

Parallelization of Backpropagation Training for Feed-Forward Neural Networks

A dissertation submitted to
the Faculty of Electrical Engineering and Computer Science
of the Norwegian Institute of Technology

Jim Tørresen

Computer Systems Group
Department of Computer Systems and Telematics
The Norwegian Institute of Technology
The University of Trondheim

March 31, 1996

Abstract

The main objective of the work presented herein is to speed up neural network training using parallel processing. The back propagation trained feed-forward neural network was selected for this research, since it has attracted most interest among neural network researchers.

Today, large parallel computers are becoming more accessible to universities and research labs. Many implementations of neural networks on parallel computers have been reported. However, a survey of work on neural applications in this thesis indicates that there exists little contact between neural application developers and researchers in the field of parallel implementations of neural networks. Many implemented parallel neural training programs are tested based on unrealistic assumptions about real applications. Using large neural networks give rise to better performance than that obtainable in reality, since real neural networks are usually small.

This thesis describes experiments undertaken for both fixed and flexible implementations. The former uses a static assignment, which is independent of the network structure and training set. The latter uses an assignment strategy where the partitioning of the training set and neural network is based on the given application. As such, the parallel program can execute efficiently for a large range of neural applications.

Several real neural applications have been used in this work to test the implemented parallel algorithms. The results show that it is beneficial to use a flexible mapping. For small parallel systems, many parallel implementations can train efficiently. However, the importance of a flexible job assignment is more prominent as the number of processors increases. Thus, to gain the full benefit of a large parallel system the multiple inherent degrees of parallelism in the training algorithm must be combined. Also the method of combining the parallel aspects of the training algorithm should be adaptable according to the given neural application.

Two tests on convergence of back propagation trained neural network are also reported. The results indicate that if training set partitioning is used, in the parallel scheme, a larger number of iterations is needed for convergence. However, still the total training time required, by using a parallel computer, is reduced from hours to minutes with respect to a sequential computer.

Contents

1	Introduction	1
1.1	Objectives	3
1.2	Limitations	3
1.3	Contribution of this Work	3
1.4	Outline of this Thesis	5
2	Feed-Forward Neural Networks	7
2.1	The Multi-Layer Neural Network	7
2.1.1	A Detailed Description of the BP Learning algorithm	9
2.1.2	Basic Notation for the BP Algorithm	13
2.1.3	Momentum	14
2.1.4	Learning Performance	14
2.1.5	Generalization	17
2.1.6	Improvements of Feed-Forward Neural Networks	17
2.1.7	The Effect of the Weight Update Interval	18
2.2	Neural Network Applications	20
2.2.1	Speech	20
2.2.2	Image Processing	21
2.2.3	Miscellaneous Applications	24
2.2.4	Commercial Products	24
2.2.5	Summary of Applications	25

2.2.6	Do Neural Network Applications Need Parallel Hardware? . . .	26
3	Parallelization of Feed-Forward Neural Networks	27
3.1	Distributed Computing for Each Degree of BP Parallelism	27
3.1.1	Training Set Parallelism	27
3.1.2	Pipelining	28
3.1.3	Node Parallelism	28
3.1.4	Dimension of Each Parallel Degree	31
3.2	Parallel Computers for Simulating ANNs	31
3.2.1	General Aspects of Parallel Processing	31
3.3	A Survey of Different Parallel Implementations	34
3.3.1	General Purpose Computer Implementations	34
3.3.2	Interconnected Workstations	39
3.3.3	Research by Use of Models of Parallel Machines	39
3.3.4	FPGA Implementations	44
3.3.5	Transputer Implementations	45
3.3.6	Digital Signal Processor (DSP) Based Systems	49
3.3.7	Commercially Available Digital Neurocomputers	50
3.3.8	Other Technologies	53
3.3.9	General versus Special Purpose Hardware	55
3.3.10	Summary of the Parallel Mapping Schemes	55
3.3.11	Summary of the Performance	57
4	Hardware for Running Neural Networks	59
4.1	Fujitsu AP1000	59
4.1.1	Programming the System	60
4.1.2	Neural Network Implementations	61
4.2	The RENNS Computer System	61
4.2.1	Programming the System	63

4.2.2	Neural Network Implementations	63
4.3	Applications that Need Parallel Hardware	63
4.3.1	Neural Network Applications Used in this Work	64
4.4	Experimental Conditions in this Work	64
4.4.1	AP1000	64
4.4.2	RENNS	65
5	Fixed Mappings of BP Onto 2D-torus MIMD Architectures	67
5.1	Single Parallel Degree Solutions	67
5.1.1	Training Set Parallelism	67
5.1.2	Node Parallelism	68
5.2	Combined Solutions	70
5.2.1	Combining Two Degrees of Parallelism – Training Set and Neuron Parallelism (2APC)	70
5.2.2	Combining Three Degrees of Parallelism – Pipelining, Training Set Parallelism and Neuron Parallelism (3APC)	74
5.3	Summary	84
6	Comparing Fixed Parallel Implementations on AP1000	85
6.1	NETtalk	85
6.1.1	Training Set Parallelism	86
6.1.2	Node Parallelism	86
6.1.3	Combined Solutions	87
6.1.4	Comparing Load Balance Between the Hidden- and Output-Layer Processors	89
6.2	Other Applications	91
6.3	Minimizing the Total Training Time	94
6.4	Summary and Discussion	94
7	General Mapping onto 2D-torus MIMD Computers	97

7.1	The Proposed Mapping Scheme	98
7.1.1	The Implemented Mapping	101
7.1.2	Improvements of the Mapping	101
7.1.3	Training Session Parallel Scheme	104
7.2	Heuristic for Selection of the Best Mapping	105
7.2.1	Selection of Back Propagation Parameters	107
7.2.2	Estimation of Execution Time	108
7.2.3	Convergence Estimation	110
7.2.4	Modeling the Memory Usage	110
7.3	Summary	110
8	Results on the General and Flexible BP Mapping	113
8.1	NETtalk - Results and Discussion	113
8.1.1	Training Speed	114
8.1.2	Estimation of the Performance	117
8.1.3	The Convergence of NETtalk Learning	122
8.1.4	Minimizing the Total Training Time	132
8.2	Neural Network Trained to Classify Sonar Targets	134
8.2.1	Results on Sonar Target Classification	134
8.2.2	Comparing Sonar Results to NETtalk Results	137
8.3	Speech Recognition Network	138
8.4	Image Compression	139
8.5	Summary of the Application Adaptable Mapping	142
9	Implementation of BP on RENNS	145
9.1	Neuron Parallelism	145
9.2	Neuron Parallelism and Pipelining Combined	146
9.3	Comparison of Communication	151
9.4	Results	153

9.5	Summary	162
10	Concluding Remarks and Further Work	163
10.1	AP1000 Implementation	163
10.2	RENNS Implementation	164
10.3	General Comments	164
10.4	Future Work	165
A	Source code	167
B	Tables	199
B.1	Results for the Fixed Implementations On AP1000	199
B.2	Results for the Flexible Implementations On AP1000	204
B.3	Implementation On RENNS	221

Preface

This is a doctoral thesis submitted to the Norwegian Institute of Technology for the doctoral degree “doktor ingeniør”. The research work presented in this dissertation was carried out between October 1993 and October 1994 at the Department of Information Science, Kyoto University, Japan and between November 1994 and March 1996 at the Department of Computer Systems and Telematics (IDT), The Norwegian Institute of Technology (NTH). Before starting this study, I carried out the doctor course part, starting January 1992.

At Kyoto University, the work started by implementing backpropagation neural network training on the general purpose computer Fujitsu AP1000. Several different parallel programs were written. At NTH, I was able to do experiments on different target architectures. I continued my work on AP1000 by use of *telnet* connection over Internet to Japan and also conducted experiments using the RENNS computer. The AP1000 computer consists of 512 processing elements, while RENNS is operative with 15 processing elements.

Program development was in the C programming language and carried out on various UNIX workstations. For writing this thesis, \LaTeX text formatting program on a UNIX workstation has been adopted. Descriptive figures have been drawn by using *Idraw*. Result figures have been generated by *Gnuplot/Tgif* and *Microsoft Excel*.

Acknowledgments

Professor Olav Landsverk at NTH has supported the work of this thesis through his deep knowledge and long experience in computer architecture and design. I owe him my best thanks for his encouragement and supervision. At Kyoto University my supervisor was professor Shinji Tomita. I am very thankful to him for inviting me to his lab and making my stay in Japan a success both scientifically and socially – showing me the the interesting Japanese culture. His deep knowledge in parallel processing and computer architecture design was a great help to my work.

Many people made my stay at Kyoto University easy and fruitful. I would like to thank associate professor Hiroshi Nakashima and assistant professor Shin-Ichiro Mori for giving me useful comments during my preperation of articles. I would also like to thank Hesham

Keshk for helping me with using the computers and all our interesting discussions. All the other laboratory members were very helpful and many thanks to all of you. Much of this study is undertaken by use of the computer AP1000 and I will express my gratefulness to Fujitsu Ltd for offering me access to the computer. Without the NTNF Research Fellowship from The Research Council of Norway, Dept. for Scientific and Industrial Research (NTNF), my visit to Japan would not have been possible. Thus, I am most thankful for their support.

I am most grateful to the Department of Computer Systems and Telematics (IDT) at NTH for financial support throughout my doctoral studies. I acknowledge the inspiring and supportive environment provided by the students as well as staff in the Computer Systems Group at IDT. My fellow Ph.D. students Jarle Greipsland, Gaute Myklebust, Jon Solheim and Lisbet Utne have inspired me, and I appreciate their co-operation and feedback on my work. Thanks to Pauline Haddow and Inge Viken for their valuable and thorough comments on the pre-printed version of this thesis.

My family has always encouraged me and been helpful to me throughout my life. This has been of a great value and I would like to thank my parents, brother, and sister.

Finally, special thanks to Kirsten for her constant support and encouragement.

