Parallel back propagation training algorithm for MIMD computer with 2D-torus network

Jim Torresen[1/2]*, Shin-ichiro Mori[1],
Hiroshi Nakashima[1], Shinji Tomita[1], Olav Landsverk[2]
[1]Department of Information Science, Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01, Japan
[2]Department of Computer Systems and Telematics, The Norwegian Institute of Technology,
University of Trondheim, N-7023 Trondheim, Norway
Email: jimtoer@idt.unit.no

***Abstract*— This paper describes several algorithms, mapping the back propagation learning algorithm onto a *large* 2-D torus architecture. To obtain high speedup, we have suggested an approach to combine the possible parallel aspects (training set parallelism, node parallelism and pipelining of training patterns) of the algorithm. Several algorithms were implemented on a 512 processor Fujitsu AP1000 and compared using NETtalk, a network that transforms text to speech.**

For a large number of processors we obtained best performance for the proposed combined algorithm, reaching 76 million weights updated per second (MCUPS) using 512 processors. Our results show that to obtain high speedup on a large number of processors, all types of parallelism in the back propagation algorithm ought to be considered to be combined. Also, highly parallel general purpose computers can compete with the performance of neurocomputers.

1 Introduction

One of the most popular artificial neural networks (ANN) is the multi layer perceptron (MLP) [1]. Back propagation (BP) is the main algorithm for training MLP. The algorithm is very computationally demanding, thus to reduce the long training time, parallel computation is mandatory.

Several approaches to parallelize BP have been suggested. The topology of MLP is not fixed. Every real ANN application requires different number of nodes and layers. It is difficult to know which parallel algorithm to choose, because the effectiveness of each approach will vary according to the MLP topology and the architecture of the target machine. Thus mainly experimental results can give an idea of the best mapping strategy. The most important factors to increase speedup are to minimize communication between processing elements (cells) and to avoid idle time for each cell.

In this paper several parallel BP algorithms will be compared on Fujitsu AP1000 ([2]), a message passing MIMD computer with two dimensional torus topology network. The system used in this research consists of 512 cells. AP1000 has distributed memory and each cell has 16 MB of local memory, of which approximately 10 MB is offered as user memory. Message routing between cells is done by worm-hole routing, thus path length has little effect on communication time.

In the following section the three different degrees of parallelism in the BP algorithm are outlined and four different approaches implemented on AP1000 are explained. The performance of each of them is given in section 3 followed by conclusion in section 4.

2 Parallel BP implementations on a 2D-torus MIMD computer

Many factors affect the design of a parallel BP algorithm. The most important issues are:

Weight updating strategies, The total gradient descent algorithm (*learning by epoch*) has a slower convergence rate than stochastic gradient descent (*learning by pattern*), especially for a large training set [3]. However, *learning by epoch* is well suited to parallelization. To improve convergence rate, the weights may have to be updated more frequently than once per training set epoch using a hybrid algorithm (*learning by block*). In this method, the weights are updated after a subset of the training patterns have been presented.

*Research Student at Kyoto University, PhD Student at The Norwegian Institute of Technology

Degree of parallelism, The BP algorithm reveals several different degrees of parallelism, as proposed by [4]. A first possible parallel approach, *training set parallelism*, is to split the training set across the cells. The weights are updated using *learning by block/epoch*. Second, the nodes within a layer and computation within each node may run in parallel. This is called *node parallelism*. In this method, weights are updated using *learning by pattern*. A third parallel aspect is *pipelining* the training patterns in the layers, i.e. while the output layer calculates output and error values, the hidden layer concurrently processes the next training pattern. Pipelining requires *learning by block/epoch*.

Several BP implementations have been done on general purpose computers. Node parallelism is used to implement BP on Intel iPSC/860 hypercube in [5] and on MasPar MP-1 SIMD computer in [6]. A combination of training set and node parallelism for MasPar MP-1216 is given in [7]. Pipelining and node parallelism is used in [8].

Below we describe several different mapping approaches onto a 2D-torus MIMD computer. First two methods exploiting training set parallelism and node parallelism respectively are explained, followed by two approaches combining several degrees of parallelism.

