EXPLOITING MULTIPLE DEGREES OF BP PARALLELISM ON THE HIGHLY PARALLEL COMPUTER AP1000

J Torresen[1/2], S Mori[1], H Nakashima[1], S Tomita[1], O Landsverk[2]

[1]Kyoto University, Japan [2]The Norwegian Institute of Technology, Norway

ABSTRACT

During the last years, several neurocomputers have been developed, but still general purpose computers are an alternative to these special purpose computers. This paper describes a mapping of the backpropagation learning algorithm onto a large 2-D torus architecture. The parallel algorithm was implemented on a 512 processor AP1000 and evaluated using NETtalk and other applications. To obtain high speedup, we have suggested an approach to combine the multiple parallel degrees (training set parallelism, node parallelism and pipelining of the training patterns) of the algorithm.

For a large number of processors, we obtained a performance of 81 million weight updates per second using 512 processors, when running the NETtalk network. Our results show that to obtain the best performance on a large number of processors, a combination of multiple degrees of parallelism in the backpropagation algorithm ought to be considered.

INTRODUCTION

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

The topology of an MLP is not fixed. Every real ANN application require different number of neurons (nodes) and layers. It is difficult to know which parallel algorithm to implement, because the effectiveness of each approach will vary according to the mapping of the neural network application onto 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. Since the number of weights is much larger than the number of nodes in an MLP, it is beneficial to store the weight matrices statically (i.e. rotate the input elements between processors, instead of the weight matrices).

In this paper a parallel BP algorithm is evaluated on Fujitsu AP1000, Ishihata et al [2], a message passing MIMD computer with two dimensional torus topology network. Figure 1 depicts the architecture. AP1000 has distributed memory and each cell consists of a Sparc CPU, a FPU, a message passing chip, 128 KB cache and 16 MB main memory. The system used in this research consists of 512 cells. Message routing between cells is done by worm-hole routing, thus the path length has little effect on communication time.

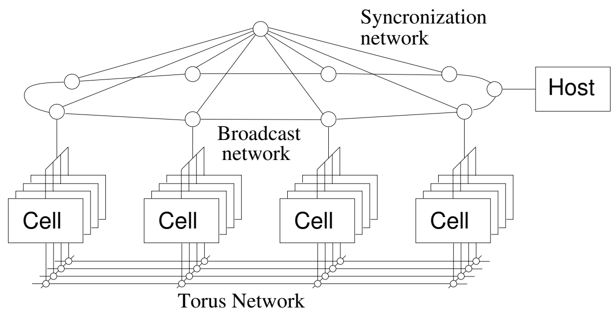

Figure 1: The AP1000 architecture.

In the following section we describe an algorithm that combines multiple degrees of BP parallelism. Then, we show how the performance varies for different MLPs and different number of cells. Finally, conclusions are given.

MAPPING OF BP NETWORK ONTO A LARGE 2D-TORUS MIMD COMPUTER

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

Weight updating strategy. Three different approaches:

- *Learning by pattern*, update the weights after *each* training pattern has been presented.
- *Learning by block*, update the weights after a subset of the training patterns have been presented.
- *Learning by epoch*, update the weights after all patterns have been presented

Learning by epoch has a slower convergence rate than *learning by pattern*, especially for a large training set, Paugam–Moisy [3]. However, *learning by epoch* is beneficial in parallel implementations.

To improve the convergence rate, the weights may have to be updated more frequently than once per training set epoch using the intermediate strategy *learning by block*.

Degree of parallelism, The BP algorithm reveals several different kinds of parallelism, as described by Singer [4]:

- *Training set parallelism*, Splits the training set across the cells, Weber [5]. Each cell has a local copy of the complete weight matrix and accumulate weight change values for the given training patterns. The weights are updated by using *learning by block/epoch*.
- *Node parallelism*, The nodes within a layer run in parallel. Further, the computation within each node may also run in parallel, Yukawa and Ishikawa [6]. In this method, the weights can be updated by using *learning by pattern*.
- *Pipelining*, Pipelines the training patterns in the layers, i.e. while the output layer processor calculates output and error values, the hidden layer processor concurrently processes the next training pattern. Pipelining requires a delayed weight update or *learning by block/epoch*.

