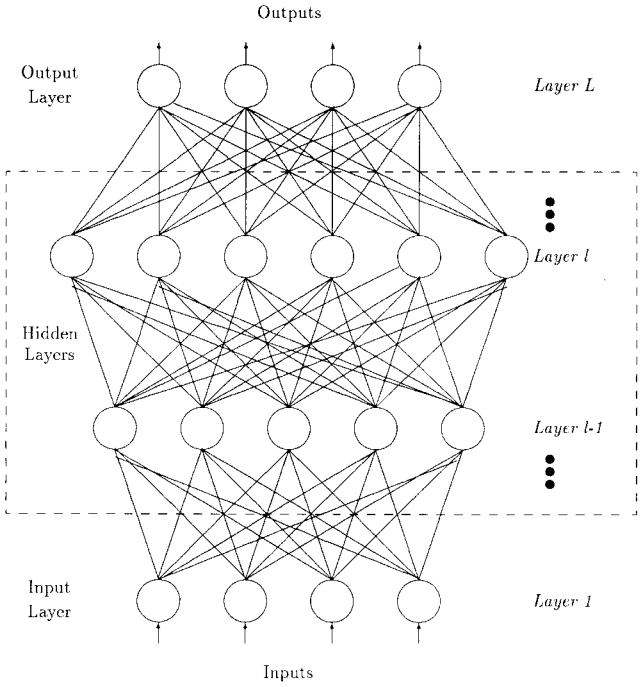


Fig. 1. A fully connected multilayered neural network.

discuss various parallel formulations of the BP algorithm available in the literature.

A. The Multilayered Neural Network

The neural network for which we consider mapping the BP algorithm is a fully connected multilayered neural network. The network consists of L layers as shown in Fig. 1. The bottom layer $(l = 1)$ is the input layer and the top layer $(l = L)$ is the output layer. The remaining intermediate layers are called hidden layers. The lth layer has n_l neurons. Each neuron in a layer is connected to all of the other neurons in the previous and next layers. Associated with each neuron i in layer l is an activation value $a_i(l)$ and an error value $\delta_i(l)$. Attached with each connection between neuron j in layer $(l+1)$ and neuron i in layer l is a weight $w_{ji}(l)$.

B. The Sequential BP Algorithm

The BP algorithm is a supervised training algorithm which uses a set of input/output training patterns to train a multilayered neural network. The training can be viewed as a procedure to find a set of weights for the network. The algorithm has three phases: feedforward execution, backpropagation of error, and weight update. In the feedforward phase, the input portion of a training pattern is fed to the input layer of the network. It is propagated through the layers to compute the activation values of the nodes in each layer. The difference between the activation values of nodes in the output layer and the expected output value (output part of the training pattern) defines the error in the output layer. In the backpropagation of error phase, the error in the output layer is propagated to the nodes in the layers below it to compute the error associated with each neuron in the layers below. The third phase updates the weights based upon the new error and activation values.

In the feedforward execution phase, the activation value of a neuron j at layer $(l+1)$, denoted by $a_j(l+1)$, is given by the following

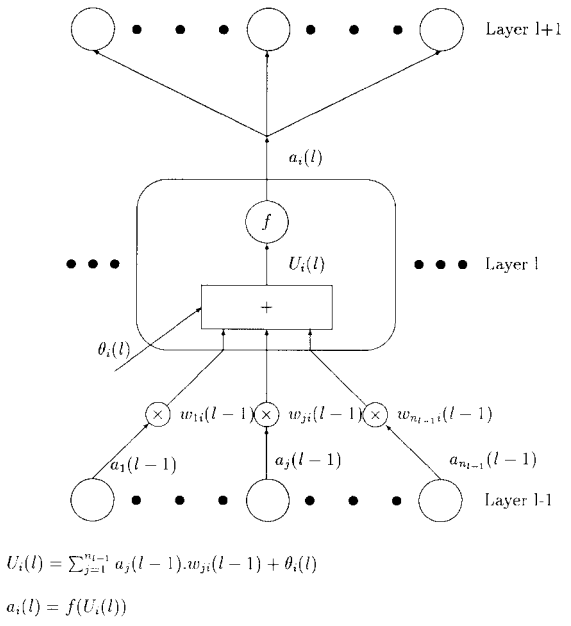


Fig. 2. Forward execution phase.

feedforward equation,

$$a_j(l+1) = f\left(\sum_{i=1}^{n_l} w_{ji}(l)\cdot a_i(l) + \theta_j(l+1)\right), \quad j = 1,\cdots,n_{l+1} \quad \text{and} \quad l = 1,\cdots,L-1, \tag{1}$$

where f is a nonlinear sigmoid function of the form $f(x) = (1 + e^{-x})^{-1}$. This is represented in Fig. 2. We note that the computation of the activation value of a neuron in layer $(l+1)$ is a function of the activation values of all the neurons in layer l and the weight values of the connections joining them, i.e., the input weights of the neuron.