Training set parallelism

The easiest way to make a serial program execute in parallel is usually by coarse grain splitting. The most obvious way that can be done in BP is by splitting the training set across the available processors [9]. Let C denote the total number of cells (i.e. processors) and P the total number of training patterns. Then, P_i where $i \in \{0, \ldots, C-1\}$, the number of training patterns on processor i is given by:

$$P_i = \lfloor \tfrac{P}{C} \rfloor + k \quad \text{where} \quad k = \begin{cases} 0 & \text{if } i \geq (P \bmod C) \\ 1 & \text{if } i < (P \bmod C) \end{cases} \tag{1}$$

Each cell has a local copy of the complete weight matrices and accumulate weight change values for the given training patterns. The neural network weights must be consistent across all the cells, thus weights are updated in a global operation (*learning by block/epoch*).

If the number of processors increases, the number of training patterns per processor decreases according to (1). That means the computation/communication ratio decreases and limits the speedup. Another problem, that may arise if the neural network is large, is that much local memory is necessary in each node to store the weight matrices. However, the AP1000 cell memory is large enough to store most of the networks for real applications.

Node parallelism

A fine grain BP algorithm using node parallelism for mesh-connected multiprocessors is proposed in [10]. The main calculations in BP are a series of *matrix-vector* multiplications. An efficient algorithm for this calculation, where communication is minimized, is suggested in the paper. The algorithm can be explained by a simple 4 input, 4 hidden and 4 output nodes network and a system of 4 cells. The weight matrices are partitioned into sub-matrices and multiplied by the input vectors as follows:

$$\left[\begin{array}{c} y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \end{array}\right] = \left[\begin{array}{cc|cc} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ \hline w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{array}\right] \left[\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \end{array}\right]$$

The element w_{ji} is the weight connecting input node i to output node j. Each cell stores a sub-matrix statically for the hidden layer and the output layer. Each element, x_i, of the input vector is mapped only to one processor. If the number of elements in the input vectors is larger than the number of processors, each processor will keep more than one element of the vector. During matrix-vector multiplication the input x_i is rotated *vertically* between the processors, onto which elements $w_{h,*i}$ are mapped. Partial results are accumulated horizontally from each processor, onto which elements $w_{h,j*}$ are mapped, to calculate the output of the hidden layer, $y_{h,j}$. The output layer, $y_{o,k}$, is calculated in a similar way. The weights are updated for every pattern and the backward phase is computed in a similar way as in the forward phase by rotating the elements between cells.

The algorithm can be effective if the network matches the architecture in a way that divides matrices and vectors into *equal* parts for each processing element. If not, some of the processors may be idle or have long idle time. Although the communication is minimized in the algorithm, the communication overhead will be large for most real neural networks and parallel machines, because the computation grains become too small.

Combined solutions

The two former solutions will be able to obtain fairly high speedup on a *small* parallel computer. But the many interconnections in the MLP network result in much communication between cells. The fixed structure of the target machine makes it impossible to increase the number of cell interconnections. When the network/training set is dispersed to many cells, the granularity becomes fine and limits speedup. The obvious way to partly overcome this problem is to combine the *different* parallel aspects of the BP algorithm. Combining training set and node parallelism for the Connection Machine is proposed in [11]. The network is replicated in each column of processors as shown in Figure 1.

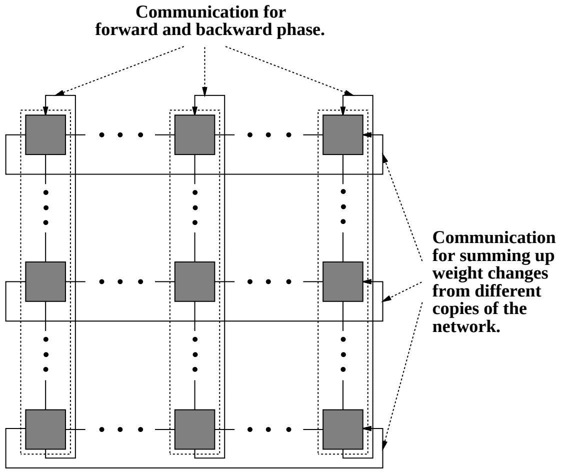

Figure 1: **Combination of training set and node parallelism. Communication along the column direction are for forward and backward phase; communication along row direction are for summing up weight changes from different copies of the network.**

