Exploiting parallel computers to reduce neural network training time of real applications

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 and Information Science
Norwegian University of Science and Technology,
N-7034 Trondheim, Norway
Email: jim@nera.no

Abstract. Neural networks have been proposed to solve difficult problems like speech and character recognition. However, there has so far not come up any revolutionary system. This paper gives the results of a survey of the ongoing research on neural network applications. Moreover, we point out the demands for the mapping of neural applications onto parallel computer hardware. We propose a flexible mapping of back propagation trained neural networks onto a highly parallel computer.

The experiments undertaken show the need for application specific mapping of the given neural network and training set.

1 Introduction

Lots of neural network research has taken place for the last ten years. Some ask if neural networks really can do more than well-known statistical methods [1]. This paper surveys some of the present neural network applications trained by the backpropagation (BP) algorithm [2].

Speeding up neural network computation by using parallel processing machines has been deeply studied. Several neurocomputers and neurochips have been designed. However, there seems to be a gap between the research on computer architecture and neural applications. The performance of parallel implementations are often measured by non-real large network sizes [3]. Whereas, real applications often use small networks. One reason for this may be that application researchers often implement their programs on a single processor machine like a PC or a UNIX workstation. Therefore, they may have to scale down the network size. From the applications studied in this paper it is not possible to draw a single conclusion that recognition rate can be improved be making the networks larger. However, some applications will require parallel hardware to perform real-time recognition.

In this paper we propose a parallel BP implementation that runs real neural network applications efficiently. It is evaluated on Fujitsu AP1000, a message passing MIMD computer with two dimensional torus topology network [4]. 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.

2 Neural network applications

The main contributors to the neural network research, Widrow and Rumelhart, recently published a paper: "Neural networks: Applications in Industry, Business and Science" [5]. They list a wide variety of commercial applications of neural networks. The increase in products in the last years is partly explained by the availability of an increasingly wide array of dedicated hardware. However, many of the applications (e.g. control application) ought to be cheap to be of commercial interest. In [6] a survey of neural network applications, trained by backpropagation, is given. Mainly large applications — i.e. where parallel processing is of interest, was included.

Application	Network size (I x H x O)	No of tr. pat.
Cancer cell classification	3600 x 20 x 1	467
Coin recognition	259 x 5 x 6	
Handwritten digit recognition (P)	32 x 15 x 10	2000
Image compression	64 x 8,6,4 (x 64)	
Locating plant cells in image	100 x 15 x 10	52;67;69
NETtalk text-to-phonemes	203 x 60-120 x 26	5438
Object detector	961 x 50 x 1	50
Object inspection	4096 x 64 x 2	
Optical Character Recognition	3000 x 20 x 94	94
Optical Character Recognition	2500 x 100 x 94	1,128
Papanicoloau Smear Cell Classif.	80 x 6 x 1	702
Paper currency recognition	128 x 64 x 12	
Road classifications in satellite images	56 x 20 x1	552
Speech recognition	234 x 1000 x 61	1,300,000
Speech recognition	351 x 4000 x 61	6,000,000
Welding Defect Indentific.	15 x 10 x 9	1024

Table 1. Some neural network applications based on feed-forward neural networks. (P) indicate that a pre-processing network is used in front of the feed-forward meural network inputs. I, H, O indicate the number of input, hidden and output neurons, respectively.

Table 1 lists a summary of the survey. As far as possible, variations in the network structure are indicated. See [6] for references to each work.

3 Parallel processing for neural networks

From the survey in the previous section we see at least two common characteristics of many applications. First, some of the networks are small. The survey was concentrating on large networks, thus many other applications using small networks were omitted. Second, the output layer often consists of a small number of neurons. The latter indicates that it may be in vain to allocate many processors for computing the output layer.

It is obvious that some of the neural networks, like those recognition speech and vision applications, need parallel processing. This is needed both for training and when performing recognition. However, quite many applications can do without expensive parallel hardware. This may also be necessary for a product to be marketable, e.g. control applications in consumer products. Still, parallel processing can be used to reduce the training time of an application.

Below, first the parallel nature of the BP algorithm is described. Then, it is proposed how to train both small networks and networks with few output units on a parallel computer.