Trondheim, March 1996
Jim Tørresen

Nomenclature

Words and Phrases

Cell	Processing Element
Epoch	See Iteration
Iteration	A complete cycle of training pattern presentation
Neural networks	Artificial neural networks
Unit	Neuron

Acronyms

ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
CPS	Connections Per Second
CPU	Central Processing Unit
CUPS	Connections Updated Per Second
DRAM	Dynamic RAM
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FLOPS	FLoating-point OPerations Per Second
LCA	Logic Cell Array
MIMD	Multiple Instruction stream Multiple Data stream
MLP	Multi-Layer Perceptron network
NTH	Norwegian Institute of Technology
PE	Processing Element
RAM	Random-Access Memory
RENNS	REconfigurable Neural Network Server
SIMD	Single Instruction stream Multiple Data stream
SRAM	Static RAM
VLSI	Very Large Scale Integration
VRAM	Video RAM

Symbols:

\propto	Proportional to
C	Total number of processing elements (cells)
C_x, C_y	Total number of PEs in x,y dimension
$c_{x,y}$	2-D processing cell index
$d_{p,k}$	Target value for pattern p and output unit k
e	Error threshold
E_p	Error measure for pattern p
E	Error measure for the complete training set
$E\%$	Percentage of patterns not properly trained
E_{RMSE}	Root Mean Square Error
f	Sigmoid function
F	Number of floating point operations
k_e	Ratio for number of iteration for <i>learning by epoch</i> versus <i>learning by pattern</i>
L	Vector length
m	The number of intermediate processor modules
N	Total no of training iterations needed
N_i	Number of input units
N_h	Number of hidden units
N_o	Number of output units
n	Training iteration index
n_i	Number of input units assigned to a processor
n_h	Number of hidden units assigned to a processor
n_o	Number of output units assigned to a processor
P	Total number of patterns in the training set
P_c	Number of training patterns on processor c
p	Pattern index
t_x	Time used for operation x in seconds
T_x	Total time used for operation x in seconds
$w_{h,ji}$	Weight value, connecting input unit i and hidden unit j
$w_{o,kj}$	Weight value, connecting hidden unit j and output unit k
$\Delta w_{h,ji}$	Weight change value for the weight $w_{h,ji}$
$\Delta w_{o,kj}$	Weight change value for the weight $w_{o,kj}$
x_i	Input value to neuron i .
$y_{h,j}$	Output value from hidden unit j .
$y_{o,k}$	Output value from output unit j .

Greek letters:

α	Momentum
γ	Expression for training iterations as a function of μ
θ	Bias value
$\delta_{h,j}$	Delta error value for hidden unit j
$\delta_{o,k}$	Delta error value for output unit k
η	Learning rate
μ	Weight update interval

Chapter 1

Introduction

Artificial neural networks (ANNs) have recently become a promising solution to problems for which the human brain is superior to conventional computing.

ANNs are based on the present understanding of the human biological nervous system, see Figure 1.1. It is built of cells, also called neurons. Dendrites extend from the cell body and connect to other cell inputs (axons) through synapses. It is estimated that the human

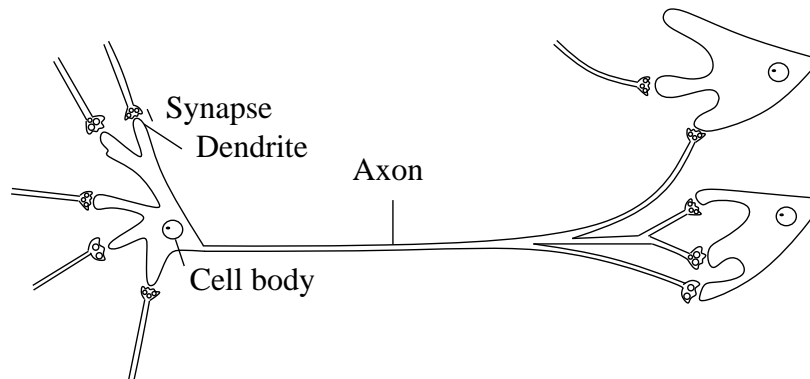


Figure 1.1: A simplified model of biological neurons.

brain contains 10^{11} neurons each with up to 10,000 connections to other neurons. As such it represents a massively parallel system with hierarchical structured interconnection. This feature inspires the ANN researchers to design systems that are based on parallel processing as an alternative to the single processing architecture.

One of the problem areas where ANNs have shown good results is pattern classification, i.e. non-linear separation of the feature vectors. It is superior to the traditional knowledge-based Artificial Intelligence (AI) method in the way of “black-box” modeling a system. The methods differ in that ANN specify the system by *presenting* a set of input and output

vectors to which the system should adjust. AI, on the other hand, requires a *specification* of the internal rules of the system, which is a much harder task. A description of various neural networks is given in several books [38, 49, 71, 149] and articles [74]. A neural network must be trained (learning phase) before it can be applied (recall phase). The goal in training the network is to adjust the connections (weights) so that application of a set of inputs produces the desired set of outputs.

ANNs can be subdivided into three main groups according to the training approach: supervised, reinforcement and unsupervised trained networks. In the former case – supervised training, the training data consists of pairs of input/output patterns, whereas, in the latter case – unsupervised training, it consists of input patterns only. Also for the case of reinforcement training, only input patterns are supplied. However, occasionally a “performance score” is given that tells how well the training of the network has done since the last score was given.

Unsupervised training is based on clustering, that is, the weights are adjusted so that similar input patterns lead to a response in the same cluster of neurons. Adaptive Resonance Theory (ART) and Self-Organizing Feature Map (SOM) are neural network models based on unsupervised learning.

The most popular supervised trained artificial neural network¹ is the Multi-Layer Perceptron network (MLP) [112]. It consists of several layers of computational elements. The non-recurrent version of the network is also called *feed-forward neural network*, since an output cannot influence its neuron input values. By contrast, recurrent networks has feedback connections so that outputs may be determined by their current input and previous outputs. Neural network research grew rapidly with the introduction of the backpropagation (BP) algorithm for training the feed-forward neural network. This training algorithm is applied for the research presented in this thesis, and the algorithm will be thoroughly explained in the next chapter.

Simulating neural networks is very computationally demanding, thus, to reduce long training and recall time, parallel computation is mandatory. Many researchers have proposed parallel hardware implementations (see Section 3.3). Neural network training can be parallelized either by network partitioning or by training set partitioning. The parallel aspects of feed-forward neural network training will be further discussed in the next chapters.

There seems to be disjoint interests among neural application developers and researchers in the field of parallel implementations of neural networks. Neural applications can for many instances do well without parallel processing. Many proposed parallel implementations, on the other hand, are tested on non-existing neural networks. Thus, there are a limited number of researchers combining real neural network applications and parallel processing. To overcome this, parallel implementations ought be made more general and adaptable to the neural network applications that require parallel processing.

¹Hopfield net and Hamming net are other networks based on supervised learning.

1.1 Objectives

The main objective of this study was to provide a more solid basis for parallel processing of neural network training. The goal was to implement several different parallel programs for training neural networks and test them using real neural network applications. Based on the initial implementations and results from other researchers a more general mapping scheme could be searched for. This can be formulated by the following sub-objectives:

- Survey the neural network applications to determine the demands for parallel processing.
- Study the published work on parallel implementations of neural networks.
- Based on the applications requiring parallel processing and the limitations of the present parallel implementations, search for an appropriate mapping strategy.

1.2 Limitations

In this study, the basic back propagation algorithm for training feed-forward neural networks has been applied. Preprocessing techniques and variants of the training algorithm are outside the scope of this thesis. Training sets applied consists of those freely available.

The desire of this work has been to find an effective parallel backpropagation training algorithm. To do this in the available frame of time, it was necessary to limit the number of experiments for some of the initial implementations. Further, estimation of execution time and computation of total training time were only made for the case of the most flexible implementation.

Many interesting problems encountered during this work were unable to be solved in the time limit. These are therefore recommended as future work – see Chapter 10.

1.3 Contribution of this Work

In this work several parallel BP neural network algorithms are proposed and implemented. They are written by the author of this thesis, if not otherwise stated.

The contributions can be summarized in the following list:

- Several parallel BP schemes have been implemented and compared (Chapter 5). This includes a proposed combination of three degrees of backpropagation parallelism. The work was published in the 1994 International Conference On Neural Information

Processing [135], two domestic Japanese conferences [136, 137], and one domestic Norwegian conference [138].

- A weight change summing scheme is proposed (Section 5.2.2) and presented in the Fourth International Conference on Artificial Neural Networks (1995) [139].
- Several parallel BP algorithms are proposed that assign the best number of processors to each degree of parallelism to minimize training time for all kinds of feed-forward networks (Chapter 7). This work is partly published in the 1995 IEEE International Conference on Neural Networks [140].
- The accuracy of execution time estimation based on the number of floating point operations and communication time has been tested (Section 7.2.2 and 8.1.2).
- For two applications (NETtalk and sonar target classification), the relation between the weight update interval and convergence has been determined (Section 8.1.3 and 8.2). For the NETtalk application the number of training iterations has been estimated based on the initial error (Section 8.1.3). The NETtalk results were partly published in the 1995 World Congress on Neural Networks [142].
- The near-optimal weight update interval for implementing the NETtalk learning on AP1000 is found (Section 8.1.4) [140].
- It is shown that distributing the computation in each layer to different processors improves the performance (Chapter 9). A paper on this study has been submitted [141].

For the international conferences ([135, 139, 140, 142]) the whole content the papers has been reviewed by referees and printed in the conference proceedings.

In my first period at NTH, I participated in the assembly of the RENNS computer by writing a down-loading program for the configuration file for the communication subsystem. Moreover, I wrote DRAM testing programs for the main board.

In have started to look into the research of Artificial Life and a popular science article on the topic has been written [134].

A chapter is invited for a book proposal to IEEE Press covering the parallel implementations of different neural networks. The chapter to be contributed is named “Implementation of Backpropagation training of Neural Networks On Large Parallel Computers”. The editors of the book are professors at Nanyang Technological University, Singapore.

1.4 Outline of this Thesis

The thesis is divided into a background part (Chapter 1 – 4) followed by three research parts (Chapter 5 – 10). After this introductory chapter, the second chapter details the back propagation neural network training algorithm used in this study. A survey of applications for this neural network is then included.

The background part is continued in Chapter 3 and starts by describing the degrees of parallelism in the backpropagation algorithm. An overview of parallel processing is also included, followed by a description of work published by other researchers on parallel implementations of BP. Chapter 4 contains a presentation of the parallel computers used in this work.

The first research part is outlined in Chapters 5 and 6, which include a description and results of four algorithms implemented on the general purpose computer AP1000. These are based on a fixed assignment of processors, independent of the neural application to be trained.

The limitations of these mappings are described and in the second research part a dynamic mapping is proposed (Chapters 7). The implementation of it is tested on AP1000 and results are included in Chapter 8.