The second phase involves the comparison between the actual output pattern and the desired one, and the propagation of the error, which is governed by the following equations:

$$\delta_i(l) = [t_i(l) - a_i(l)]\cdot[a_i(l)\cdot(1-a_i(l))], \qquad l = L \tag{2}$$

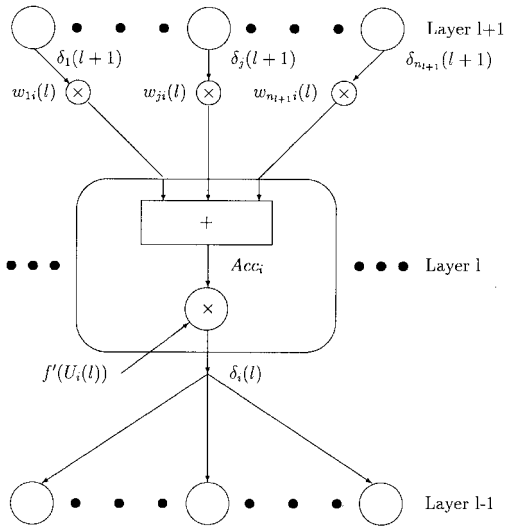
$$= \left[\sum_{j=1}^{n_{l+1}} \delta_j(l+1)\cdot w_{ji}(l)\right] \cdot [a_i(l)\cdot(1-a_i(l))], \qquad l = L-1,\cdots,1 \tag{3}$$

where $\delta_i(l)$ is the error value of neuron i in layer l and $t_i(L)$ is the desired value of neuron i in the output layer. We note that the computation of the error value of a neuron in layer l is a function of the error values of all the neurons in the layer $(l+1)$ and the weight values of the connections joining them, i.e., the output weights of the neuron.

In the final phase, weight updates are performed according to the following equation:

$$\Delta w_{ji}(l) = \eta\cdot\delta_j(l+1)\cdot a_i(l) \tag{4}$$

where η is the learning rate. We see that the weight update of a connection between a neuron in layer $(l+1)$ and a neuron in layer l is a function of the error value of the neuron in layer $(l+1)$ and the activation value of the neuron in layer l. The second and third phases are depicted in Fig. 3. We also see from this figure that the second and third phases can be combined into a single phase, called the backward execution phase.



$$\delta_i(l) = f'(U_i(l)).Acc_i$$

$$Acc_i = \sum_{j=1}^{n_{l+1}} \delta_j(l+1).w_{ji}(l)$$

$$w_{ji}(l) = w_{ji}(l) + \eta\delta_j(l+1)a_i(l)$$

Fig. 3. Backward execution phase.

C. Parallel Schemes for BP Algorithm

In this section we present a brief taxonomy of the existing schemes to parallelize a BP algorithm. Fig. 4 gives a classification of the existing BP algorithms. We notice that the parallelization schemes for a BP algorithm can be broadly classified into four categories: network partitioning, pattern partitioning, hybrid partitioning integrating the previous two strategies, and heuristic partitioning schemes. The network partitioning schemes take advantage of the inherent parallelism in the node and the weight value computations present in the BP algorithm. These schemes distribute both the nodes and the weights of the network to different processors. This distribution of nodes and weights can be carried out in various ways. Nodes of the multilayered network can be either completely partitioned, i.e., each node is assigned to a different processor [1], or can be partitioned using the vertical sectioning scheme where each processor gets some nodes from each layer [37], [38]. The weights of the neural network can be partitioned using four strategies: complete partitioning, inset grouping, outset grouping, and checkerboarding. Complete partitioning allocates one processor per weight [1] and exploits maximum concurrency in the weights computation but suffers from heavy communication overhead. The inset and outset grouping schemes are used with the vertical sectioning scheme for partitioning the nodes. Each processor keeps either the input or the output weights connecting all the nodes mapped onto it. We note that while inset grouping reduces communication during the activation value computation (forward phase) [40], the outset grouping decreases communication overhead during the error propagation phase (backward phase). Both inset and outset grouping can be combined to increase efficiency in both the forward and backward phases [37], [38], though this introduces an additional overhead in maintaining the consistency of the redundant weight sets. Checkerboarding partitions the weights of a neural network by grouping the rows and the columns of the weight matrix. It has been used for systolic arrays connected in a mesh configuration [18] and for the nCUBE, a hypercube configuration [15].

Pattern partitioning divides the pattern set equally among all processors. This division of the patterns can be achieved either by replicating the network nodes and weights at each processor, where each processor carries out the forward and backward phases for the local set of patterns [35], or by pipelining the computation at each layer, i.e., while one pattern is being processed in some layer, another pattern can be processed in the preceding layer [18].

Hybrid schemes combine pattern partitioning with network partitioning. For example, some implementations include, pipelining combined with vertical sectioning [18], combining vertical sectioning with pattern partitioning involving network duplication [40], and checkerboarding combined with the network duplication scheme [15].

Heuristic partitioning techniques involve the use of heuristics for partitioning the neural network graph for mapping onto a general or specific architecture represented by a processor graph [8], [18], [35]. These techniques try to find an efficient way of partitioning/mapping the neural graphs in such a way that it reduces the inter-processor communication and balances the load on the processors.

In this paper, we employ a vertical sectioning scheme for partitioning the nodes, and a combination of inset and outset grouping of weights, for efficient mapping of multilayered neural network onto a NOW's. This scheme is particularly suitable for a NOW's, because it is architecture independent.

D. Training Regimes

There are two variations in training regimes available in the literature for training a neural network using a BP algorithm: the per-pattern or data-update training regime and the set-training or the block-update training regime. In the per-pattern training regime, the weight changes computed for a particular pattern are affected before processing the next pattern [29]. We note that the per-pattern training regime is not amenable for the pattern partitioning mapping technique for parallelization of the BP algorithm. The set-training regime accumulates the weight changes over a set of patterns before applying these to update the weight values [15]. Set-training regimes can be utilized with both the pattern partitioning and the network partitioning schemes for mapping the BP algorithm onto a multiprocessor. The set-training scheme may produce results which differ from the results obtained by the per-pattern training regime and may also take a greater number of iterations than the per-pattern training regime for convergence, a fact ignored by many researchers while presenting performance comparison figures. We have used the per-pattern training regime for training our network.

