Parallel Implementation of the Backpropagation Algorithm on Hypercube Systems

Cevdet Aykanat, Kemal Oflazer, Radwan Tahboub

Department of Computer Engineering and Information Science
Bilkent University
Bilkent, Ankara, 06533 Turkey

Abstract Backpropagation is a supervised learning procedure for a class of artificial neural networks which has recently been widely used in training such neural networks to perform relatively nontrivial tasks like text-to-speech conversion or autonomous land vehicle control. However, the slow rate of convergence of the backpropagation algorithm has limited its application to rather small networks and various researchers have implemented parallel versions on a number of different parallel platforms. This work presents experimental speed-up performance results from a parallel implementation of the backpropagation learning algorithm on an Intel iPSC/2 hypercube parallel processor, for such well-known neural nets like NETTalk and extrapolated speed-up results for large scale hypercube systems from analytic performance models of the implementation.

Keywords: Parallel Neural Network Training, Backpropagation Algorithm, Neural Networks, Hypercube, Parallel Processing.

Introduction

Backpropagation is a supervised learning procedure for a class of artificial neural networks [15]. It has recently been widely used in training such neural networks to perform relatively nontrivial tasks (e.g., text-to-speech conversion [16], autonomous land vehicle control [10].) However, the slow rate of convergence of the basic backpropagation algorithm coupled with the substantial computational requirements has limited its application to rather small networks. Hence, the backpropagation algorithm for training large networks is a good candidate for parallelization.

In this work, we present efficient parallel backpropagation algorithms for hypercube architectures. The proposed parallel algorithms are implemented on an 8-processor Intel iPSC/2 multicomputer available at Bilkent University. Using a combination of network and training set partitioning and a parallel algorithm that minimizes the amount of communication, our approach can utilize a large number of processors to achieve an almost linear speed-up for large networks and large training sets. Detailed performance models for alternate parallel processing approaches extrapolate the performance of large scale hypercube systems while training large networks. Using these models the combination of the parallel processing techniques to get the best speed-up for a given network can be predicted accurately [1].

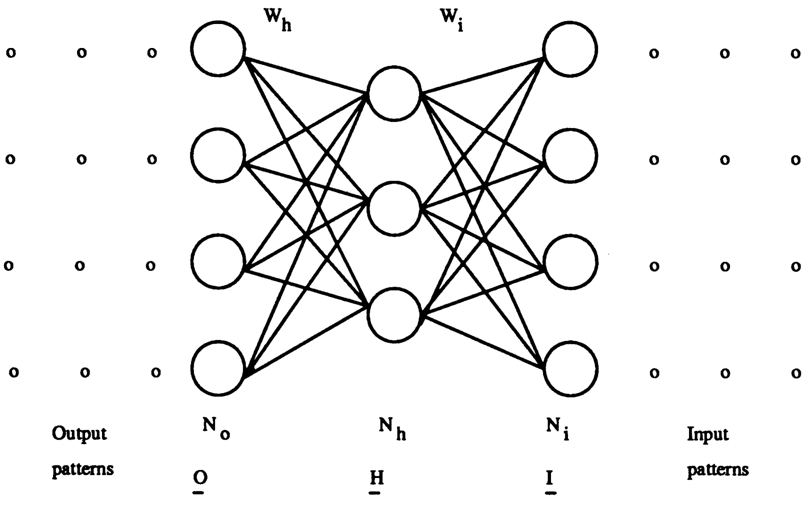


Figure 1: Structure of a simple backpropagation network with a single hidden layer