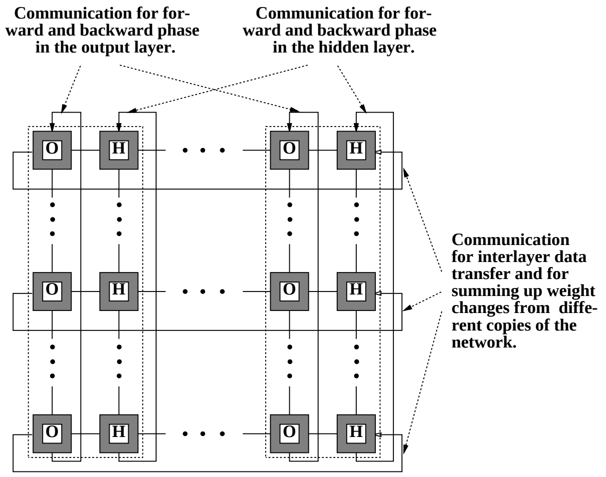

Figure 2: **Combination of all three types of parallelism in the BP algorithm; Training set parallelism, node parallelism and pipelining of the training patterns are combined in a way that minimizes communication conflicts.**

Each column contains different training patterns, thus weight update is done by communication along row direction to sum up weight changes (*learning by block/epoch*) from the different copies of the network. Each cell in a column computes the output of certain nodes, thus the weight matrices are partitioned horizontally and mapped statically.

As an example, a column of 2 processors has the weight matrices partitioned and multiplied by the input vectors as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ \hline y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ \hline w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

All input training pattern elements are stored in *each* cell to reduce communication. Thus only hidden layer output elements have to be rotated between the processors. The previously described node parallel algorithm can be transformed into this by using only one column of processors. The output of each node will be given directly, because complete rows of the weight matrix are stored in each cell. No accumulation of partial results is necessary.

Although this solution will give better speedup ratio than previously described algorithms, it doesn't benefit from pipelining the training patterns. We here propose to combine *all* the three degrees of parallelism in the BP algorithm to obtain higher speedup ratio on a computer with a *large* number of processors.

The suggested processor assignment is shown in Figure 2. Two processor columns compute the weight change for one training pattern group. The **H** column acts as the hidden layer, and the **O** column acts as the output layer. We have chosen to let the interlayer communication be along the row direction because the training set parallelism only requires communication when the weights are updated. This layout reduces the degree of training set parallelism, thus it is required to double the number of training patterns assigned to each column. The problem of grain size reduction for a large number of processors is then reduced.

O:		C: $(y_{o,k}, \delta_{o,k}, \delta_{h,j})^1$		Awc: $(w_{o,kl})^1$	C: $(y_{o,k}, \delta_{o,k}, \delta_{h,j})^2$	Awc: $(w_{o,kl})^2$	$\cdots$
	$y_{h,j} \uparrow$	$y_{h,j} \uparrow$		$\downarrow \delta_{h,j}$		$y_{h,j} \uparrow\downarrow \delta_{h,j}$	
H:	C: $(y_{h,j})^1$	C: $(y_{h,j})^2$		Awc: $(w_{h,ji})^1$	C: $(y_{h,j})^3$	Awc: $(w_{h,ji})^2$	

where:

H: Hidden layer
O: Output layer
$1 \leq i \leq number\_of\_input\_nodes$
$1 \leq j \leq number\_of\_hidden\_nodes\_in\_cell,$
$1 \leq k \leq number\_of\_output\_nodes\_in\_cell$

C: $(z)^p$: Compute z for pattern p
Awc: $(w)^p$: Accumulate weight change values for pattern p
$1 \leq l \leq number\_of\_hidden\_nodes$

Figure 3: **Pipelining of hidden and output layer calculation.**

The computation, in one processor, in each of the layers is shown in Figure 3. The hidden layer is dispersed in the lower row, while the upper row indicates the output layer. Communication between the layers is needed only to transfer $y_{h,j}$ to the output layer and the hidden layer error $\delta_{h,j}$ back to the input layer. The balance of computation load between the layers depends on the number of nodes in each layer. The output layer processors do most of the backward phase computation, but usually the hidden layer has a much larger number of nodes (i.e. larger weight matrix). Thus hidden layer processors require *more* time to compute outputs and accumulate weight change values than the output layer processors. Parts of the $\delta_{h,j}$ computation may be moved to the layer having least computation.

When weights are updated, the communication is between every other column. Since inter-cell communication time on AP1000 is independent of distance, the updating time of weights will not essentially increase.