III. Our Distributed Algorithm

We now investigate a distributed implementation of the BP algorithm on a NOW's for training a fully connected multilayered neural network. In this section, we first describe our partitioning strategy, then discuss our fully distributed implementation, and finally present a speedup analysis for the algorithm.

A. Partitioning Scheme

We have used a vertical partitioning scheme for partitioning the node set of the multilayered neural network in our implementation. We have used a per-pattern training regime to train the neural network and only a network partitioning scheme can be used for per-pattern training of a neural network. In the vertical partitioning scheme, each layer (l), having n_l neurons, is divided into p partitions, where p is the number of processors in our distributed implementation. Each partition has n_l/p neurons that are assigned to a processor as shown in Fig. 5. We have adopted a combination of the inset grouping and outset grouping for partitioning the weight set of the neural network. Each processor maintains in its local memory the activation values, the error values, and the input and the output weight vectors of

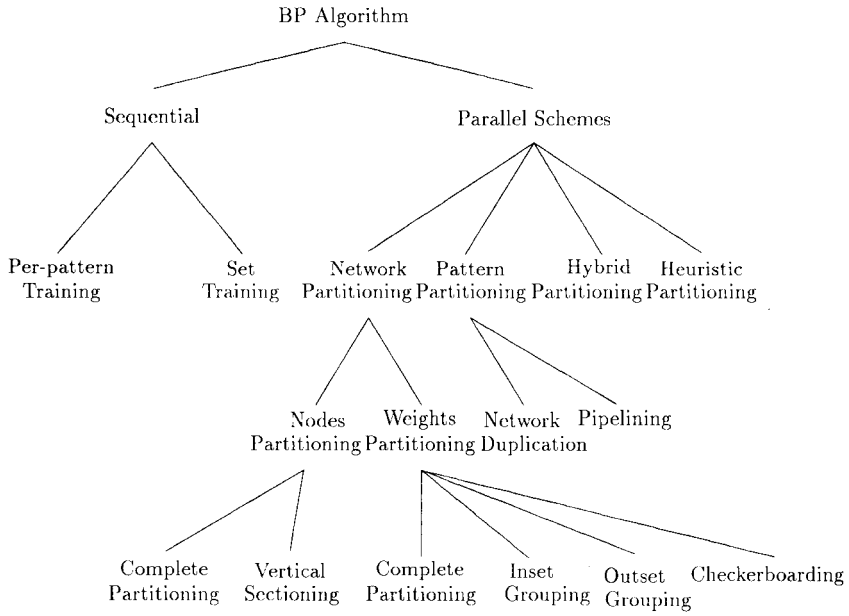


Fig. 4. Classification of the various BP algorithms.

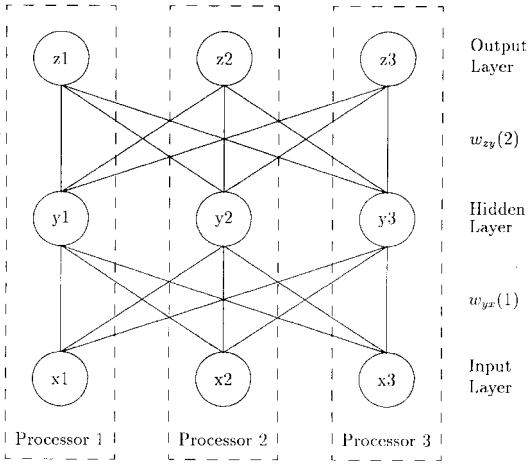


Fig. 5. Vertical partitioning of a multilayered neural network.

the assigned neurons. We observe that the input weight values of neurons in layer $(l+1)$ are the output weight values of the neurons in layer l, and, hence, in our scheme, the same weight value is stored at two processors. Even though, this partitioning scheme results in the duplication of weight values, we prevent inconsistency amongst them by employing recomputation of the weights and thus avoid communication of the weight values during the execution of the algorithm. We also note that all the activation values and the error values are completely partitioned into p disjoint sets.

B. The Distributed Algorithm

Since we have employed a partitioning strategy where the neural network is partitioned into p subnetworks and then mapped onto p processors, each processor is required to cooperate with every other processor to simulate the complete network. Every processor in the distributed system executes the three phases mentioned in Section II-B, but some inter-processor communication is required to acquire the activation and error values of neurons present on other processors as the data in a layer is distributed to all the processors. Thus, each phase is conceptually divided into two subphases: communication and computation. As noted earlier, the second and third phases of the BP algorithm can be combined into a single phase. Hence, we present our distributed algorithm in only two phases, which we call the forward execution phase and the backward execution phase. We will discuss below the necessary computations and communications involved in both the phases of our algorithm.

The Forward Execution Phase: In this phase, the activation values of all the neurons in the hidden layers and the output layer are computed. To compute the activation value of a neuron i in layer l, $a_i(l)$, we require the activation values of all the neurons in layer $(l-1)$ and the input weights of the neuron i (1). So, before we start computing the activation values of the neurons on a processor for the layer l, we broadcast the activation values of all the neurons present on the local processor in layer $(l-1)$ and receive broadcasts from all the other $(p-1)$ processors. After this communication step is complete, all the processors will have all activation values of the neurons in layer $(l-1)$, so that the activation values of the neurons local to the processor for layer l can be computed. This broadcast and receive step, in which a set of distinct messages initially residing at each processor is disseminated so that eventually a copy of each message resides with all the processors, is called *all-to-all broadcasting* (AAB) [16]. For an Ethernet network of workstations (NOW's), this can be achieved in p steps, one for broadcasting the local values and $(p-1)$ steps to receive messages from the rest of the processors. We describe our algorithm for the forward execution phase below.