BP has been implemented on many general purpose computers. However, none (as known to us) of them have combined all degrees of parallelism. In Torresen et al [7], it was shown that to obtain the highest possible performance on a highly parallel computer, a combination of all degrees of BP parallelism ought to be considered. Below, the details of the proposed mapping is outlined.

Combined solution

To be able to exploit all degrees of parallelism, the network is partitioned as shown in Figure 2.

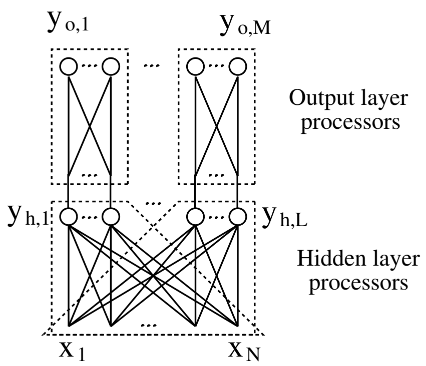

Figure 2: Partitioning of the network, when pipelining and node parallelism are combined. The dotted lines show which *nodes* are mapped to each processor.

Notice that *all* input training pattern elements are stored in *each* hidden layer processor to reduce communication. The output layer processors *must* rotate their input elements (hidden layer output elements) between themselves. Training set parallelism means that several copies of the network are made.

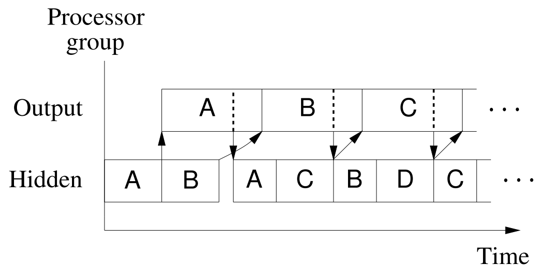

Figure 3: Pipelining of the training patterns.

Figure 3 shows a pipelining example. First, the hidden layer processor computes output values of training pattern A. The output processor reads the values and computes output and error values of A. The hidden processor concurrently process the next training pattern (B). Then, it reads the hidden error for A and both processors accumulate the weight change values for A.

The separation of layer computation to different processors reduces the memory requirements. The program code size is smaller and either the input *or* the output training vectors (not both) have to be stored in a cell.

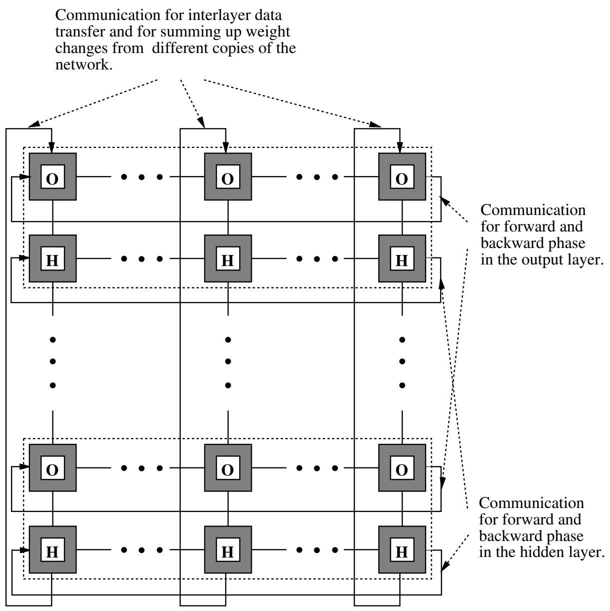

Figure 4: Combination of all three types of parallelism in the BP algorithm; Training set parallelism, Node parallelism and Pipelining of the training patterns are combined in a way that minimizes communication conflicts.