3 Comparing parallel implementations on AP1000

The performance of the suggested parallel algorithms on AP1000 are measured using NETtalk [12] as an application. It is a two layer MLP neural network that transforms text to speech, using 203 input nodes, 80 hidden nodes and 26 output nodes. NETtalk has been used to test the performance of both general purpose computers ([13],[5],[7] and [11]) and neurocomputers ([14] and [15]). Since it has become widely used, it is regarded as a benchmark for comparing computers running the BP algorithm, although some differences in the number of hidden nodes and weight updating frequency exist. Performance during training is measured in CUPS (Connections/Weights Updated Per Second). In the case *learning by block/epoch* is used, CUPS is given by the number of weight change values computed per second. See Table 1 for the performance of some computers.

Computer	No of cells	MCUPS
Sun-3 [5]	1	0.034
iPSC/860 [5]	32	11
CM-1 [16]		2.8
Warp [13]	20	32
CM-2 [11]	65536	40
MP-1216 [7]	16384	41
Cellular arch. [14]	4175	52
Sandy [15]	60	135 (Est.)

Table 1: **Performance of other computers running NETtalk.**

In order to compare parallel BP algorithms and show how well they performed for an increasing number of processors, four algorithms were implemented: training set parallelism, node parallelism, combination of training set and node parallelism (*2 Aspects of Parallelism Combined*, 2APC) and it's improvement using pipelining technique (*3 Aspects of Parallelism Combined*, 3APC). Except for the node parallel algorithm, *learning by block* was used. Few guidelines for choosing proper weight updating frequency exist. In [5] the weights were updated after 2000 training patterns. In our experiments the weights were updated for every 1024 patterns. The implementations were first trained and tested by a small character recognition problem. Then the NETtalk performances were measured using randomly generated training patterns.

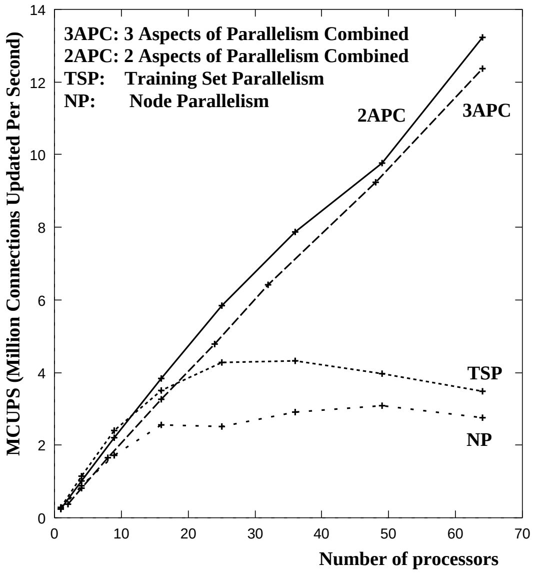

Figure 4: **CUPS for the implemented algorithms running NETtalk.**

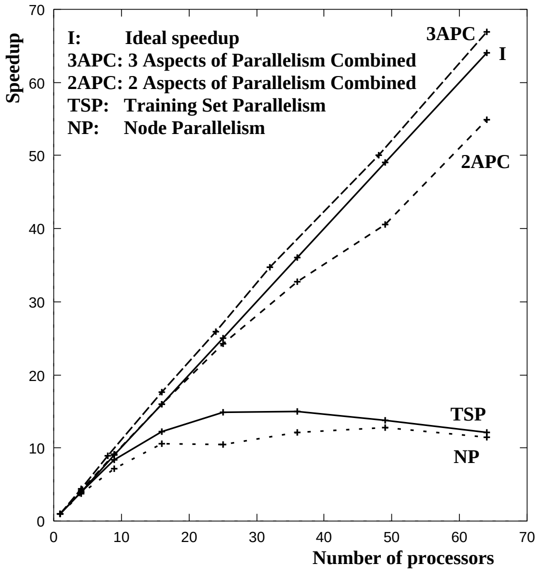

Figure 5: **Speedup for the implemented algorithms running NETtalk.**

Figure 4 shows the performance of the algorithms for 1 to 64 processors. Only the combined solutions show increasing CUPS values for a large number of processors. Although the 2APC shows the best performance, the speedup, shown in Figure 5, for 3APC is definitely the highest. The speedup larger than the number of processors can be explained by load imbalance between layers, as explained in the discussion on combined solutions, and cache miss rate. For a small number of processors the cache miss occurs more frequently, due to the large weight matrices and many training patterns stored in each cell. As the number of processors increases, the size of the matrices and number of training patterns stored in each cell decreases. Due to the large cell memory, the *whole* training set can be stored in the cells, thus down-loading is only needed initially. That means that the down-loading time is very short compared to the total training time and is omitted in the time measurements.