Artificial Neural Networks

Artificial neural networks constitute an alternative knowledge representation paradigm where knowledge is represented in a distributed fashion by the strength of connections and interactions among a massive number of very simple processing units called *neurons.* [6, 7, 9]. Neural networks "learn" their behavior through a process called *training.* Multilayer perceptrons are extensions to the basic perceptron [14] where the outputs of a perceptron can drive the inputs of other perceptrons. A certain class of multilayer perceptrons has received significant attention in recent years mainly because of the proposal of a supervised learning algorithm known as *backpropagation* [15]. Such networks are *layered* and *fully connected* between layers, as shown in Figure 1. They consist of an *input layer* and *output layer* and a series of *hidden layers.* The input and output layers, as their names suggest, are the layers where the inputs are presented and the outputs are observed. The units in the hidden layer configure themselves into complex feature detectors, computing internal representations of unobvious but salient features of the pattern set. The output of each unit in the input and hidden layers is connected to all the units of the next layer (which may be another hidden layer or the output layer). When an input pattern (a vector of activation values for the input layer) is applied to the network, each node in the hidden and the output layers computes a weighted sum of its inputs and then passes the resulting sum through an activation function which limits the output activation values of that unit to a small range. The activation function in backpropagation networks is slightly different from the hard-limiter activation functions used in perceptrons since the mathematical derivation of the backpropagation algorithm requires that the activation function be continuous and differentiable.[1]

[1]The sigmoid function $y = \frac{1}{1+e^{-x}}$ has these features and has been used in backpropagation networks. Here x denotes the weighted sum that is computed by the neuron unit and y denotes the output of the unit after it goes through this limiting function.

Backpropagation algorithm

The backpropagation algorithm is a supervised learning method for the class of networks described above. Starting with a given network structure, and with weights initialized to small random values, the backpropagation algorithm updates the weights between units as a given set input/output pattern associations – known as the *training set* – is presented to the network. Presentation of all the patterns in the training set to the network is known as an *epoch.* The backpropagation algorithm can be implemented as either an *on-line* or an *off-line* version. In the on-line version, the weights in the network are adjusted after each input/output pair is presented, while in the off-line method the weights are adjusted after an epoch. Training a backpropagation network typically takes many epochs. The network weights may never settle into stable values for certain problems as there is no corresponding convergence theorem for this algorithm. However, in practice most networks converge to configurations with reasonable performance. Below, we will describe the on-line method – the off-line method is very similar.

In on-line training, each input/output presentation consists of two stages:

1. a *forward pass* during which inputs are presented to the input layer and the activation values for the hidden and the output layers are computed, and
2. a *backward pass* during which the errors between the computed and expected outputs are propagated to the previous layers and the weights between the units are appropriately updated.

The following is a formal description of the backpropagation algorithm for networks with a single hidden layer – extension to networks with multiple hidden layers is straightforward. Let N_i, N_h and N_o be the number of units at the input, hidden and the output layers of a network respectively. The matrix formulation of the backpropagation learning algorithm is presented below for the sake of clarity in the presentation of the parallelization phase (See [15, 1] for a detailed description which is not presented here due to space limitations). The forward pass of the backpropagation computation is essentially a sequence of matrix-vector multiplications with intervening applications of the sigmoidal activation function to each element of the resulting vectors. For example, for a network with one hidden layer, N_i inputs, N_h hidden units and N_o output units, the outputs of the first hidden layer $\mathbf{h} = [h_1, h_2, \ldots, h_{N_h}]^T$ can be written as

$$\begin{aligned} \mathbf{r^h} &= \mathbf{W_I} \times \mathbf{I} & (1) \\ \mathbf{h} &= Sig(\mathbf{r^h}) & (2) \end{aligned}$$

where $\mathbf{W_I}$ denotes the $N_h \times N_i$ matrix[2] of weights between the input and the hidden layers. Here, the input-vector $\mathbf{I} = [i_1 = 1, i_2, \ldots, i_{N_i}]^T$ denotes the activation values at the input layer for a particular input pattern. The function Sig denotes the sigmoidal activation function applied to each element of the resultant vector $\mathbf{r^h}$. For the output layer one can similarly write

$$\begin{aligned} \mathbf{r^o} &= \mathbf{W_H} \times \mathbf{h} & (3) \\ \mathbf{o} &= Sig(\mathbf{r^o}) & (4) \\ \delta^{\mathbf{o}} &= Error(\mathbf{o}). & (5) \end{aligned}$$

Here, $\mathbf{W_H}$ denotes the $N_o \times N_h$ weight matrix between the hidden and the output layers, $\mathbf{o} = [o_1, o_2, \ldots, o_{N_o}]^T$ denotes the output activation vector whose elements are the activation values of the neuron units at the output layer, and $\delta^{\mathbf{o}} = [\delta_1^O, \delta_2^O, \ldots, \delta_{N_o}^O]$ denotes the output

[2]We will assume that the extra bias units are included in the unit counts.

error vector whose elements are the error terms for the neuron units at the output layer. The function *Error* denotes the application of the error function to individual entries of the output activation vector.