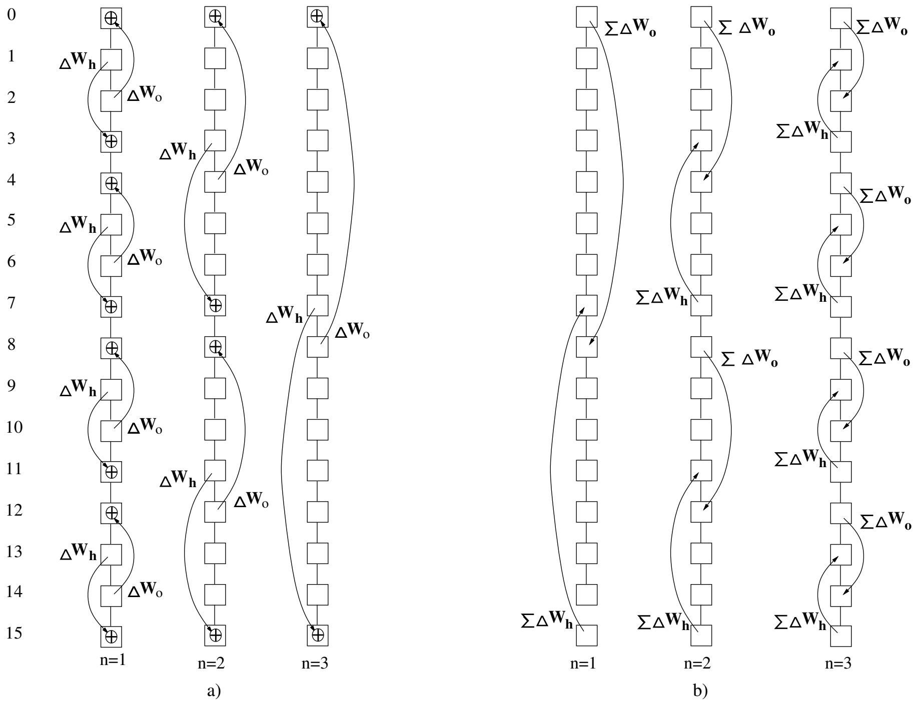

Figure 5: a) Summing the weight matrices before weight update in a system of 16 vertical cells. b) Broadcasting the summed weight matrices back to each cell.

The suggested processor assignment is shown in Figure 4. Two processor rows compute the weight change matrix for one training pattern group. The **H** row acts as the hidden layer, and the **O** row acts as the output layer. We have chosen to let the interlayer communication be along the column direction, because the training set parallelism only requires communication when the weights are updated.

The balance of computation load between the layers depends on the number of processors and 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. This will be investigated in more detail in the result section.

Optimizing the weight update

Since the convergence rate improves when the weights are updated frequently, it is of major importance to minimize the weight update time. To avoid weight updating to become a bottleneck in a large system, we use a $\log n$ step summing technique, shown in Figure 5a).

The summing starts in the leftmost column. Hidden layer cells send their matrices to cells south of themselves, while output layer cells send to cells further north. We call the solution *edge summing*, since we sum the matrices in the north-most cell and south-most cell for the output and hidden layer, respectively. In the given system, only 3 steps are required for summing the weights matrices. Then the result is (Figure 5b) sent back to each hidden and output layer cell. This is done in a similar way as the summing, but opposite direction for sending the data is used.

The broadcasting part can be omitted, if we make the communication in part a) bi-directional and duplicate the summing. However, this introduces more overhead, communication conflicts and redundant computation.

RESULTS AND DISCUSSION

The performance on AP1000 is measured using the NETtalk application proposed by Sejnowski and Rosenberg [8]. It is a two layer MLP neural network with 203 input, 60–120 hidden and 26 output nodes. Performance during training is measured in MCUPS (Million Connections (or weights) Updated Per Second). In the case *learning by block/epoch* is used, MCUPS is given by the number of weight change values computed per second. We have chosen to implement *several* different algorithms to compare the parallelization schemes. The main purpose has not been to show the computer speed. Thus, little time has been used on optimizing the program code.

In the following, we compare the solution combining all three degrees of parallelism (*3 Aspects of Paral-*

lelism Combined, 3APC) with a combination of only training set and node parallelism (*2 Aspects of Parallelism Combined*, 2APC). The weights were updated for every 1024 patterns.