Training set parallelism

The algorithm performs well for a *small* number of processors, but if more than 36 processors (28 training patterns per cell) are used, the training time *increases*. The reason is that the weight update overhead becomes large, because the training patterns are dispersed to many cells. Although the algorithm gives better performance than node parallelism, *learning by block* means slower convergence rate and more training iterations are required.

Node parallelism

The main problem of the node parallel algorithm is that the granularity becomes too fine, so that it is impossible to obtain high speedup for a large number of processors. The nonlinearity in the result curves shows that the performance is dependent on how well the perceptron network is mapped to the architecture. The load balancing easily becomes uneven when the NETtalk network is to be mapped onto many cells. One optimization we included was to store the whole training input sub-vector in each cell and not rotate the elements during hidden layer output computation. The algorithm benefit from using *learning by pattern*.

Combined solutions

The results show the need to combine different kinds of parallelism to obtain acceptable speedup for a large number of cells. The 2APC algorithm is faster than the 3APC algorithm, because the latter suffers

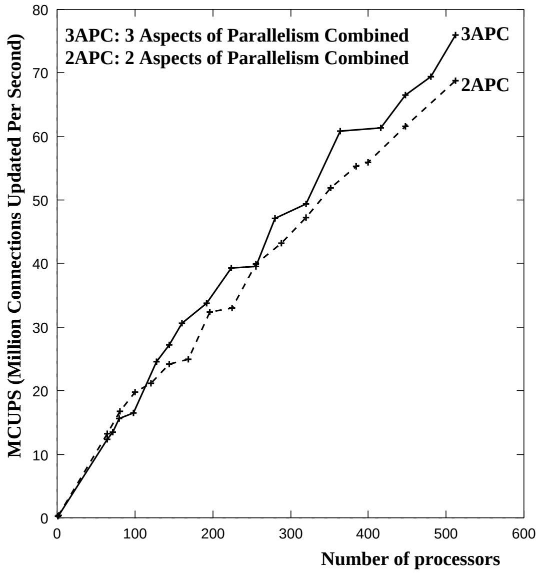

Figure 6: **CUPS for the two algorithms combining different parallel aspects running NETtalk on a large number of processors.**

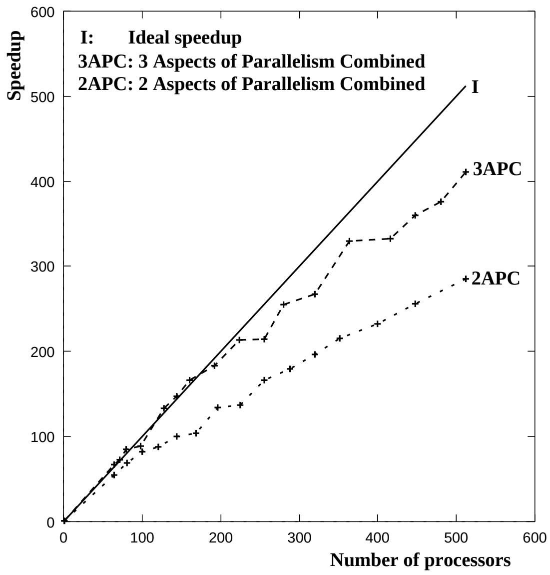

Figure 7: **Speedup for the implemented algorithms in Figure 6.**

from uneven load balance between hidden and output layer. Performance analysis showed that the hidden layer had higher load than the output layer. Since less communication is done between the hidden layer processors, compared to the output layer processors, the load becomes more even as the number of processors increases. The performance of the combined solutions on 1 to 512 processors is shown in Figure 6. When more than 110 processors are used, the 3APC becomes faster than 2APC, reaching 76 MCUPS on 512 processors. Both algorithms are sensitive to how well the perceptron network/training set is mapped to the architecture. Thus the performance is not strictly linear. The speedup, shown in Figure 7, is 411 on 512 processors for 3APC compared to 289 for 2APC. No larger AP1000 system was available to us, but considering the speedup of 2APC, 3APC (larger training set in each cell) should be able to gain even more speedup on a system of 1024 cells.