The final research part (Chapter 9) of this thesis reports a proposed mapping for the reconfigurable computer RENNS. This is a mapping utilizing the flexibility in the interconnection of the processors.

Finally, closing comments and further work are given in Chapter 10.

Chapter 2

Feed-Forward Neural Networks

Artificial neural network research was first initiated in the 1940s, when McCulloch and Pitts published their study [78]. They started the development of the perceptron model, shown in Figure 2.1. In the 1960s, convincing demonstrations using the model were made

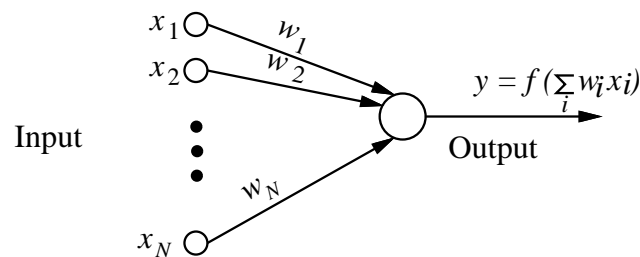


Figure 2.1: The perceptron.

[149]. However, Minsky and Papert, in the late 1960s, proved that severe restrictions in the learning capabilities of a single layer model exist [83]. For the following twenty years little research was undertaken. Then, the learning of multi-layer perceptron network, called backpropagation (BP), was introduced by Rumelhart et al. in 1986 [112]. This initiated a remarkable increase in the neural network research.

2.1 The Multi-Layer Neural Network

A two layer¹ feed-forward network is shown in Figure 2.2. The network is called fully connected, since there are all-to-all connections between two adjacent neuron layers. The

¹In this thesis, the word layer is used about the number of layers of weights in the network. This is equal to the number of layers of neurons, when excluding the input neuron layer.

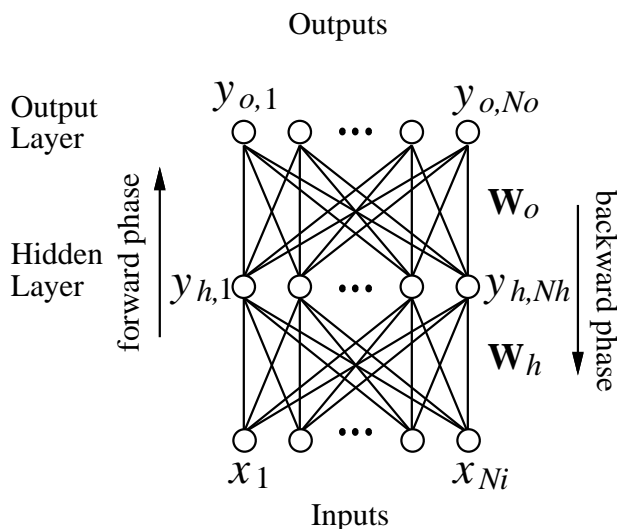


Figure 2.2: A two weight layer feed-forward neural network.

number of neurons (also called units) in each layer is N_i , N_h , and N_o for the input, hidden, and output neuron layer, respectively. The network can be extended to any number of layers, however, most applications (see Section 2.2) use two weight layers. Therefore, this work has been restricted to two layer networks. The BP learning phase for a pattern consists of a forward phase followed by a backward phase. The main steps are given in Figure 2.3.

-
1. Initialize the weights to small random values.
 2. Select a training vector pair (input and the corresponding output) from the training set and present the input vector to the inputs of the network.
 3. Calculate the actual outputs - *forward phase*.
 4. According to the difference between actual and desired outputs (error), adjust the weights \mathbf{W}_o and \mathbf{W}_h to reduce the difference - *backward phase*.
 5. Repeat from step 2 for all training vectors.
 6. Repeat from step 2 until the error is acceptably small.
-

Figure 2.3: Backpropagation learning.

The weight updating scheme used is called *learning by pattern* (or online weight update) and

updates the weights after *each* training pattern has been presented. This update method, which is stochastic gradient descent [152], has been shown by experiments to converge faster than the total gradient descent – which updates the weights after all training patterns have been presented. However, only the latter method, called *learning by epoch*, has been proved to converge². An intermediate method is called *learning by block* and it updates the weights after a certain number of patterns have been presented [88].

2.1.1 A Detailed Description of the BP Learning algorithm

In the forward phase the hidden layer weight matrix \mathbf{W}_h is multiplied by the input vector $\mathbf{X} = (x_1, x_2, \dots, x_{N_i})^T$, to calculate the hidden layer output

$$y_{h,j} = f\left(\sum_{i=1}^{N_i} w_{h,ji}x_i - \theta\right) \quad (2.1)$$

where $w_{h,ji}$ is the weight connecting input unit i to unit j in the hidden neuron layer³. The θ is an offset termed bias, incorporated into the training algorithm by a weight connected to +1 for each neuron [149]. This bias-weight is trained like an ordinary weight.

The function f is a nonlinear activation function. Normally the S-shaped sigmoid function

$$f(\alpha) = \frac{1}{1 + e^{-\alpha}} \quad (2.2)$$

is used. It compresses the output value to lie in $< 0, 1 >$, as shown in Figure 2.4. Moreover, the function is differentiable, which is a demand of the training algorithm – see Equation 2.13.

The output from the hidden layer, $y_{h,j}$, is used to calculate the output of the network, $y_{o,k}$

$$y_{o,k} = f\left(\sum_{j=1}^{N_h} w_{o,kj}y_{h,j} - \theta\right) \quad (2.3)$$

The error measure E_p for a training pattern p is given by

$$E_p = \frac{1}{2} \sum_{k=1}^{N_o} (d_{p,k} - y_{p,o,k})^2 \quad (2.4)$$

The overall measure for a training set of P patterns is

$$E = \sum_{p=1}^P E_p \quad (2.5)$$

²However, there is not known any theoretical bound on the number of presentations required for convergence.

³To distinguish the different neuron layers, the indices i , j , and k are used for indexing the input, hidden, and output neuron layer, respectively.

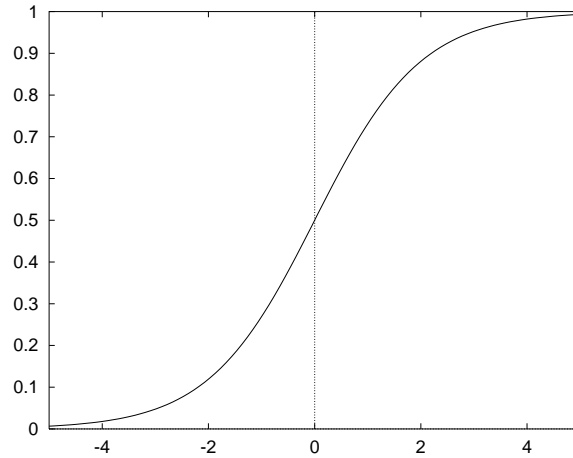


Figure 2.4: The sigmoid function $f(\alpha) = \frac{1}{1+e^{-\alpha}}$

In the expressions below the pattern index p has been omitted on all variables except for E_p to improve clarity. In the backward phase the target, d , and output, y_o , are compared and the difference (error) is used to adapt the weights to reduce the error. The weights are adjusted according to the *generalized delta rule*. This differs from the original delta rule, by the inclusion of non-linear activation function and hidden units. The rule is also called gradient descent, since it corresponds to performing the steepest descent in weight space where the height is equal to the error measure. It requires that the derivative of the error measure, with respect to each weight, is proportional (with negative constant) to the weight change computed by the delta rule, as given by

$$\Delta w_{o,kj} \propto -\frac{\partial E_p}{\partial w_{o,kj}} \quad \wedge \quad \Delta w_{h,ji} \propto -\frac{\partial E_p}{\partial w_{h,ji}} \quad (2.6)$$

The $\frac{\partial E_p}{\partial w}$ expression indicates how much a change in a weight alters – preferably reduces, the output error. The sum of the weight-input product for the output layer is

$$net_{o,k} = \sum_{j=1}^{N_h} w_{o,kj} y_{h,j} - \theta \quad (2.7)$$

The chain rule is used to write the derivative of the left-hand expression in Equation 2.6 as a product of two parts

$$\frac{\partial E_p}{\partial w_{o,kj}} = \frac{\partial E_p}{\partial net_{o,k}} \frac{\partial net_{o,k}}{\partial w_{o,kj}} \quad (2.8)$$

The first part represents the change in error as a function of the change in the network output and the second part represents the effect of changing a particular weight on the network output. Similarly, the chain rule can be applied to the hidden layer.

From Equation 2.7 the second factor is given by

$$\frac{\partial net_{o,k}}{\partial w_{o,kj}} = \frac{\partial}{\partial w_{o,kj}} \left(\sum_{j=1}^{N_h} w_{o,kj} y_{h,j} - \theta \right) = y_{h,j} \quad (2.9)$$

The first factor is used to define the delta error

$$\delta_{o,k} = -\frac{\partial E_p}{\partial net_{o,k}} \quad (2.10)$$

The chain rule can be applied to compute

$$\frac{\partial E_p}{\partial net_{o,k}} = \frac{\partial E_p}{\partial y_{o,k}} \frac{\partial y_{o,k}}{\partial net_{o,k}} \quad (2.11)$$

Differentiating Equation 2.4 for a pattern p leads to

$$\frac{\partial E_p}{\partial y_{o,k}} = -(d_k - y_{o,k}) \quad (2.12)$$

Equation 2.2 is differentiated to get the value

$$\frac{\partial y_{o,k}}{\partial net_{o,k}} = y_{o,k}(1 - y_{o,k}) \quad (2.13)$$

Substitution in Equation 2.10 and rewriting the expression gives

$$\delta_{o,k} = y_{o,k}(1 - y_{o,k})(d_k - y_{o,k}) \quad (2.14)$$

The chain rule is applied for the hidden units

$$\frac{\partial E_p}{\partial y_{h,j}} = \sum_{k=1}^{N_o} \frac{\partial E_p}{\partial net_{o,k}} \frac{\partial net_{o,k}}{\partial y_{h,j}} = \sum_{k=1}^{N_o} \frac{\partial E_p}{\partial net_{o,k}} \frac{\partial}{\partial y_{h,j}} \left(\sum_{j=1}^{N_h} w_{o,kj} y_{h,j} - \theta \right) = -\sum_{k=1}^{N_o} \delta_{o,k} w_{o,kj} \quad (2.15)$$

Similar to computing the output delta error in Equation 2.10 and 2.11, the hidden delta error value for neuron j is

$$\delta_{h,j} = -\frac{\partial E_p}{\partial y_{h,j}} \frac{\partial y_{h,j}}{\partial net_{h,j}} = y_{h,j}(1 - y_{h,j}) \sum_{k=1}^{N_o} \delta_{o,k} w_{o,kj} \quad (2.16)$$

The expressions in Equation 2.6 can then be rewritten as

$$\Delta w_{o,kj} = \eta \delta_{o,k} y_{h,j} \quad \wedge \quad \Delta w_{h,ji} = \eta \delta_{h,j} x_i \quad (2.17)$$

where η is the learning rate coefficient.