Distributed forward execution algorithm
Read the input pattern.
for (*start* $=1$ to $(L-1)$) **do**
stop $=$ *start* $+1$.
Broadcast local activation values of *start* layer.
Receive activation values from the other $(p-1)$ processors.
Compute activation values of *stop* layer.

The Backward Execution Phase: As seen earlier, the backward execution phase comprises the back propagation of the error at the output layer and the update of weights. After the completion of the

forward execution phase we have the activation values of the neurons in the local processor for the output layer. So, for the neurons in the output layer, we can compute the error values using (2). When computing the error value $\delta_i(l)$ for a neuron i local to the processor for a layer l, we require the error values at all the neurons in the layer $(l+1)$ and the output weights of the neuron i (3). Hence, in this phase, too, we have to execute a similar broadcast/receive step as in the previous phase, for broadcasting the error values at neurons local to the processor for the layer $(l+1)$ and receiving the same from all the other $(p-1)$ processors. For updating a weight $w_{ji}(l)$ of a connection joining a neuron j in layer $(l+1)$ and a neuron i in layer l, we need the error value at the neuron j and the activation value of the neuron i. We observe that, the activation values of the neurons in the layer l are propagated to all the processors in the forward execution phase and the error values of the neurons in the layer $(l+1)$ are propagated to all the processors in the backward execution phase. Hence, we avoid communication of any activation or error values while updating the weight values. Starting from layer $(L-1)$, for all layers until the input layer, at every step we update the output weights of neurons local to the processor for the current layer l and the input weights of the neurons local to the processor for layer $(l+1)$. Hence, at each step, we compute the input and output weights of all the neurons mapped to the local processor, connecting the current layer l and the layer $(l+1)$. At the end of this phase, each weight has been computed twice: once as an output weight and once again as an input weight. Assume that neuron j in layer $(l+1)$ and neuron i in layer l are mapped onto processor p_x and processor p_y, respectively. The difference in weight, $\Delta w_{ji}(l)$, can be computed at processor p_x, if the processor knows the activation value $a_i(l)$ since it already has the error value $\delta_j(l+1)$ stored locally with it. Similarly, the same difference, $\Delta w_{ji}(l)$, can be computed at processor p_y, if it knows the error value $\delta_j(l+1)$, since it already has the activation value $a_i(l)$ stored locally with it. Both the above updates are identical and the consistency of the two updates is guaranteed, as the weight update value computed in both cases uses the same error and activation values for computation. We have also carefully avoided recomputing the weights joining the neurons present on the same processor. For a given processor, for weights between the layer $(l+1)$ and layer l, we compute the weight change for all the weights which connect a neuron local to the processor and a neuron which is mapped onto some other processor, and for weights which connect local neurons, we compute the weight change only once. In earlier implementations [37], [38], the weights connecting the local neurons were updated twice, once as an input weight to neurons in the layer $(l+1)$ and the next time as an output weight to neurons in the layer l. We save on recomputation of the weights connecting the local neurons as these are computed only once. We present our algorithm for the backward execution phase as follows.

Distributed backward execution algorithm

```
Compute error values of the local neurons at the
output layer.
for (stop = L to 2) do
  start = stop −1.
  if (start > 1) do
    Broadcast local error values of stop layer.
    Receive error values from other processors.
    Compute error values for the start layer.
  for (j = 1 to n_stop) do
    for (i = 1 to n_start) do
      if (i == local or j == local)
        update w_ji(start).
```

We notice from the above (distributed forward and backward execution) algorithms that we require two sets of communications, one in the forward execution phase for the activation values, and the other in the backward execution phase for the error values.

C. Improvements Over a Recent Algorithm

As mentioned earlier, we adopted a simple vertical partitioning scheme for mapping the multilayered neural network onto a NOW's. One critical issue here is that of mapping the data, especially the weights, onto the processors of the network. A partitioning scheme in which the processor keeps either the input or the output weights of the neurons mapped onto it incurs a significant amount of communication overhead in the backpropagation or the feedforward cases [23], [40]. The logical extension to this scheme is to keep both the input and the output weights of the neurons in the same processor [37]. Here too, there is heavy communication overhead incurred in order to maintain consistency among the replicated weight set, if weights are to be communicated. A modification to the above is proposed in [38], where weight recomputation is suggested as a compromise for weight communication in order to update the duplicated weight set. This algorithm makes use of three sets of communication, each in the forward execution, error propagation, and weight update phase, and four sets of computation for one training cycle of the algorithm which consists of activation values, error values, and weight update computations. Our algorithm makes use of only two sets of communications, one each in the forward execution and backward execution phases; moreover, it also avoids some redundancy in weight update during the weight recomputation phase, per cycle of the algorithm. We also employ a grouped-broadcast strategy to broadcast all the values at a processor, instead of the one-by-one AAB employed in [37], to reduce the communication setup time.

The following are the improvements of our algorithm over a recent algorithm [37], [38] available in the literature employing a vertical partitioning scheme for a distributed implementation of the BP algorithm.

1) Our algorithm uses only two sets of communication, compared to the three sets of communication used by the earlier algorithm.
2) We also save on the computation of the weights, by avoiding recomputing weights joining the neurons on the same processor.
3) We employ grouped-broadcast strategy, to broadcast all the values at a processor, instead of the one-by-one AAB, to reduce the communication setup time.

D. Speedup Analysis

For the time complexity analysis of our model we assume a fully connected multilayered neural network with L layers and, without loss of generality, n neurons per layer. The neural network is partitioned vertically into p partitions and mapped onto p processors with one partition per processor. There are n/p neurons from each layer per processor.