The performance of each parallel mapping will vary according to the network and training set of each application. Thus to make an optimal combination one should include, prior to start of training, a comparison of the time used for 1 iteration for various possible parallel implementations on the target machine. The times can be found either by running one iteration of each or by using performance estimation. Convergence rate should also be considered. Many parameters affect the convergence rate, like the number of hidden nodes, initial weight values, learning rate and the weight updating frequency. Thus it is difficult to estimate how often the weights should be updated to minimize the total training time.

The NETtalk network is quite small and larger perceptron networks/training sets will be able to obtain higher performance, as the granularity becomes coarser.

4 Conclusion

An overview of parallelizing the BP algorithm on a MIMD computer with torus topology has been given. We have evaluated several approaches on AP1000, a highly parallel computer.

Earlier research on parallelizing BP exploits only one or two types of parallelism, while we have combined three types: training set parallelism, node parallelism and pipelining of the training patterns. Our results show that this is necessary to obtain the highest possible speedup on a large number of processors. By combining the aspects of BP parallelism, computation granularity can be kept coarse even on a large number of processors. Although we have been using a general purpose computer in our research, the way of exploiting different types of parallelism also ought to be considered when neurocomputers are to be designed.

We obtained a speedup of 411, running NETtalk on a 512 cell AP1000, a very satisfying result considering the restricted connectivity of a 2D-torus computer. A NETtalk performance of 76 MCUPS is faster than other general purpose computers and compete-able with neurocomputers.

The study of convergence rate and optimal combination of types of BP parallelism is a subject for future research.

References

[1] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representation by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. The MIT Press, 1986.

[2] Hiroaki Ishihata et al. Third generation message passing computer AP1000. In *Proc. of the International Symposium on Supercomputing*, pages 46–55, Nov. 1991.

[3] Helene Paugam-Moisy. Parallel neural computing based on neural network duplicating. In Ioannis Pitas, editor, *Parallel algorithms for digital image processing, computer vision and neural networks*, chapter 10, pages 305–340. John Wiley & Sons, 1993.

[4] Alexander Singer. Implementation of artificial neural networks on the Connection Machine. *Parallel Computing*, 14:305–315, Summer 1990.

[5] Darin Jackson and Dan Hammerstrom. Distributing back propagation networks over the intel iPSC/860 hypercube. In *Proc. of International Joint Conference on Neural Networks*, volume I, pages 569–574, 1991.

[6] G. Chinn et al. Systolic array implementations of neural nets on the MasPar MP-1 massively parallel processor. In *Proc. of International Joint Conference on Neural Networks*, volume II, pages 169–173, 1990.

[7] Andreas Zell et al. Problems of massive parallelism in neural network simulation. In *Proc. of IEEE International Conference on Neural Networks*, pages 1890–1895, 1993.

[8] Peter Wohl and Thomas W. Christopher. MIMD implementation of neural networks through pipelined, parallel communication trees. In *Proc. of the 1991 IEEE Int. conference on tools for AI*, pages 82–89, November 1991.

[9] D.M. Weber. Parallel implementation of time delay neural networks for phoneme recognition. In *Proc. of IEEE International Conference on Neural Networks*, pages 1583–1587, 1993.

[10] Takashi Yukawa and Tsutomu Ishikawa. Optimal parallel back-propagation schemes for mesh-connected and bus-connected multiprocessors. In *Proc. of IEEE International Conference on Neural Networks*, pages 1748–1753, 1993.

[11] Xiru Zhang. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14:317–327, Summer 1990.

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

[13] D. A. Pommerleau et al. Neural network simulation at Warp speed: How we got 17 million connections per second. In *Proc. of IEEE International Conference on Neural Networks*, 1988.

[14] Bernard Faure and Guy Mazare. A cellular architecture dedicated to neural net emulation. *Microprocessing and Microprogramming*, 30(1-5):249–256, August 1990.

[15] Hideki Kato et al. A parallel neurocomputer architecture with ring registers. In *Proc. of an international conference organized by the IPSJ to commemorate the 30th anniversary*, pages 233–240, 1990.

[16] G. Blelloch and C. R. Rosenberg. Network learning on the Connection Machine. In *Proc. of IJCAI87*, pages 323–326, 1987.