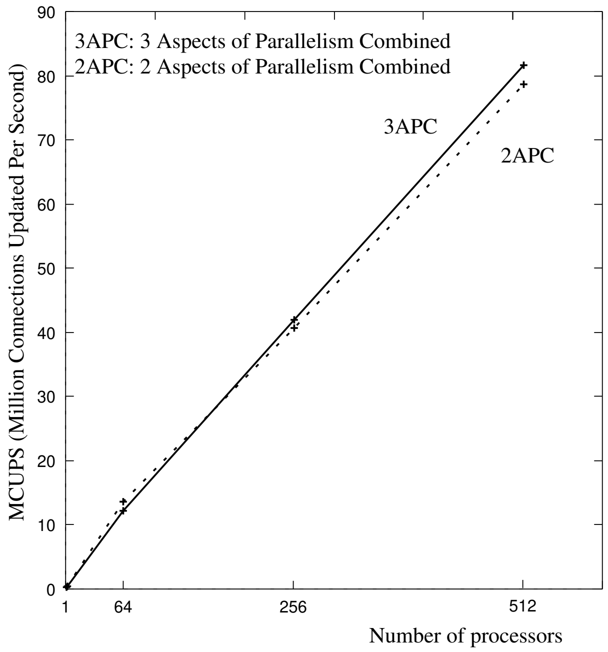

Figure 6: Performance for the two algorithms combining different parallel aspects, running NETtalk with 80 hidden nodes.

The performance, for 80 hidden nodes, of each of them on 1 to 512 processors is shown in Figure 6. Both algorithms have almost linear speedup on a system of many cells. When more than 64 processors are used, the 3APC becomes faster than 2APC, reaching 81 MCUPS on 512 processors. This can partly be explained by the small grains in computation for the 26 output neurons. Instead of using all the processors for this task, 3APC uses half of the processors to work on the intensive computing of the hidden layer. Moreover, there are twice as many training patterns computed on *each* processor, between weight updates, compared to for 2APC. The reason is that, 3APC uses *two* rows for each network copy, while 2APC needs only one. In comparison, the NETtalk application on a Sparc 10 workstation got a performance of 0.6 MCUPS, using *learning by pattern*.

On 512 processors, Figure 7 shows how the performance decreases, when the weights are updated more frequently than once per 1024 patterns. 2APC updates the weights in less time than 3APC. This is due to the initial pipeline delay and unequal load balance, when then hidden and output layer weights are updated concurrently for 3APC.

Figure 8 shows the NETtalk performance for 60 and 120 hidden neorons. 3APC performs better than 2APC for 120 hidden neurons. When the number

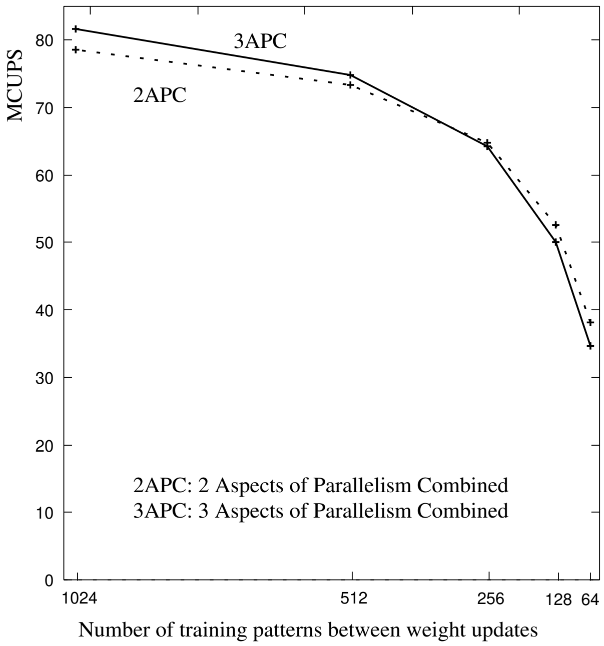

Figure 7: Performance for different numbers of weight updates per iteration in the NETtalk experiment on a 512 cells system.

of hidden neurons is increased, both the number of hidden and output *forward* computation increases. The increase is of equal proportion (e.g. doubled for 120 neurons, compared to 60) for each of the layers. However, the *backward* phase requires more computation by the output layer processors. This is because the main part of the backward pass is done by the output processors. A large number of hidden neurons seems to be beneficial for the hidden-output load balance in our application. In general, a decrease in the number of input nodes or an increase in the number of hidden or output nodes moves computation load from the hidden to the output processors.