If *learning by pattern* is applied, the output layer weights are changed to $w'_{o,kj}$

$$w'_{o,kj} = w_{o,kj} + \eta \delta_{o,k} y_{h,j} \quad (2.18)$$

The hidden layer weights are updated accordingly

$$w'_{h,ji} = w_{h,ji} + \eta \delta_{h,j} x_i \quad (2.19)$$

The training continues for each vector in the training set until the error for the entire set becomes acceptably small.

Instead of updating the weights after each training pattern presentation, they can be updated less frequently by using *learning by block*. For updates after μ patterns have been presented. Equations 2.18 and 2.19 become 2.20 and 2.21

$$w'_{o,kj} = w_{o,kj} + \eta \sum_{p=p'+1}^{p'+\mu} \delta_{p,o,k} y_{p,h,j} \quad (2.20)$$

$$w'_{h,ji} = w_{h,ji} + \eta \sum_{p=p'+1}^{p'+\mu} \delta_{p,h,j} x_{p,i} \quad (2.21)$$

The total number of training patterns is P and $\mu \leq P$. *Learning by block* is well suited to parallelization, but as shown by Paugam-Moisy [105], the convergence rate declines as μ gets larger. Therefore, an appropriate value for μ need to be chosen.

The error measure in equation 2.5 is dependent on N_o and P . Thus, large numbers result in a large error. Another measure for the total error is the root mean square error (RMSE)

$$E_{RMSE} = \sqrt{\frac{1}{PN_o} \sum_{p=1}^P \sum_{k=1}^{N_o} (d_{p,k} - y_{p,o,k})^2} \quad (2.22)$$

The error threshold e for a pattern is in this thesis defined by

$$E_p \leq e \text{ when pattern } p \text{ is learned} \quad (2.23)$$

Moreover, the pattern error for the training set is defined by

$$E_{\%} = \text{Number of patterns with } E_p > e \text{ in percentage of all training patterns} \quad (2.24)$$

As stated, there is no proven convergence⁴ for the backpropagation algorithm and therefore the word *convergence* is defined as reaching a pre-determined stopping criteria [105].

⁴An exception is *Learning by epoch*, but still there is no bound in the number of training iterations required.

2.1.2 Basic Notation for the BP Algorithm

In the following chapters, reference is made to the neural network terminology. Therefore it has been chosen to include the following summary of the most common terms.

Training Set. Consists of a number of training patterns, each given by an input vector and the corresponding output vector.

Network size. A network of N_i input units, N_h hidden units, and N_o output units is for short written $N_i \times N_h \times N_o$. Note that the word *network* is used for neural network in this thesis and not for processor topology network. In the case of the latter use of the word it will be explicitly stated.

(Training) Iteration. Denotes *one* presentation of the whole training set.

Weight updating strategies. Three different approaches:

- *Learning by pattern (lbp)*, update the weights after *each* training pattern has been presented.
- *Learning by block (lbb)*, update the weights after a subset of the training patterns has been presented.
- *Learning by epoch (lbe)*, update the weights after all patterns have been presented (i.e. one training iteration).

Weight update interval. The number of training patterns that is presented between weight updates is termed μ . For *lbp*, $\mu = 1$, while for *lbe* $\mu = P$, where P is the number of training patterns in the training set.

Degrees of parallelism, The BP algorithm reveals four different kinds of parallelism, as described in [101, 122]. This topic is explained in greater detail in Chapter 3.

- *Training session parallelism*, Starts training sessions with different initial training parameters on different processing elements.
- *Training set parallelism*, Splits the training set across the processing elements. Each element has a local copy of the complete weight matrix and accumulates weight change values for the given training patterns. The weights are updated using *lbb/lbe*.
- *Pipelining*, Pipelines the training patterns between the layers, i.e. compute hidden and output layer on *different* processors. While the output layer processor calculates output and error values for the present training pattern, the hidden layer processor processes the *next* training pattern. The forward and backward phase may also be parallelized in a pipeline. Pipelining requires a delayed weight update or *lbb/lbe*.

- *Node parallelism*, The neurons within a layer is computed in parallel (named neuron parallelism). Further, the computation within each neuron may also run in parallel (Nordström [101] names this *weight* or *synapse* parallelism). In this method, the weights can be updated using *lbp*.

2.1.3 Momentum

To obtain true gradient descent requires infinitesimal small changes of the weights. This is obtained by selecting a small value for the learning rate. However, we want to choose a learning rate as large as possible without leading to oscillation⁵, since experiments show that this offer the most rapid learning [112]. To increase the learning rate and avoid oscillation, a momentum can be included. Rumelhart et al. proposed [112] to add a fraction, equal to α , of the previous weight update value to the current weight change

$$\Delta w_{o,kj}(p+1) = \eta \delta_{o,k}(p+1) y_{h,j}(p+1) + \alpha \Delta w_{o,kj}(p) \quad (2.25)$$

where p is the training pattern index. The weights are updated

$$w_{o,kj}(p+1)' = w_{o,kj}(p) + \Delta w_{o,kj}(p+1) \quad (2.26)$$

Similarly, Sejnowski et al. [118] proposed a smoothing term, α

$$\Delta w_{o,kj}(p+1) = \alpha \Delta w_{o,kj}(p) + (1 - \alpha) \delta_{o,k}(p+1) y_{h,j}(p+1) \quad (2.27)$$

The smoothing makes it less necessary to scale the learning rate according the weight update interval.

$$w_{o,kj}(p+1)' = w_{o,kj}(p) + \eta \Delta w_{o,kj}(p+1) \quad (2.28)$$

The equations for updating of the hidden weights can be similarly derived. The term α is normally set to around 0.9.

2.1.4 Learning Performance

There exist two commonly used metrics for the speed of neural network simulations. For the recall phase performance, Connections Per Second (CPS) are used, which describe the number of weight multiplications in the forward pass per second. Performance during training is measured in Connections Updated Per Second (CUPS). This accounts for the number of weights updated per second.

According to Crowl [21], presentation of parallel performance can be easily and unintentionally distorted. To avoid this, he suggests presenting the elapsed time as opposed to

⁵The error is not constantly decreasing but is oscillating between large and small error values without reaching convergence.

speedup where possible. The use of speedup to define performance is limited by the lack of consensus for speedup definition. Crowl is of the opinion that many machines sacrifice sequential performance for parallel scalability and that this gives rise to overestimated speedup. Linear speed, i.e. solutions per time unit, is visually similar to speedup and may be used instead. CPS and CUPS measure linear speed by connections computed or updated per second.

The CUPS measure is sensitive to several features. Even though the inclusion of momentum increases the convergence, the training *speed* is slowed down. That is, there are more computation per iteration, leading to reduced CUPS performance. The output layer has less backward error computation than the hidden layer. Thus, if more hidden layers are added to a network, the CUPS performance will be reduced. When the number of neurons is large, the computation grains become large, which in most cases improve performance compared to that obtained from a small number of neurons. However, if the network is too large to be stored in main memory the training is slowed down.

The number of floating point operations used for weight updating for *learning by pattern* differs from the number for *learning by block/epoch*. In this section, expressions will be derived to show the difference in computation between the different weight update strategies. The expressions are based on serial executing of the training program. On a parallel computer, additional time is used for communication. The NETtalk application with number of neurons $N_i = 203$, $N_h = 120$, $N_o = 26$, and $P = 5438$ training patterns is used below to illustrate the difference. The NETtalk training uses the Sejnowski momentum (Equation 2.27 and 2.28) for weight update. The number of floating point operations used for *lbp* weight updating is then 6 per weight, leading to a total number for one training iteration

$$F_{lbp} = 6P(N_i + N_o)N_h \quad (2.29)$$

$$= 6 \cdot 5438(203 + 26)120 = 896,617,440 \quad (2.30)$$

For *lbb* the number of floating point operations for weight accumulation and updating, 2 and 5 respectively, is found from Equation 2.20, 2.21, 2.27 and 2.28. This leads to a total floating point operation count of

$$F_{lbb} = 2P(N_i + N_o)N_h + 5 \left[\frac{P}{\mu} \right] (N_i + N_o)N_h \quad (2.31)$$

$$= (2P + 5 \left[\frac{P}{\mu} \right]) (N_i + N_o)N_h \quad (2.32)$$

$$= (2 \cdot 5438 + 5 \left[\frac{5438}{\mu} \right]) (203 + 26)120 \quad (2.33)$$

$$= 298,872,480 + \frac{1}{\mu} 747,181,200 \quad (2.34)$$

The first expression in Equation 2.31 represents the weight accumulation and the latter represents weight updating. The number of operations for weight accumulation is less

than for weight update. Thus, if the weights are infrequently updated we get $F_{lbp} > F_{lbb}$. Figure 2.5 shows the number of floating point operations for weight accumulation and update for different weight update intervals, μ .