The time required for the sequential execution of the BP algorithm on a uniprocessor for one layer can be represented as $T_1 = t_1 + t_2 + t_3$, where t_i is the time taken to execute the ith phase of the BP algorithm for the layer. The times can be approximately expressed as follows:

$$\begin{aligned} t_1 &= n \cdot (n \cdot M_a + F) \\ t_2 &= n \cdot (n \cdot M_a) \\ t_3 &= n \cdot (n \cdot M_a) \\ T_1 &= t_1 + t_2 + t_3 \\ &= n \cdot (3 \cdot n \cdot M_a + F) \end{aligned} \tag{5}$$

where M_a is the time taken for one multiply and one add function for multiplying and adding two floating point numbers, and F is the time taken to execute the sigmoid function. We have ignored for simplicity, and without loss of generality, the time taken for the addition of θ, the neuron threshold value.

We now analyze our distributed algorithm having two phases, the forward execution and the backward execution. The time taken for executing our distributed BP algorithm for a layer of n/p neurons on p processors can be represented as, $T_p = t'_1 + t'_2$, where t'_i is the time taken for the execution of the ith phase on a processor, which can be expressed as follows:

$$\begin{aligned} t'_1 &= AAB(p) + \frac{n}{p} \cdot (n \cdot M_a + F) \\ t'_2 &= \left(AAB(p) + \frac{n}{p} \cdot (n \cdot M_a) \right) \\ &\quad + \frac{n}{p} \cdot \left(n \cdot \left(2 - \frac{1}{p} \right) \cdot M_a \right) \\ &= AAB(p) + \frac{n}{p} \cdot \left(3 \cdot n \cdot M_a - \frac{n}{p} \cdot M_a \right) \\ T_p &= t'_1 + t'_2 \\ &= 2 \cdot AAB(p) + \frac{n}{p} \cdot \left(4 \cdot n \cdot M_a + F - \frac{n}{p} \cdot M_a \right) \end{aligned} \tag{6}$$

We will now take a look at the AAB employed by our algorithm. We too assume as in [38] a single-port communication, where, each workstation in the NOW's can send/receive one unit of message on its communication port at a given instance of time. This sets a lower bound of $O(p)$ for the AAB, because each workstation has to receive data from all the other $(p-1)$ workstations. This bound for AAB can be achieved on an Ethernet NOW's. The unit of time for sending/receiving one unit of data (a floating point number) is defined as C. We have assumed a grouped broadcast for modeling communication, where all the values on a processor are grouped together and broadcast as one message. This reduces the overhead for processing each broadcast. Hence an $AAB(p)$ to broadcast a message of size n/p to all the p processors, can be represented by, $AAB(p) = C \cdot p \cdot f(n/p)$, where $f(n/p)$ represents the scaling of the grouped broadcast as the items in the group to be broadcast are increased. Normally $f(n/p)$ is much less than n/p which is the worst case for one-by-one $AAB(p)$.

From (5) and (6), speedup, S_p, of our algorithm can be formulated as below:

$$\begin{aligned} S_p &= \frac{T_1}{T_p} \\ &= \frac{n \cdot (3 \cdot n \cdot M_a + F)}{2 \cdot AAB(p) + \frac{n}{p} \cdot \left(4 \cdot n \cdot M_a + F - \frac{n}{p} \cdot M_a \right)} \\ &= \frac{n \cdot (3 \cdot n \cdot M_a + F)}{2 \cdot C \cdot p \cdot f\left(\frac{n}{p}\right) + \frac{n}{p} \cdot \left(4 \cdot n \cdot M_a + F - \frac{n}{p} \cdot M_a \right)} \\ &\quad \cdot \left(by\ AAB(p) = C \cdot p \cdot f\left(\frac{n}{p}\right) \right) \\ &= \frac{n \cdot (3 \cdot n \cdot M_a + \beta \cdot M_a)}{2 \cdot \alpha \cdot p \cdot M_a \cdot f\left(\frac{n}{p}\right) + \frac{n}{p} \cdot \left(4 \cdot n \cdot M_a + \beta \cdot M_a - \frac{n}{p} \cdot M_a \right)} \\ &\quad (by\ C = \alpha \cdot M_a, F = \beta \cdot M_a) \\ &= \frac{n \cdot (3 \cdot n + \beta)}{2 \cdot \alpha \cdot p \cdot f\left(\frac{n}{p}\right) + \frac{n}{p} \cdot \left(4 \cdot n \cdot + \beta - \frac{n}{p} \right)} \end{aligned} \tag{7}$$

TABLE I
CHARACTERISTICS OF THE APPLICATION

	Structure (Input × Hidden × Output)	Total Connections	Input Patterns
Num 6 × 8	48 × 48 × 6	3592	6 × 8 Numbers
Num 12 × 12	144 × 144 × 12	20736	12 × 12 Numbers

In (7) the most important parameter is the communication/computation ratio α. It lies between 0.5 and 256 for various currently available architectures. For a NOW's this value is very high and lies in the range 32–256.

IV. PERFORMANCE EVALUATION

In this section we present the results obtained from implementing our distributed algorithm on a network of Sun workstations. We compare the results with those obtained by Yoon *et al.* [38]. We also conduct an analytical comparison of the two algorithms for studying the scaleup of the algorithms with an increasing number of processors and size of the neural network.