3.1 Parallel implementations of BP

The BP algorithm reveals several different kinds of parallelism, as described in [8, 9]:

- *Training session parallelism*, Starts training sessions with different initial starting values on different processors.
- *Training set parallelism*, Splits the training set across the processors. Each processor has a local copy of the complete weight matrix and accumulate weight change values for the given training patterns.
- *Pipelining*, Pipelines the training patterns in the layers, i.e. compute hidden and output layer on *different* processors. While the output layer processor calculates output and error values, the hidden layer processor concurrently processes the next training pattern.
- *Node parallelism*, The neurons within a layer run in parallel (named neuron parallelism). Further, the computation within each neuron may also run in parallel (Nordström [9] names this weight or synapse parallelism).

The weight update interval, μ, denotes the number of training patterns that is presented between weight updates.

Several BP implementations have been conducted on general purpose computers. Node parallelism is used to implement BP on Intel iPSC/860 hypercube in [10] and on MasPar MP-1 SIMD computer in [11]. A combination of training set and node parallelism for MasPar MP-1216 is given in [12]. Common to all these implementations is the lack of possible adaption to the large range of neural network applications. E.g., in the case of a node parallel implementation, it scales badly for *small* feed-forward networks.

Kumar et al. [3] propose a hybrid scheme using node and training set parallelism for hypercube architectures. The method makes it possible to vary the number of processors assigned to each degree of parallelism and utilizes the hypercube communication network. However, the effect of μ is not considered. As shown in [13], less frequent weight updates makes the convergence rate slower (i.e. the decrease in error per iteration is smaller) during training. Since the main interest is to minimize the *total* training time, it is impossible to omit the selection of a proper μ, when training set parallelism is used.

3.2 Small networks

To obtain high speedup on a highly parallel computer, we showed [14, 15] that *multiple* inherent degrees of parallelism in the BP algorithm should be combined. This is highly needed if the neural networks are of small size. The partitioning of the network is given in Figure 1, where pipelining and neuron parallelism are combined. Moreover, we included training set parallelism. In this paper, we propose a flexible and scalable mapping based on these results. While the previous work was on fixed combinations of the degrees of parallelism, the new scheme allows arbitrary combinations of training set parallelism and neuron parallelism. The proposed the mapping is shown in Figure 2. It combines neuron parallelism and training set parallelism. We allow the number of processors assigned for each degree of parallelism to be changeable. Within each dotted rectangle, neuron parallelism is used. Pipelining may also be included [16].

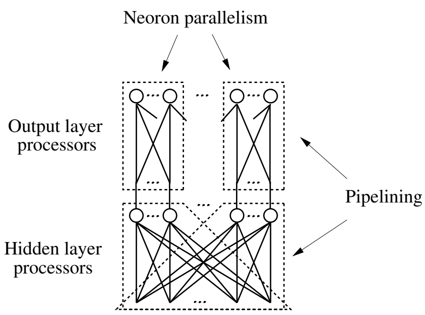

Fig. 1. Partitioning of the network, when pipelining and neuron parallelism are combined. The dotted lines show which *neurons* are mapped to each processor.

Below each processor mapping, the number of training set partitions (N_{TSP}) and the number of processors for neuron parallelism (N_{NP}) are given. The relation between them can be expressed by:

$$N_{TSP} = \frac{C_x C_y}{N_{NP}} \tag{1}$$

where C_x and C_y is the number of processors in the horizontal and vertical directions, respectively. For a given weight update interval, a larger number