Performance for a large neural network (e.g. an image recognition application) is given in Figure 9. The network consists of 1024 input, 512 hidden and 64 output neurons. 4096 training patterns were used and the weights were updated for every 512 patterns. The 2APC outpace 3APC, which is understandable from the non-even number of nodes in each layer. For such a large network it would be interesting to see if it is possible to get high performance by using only node parallelism (i.e. both nodes within a layer and computation within each node run in parallel, Yukawa and Ishikawa [6]). For 64 processors the performance is equal to 2APC. For 256 processors the performance is much less than 2APC. This indicate that the computation grains become small, even for a relatively large network. However, the algorithm may need less number of training iterations, since *learning by pattern* is used. It was impossible to run the node parallel implementation on the 512 cell configuration.

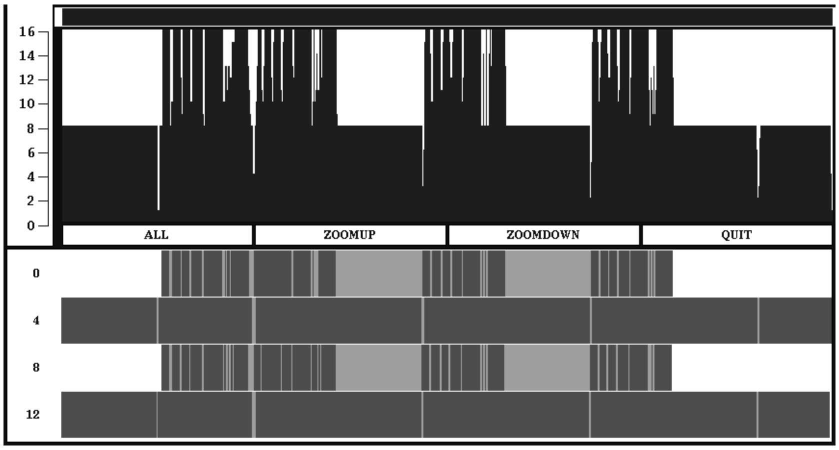

Figure 10: Performance on a 16 processor configuration, for the first 4 training patterns of NETtalk.

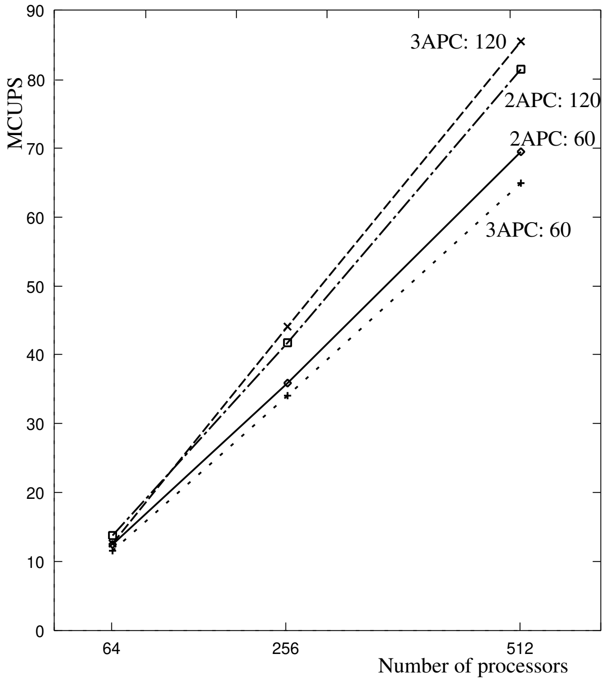

Figure 8: Performance for the 2APC and 3 APC algorithms for the NETtalk network with 60 and 120 hidden nodes.

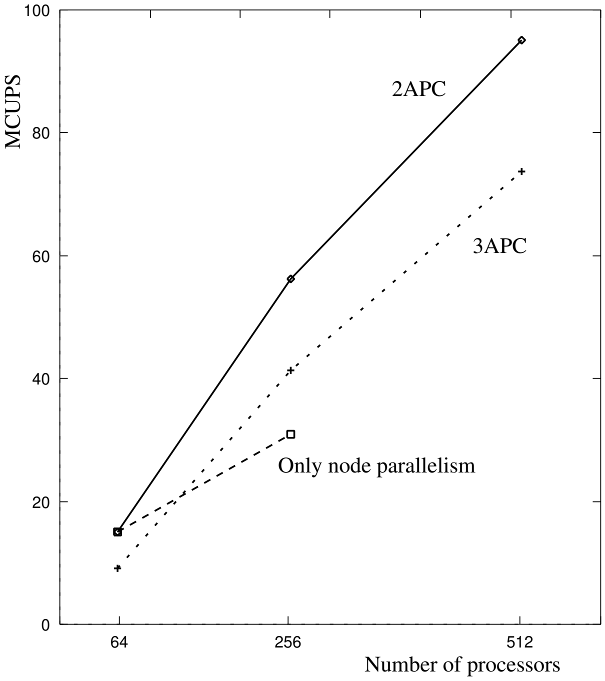

Figure 9: Performance for a large MLP network with 1024 input, 512 hidden and 64 output neurons.