A. Experimental Comparison

In this subsection we present the performance of our algorithm on a NOW's and also compare the results with a recently proposed BP algorithm for a DMM [38]. We implemented both algorithms on a 10 Mb/s Ethernet network of Sun 3/50 workstations. We conducted several experiments to obtain a suitable value for α. We have found that for a grouped AAB, the cost of communication is 50 to 60 times the cost of a multiply-and-add operation, given the size of messages in our experiments. Since the variation in the cost of communication is very small with an increase in the size of the message for a grouped AAB, we have assumed the value of $C \cdot f(n/p)$ to be 55, for our analytical studies. We have kept the value of β at 40 as in the earlier implementation [38]. We have tested the algorithms for classifying arabic numeral digits. The characteristics of the application are listed in Table I. Binary images of the numbers are used as inputs to a neural network with 3 layers. *Structure* represents the number of neurons in the input, the hidden, and the output layer. We evaluated the performance of the algorithms for classifying 6×8 numbers (*Num* 6×8) and 12×12 numbers (*Num* 12×12).

Fig. 6 shows the speedup of the algorithms for the application (*Num* 6×8). We observe from the graph that, our experimental values closely agree with the analytical curve. The difference in the analytical and experimental speedup can be attributed to the assumptions made to simplify the derivation of the speedup factor. We ignore the processing time for θ, and deal with a simplified communication to computation ratio while computing the speedup factor for our distributed algorithm. We have experimented with small values in the number of processors to fully exploit the power of the distributed system, i.e., to keep the processors uniformly balanced [15], [38]. It is seen that, with an increase in the number of processors, the vertical partitioning scheme does not uniformly balance the distributed system, resulting in degradation in performance [15].

Fig. 7 shows the speedup of the algorithms for the (*Num* 12×12) application. We see that, for a small value in the number of processors, our algorithm shows only slight improvement over the algorithm of Yoon *et al.* because there is much less communication overhead, and the slight gain is attributed to the saving in the recomputation step. As the number of processors increases, the performance of our algorithm improves. The experimental curve tends to agree with the analytical

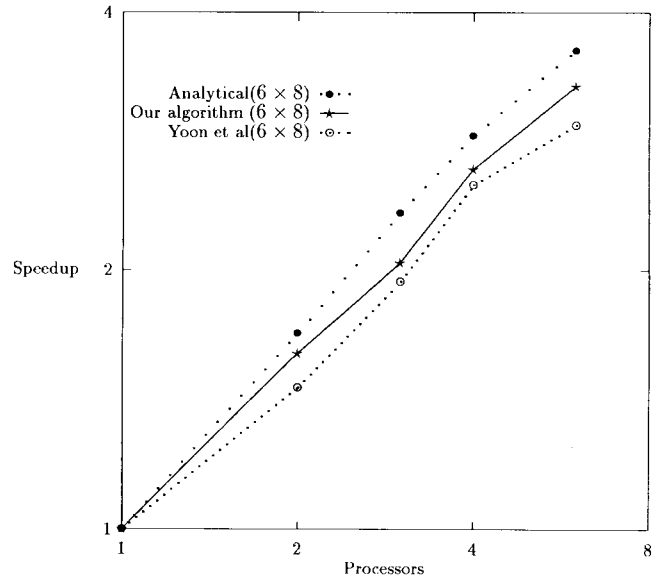


Fig. 6. Experimental speedup for Num 6×8.

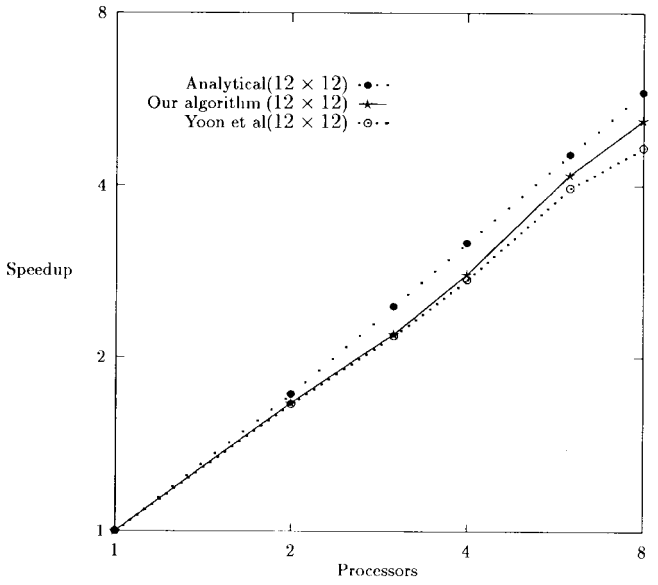


Fig. 7. Experimental speedup for Num 12×12.

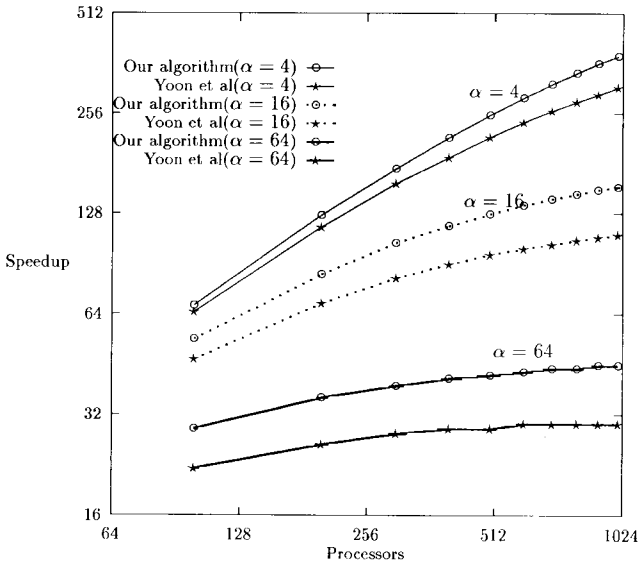


Fig. 8. Analytical speedup when $n = 2048$.