During the backward pass, the error terms computed at the output layer are propagated to the hidden layer with:

$$\begin{aligned} \mathbf{s^h} &= \mathbf{W_H^T} \times \delta^{\mathbf{o}} & (6) \\ \delta^{\mathbf{h}} &= Scale(\mathbf{s^h}) & (7) \end{aligned}$$

where $\mathbf{W_H^T}$ is the transpose of the weight matrix between the hidden layer and the output layer. Here, the function *Scale* denotes the scaling performed by multiplying each entry s_i^h of the resultant vector $\mathbf{s^h}$ by $h_i(1 - h_i)$.

Once these error vectors are computed then the weight change matrices, $\triangle\mathbf{W_H}$ and $\triangle\mathbf{W_I}$ are computed. A closer look to these, show that these computations are essentially outer product of two vectors, i.e.

$$\begin{aligned} \triangle\mathbf{W_H} &= \eta\delta^{\mathbf{o}} \times \mathbf{h^T} & (8) \\ \triangle\mathbf{W_I} &= \eta\delta^{\mathbf{h}} \times \mathbf{I^T} & (9) \end{aligned}$$

Then, the weight matrices are updated by performing the following matrix additions.

$$\begin{aligned} \mathbf{W_H} &= \mathbf{W_H} + \triangle\mathbf{W_H} & (10) \\ \mathbf{W_I} &= \mathbf{W_I} + \triangle\mathbf{W_I} & (11) \end{aligned}$$

The off-line version of the algorithm accumulates these changes for all the patterns during an epoch and performs a weight update only at the end of the epoch with the accumulated changes.

The Intel iPSC/2 System

The Intel Personal Super Computer iPSC/2 is a distributed memory multicomputer system. The processors (or nodes) are connected in an p-dimensional hypercube topology with $P = 2^p$ nodes labeled from 0 to $2^p - 1$. Each node is a ($\approx$ 4 MIPS) Intel 80386 based processor with a 80387 math floating point coprocessor and 4 megabytes of memory. The nodes are controlled by a front-end processor (the host), and all communications are done by message passing communication system. This system can be extended up to 32 processors within the same backplane interconnect.

Critical communication issues on the hypercube

One of the crucial limitations of distributed memory parallel processor systems is the speed of interprocessor communication. Unless parallel algorithms reduce interprocessor communication through problem partitioning with judicious use of communication, the benefits of parallel processing can easily be offset by the overhead of communication. Hypercube systems provide a communication facility which can transmit messages between arbitrary processors in a small number of hops (p).

The communication between the processors in iPSC/2 system is done by high speed message passing using the Direct Connect Module facility. The system libraries provide a variety of communication primitives like SEND, RECV, GSUM, GCOL. The GSUM and GCOL functions are higher level function which implement global summation and collection of vectors, respectively, followed by a broadcast. They will be briefly described here because of their critical use in our implementation.

- GSUM: This function does a global vector sum operation. Initially each processor has its own vector of length n, and after calling this function all processors will have a vector of length n containing the sum of all these vectors.
- GCOL: This function does a global concatenation operation . The subvectors from each node processors is concatenated in order and the resulting vector is broadcast back to each processor. The length of the resulting vector will be the summation of the lengths of the initial vectors found in each processor.

The communication time of these two functions depends on the number of processors and the length of the vectors. We have found experimentally that this time is of the form $T(p,N) \approx a + N \times b$ msecs where N is the final length of the vector (in 4-byte words) involved, and a and b depend on the number of processors and vector length ranges and grow logarithmically with the number of processors for a given vector size range. [4].