Comparing load balance between the hidden- and output-layer processors

An even load balance is very important for the 3APC algorithm to obtain good performance. In the following, we show the results for two performance analysis, given by a AP1000 performance analyser tool.

Figure 10 shows the processor utilization in a 16 processor (4x4) configuration, running NETtalk (80 hidden nodes), for the first 4 training patterns. The time axis, in the figure, is horizontal and the upper part of the figure shows the number of active cells. The lower part shows the activity for the hidden cells 4 and 12, and the output cells 0 and 8. Dark gray areas indicate active processor (i.e. computing), while light gray areas means that the processor is idle (i.e. waiting for data). The white areas in the beginning and end (for cell 0 and 8) indicate waiting to start and finished, respectively. The hidden processors are computing almost 100 % of the time. We see that after two patterns, the output layer has to wait for the hidden layer output to arrive (large light gray area). The picture corresponds to the earlier given explanation in Figure 3. The white thin columns in the upper part of the performance plot arise from the communication between the output layer processors.

Figure 11 shows how the idle time for the output layer processors are reduced, when the number of processors in the system increases. The output cells becomes more active, since they do more commmu-

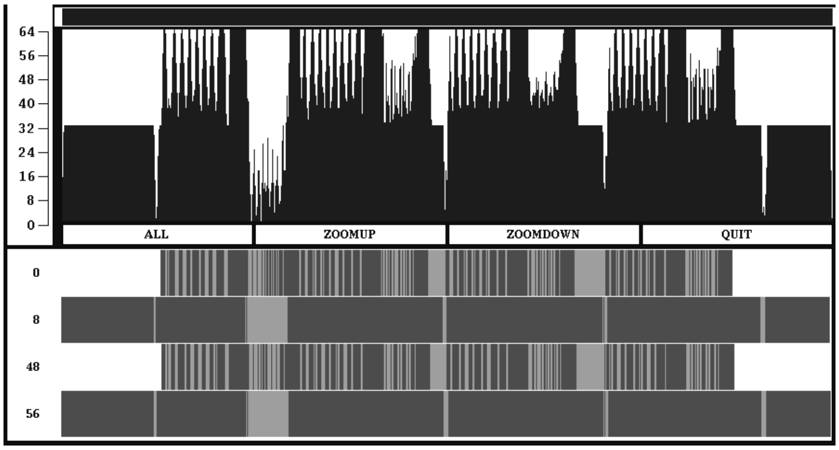

Figure 11: Performance on a 64 processor configuration, for the first 4 training patterns of NETtalk.

nication (error computation) than the hidden layer cells. Exhanging values between 8 processors need more time than between 4. The hidden layer processors becomes idle, while waiting for the error of the first pattern. Usually, there will be quite many training patterns between weight updating, thus this idle time will be a minor problem. In general, an application on a large parallel system needs to be of a smaller number of hidden and output neurons (compared to the number of input neurons), to obtain an equal load balance, compared to on a smaller system.

Since both the number of processors and neurons affect the load balance it is difficult to tell when to use 3APC instead of 2APC. However, this can easily be decided by comparing the time of one training iteration.

CONCLUSIONS

A description of how to exploit multiple degrees of BP parallelism on the AP1000 computer has been given. We have shown how the performance varies for different MLP network sizes and number of processors in the system.

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 performance 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. For the NETtalk application, we show that the load balance between hidden and output layer computation becomes more equal as the number of processors is increased.

The results should be of interest both to other MIMD computers with 2D-torus network and other large general purpose computers.

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] 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.

[6] 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.

[7] Jim Torresen et al. Parallel back propagation training algorithm for MIMD computer with 2D-torus network. In *Proceedings of International Conference On Neural Network Processing, Seoul, Korea*, volume 1, pages 140–145, October 1994.

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