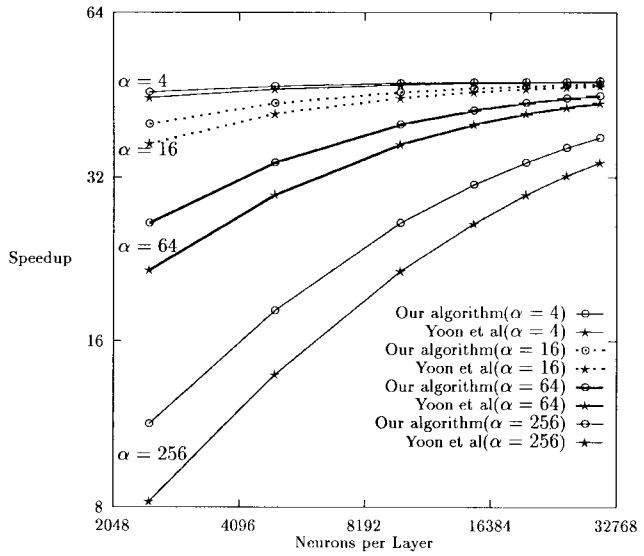


Fig. 9. Analytical speedup when $p = 64$.

plot, with a marginal error, attributed to simplification steps during the speedup factor derivation. We used log-log graphs to plot the relative performance of the two algorithms [2].

B. Analytical Comparison

Because of system limitations such as main memory size and the broadcast packet size, we could not study the experimental scaleup of our algorithm. We have, however, compared the analytical models of our algorithm and the algorithm of Yoon *et al.* [38] to study the speedup of the algorithms against the number of processors and the number of neurons in the neural network. Equation (7) gives the speedup analysis for our algorithm. The speedup analysis for the algorithm of Yoon *et al.* can be found in [38]. We have assumed $f(n/p)$ to be equal to (n/p), when comparing the two analytical models. The theoretical speedups for both the algorithms for various α values are shown in Fig. 8, when $n = 2048$ neurons/layer and $\beta = 40$. The graph clearly shows that our algorithm performs better than the algorithm of Yoon *et al.*, especially for higher number of processors and large α values. Note that for loosely coupled distributed systems, the α values are very high. We had an α value of 55 for workstations without math co-processors. With faster processing workstations, the α values tend to be higher, and hence the need for having an algorithm minimizing the communication overhead. We also note from the graph that there is a cost-effective number of processors depending upon the α values, where, even if more processors are added to the simulation, the speedup ratio does not increase significantly.

Fig. 9 shows the comparison of the speedups of the algorithms versus the size of the neural network, when p, the number of processors, is fixed at 64. We note that the speedup is heavily influenced by α when the size of the network is small, and, hence, for smaller values of n, our algorithm performs better than the algorithm of Yoon *et al.* Similarly for larger values of α our algorithm performs better than their algorithm, because of the saving in the communication. We plotted the above graphs too on the log-log scale.

V. Conclusions

We have presented an efficient distributed algorithm for implementing the BP algorithm for training neural networks on a network of Sun 3/50 workstations. We have used the vertical partitioning scheme to map the multilayered neural network onto the NOW's, since a vertical partitioning scheme is amenable for per-pattern training of a multilayered neural network.

We have compared the results obtained from our implementation with a recent algorithm suggested by Yoon *et al.* [38], employing vertical partitioning scheme for implementation on a distributed memory multiprocessor. Because of hardware limitations we could not study the scaleup of our algorithm on the NOW's. Hence, we also conducted analytical studies to compare the speedup of the two algorithms with the increase in the number of processors and with the increase in the size of the neural network.

Our experimental and analytical results show that we are able to achieve better speedups than the algorithm of Yoon *et al.* We improved the communication time by using only two sets of communication instead of three used by Yoon *et al.*, per cycle of the training algorithm. We also made saving in computation by eliminating some redundancy in the recomputation step during computation of the weights. We also made use of a grouped AAB to broadcast all the values on a processor as a group, to decrease the setup time for communication.

References

[1] G. Blelloch and C. R. Rosenberg, "Network learning on the connection machine," *10th Int. Joint Conf. Artificial Intelligence*, 1987, pp. 323–326.

[2] L. A. Crowl, "How to measure, present, and compare parallel performance," *IEEE Parallel Dist. Technol.*, Spring 1994, pp. 9–25.

[3] E. Deprit, "Implementing recurrent backpropagation on the connection machine," *Neural Networks*, vol. 2, pp. 295–314, 1989.

[4] S. E. Fahlman and G. E. Hinton, "Connectionist architectures for artificial intelligence," *IEEE Comput.*, Jan. 1987, pp. 100–109.

[5] J. A. Feldman *et al.*, "Computing with structured connectionist networks," *Commun. ACM*, vol. 31, no. 2, pp. 170–187, 1988.

[6] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace, and G. V. Wilson, "Implementing neural network models on parallel computers," *Comput. J.*, vol. 30, no. 5, pp. 413–419, 1987.

[7] K. Fukushima, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural Networks*, vol. 1, no. 2, pp. 119–130, 1988.

[8] J. Ghosh and K. Hwang, "Mapping neural networks onto message passing multicomputers," *J. Parallel Dist. Comput.*, vol 6, pp. 291–330, 1989.

[9] R. P. Gorman and T. J. Sejnowski, "Learned classification of sonar targets using a massively parallel network," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 36, pp. 1135–1140, July 1988.