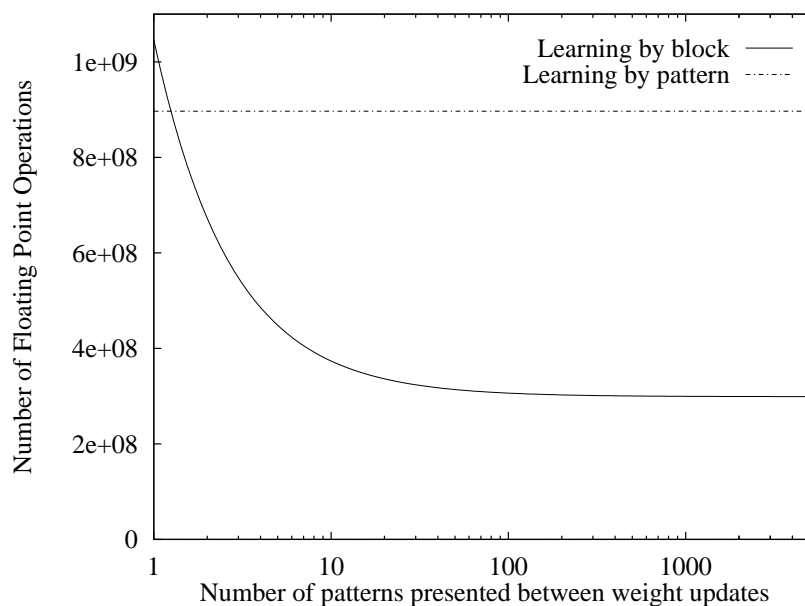


Figure 2.5: The number of floating point operations for weight accumulation and update for block updates, F_{lbb} , and for pattern updates, F_{lbp} .

The update method *learning by epoch* uses less number of operations than *learning by pattern* if the block size is larger or equals two, i.e. $\mu \geq 2$.

Where *learning by block/epoch* is used, CUPS is given by the number of weight change values computed per second. That is, the number of weights *updated* is excluded, when the CUPS value is calculated. However, the time for weight update is not excluded. Thus, only the number of weights *accumulated* per epoch is used for computing the CUPS value, i.e. like omitting the second expression in Equation 2.31.

To measure how well a feed-forward neural network has learned, a separate test set should be used in addition to the training set. The available vectors should be partitioned into disjoint sets: learning set and test set. It is preferable that several different test sets should be included. The test set should include a representative selection of patterns, e.g. for pattern classification, the test set should contain vectors from all classes. The network is trained by the training set and then tested using the test set. The test set or a separate acceptance set is used as an acceptance set for the network.

2.1.5 Generalization

Although the patterns in the training set produce the correct output, this may not be the case for other patterns, as indicated by the property of generalization. Good generalization cannot be achieved unless some a priori knowledge about the task is built into the system [26]. For feed-forward networks this knowledge may be specified by imposing constraints on both the architecture of the network and its weights. To achieve neural network generalization from the training set examples to the entire problem environment, the amount of available training data must be large enough [49]. Moreover, overtraining must be avoided. The problem of overtraining is illustrated in Figure 2.6. As training progresses the network tries to fit to the learning vectors. After a certain number of iterations, the network learns mainly small details about the training vectors, and the general classification ability decreases. Thus, training should be stopped when the error is minimized for the test set.

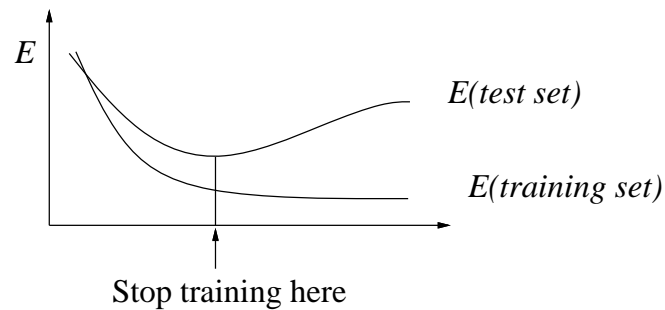


Figure 2.6: Training set error versus test set error as a function of the number of training iteration (from [49]).

2.1.6 Improvements of Feed-Forward Neural Networks

This section briefly describes preprocessing techniques and learning algorithm modifications which have been proposed in the literature to improve the generalization performance of networks.

According to Means [81], preprocessing for neural network classification usually requires as much as 80 % of the total computational cycles. However, preprocessing is parallelizable. New variants of neural networks can also be expressed as standard linear algebra functions and as such are parallelizable.

Preprocessing Techniques

Receptive fields is a technique which restricts the input of each hidden neuron to only a small portion of the input pattern. Thus, each hidden neuron is connected to a subset of the input neurons. This reduces the number of weights and increases the likelihood of correct generalization [26]. The output layer is fully connected. One of the mathematical definitions of 2-D receptive fields – the Gabor functions, is proposed by Daugman [24]. The functions projects an image into set vectors. These are represented by projection coefficients.

Wavelet Transforms (WTs) capture mathematically the functionality of eyes and ears in order to produce multiple resolution inputs to ANN [129]. This preprocessing technique implies feature-preserving data compression. The Gabor function is based on a fixed window, while WT uses varying window size for Fourier Transform (FT). The input to the human eye is based on neighborhoods, not line-by-line information as digital images are stored. Using WT, the frequencies with the best separability can be selected.

Variants of the BP Algorithm

During recent year, several variants of BP learning have been proposed, e.g. quick propagation and conjugate gradient method. A comparison of learning algorithms for feed-forward networks by Nesvik [100] showed that some of the new algorithms were less sensitive to the selection of the learning parameters.

Networks using Radial-Basis Functions (RBFs) are shown [85] to exhibit universal approximation abilities and converge faster than standard MLPs. The network consists of a single layer of locally-tuned neurons. For any given input, only a small fraction of the neurons will respond. Thus, only these units need to be evaluated and trained.

2.1.7 The Effect of the Weight Update Interval

It is shown for one neural application by Paugam–Moisy – Figure 2.7 that less frequent weight updates during training reduces the convergence rate. That is, the decrease in error per training iteration is smaller. The network classified patterns into three classes. To avoid unstable behavior during training, the learning rate had to be reduced when the weight update interval was increased. The reason for this can be explained by network paralysis [149]. Adding weight changes for many training vectors together may result in large weight change values. This may lead to large weight values, if the learning rate is not reduced. Large weights can lead to large output values – Equation 2.7, to be input to the non-linear function. The derivative of the function, which is used for computing the delta error, approaches zero for large values. This results in a very small change in the weights, and the training can come to a virtual standstill.

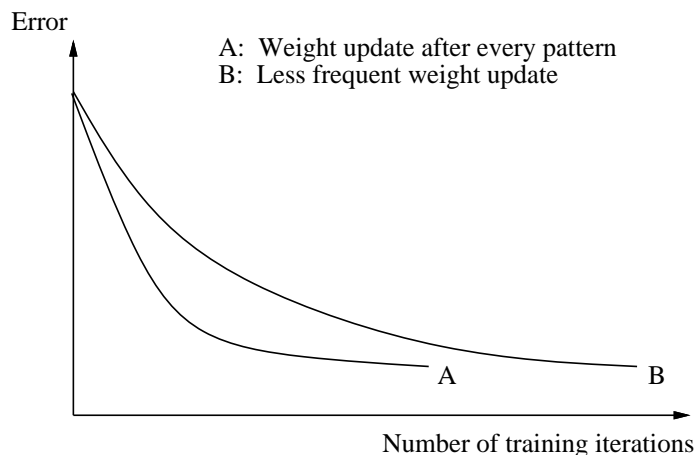


Figure 2.7: Error throughout training for different weight update frequencies [105].

A special case of *learning by pattern* is *delayed weight update* in which the weights are updated for pattern p after the forward pass for the next pattern $p+1$, has been computed. This method may be used for weight updating when the computation of each layer is distributed over different processors. The delta weight change values are small compared to the weights and thus the convergence should be very close to the convergence of ordinary pattern weight updates.

For some neural application experiments, like training recurrent networks for speech recognition [154], *lbp* updating results in a stagnation of the error that does not occur for *lbe* updating. The quick-propagation (quickprop) algorithm, proposed by Fahlman [30], is based on *learning by epoch*. That is, all training vectors are applied before the new weights are computed. Thus, training set parallelism can be used in parallel implementation of quickprop without leading to any reduction in the convergence rate.

The redundancy in large training sets slows down the convergence of *learning by epoch* based algorithms according to Møller [84]. Accumulating redundant gradient weight vectors implies redundant computation. This is not a problem if the weights are updated after each training vector. In fact, redundancy has a positive effect in the beginning of the training. However, simulations show that a conjugate gradient training algorithm – which is based on epoch learning, is more efficient in the end of training even though there is redundancy in the training set. Thus, a *learning by block* approach is proposed, where the block size varies throughout training. Since the redundancy is dependent on the problem, the block size has to be selected by estimation. For each iteration the block size that lead to a confident decrease in the total error of the training set is determined. Simulation using NETtalk (described in Section 2.2) shows that the training starts with a very small block size and ends by updating weights only two or three times per epoch.

2.2 Neural Network Applications

To efficiently utilize parallel processing for speeding up neural network training, we should be aware of which types of networks and training sets are used in today's neural applications. This section provides a survey of neural network applications. Mainly large applications – where parallel processing is of interest, will be described. Feed-forward neural network (FF-NN) with a single hidden layer is assumed, unless otherwise stated.

Two of the main contributors to the neural network research, Widrow and Rumelhart, recently published the paper called “Neural networks: Applications in Industry, Business and Science” [153]. They list a wide variety of commercial applications for neural networks. The increase in commercial products in the last years is partly explained by the availability of an increasingly wider array of dedicated hardware. Many of the products have to be cheap to be of commercial interest.

2.2.1 Speech

Speech Recognition

Several research groups are working on the difficult task of continuous speech recognition. Promising results using neural networks are described in [86]. However, the network and training set need to be large. For recognizing 300 sentences – speaker independently, the system achieved 4-5% error, which is competitive with statistically based systems. For a larger recognition problem, the error for the neural network system became twice that of the best mainstream system. However, the mainstream system is larger than the neural network based system. As the speech networks get larger they tend more towards networks that are not fully connected [5]. This is in the form of fully-connected subnets.

Speech Synthesis

NETtalk is a two layer feed-forward network that transforms text to phonemes⁶, using 203 input units, 60–120 hidden units and 26 output units [118]. The input to the network is series of 7 consecutive letters from one of the training words, see Figure 2.8.

The central letter is the one for which the phonetic output is to be produced. The three letters on each side of central character help to determine the pronunciation. The input word is scrolled through the window of 7 letters so that each letter, one after another, is placed in the center of the window. The network can be used for continuous text or a dictionary of words. For the latter approach, the words are moved through the window individually. Thus, empty letters are added in front of and after the input word as seen in

⁶Elementary speech sounds.