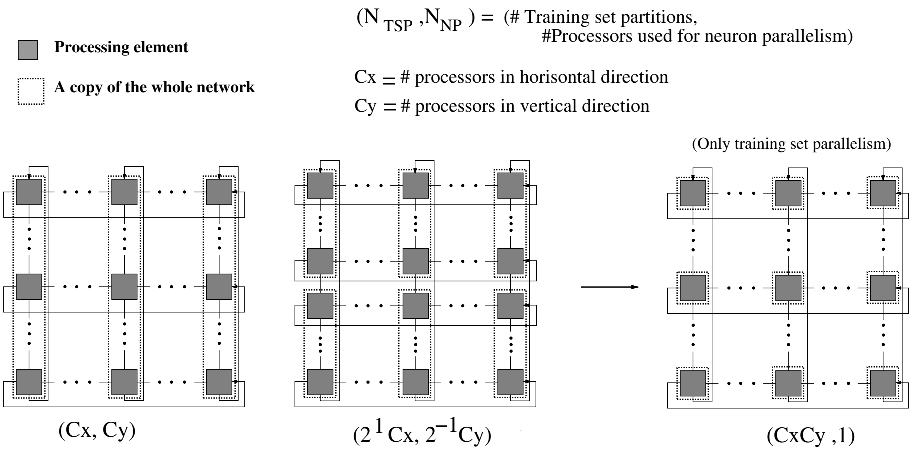

Fig. 2. Combination of training set and neuron parallelism.

of training set partitions imply fewer training pattern computations on each processor between weight updates.

Before running a neural application, we have to select the mapping giving shortest possible total training time. Thus, for each mapping, we have to predict the total training time for a set of weight update intervals. The total training time is given by:

$$T_{total} = T_{1it}(\mu)N(\mu) \tag{2}$$

where T_{1it} is the time for one iteration and $N(\mu)$ is the number of iterations needed. T_{1it} can be found either by estimation or by running each of the possible mappings for one iteration. $N(\mu)$ can be estimated based on the error after a few iterations [17]. We should select the mapping giving smallest possible T_{total} and use the corresponding μ-value.

3.3 Few output units

To train a network efficiently, when few output units is present, the number of processors assigned to the output layer computation should be reduced. Thus, we have to pipeline the computation so that hidden and output layer outputs are computed on different processors.

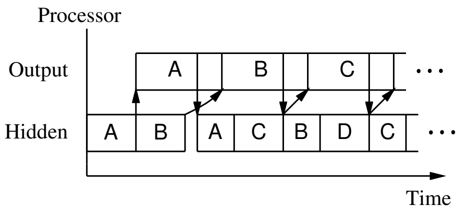

Fig. 3. Pipelining of the training patterns.

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.

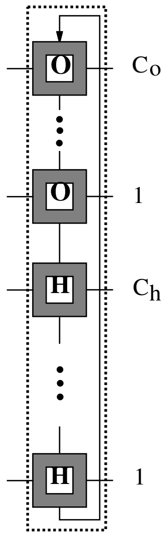

Fig. 4. Pipelined mapping of BP.

Figure 4 shows a possible pipelined mapping. The dotted rectangle represent one dotted rectangle in Figure 2. The processors marked **H** compute the hidden layer part, while the output layer part is computed on the processors marked **O**. For most cases, the number of processors assigned to the hidden layer, C_h, will be larger than the number of processors assigned to the output layer, C_o. The best configuration can be found by measuring the time for one pass through the training set for different processor assignments. Network contention occurs in this implementation, i.e. several processors may send messages concurrently to the same link. However, the communication time on AP1000 is reasonable, when network contention exists. It is approximately doubled if 5 messages are on the same channel, indicating that much time is spent within each processing element for communication overhead and less on the physical transfer of data [18].

4 Results and discussion

To be able to evaluate the benefit of a parallel back propagation algorithm that can vary the amount of each degree of parallelism, we implemented the mapping shown in Figure 2 on AP1000.

NETtalk

First, we use the NETtalk [19] neural network application in our experiments. The number of hidden units is 120. The training set consists of the 1000 most common English words (total 5438 characters). We have undertaken experiments with $\mu = 63, 259, 494, 906, 1360, 2719, 5438$.