Parallel Implementation of Backpropagation

Parallelism in backpropagation

The backpropagation algorithm described above has opportunities for parallel processing at a number of levels. The on-line version of the algorithm is more limited than the off-line version in the ways parallel processing can be applied because of the necessity of updating the weights at every step. The following is a list of possible approaches to parallel processing in backpropagation algorithm:

1. Each unit at the hidden and output layers can perform the computation of the weighted sum (which is actually a dot-product computation) using a parallel scheme. For example each multiplication can be performed in parallel and the results can be added with a tree-like structure in logarithmic number of steps. Here, the granularity of parallel computation is a single arithmetic operation like multiplication or addition.
2. Within each layer, all the units can compute their outputs in parallel once the outputs of the previous level are available. Here the granularity of parallel computation is the computation performed by a single neuron unit.
3. With the off-line version where weight updates are performed once per epoch, all the patterns in the training set can potentially be applied to multiple copies of the network in parallel and the resulting errors can be combined together at the end of an epoch. The granularity of parallel computation is a complete forward pass of the pattern through the network followed by the computation of changes in weight (but not the update of the weights.)

Some or all of these parallel processing approaches can be used together depending on the available resources and the configuration of the parallel implementation platform available to improve the training times for complex networks.

Related work

There have been a number of implementations of backpropagation learning algorithm on a number of different parallel platforms. These include implementation of the NETTalk system on the massively parallel Connection Machine [2], the implementation on the WARP array processor for a neural network for road recognition and autonomous land vehicle control [10, 11], the implementation of backpropagation on IBM's experimental GF11 system with 566 processors capable of delivering 11 gigaflops [17], implementations of backpropagation on an array of Transputers [13, 3]. Other massively parallel systolic VLSI emulators for artificial neural networks have also been proposed [12, 8].

The implementations above are on massively parallel machines with substantial computing power. For example the implementation of NETTalk on Connection Machine with 16K processors offers a speed-up of 500-fold over a VAX 11/780 implementation and two-fold over a Cray-2. The implementation on WARP using 10 processors is reported to have at least twice the performance of the original Connection Machine implementation for the same problem. The GF11 implementation for the off-line version of the algorithm is estimated to deliver 50 to 100 times the performance of the WARP implementation.

Network partitioning

This approach essentially partition the backpropagation network to the processors so that each processor gets a portion of the neurons at each layer. This approach is applicable for both the on-line and the off-line versions of the backpropagation algorithm. Here we present the 3–layer network case and extensions to 4-or more layers is straightforward.

Assuming we have $P = 2^p$ processors available, we partition units in each hidden layer and the output layer so that (approximately) the same number of units from each layer are distributed to each processor. This is equivalent to partitioning the weight matrices into strips assigning each strip to a processor. Figure 2 show the basic idea for network partitioning. Each processor is assigned $\lceil N_H/P \rceil$ rows from the matrix $\mathbf{W_I}$.[3] The matrix $\mathbf{W_H}$ is however partitioned **columnwise** with $\lceil N_H/P \rceil$ columns to each processor. The input vector **I** is available globally to all processors and is not partitioned because each processor should be able to access all its elements.

The processors then proceed as follows:

1. Each processor initializes (to small random values) the allocated strip of $\mathbf{W_H}$ and $\mathbf{W_I}$ matrices.
2. After initialization, each processor starts computing the forward pass as follows:
   (a) Using the scalar accumulation scheme, each processor performs a matrix–vector product (Step 1 in Figure 2) to compute its portion of the **H** vector via passing each element of the product vector through the sigmoid function.
   (b) For the hidden layer to output layer computation, each processor (using a vector accumulation scheme) now performs partial matrix–vector product accumulating a partial sum for each element of the output vector. This is followed by a GSUM operation over the processors involved, which sums the processor partial sums and then broadcasts the resulting output vector to all processors (Steps 2 and 3 in Figure 2).

[3]Some of them may get an extra row.