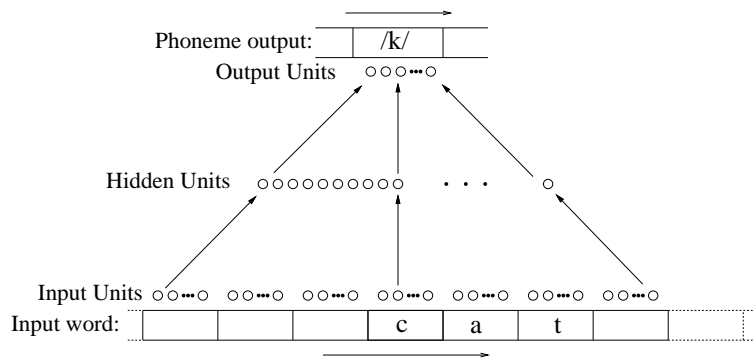


Figure 2.8: Schematic drawing of the NETtalk network.

the Figure 2.8. For a 1000 word training set, 98 % correct transformation was reported. The training set is freely available and is often used as a benchmark.

2.2.2 Image Processing

Satellite Images

Remote sensing is of interest for agricultural, environmental, and weather control applications. For some of these it is necessary to identify human-made structures like roads or residential areas. Wolfer has shown that neural networks can be used to extract roads in raw Landsat TM images [155].

Face Recognition

High order neural networks have been used for human face recognition [33]. The network consists of a preprocessing network followed by a feed-forward network with one or two hidden layers. Four image planes, each of 16 x 16 elements, are input to the network. The originally proposed network had to be reduced to make computer simulations possible on a i486 personal computer. The recognition rate varied according to the transforming – scaling and rotation, of the input image. In the best case, over 96 % of the transformed training patterns were classified correctly.

Medical Imaging

Much research is published within the field of medical imaging [151]. Molecular biology is one area in which neural networks often outpace traditional tried methods. However, in most other areas, neural networks have not yet been shown to outperform statistical

techniques. A feed-forward neural network used for cancer cell classification is described in [52]. A classification accuracy of as high as 96.8 % was obtained. The method is based on using extracted features from the raw image as input to the network.

Analyses of smears for cervical cancer is investigated by McKenna [80]. 80 features were derived from the Fourier spectrum of each 256x256 pixel image and used as input to the network. The classification rate was 92.9 %, higher than the 91.3 % achieved by the Bayesian classifier⁷. One hidden layer resulted in better performance than two hidden layers. The classification rate by using the neural network was for all experiments higher than by using the Bayesian classifier. A second experiment indicated that images of higher resolution improved the classification rate.

Blood vessel detection in angiograms has been shown to obtain a superior 92% classification performance by using BP trained networks, compared to 68% and 83% for Bayesian maximum likelihood classifier (MLE) and iterative ternary classification (ITC), respectively [99]. Including more hidden layers was shown not to result in a performance improvement.

Object Inspection

Object recognition is one of the key problems in factory automation. Kang [59] proposes a system for inspection of objects moving down a conveyor belt based on a feed-forward neural network. 94% correct recognition was obtained for the two objects used in the experiments.

Onda et al. [104] have designed a neural system for identification of metal welding defects. The average identification rate is 75 %, reported to be approximately equivalent to that obtained by expert engineers. Training of the network required about three hours on their neuroboard based system.

Shiotani et al. [120] show that a FF-NN can be used to speed up the recognition of overlapping plant cell objects. The neural network directs the movement of a window to place an object of interest within the window. Afterwards, a template matching system is used to check the located object.

Road and Obstacle Recognition

A vision system for real time recognition of traffic situations is described by Tsinas [145]. Separate FF-NNs are used for recognizing the driving lane and obstacles on the lane, respectively. The time for processing an image of a road scene varied between 0.1 s and 0.65 s on a 486 PC running at 50 MHz. Recognition of 2 minutes of real time video recording reported the one obstacle present and produced one false alarm.

⁷Statistical pattern recognition.

Image Compression

Multi-layer neural networks can be used to transform image data into compressed data by reduction of the spatial redundancy [128]. The number of neurons in the hidden layer is chosen to be smaller than in the input and output layer. Thus, the output of the hidden layer is a compressed image, and the output layer de-compresses the image. In order to increase the generalization properties, the image is divided into blocks. By dividing the original image into 8x8 pixel blocks and using an adaptive compression ratio according to the complexity of each block, 25:1 compression ratio was obtained by Cho et al. [18]. However, they report higher ratios by use of Self-organizing Maps.

Optical Character Recognition

Recognition of characters in printed documents seems to be almost error free, whereas handwritten characters still make recognition prone to errors. In the work of Diep the network recognizes multi-size and multi-font printed characters successfully [27]. Garis achieved handwritten writer-independent digit recognition of 92% [41]. The network consists of a preprocessing network followed by a single hidden layer network. However, combining two neural networks, one which inputs the digit image and one that inputs the contour of the digits, resulted in 97.8% correct recognition [133].

Recognizing Paper Bills and Coins

Automatic recognition of paper bills and coins is an important issue for vending machines and within banking. Error free recognition of Japanese or U.S. paper currency is proposed by Takeda and Omatu [131]. To reduce the size of the network a technique called *random masks* is used. Some parts of the scanned input are covered using random masks. The sum of the pixels not covered is used as input to one input unit. Other masks are used for the other input units. The masks used in the training of the network are also applied when the network is used for recognition. This method implies that the number of weights in the network can be reduced to 1/10 of the original number, without increasing the recognition error.

A coin classification system can be based on several different kinds of sensors – acoustic and weight based, used as inputs to a FF-NN. 100% accuracy was obtained by this kind of system for recognition of British coins [29].

2.2.3 Miscellaneous Applications

Weather Prediction

Weather prediction normally involves computations using approximations of complex mathematical models. The prediction models require detailed information of the areas to be studied. Carpintero et al. [14] used Adaptive Time-Delay neural networks for local, short-term weather forecasting. The input is METOSAT images and local information about temperature, pressure, speed and wind direction, and visibility. Promising predictions of temperature and wind speed were obtained, although no comparison with traditional methods was undertaken.

Rainfall prediction using a neural network with 2 hidden layers is presented in [16]. Infrared-light and visible-light images from geostationary meteorological satellite (GMS) are used as input data. An average classification accuracy of approximately 90 % is reported.

Sonar Return Classification

Gorman et al. [43, 44] used feed-forward neural network to the classification of sonar returns from two underwater targets, a metal cylinder and a similar shaped rock. The network classification performance was shown to be better than that of trained human listeners. The network uses one hidden layer and 60 continuous-valued input units. The best performance was achieved in the case of 24 hidden units. However, this was only slightly better than that achieved with only 12 hidden units. Two output units were used, where (1,0) represented a return signal from a metal cylinder, and (0,1) represented a return signal from a rock. A set of 208 returns (111 cylinder returns and 97 rock returns) were used in the experiments.

2.2.4 Commercial Products

Magnetic Character Reader

Verifone has developed the Onyx check reader, which does Magnetic-Inc Character Recognition (MICR) of the digits on the bottom of checks. The company claims 99.6 % accuracy even with checks that are crumpled or overwritten [48]. The character reader uses a custom chip designed by Synaptics. The chip combines a retina-like optical sensor with a neural network.

Financial Forecasting

By training a neural network to mimic a market, its prediction can be used to guide investments. As described by Hammerstrom [48], many companies are secretly using neural networks for various financial tasks. However, little information is available on their methods and results, since neural networks have demonstrated an outstanding performance. A system mentioned in the article was tried for two months with a \$ 10 million investment. The performance was 2.8 % above bench-marked performance.

Process Control

Neural networks are ideal for process control because they can build predictive models of the process directly from multidimensional data collected from sensors [48]. The networks need history – which is often abundant, and not theory – which is often absent. Thus, neural networks are well-suited for predicting, controlling, and optimizing industrial processes.

2.2.5 Summary of Applications

Application	Data set	Network size (I x H x O)	No of tr. pat.
Blood vessel det. in angiograms [99]		121 x 17 x 2	75
Cancer cell classification [52]		3600 x 20 x 1	467
Coin recognition [29]		259 x 5 x 6	–
Handwritten digit recognition (P) [41]	[102]	32 x 15 x 10	2000
Image compression [18]		64 x 8,6,4 (x 64)	–
Location of plant cells in images [120]		100 x 15 x 10	52;67;69
NETtalk text-to-phonemes [118]	[116]	203 x 60-120 x 26	5438
Object detector [145]		961 x 50 x 1	50
Object inspection [59]		4096 x 64 x 2	–
Optical character recognition [27]		3000 x 20 x 94	94
Optical character recognition [27]		2500 x 100 x 94	1,128
Papanicoloau smear cell classification [80]		80 x 6 x 1	702
Paper currency recognition [131]		128 x 64 x 12	–
Road classifications in satellite images [155]		56 x 20 x 1	552
Speech recognition [86]		234 x 1000 x 61	1,300,000
Speech recognition [86]		351 x 4000 x 61	6,000,000
Welding defect identification [104]		15 x 10 x 9	1024

Table 2.1: Neural network applications using feed-forward neural networks. A reference to the data set is given if it is freely available. (P) indicates that a preprocessing network is used in front of the FF-NN inputs.

Data set	No of elements per pat.	No of patterns
26 capital letters, 20 different fonts [123]	16 units	20,000
Isolated handwritten digits (49 writers) [102]	32 x 32 pixels	3,471 characters
NETtalk corpus [116]	29 per character	20,008 English words
Sonar return data [117]	60 units	208

Table 2.2: Freely available data training sets.

Table 2.1 lists a summary of back propagation applications. As far as possible, variations in the network structure are indicated. Some freely available training sets are given in Table 2.2.

2.2.6 Do Neural Network Applications Need Parallel Hardware?

From the survey in the previous section, at least two common characteristics of many applications are noticeable. They indicate that the ANNs are quite different from the complex organization of the human brain. First, the networks are small enough to be trained and applied on a computer with a single processor. Only in a few instances has parallel processing been used. The survey concentrated on large networks, thus many other applications using small networks have been omitted. Second, the output neuron layer often consists of a small number of neurons. The latter indicates that allocation of many processors for computing the output layer is unnecessary.

Several researchers have reported the need for selection of good feature vectors to make the network smaller and executable on their serial computers. They demonstrate that the same level of performance can be obtained as when using larger networks. Reports about improved performance by using larger neural network are few. Mainly all are based on a single hidden weight layer. Most of the published applications are from academic work. Since commercial companies rarely publish in scientific conferences, there probably exist many commercial applications which are not described here.