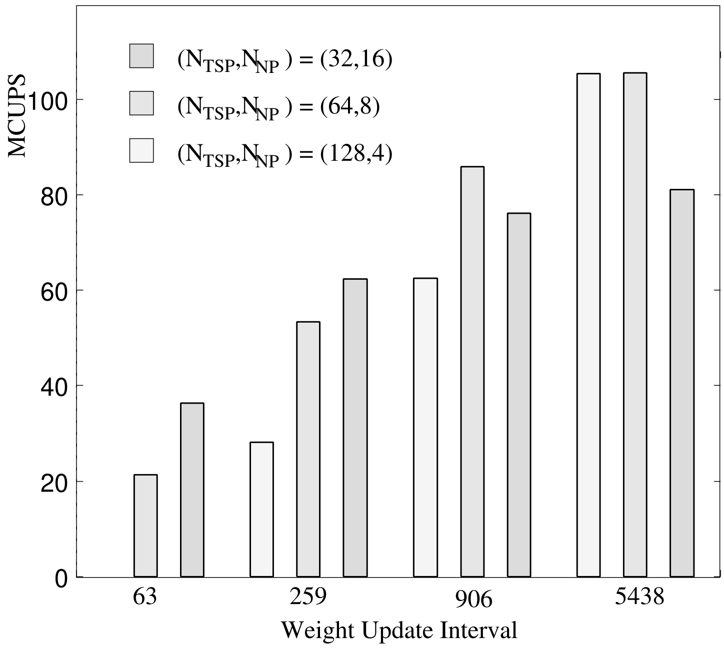

Fig. 5. MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 processor configuration running the NETtalk application. The number of hidden neurons is 120 and the performance for $\mu = 63, 259, 906, 5438$ are given.

Figure 5 shows the performance of NETtalk on a large system, for some μ-values. Performance is measured in MCUPS (Million Connections Updated Per Second). The configuration for each bar is indicated by (N_{TSP}, N_{NP}), as explained in the theory section. The performance improves with less frequent weight updates. The most efficient system is $(N_{TSP}, N_{NP}) = (32, 16)$ for frequent updates and $(N_{TSP}, N_{NP}) = (64, 8)$ for rare updates. This means that using 64 training set groups is better than using 32, even though network contention occurs.

Minimizing the total training time

In this section we show the relation between weight update interval and the number of iterations needed to obtain convergence for the NETtalk training set.

Moreover, we find the best weight update interval when running on AP1000.

The convergence was measured in percentage of the characters that have obtained an error of less than 0.1. Due to representation limitations of the network, the number of patterns that is not trained stabilized slightly below 5%. Figure 6 shows the number of iterations needed ($N(\mu)$) to reduce the error to 5% for the investigated μ-values.

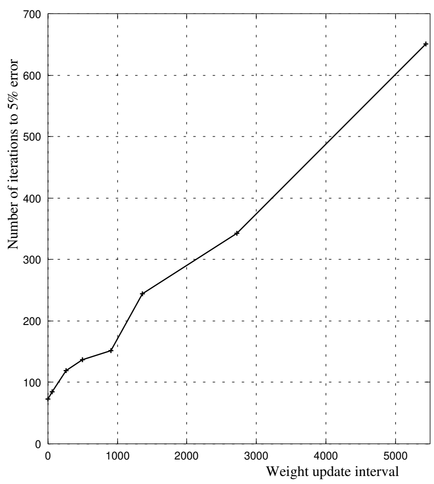

Fig. 6. Number of iterations needed to obtain convergence.

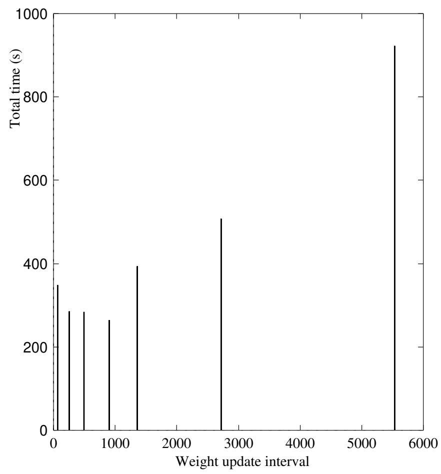

Fig. 7. Total training time for NETtalk running on 512 cells, using 120 hidden units.

The time for one iteration, T_{1it}, can be computed from Figure 5 (not all μ-values are shown). Hence, we can find the best weight update interval by employing (2) for μ= 63, 259, 494, 906, 1360, 2719, 5438. We get the total times as shown in Figure 7. We omit to accumulate weight changes for a pattern when the error is sufficiently small to avoid over-training, thus T_{1it} is decreasing with the convergence. Since we use a constant T_{1it}, the T_{total} will be larger than the real execution time. The best weight update interval is for every 906 pattern when $(N_{TSP}, N_{NP}) = (64, 8)$. Convergence is obtained after 265 s.