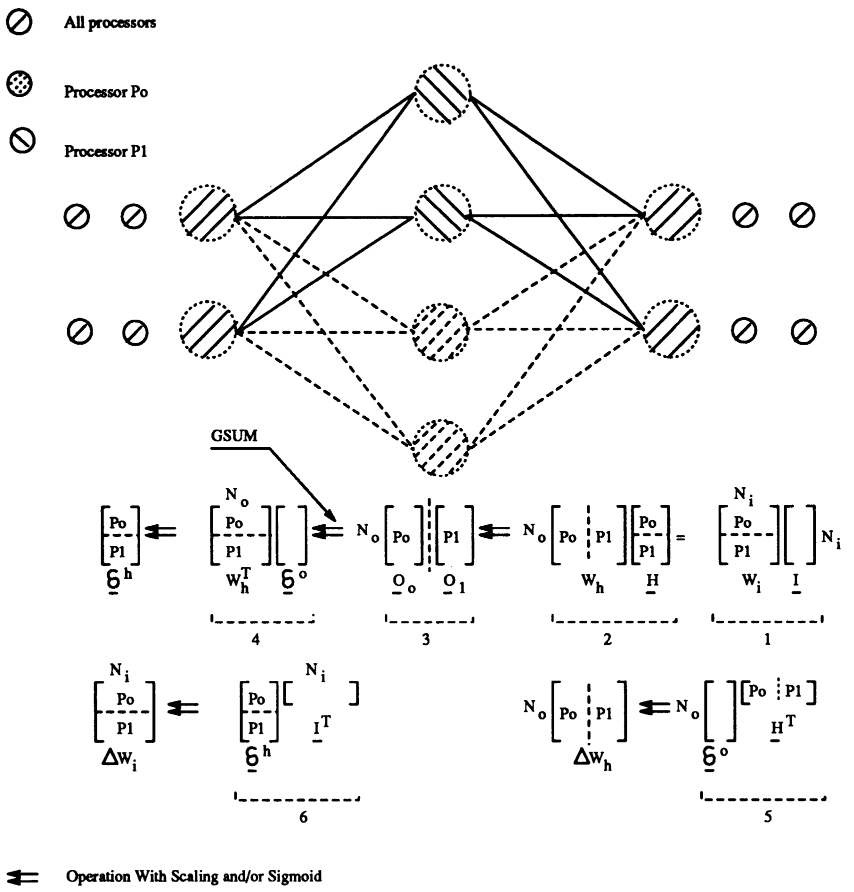


Figure 2: Network Partitioning

(c) Each processor now applies the sigmoid function to **all** the elements of the output vector. It can be seen that each processor performs some redundant sigmoid computation – in fact $\approx N_o - \lceil N_o/P \rceil$ sigmoid computations are done redundantly. An alternative to this to perform sigmoid computation only on the relevant portions of the output vector and then use GCOL to broadcast the resulting vector. This trades off communication to extra computation and usually is beneficial when the number of nodes in the output layer is large.

By the end of the forward pass stage each processor has a complete copy of the $\delta^{\mathbf{o}}$ vector and can start computing the backward pass as follows:

3. Each processor performs a local vector–matrix product to compute the product of $\mathbf{W_H^T}$ and $\delta_{\mathbf{o}}$. Note that this can be done trivially as the procesors have the matrix in the right form for transpose multiplication (Step 4 in Figure 2).
4. At this point all processors have the necessary information to compute the respective changes to their portions of the weight matrices and perform the necessary outer products to compute the changes (Steps 5 and 6 in Figure 2).
5. For the on-line version of the algorithm, the weight changes are applied to the respective weight matrices and for the off-line version they are accumulated and applied at the end of the epoch.

Training set partitioning

For networks with large training sets, it may be better to partition the training set and apply them to different copies of the network in different processors in parallel. In this case, each processor accumulates its local weight changes during an epoch. After the epoch, the weight changes in each processor are accumulated and then broadcast to all the processors for the next epoch using the GSUM primitive. For networks with small size but with large training sets, this approach is suitable as there is no processor communication overhead within an epoch. Figure 3 depicts this approach.

Combining network and training set partitioning

Training set partitioning method can actually be combined with network partitioning described above to exploit multiple levels of parallelism. In this case we first partition the training set to groups of processors and then partition thenetwork set within these groups. The best combination of both kinds of partitioning can be estimated by using the performance models [1]. Figure 4 depicts this approach. To clarify the idea, let $P = 2^p$ be the number of processors available. Those P processors can be divided into $G = 2^g$ groups (subcubes) each with $Q = 2^q$ processors, such that $p = q + g$.

Each group of processors will take the same copy of the network, and $m = \lceil \frac{M}{G} \rceil$ input/output patterns from the training set. The algorithm described earlier is used. At the end of the epoch a total of q *concurrent* GSUMs to accumulate and broadcast the weights have to be done. These concurrent GSUMs are across the groups and involve the respective processors of each group. The number of groups and training set partitions, q and g, should be chosen such that to maximize the speed up. When $Q = 1$ then this method reduces to the training set partitioning method, and when $G = 1$ then it reduces to the network partitioning method.