It is obvious that some of the neural network applications require parallel processing, i.e. for speech recognition this is required both for training of the network and when the network is applied for recognition of real time speech. However, quite many applications can be successfully implemented without parallel hardware. This may also be necessary for a product to be marketable, e.g. control applications in consumer products. However, it is still possible for parallel processing to be used to reduce the *training time* of an application.

Chapter 3

Parallelization of Feed-Forward Neural Networks

The degrees of parallelism in BP training for feed-forward neural networks were introduced in Section 2.1.2. In the first part of this chapter, each backpropagation parallel degree is described in greater detail and an overview of parallel processing is given. The second part of the chapter contains a survey of published work on mapping of parallel BP training onto parallel architectures. This includes both general purpose systems and special purpose architectures like neurocomputers.

3.1 Distributed Computing for Each Degree of BP Parallelism

This section first describes training set partitioning. Then the other main parallel degree, network partitioning, is detailed by describing pipelining and node parallelism.

3.1.1 Training Set Parallelism

Training set parallelism is also called data parallelism, since the training set is partitioned, not the training program. An example is given in Figure 3.1 for training of the English alphabet.

Each PE 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 PEs, thus weights are updated in a global operation (*learning by block/epoch*). The weight change values of each PE are summed and used to update the local weight matrices.

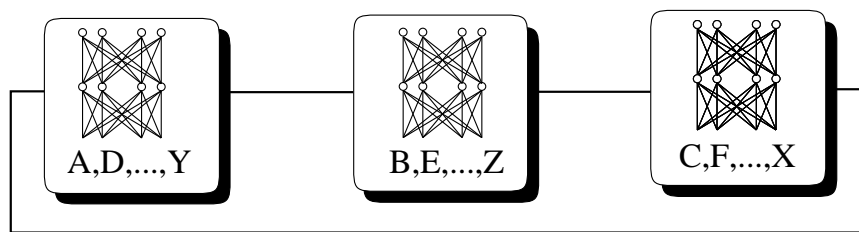


Figure 3.1: Training set parallelism for learning the English alphabet.

3.1.2 Pipelining

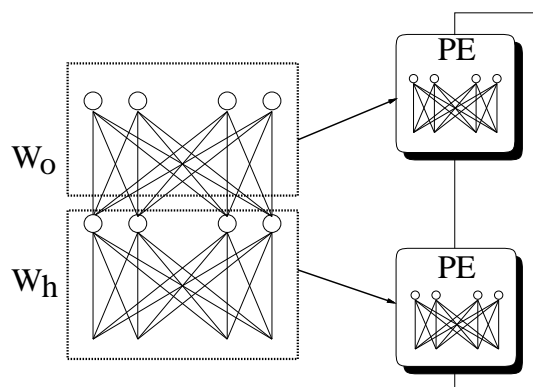


Figure 3.2: Mapping of the weight matrices for pipelining.

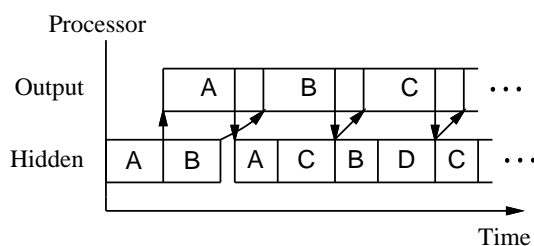


Figure 3.3: Pipelining of the training patterns.

In pipelining, the different weight layers are computed in different PEs as illustrated in Figure 3.2. Figure 3.3 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 processes the next training pattern (B). Then, it reads the hidden error for A and both processors accumulate the weight change values for A. This method interleaves the forward phase with the backward phase. A further extension is to execute the two training phases in parallel [111].

3.1.3 Node Parallelism

As described in Section 2.1.2, node parallelism contains two sub-degrees of network parallelism: neuron parallelism and synapse parallelism. The parallel mapping schemes, which use a mixture of these two, are named node parallel methods. Below, each of the two sub-degrees are explained.

Neuron Parallelism

The most common way to parallelize feed-forward network is by using neuron parallelism, also called vertical slicing. Figure 3.4 indicates the principle for an example of three processing elements.

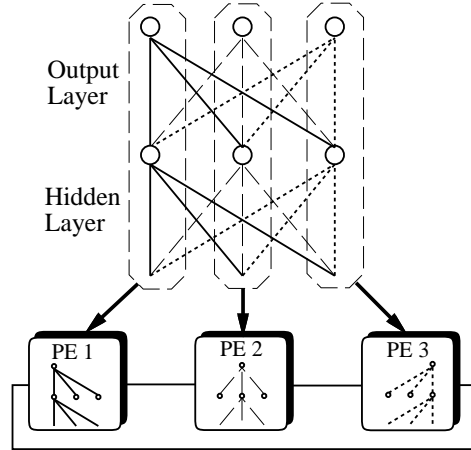


Figure 3.4: Neuron parallelism, also called vertical slicing.

All incoming weights to one hidden and one output neuron are mapped to each PE. That is, each PE stores all the incoming weights to the neuron assigned to that PE. The network slicing corresponds to storing one row of the weight matrix in each PE. The output of the neurons is computed by the matrix-vector product

$$\begin{bmatrix} y_{L,1} \\ y_{L,2} \\ y_{L,3} \end{bmatrix} = \begin{bmatrix} w_{L,11} & w_{L,12} & w_{L,13} \\ w_{L,21} & w_{L,22} & w_{L,23} \\ w_{L,31} & w_{L,32} & w_{L,33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (3.1)$$

where

$$L = \{\text{Layer} \mid \text{Layer} \in \{h, o\}\}$$

First, the value of one hidden neuron is computed in each PE. Then, each PE exchanges the values – e.g. over a ring bus, and continues by computing the value of the output neuron. The hidden layer error is a computed based on the output error. This can, according to Equation 2.16, be formulated as a vector-matrix product

$$\begin{bmatrix} \delta'_{h,1} & \delta'_{h,2} & \delta'_{h,3} \end{bmatrix} = \begin{bmatrix} \delta_{o,1} \\ \delta_{o,2} \\ \delta_{o,3} \end{bmatrix} \begin{bmatrix} w_{o,11} & w_{o,12} & w_{o,13} \\ w_{o,21} & w_{o,22} & w_{o,23} \\ w_{o,31} & w_{o,32} & w_{o,33} \end{bmatrix} \quad (3.2)$$

Since the weights are stored in row order, the sum-of-products is computed by adding partial products. However, more effective summation could be achieved if the weights were

stored in column order. Thus, some implementations store the weights twice, both by row order – for forward pass, and column order – for backward pass. Either duplicated weight updates [157, 61] or communication of the weights [89] is by this method required. The former has been shown to be the most efficient [89].

If the number of neurons in a layer is larger than the number of PEs, each PE compute more than one neuron from each layer.

Synapse Parallelism

Instead of mapping the rows of the weight matrices to each PE, the columns may be mapped. In synapse parallelism, each PE computes a partial sum of the neuron output as indicated in Figure 3.5.

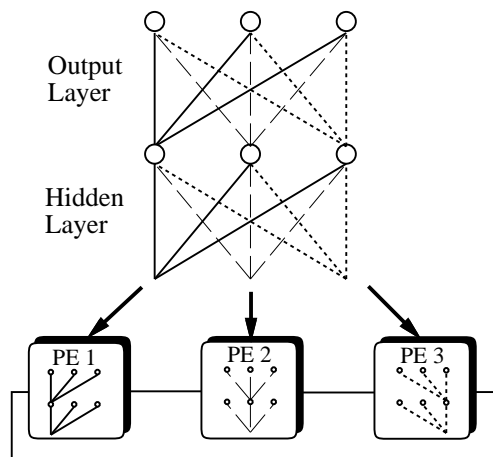


Figure 3.5: Synapse parallelism.

The computation is more fine grained than for neuron parallelism. The sub-results from each PE have to be added and broadcasted to all PEs before the next layer can be computed. The advantage is that the hidden layer error can be computed without communication

$$\begin{bmatrix} \delta_{h,1} & \delta_{h,2} & \delta_{h,3} \end{bmatrix} = \begin{bmatrix} \delta_{o,1} \\ \delta_{o,2} \\ \delta_{o,3} \end{bmatrix} \begin{bmatrix} w_{o,11} & w_{o,12} & w_{o,13} \\ w_{o,21} & w_{o,22} & w_{o,23} \\ w_{o,31} & w_{o,32} & w_{o,33} \end{bmatrix} \quad (3.3)$$

Thus, some implementations use neuron parallelism in the first layer and synapse parallelism in the second layer. The two degrees can be combined for both weight layers, as is described in Section 5.1.2.

3.1.4 Dimension of Each Parallel Degree

Each parallel degree has an inherent limitation given by

Training set parallelism: The number of patterns in the training set

Pipelining: The number of weight layers

Neuron parallelism: The number of hidden units and output units

Synapse parallelism: The number of input units and hidden units

Thus, this indicate the maximum number of processing elements that can be assigned to each degree of parallelism.

3.2 Parallel Computers for Simulating ANNs

Parallel processing of neural network simulations has attracted much interest during the past years. The training and use of neural networks can be represented mathematically as linear algebra functions that operate on vectors and matrices [81]. Thus, standard parallelization schemes can be exploited. Parallel architectures for simulating neural networks can be subdivided into general-purpose parallel computers and neurocomputers. Neurocomputers are designed as boards and systems for high speed ANN simulations [6]. Neurocomputers can be classified as general-purpose or special-purpose [144]. A general-purpose neurocomputer is programmable and is capable of supporting a large range of neural network models, whereas, a special-purpose neurocomputer implements one neural model in dedicated hardware. The latter benefits from higher *speed* than the former. Most neurocomputers are based on processing elements computing in parallel. A survey by Solheim [125] lists about eighty different digital neural hardware projects. However, only twenty of these proposed architectures have been implemented. These systems are designed and built by either research institutes or commercial companies.

3.2.1 General Aspects of Parallel Processing

A parallel computer usually consists of a number of processing elements (PEs). Each processing element¹ consists of a processor and memory. The memory can either be on the processing chip or on separate chip(s). Recently, neural circuits have been produced containing several PEs on a single chip. Since the processing elements may have to exchange data with their neighbors, a communication module may be required for each PE.

¹The name processing element was originally used for simple elements in SIMD computers, but are used today also about the more complex elements in MIMD computers.