Running the same program (with parallel parts removed) on a Sparc10 workstation, using *learning by pattern*, converged after 73 iterations and 169 minutes. AP1000 speedup based on total execution time ($T_{1it}N$) is then 38 times. This is a conservative measure, since the decreasing T_{1it} for AP1000 is not accounted for. Thus, training time can be reduced from hours to minutes by using parallel processing.

5 Conclusion

The results of a survey of neural network applications trained by back propagation have been given. A mapping of the BP neural network training algorithm onto a highly parallel computer has been proposed. The mapping is flexible and designed to efficiently train both small networks and networks with different numbers of units in each layer. The results indicate the need for making a parallel mapping flexible to run the different real applications efficiently. Moreover, we have shown how the weight update interval influences on the total training time.

References

1. Tom Kavli. Nevrale nett: Hvor vil vi de neste årene ? In *Proc. of the Norwegian Neural Network Seminar*. SINTEF Instrumentation, November 1994.
2. 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.
3. Vipin Kumar et al. A scalable parallel formulation of the back propagation algorithm for hypercubes and related architectures. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1073–1090, October 1994.
4. Hiroaki Ishihata et al. Third generation message passing computer AP1000. In *Proc. of the International Symposium on Supercomputing*, pages 46–55, Nov. 1991.
5. Bernard Widrow et al. Neural networks: Applications in industry, business and science. *Communication of ACM*, 37(3):93–105, March 1994.
6. Jim Tørresen. *Parallelization of Backpropagation Training for Feed-Forward Neural Networks*. PhD thesis, Norwegian University of Science and Technology, 1996. ISBN 82-7119-906-4.
7. Terrence J. Sejnowski. NETtalk corpus, obtainable from ftp.idiap.ch in pub/benchmarks/neural/nettalk.tar.z.
8. Alexander Singer. Implementation of artificial neural networks on the Connection Machine. *Parallel Computing*, 14:305–315, Summer 1990.
9. Tomas Nordstrom and Bertil Svensson. Using and designing massively parallel computers for artificial neural networks. *Journal of Parallel and Distributed Computing*, 14(3):260–285, March 1992.
10. Darin Jackson and Dan Hammerstrom. Distributing back propagation networks over the Intel iPSC/860 hypercube. In *Proc. of Int. Joint Conference on Neural Networks*, volume I, pages 569–574, 1991.
11. G. Chinn et al. Systolic array implementations of neural nets on the MasPar MP-1 massively parallel processor. In *Proc. of Int. Joint Conference on Neural Networks*, volume II, pages 169–173, 1990.
12. Andreas Zell et al. Problems of massive parallelism in neural network simulation. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 1890–1895, 1993.
13. 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.

14. Jim Torresen, Shin-ichiro Mori, Hiroshi Nakashima, Shinji Tomita, and Olav Landsverk. Parallel back propagation training algorithm for MIMD computer with 2D-torus network. In *Proceedings of International Conference On Neural Information Processing (ICONIP'94), Seoul, Korea*, volume 1, pages 140–145, October 1994.
15. Jim Torresen, Shin-ichiro Mori, Hiroshi Nakashima, Shinji Tomita, and Olav Landsverk. Exploiting multiple degrees of BP parallelism on the highly parallel computer AP1000. In *Fourth International Conference on Artificial Neural Networks (ANN'95)*, pages 483–488, Cambridge, UK, June 1995. IEE.
16. Jim Torresen, Hiroshi Nakashima, Shinji Tomita, and Olav Landsverk. General mapping of feed-forward neural networks onto an MIMD computer. In *Proc. of IEEE Int. Conference on Neural Networks (ICNN'95)*, Perth, Western Australia, 27 November – 1 December 1995. IEEE.
17. Jim Torresen, Shinji Tomita, and Olav Landsverk. The relation of weight update frequency to convergence of BP. In *Proc. of World Congress on Neural Networks (WCNN'95)*, volume 1, pages 679–682, Washington, D.C., July 1995. INNS Press.
18. Hiroaki Ishihata. Performance evaluation of the AP1000. In *Proc. of CAP workshop*, pages N–1–8, 1991.
19. Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168, 1987.
20. Kwang Bo Cho et al. Image compression using multi-layer perceptron with block classification and SOFM coding. In *Proc. of World Congress on Neural Networks*, volume 3, pages 26–31, 1994.

This article was processed using the LaTeX macro package with LLNCS style