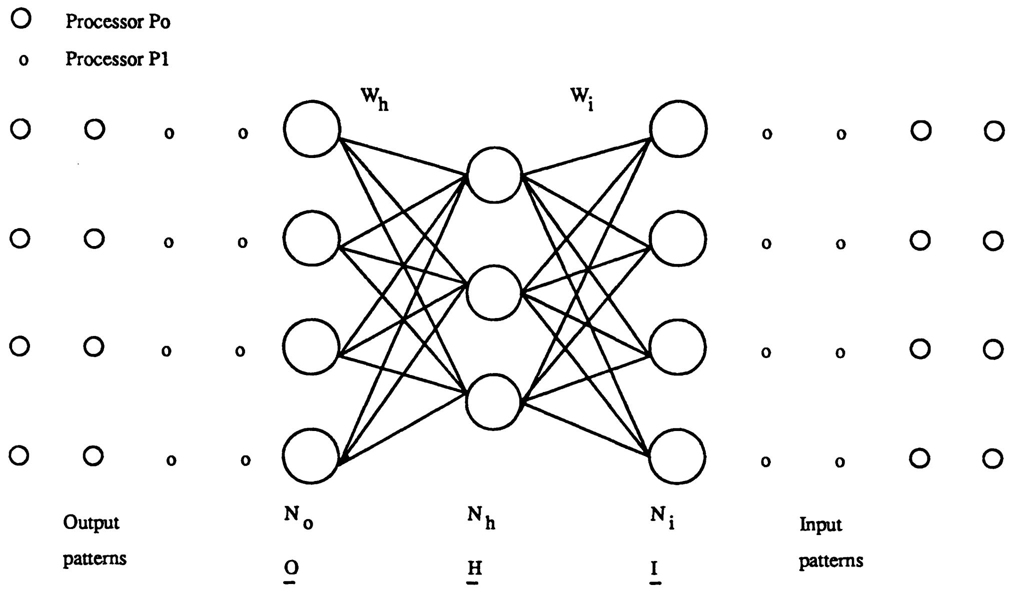


Figure 3: Training Set Partitioning

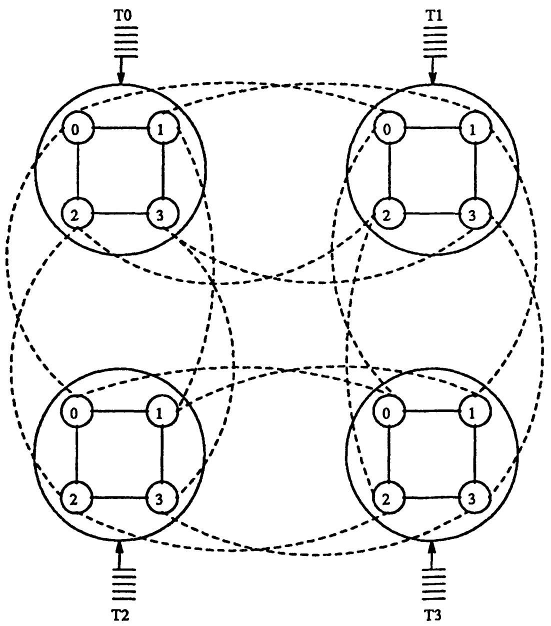


Figure 4: Network and training set partitioning

Performance Models

We have developed detailed performance models corresponding to each of the methods described above by analyzing the computational and communication requirements [1]. We have taken into account specific system related parameters such as floating point operation times, loop overhead, sigmoid function computation and interprocessor communication times, and network related parameters such as the number of layers, number of nodes at each layer and the training set size. The resulting performance models estimate the times required by an epoch of training using a given number of processors. Using these models the users can estimate which version or combinations of groups and training set partition sizes can give the better execution time. Our models estimate the parallel execution times accurately as shown later.

Results

In this section we present the experimental results obtained from our implementation on an iPSC/2 hypercube system with 8 processors for a number of benchmark neural networks. The performance models discussed earlier were also evaluated and compared with experimental results. These models are then used to extrapolate the performance of large scale hypercube systems with the communication related system parameters again extrapolated from the availabe experimental data. We present results from only network partitioning. For results on training set partitioning see [4, 5].