[10] S. Haykin and C. Deng, "Classification of radar clutter using neural networks," *IEEE Trans. Neural Networks*, vol. 2, pp. 589–600, 1991.

[11] J. Hicklin and H. Demuth, "Modeling neural networks on the MPP," in *Proc. 2nd Symp. Frontiers Massively Parallel Computation*, 1988, pp. 39–42.

[12] B. H. Juang, S. Y. Kung, and C. A. Kamm, Eds., *Neural Networks for Signal Processing*, Proc. 1991 IEEE Workshop, Princeton, NJ, 1991, pp. 7–185.

[13] B. Kosko, Ed., *Neural Networks for Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing—Design and Analysis of Algorithms*. New York: Benjamin/Cummings, 1994.

[15] V. Kumar, S. Shekhar, and M. B. Amin, "A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pp. 1073–1090, Oct. 1994.

[16] S. Y. Kung, *Digital Neural Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[17] S. Y. Kung and J. N. Hwang, "Systolic architectures for artificial neural nets," *IEEE Int. Conf. Neural Networks*, 1988.

[18] ____, "A unified systolic architecture for artificial neural networks," *J. Parallel Dist. Comput.*, vol. 6, pp. 358–387, 1989.

[19] S. P. Luttrell, "Image compression using a multilayer neural network," *Pattern Recognit.*, vol. 10, pp. 1–7, 1989.

[20] J. Millan and P. Bofill, "Learning by backpropagation: A systolic algorithm and its transputer implementation," *Neural Networks*, vol. 3, pp. 119–137, 1989.

[21] M. Misra and V. K. P. Kumar, "Neural network simulation on a reduced mesh of trees organization," in *SPIE Symp. Electronic Imaging*, 1990.

[22] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Trans. Neural Networks*, vol. 1, pp. 4–27, Mar. 1990.

[23] D. S. Newhall and J. C. Horvath, "Analysis of text using a neural network: A hypercube implementation," in *Proc. Conf. Hypercubes, Concurrent Computers, Applications*, 1989, pp. 1119–1122.

[24] D. H. Nguyen and B. Widrow, "Neural networks for self learning control systems," *IEEE Contr. Syst. Mag.*, Apr. 1990, pp. 18–23.

[25] M. Nikoonahad and D. C. Liu, "Medical ultrasound imaging using neural networks," *Electron. Lett.*, vol 26, pp. 545–546, Apr. 1990.

[26] B. P. Paris, G. Orsak, M. Varanasi, and B. Aazhang, "Neural net receivers in multiple-access communications," *Adv. Neural Inf. Processing*, vol. 1, pp. 272–280, 1989.

[27] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural network simulation at warp speed: How we got 17 million connections per second," *Proc. Int. Conf. Systolic Arrays*, 1988.

[28] M. W. Roth, "Neural networks for extraction of weak targets in high clutter environments," *IEEE Trans. Syst., Man, Cybern.*, vol. 19, pp. 1210–1217, Sept./Oct. 1989.

[29] D. E. Rumelhart and J. L. McClelland, Eds., *Parallel and Distributed Processing (PDP): Explorations in the Microstructure of the Cognitron*. Cambridge, MA: MIT Press, 1986.

[30] T. J. Sejnowski and C. R. Rosenberg, "Nettalk: A parallel network that learns to read aloud," Tech. Rep. JHU/EECS-86/01, Dept. of Elec. Engg. and Comp. Sci., John Hopkins Univ., Baltimore, MD, USA, 1986.

[31] ____, "Parallel networks that learn to pronounce English text," *Complex Syst.*, vol. 1, pp. 145–168, 1987.

[32] S. Shams and J. Gaudiot, "Efficient implementation of neural networks on the DREAM machine," in *Proc. 11th Int. Conf. Pattern Recognition*, Hague, The Netherlands, 1992, vol. 4, pp. 204–208.

[33] S. Shams and K. W. Przytula, "Implementation of multilayer neural networks on parallel programmable digital computers," in M. Bayoumi, Ed., *Parallel Algorithms and Architectures for DSP Applications*. Norwell, MA: Kluwer, 1991, pp. 225–253.

[34] V. Sunderam, "PVM: A framework for parallel and distributed computing," *Concurrency, Practice, Experience*, vol. 12, pp. 315–339, Dec. 1990.

[35] B. W. Wah and L. Chu, "Efficient mapping of neural networks on multicomputers," *Int. Conf. Parallel Processing*, 1990, pp. I234–I241.

[36] B. Widrow and R. Winter, "Neural nets for adaptive filtering and adaptive pattern recognition," *IEEE Computer*, vol. 21, pp. 25–39, Mar. 1988.

[37] H. Yoon and J. H. Nang, "Multilayer neural networks on distributed memory multiprocessors," in *Proc. Int. Neural Network Conf.*, Paris, France, July 1990, pp. 669–672.

[38] H. Yoon, J. H. Nang, and S. R. Maeng, "Parallel simulation of multilayered neural networks on distributed memory multiprocessors," *Microprocess. Microprogr.*, vol. 29, pp. 185–195, 1990.

[39] M. Zeidenberg, *Neural Networks in Artificial Intelligence*. London, U.K.: Ellis Horwood, 1990.

[40] X. Zhang and M. McKenna, "The backpropagation algorithm on grid and hypercube architectures," Tech. Rep. RL90-9, Thinking Machines Corp., 1990.

[41] Y. Zhou, R. Chellappa, A. Vaid, and R. K. Jenkins, "Image restoration using a neural network," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 36, pp. 1141–1151, July 1988.