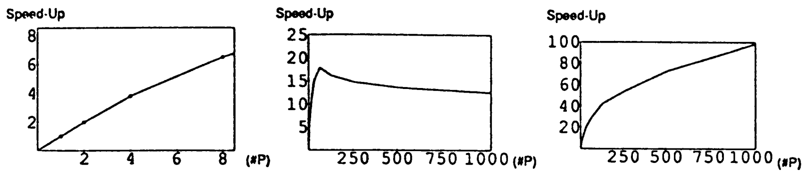
The results will be presented for the following neural net examples :

- a simple neural net to classify simple images on a retina as digits 0,...,9.
- neural net that learns to pronounce English text, NETTalk [16],
- a very large synthetic neural net to get and idea about the performance of the system on large networks.

For each network we present experimental speed-up results from our implementation in both tabular and graphical form along with a comparison with the performance model results. We also present graphically, extrapolated performance results for large scale hypercube systems for both network partitioning and both the best combination of network and training set partitioning methods. The times listed in the tabular data are per epoch times. The plots show number of processors versus speed-up graphs for experimental and theoretical network partitioning results and theoretical combined partioning.

Results for the digit recognition network

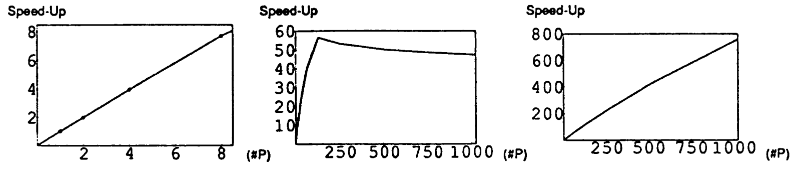
This is a simple network which is to learn and recognize simple digit patterns on 10 by 6 retina. The input layer has 60 units (logically arranged in the form of a 10 by 6 retina), the hidden layer has 60 units, and the output layer has 10 units (one unit for each digit). The training set consists of 10 patterns on the retina one for each digit. Figure 5 presents the experimental and theoretical results for this problem.



(a) (b) (c)

P	Experimental Time(Sec)	Experimental SU	Theoretical Time(Sec)	Theoretical % difference
1	1.59	1.00	1.62	-1.88
2	0.79	1.96	0.82	0.00
4	0.41	3.79	0.42	0.26
8	0.24	6.56	0.25	1.53

Figure 5: Results for digit recognition network



(a) (b) (c)

P	Experimental Time(Sec)	Experimental SU	Theoretical Time(Sec)	Theoretical % difference
1	10241	1.00	10460	-2.14
2	5128	1.99	5249	0.00
4	2587	3.96	2650	-2.45
8	1332	7.68	1356	-0.39

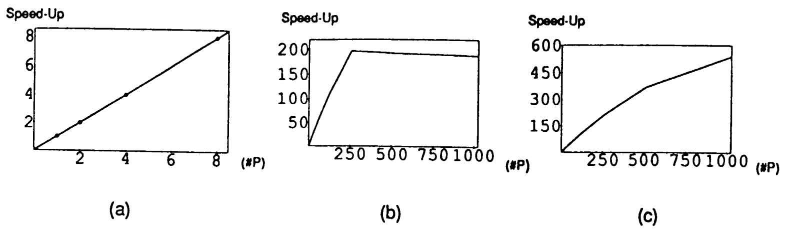
Figure 6: Results for NETTalk Network

Results for NETTalk

NETTalk[16] is backpropagation network that can be trained to convert English text to a sequence of phonemes. Its input is a string of characters forming English text and the network converts this text into phonemes representation that can be applied to the input of a speech synthesizer. This network has 203 units in the input layer, 120 units in the hidden layer, and 26 units in the output layer. The training set for this network has 1000 presentations. Figure 6 shows the results for this network for this problem.

Results for a synthetic large network

This is a large sythetic network that we have used for testing our implementation. The input layer has 1024 units, the hidden layer has 256 units, and the output layer has 32 units. Figure 7 shows the results for this network with 10 patterns in the training set. Figure 8 shows results from our models with 1000 patterns in the training set.



P	Experimental		Theoretical	
	Time(Sec)	SU	Time(Sec)	% difference
1	100.3	1.00	101.5	-1.19
2	50.2	1.99	50.8	-1.19
4	25.1	3.99	25.4	-1.19
8	12.6	7.96	12.7	-0.79

Figure 7: Results for the large synthetic network

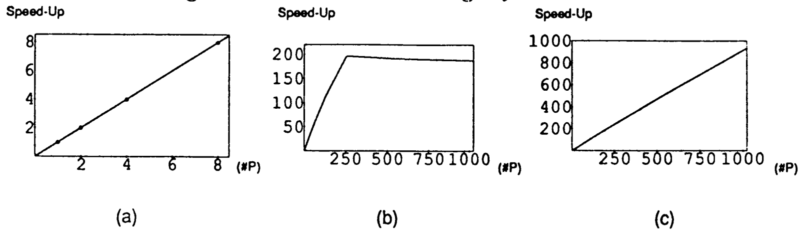


Figure 8: Theoretical results for the synthetic network with 1000 entries in the training set

Conclusions

In this paper we have presented a parallel processing approach for backpropagation algorithm with which a one can get almost linear speed-up on hypercube system for large networks using a combination of network and training set partitioning, and an algorithm that minimizes the amount of communication.

The results show that using the combined partitioning method is the best method that for large backpropagation networks with large training sets provide a large scale hypercube system is available. For small networks (couple tens of nodes) network partitioning is suitable and training set partitioning is suitable only if the training set is large.

References

[1] Cevdet Aykanat, Kemal Oflazer, and Radwan Tahboub. Parallel backpropagation algorithms for medium-to-coarse grain multicomputers. Submitted for Publication, July 1991.

[2] Guy Blelloch and Charles R. Rosenberg. Network learning on the Connection Machine. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[3] K. Diakonikolaou, S. Kollias, D. Kontoravdis, and A. Stafylopatis. Implementation of neural network learning strategies on a transputer-based parallel architecture. Technical report, Compter Science Division, Department of Electrical Engineering, National Technical University of Athens, Greece, 1991.

[4] Deniz Ercoşkun. Parallel implementation of the backpropagation learning algorithm on a hypercube parallel processor. Master's thesis, Bilkent University - Dept. of Computer Engineering and Information Science, December 1990.

[5] Deniz Ercoşkun and Kemal Oflazer. Experiments with parallel backpropagation on a hypercube parallel processor system. In *Proceedings of ICANN-91 – International Conference on Artificial Neural Networks*. Helsinki University of Technology, Elsevier Science Publishers, June 1991.

[6] Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 1988.

[7] Kevin Knight. A gentle introduction to subsymbolic computation: Connectionism for the AI researcher. Technical Report CMU-CS-89-150, School of Computer Science, Carnegie Mellon University, 1989.

[8] S. Y. Kung and J. N. Hwang. Parallel architectures for artificial neural nets. In *Proceedings of IEEE International Conference on Neural Networks*, volume 2, pages 165 – 172, 1988.

[9] Richard P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4 – 22, April 1987.

[10] Dean A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan Kaufman, 1989.

[11] Dean A. Pomerleau, George L. Gusciora, David S. Touretzky, and H. T. Kung. Neural network simulation at Warp speed: How we got 17 million connections per second. In *Proceedings of IEEE International Conference on Neural Networks*, volume 2, pages II–143 - II–150, 1988.

[12] U. Ramacher and J. Beichter. Systolic architectures for fast emulation of artificial neural networks. In *Proceedings of International Conference on Systolic Arrays*, 1989.

[13] Gareth D. Richards. Implementation of Back-Propagation on a Transputer. Edinburgh Preprint, 1989.

[14] Frank Rosenblatt. *Principles of Neurodynamics*. Spartan Books, 1962.

[15] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and John L. McClelland, editors, *Parallel Distributed Processing*, volume 1. MIT Press, 1986.

[16] Terence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145 – 168, 1987.

[17] Michael Witbrock and Marco Zagha. An implementation of Back-propagation on GF11, a large SIMD parallel computer. Technical Report CMU–CS–89–208, School of Computer Science, Carnegie Mellon University, December 1989.