A number of topologies exists for interconnecting PEs [147]. The most common ones are shown in Figure 3.6: Broadcast bus, ring, array, 2-D mesh, 2-D toroidal mesh (2D-torus), 3-D mesh and hypercube. 1-D systems have a much lower optimal processor count than 2-D and 3-D systems [6, 55]. This means that much finer grained parallel processing can be realized by using a multidimensional topology.

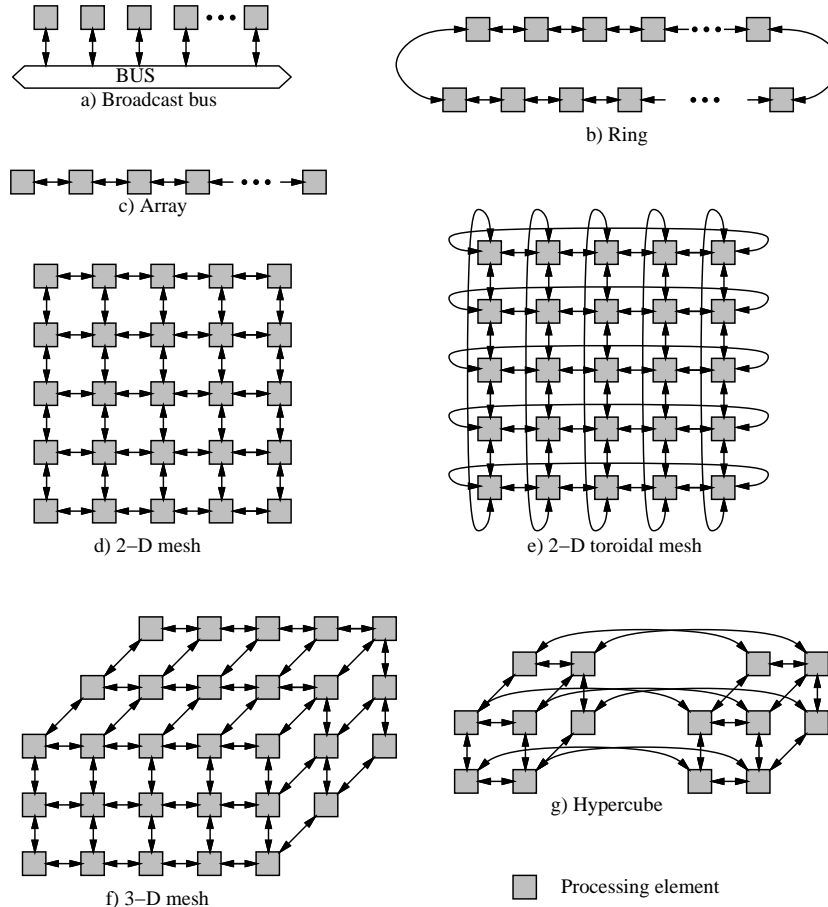


Figure 3.6: Processor topologies for simulating ANNs (from [147]).

Designers of parallel programs should be aware of Amdahl's law, whose essence is that the improvement of overall system performance due to the speeding up of one part of the system is limited by the fraction of the job that is not speeded up [5].

Parallel computers can be classified according to Flynn's classification, based on the number of simultaneous instruction and data streams[54]:

SISD Single Instruction stream Single Data stream – sequential computer with a single CPU.

SIMD Single Instruction stream Multiple Data streams – a single program controls multiple execution units.

MISD Multiple Instruction streams Single Data stream – systolic arrays with pipelined execution.

MIMD Multiple Instruction streams Multiple Data streams – computers with more than one processor and the ability to execute more than one program simultaneously. Computers in this category are also called multiprocessors or multicomputers depending on shared or distributed memory, respectively.

A special execution mode of MIMD has been defined:

SPMD Single Programs operating on Multiple Data streams – means that the same program is down-loaded onto all the processing elements. The processors are usually performing the same operations but on different parts of the data [87].

Several different methods are possible for inter-processor communication. The two major switching methods for communication [54] are:

Circuit switching, A physical path is established between the source and the destination before message is sent. SIMD machines frequently use circuit switching.

Packet switching, A message is split into fixed or flexible sized packets. Each packet is routed through the interconnection network independent of other packets. As such, packets may take different routes through the network. MIMD machines are usually based on packet switching.

Two major concerns in parallel *implementations* are:

Load balancing, To minimize idle time it is necessary to keep the processors active. Each processor should be given an equivalent computation load.

Communication, To maximize the time processors perform computation, communication should be minimized. Moreover, the communication should be distributed as evenly as possible over all the communication links [8].

As the number of processors increases, these factors become more dominating. One of the main purposes of the work presented in this thesis is to show how fixed mappings of neural networks to large parallel systems can be weakened by these problems. Further, solutions to minimize these problems are proposed.

The computation grain size is an important factor in load balancing [25]. Grain size determines the basic program segment chosen for parallel processing [53]. *Fine* grained

parallelism means that the computation is spread over a number of small tasks, whereas, for *coarse* grained parallelism, the tasks are substantially larger. Some computers are suited for coarse grained computation (i.e. message passing MIMD computers), while others are designed for fine grained computation (massively parallel SIMD computers).

Complexity modeling is a way to theoretically specify the upper bound on running time of a program. It is specified by the “big-oh” notation. For an input size n , $\mathcal{O}(n^2)$ means that there are positive constants c and n_0 , such that for n equal to or greater than n_0 , the running time for a program is $T(n) \leq cn^2$.

3.3 A Survey of Different Parallel Implementations

Many different mappings of BP onto parallel computers have been proposed and implemented – both for general purpose and specially purpose computers. In this section, a description of the published work surveyed by the author of this thesis is presented. As far as possible, both the architecture and the parallel implementation are described.

Mapping of neural network models onto parallel architectures may be categorized into two general groups: heuristic mapping and algorithmic mapping [72]. Most of the proposed mappings are of the heuristic kind. They are generated by trial and error based on knowledge about the algorithm and the target machine. The algorithmic technique relies on a systematic approach to the implementation. The work by Fujimoto et al. ([39, 40]), and Kumar et al. [65] belong to the algorithm mapping category. Moreover, some theoretical studies of parallel algorithms have been made, which are described in Section 3.3.3. The majority of other surveyed mappings assign PEs according to the number of elements in a single degree of parallelism e.g. the number of neurons. The same mapping is used for all kinds of neural applications. However, only a few networks and training sets will run optimally on a fixed mapping. What ought to be considered is what degree of parallelism should be included and how many PEs should be assigned for each of them to minimize the total training time.

Overview of implementations on the computers applied in this work – AP1000 and RENNS, will be given in the next chapter.

3.3.1 General Purpose Computer Implementations

In this section, BP neural network simulations on various general purpose machines are described. A summary of the degrees of parallelism used and performance results are given in Section 3.3.10 and 3.3.11, respectively.

MasPar MP-1

MP-1 is a massively parallel SIMD computer from MasPar Computer Corporation. A variant of the mesh topology connects PEs to their 8 nearest neighbors – horizontally, vertically, and diagonally. In addition to nearest neighbor communication, there is a global router which allows any PE to send data to any other PE. However, the communication bandwidth is much larger for the nearest neighbor communication than for the global router. A fully configured system consists of a 128x128 array of PEs. Several node parallelism based implementations have been made for the machine. Chin et al. have made a node parallel implementation where the weights are ordered so that one (or more if available) column of PEs correspond to a single neuron [17]. For a 256 x 128 x 256 neural network, based on the results from smaller systems, 10 MCUPS is estimated for a 16K PE array [45].

Node parallelism and training set parallelism are combined in [22] for a 64x64 system. Each row of the array of PEs is assigned a copy of the network. Within each row neuron parallelism is used for computing the activation values of the hidden neurons. Then, synapse parallelism is used for computing the output activation values. A master processor adds the partial sums together and computes the output error. The error values for the hidden layer can then be computed by neuron parallelism. A network of 128 input neurons, 64 hidden neurons, and 16 output neurons achieved a maximum performance of approximately 12 MCUPS for 1536 training patterns. The weights were updated by *learning by epoch*.

A neural network simulator, Stuttgart Neural Network Simulator (SNNS), has been developed at the University of Stuttgart [160]. It is a portable software tool to generate, test and visualize ANNs. Many different learning algorithms have been implemented. A parallel SNNS kernel for multilayer feed-forward networks runs on MasPar MP-1216, which is the 16K PEs version of the MasPar computer. Two mappings are proposed based on a combination of training set and neuron parallelism. The first scheme assigns one PE to each vertical slice of the feed-forward network, which includes one neuron from each layer. The number of necessary PEs is equal to the number of neurons in the largest layer, except the input layer. Further, as many multiple copies as possible of the neural network are made to utilize all the PEs. The NETtalk network with 120 hidden units was trained in 41 MCUPS (down-loading time for training patterns from the host was not accounted for).

One of the biggest problems identified was the overhead for down-loading the training patterns from the host to the parallel system. To avoid a single PE storing the remaining input values for an input layer greater than the other layers, the second implementation assigned the number of PEs equal to the largest number of neurons in *any* layer. A performance measure (including down-loading time for the training patterns) for NETtalk was 17.6 MCUPS. Fewer networks can be trained in parallel and more PEs become idle for networks with the largest number of neurons in the input layer.

A third prototype that combined training set parallelism and node parallelism gained slightly better peak performance than the second implementation. However, in this method,

a variation in the number of neurons in each layer leads to dummy weight values in some of the PEs. The results from all three schemes demonstrate a performance dependence on the number of neurons in each layer. An unequal number of neurons in each of the layers reduces the performance.

IBM GF11

IBM GF11 is an experimental SIMD machine with 566 processors interconnected by a Benes Network². BP has been implemented on this machine using training set parallelism [154]. The weight changes are summed by a binary tree processor configuration. To reduce the memory latency time, a *subset* of the weights are stored in SRAM. In this case, several patterns are processed on these weights before a new subset of weights is loaded into SRAM as opposed to finishing one training vector before a new one is input. NETtalk with 60 hidden units and a training set of 12022 patterns (epoch update) learned at 900 MCPS – no measure for CUPS given, on 356 processors. A ring summing algorithm achieved the highest performance for 128 processors of 222 MCPS, confirming the limitation of processors connected by a single ring.

Hypercube Machines

Simulation of BP trained neural networks on NCube/4⁺ is reported by Kerckhoffs [61]. NCube/4⁺ is a 4-dimensional hypercube machine with 16 processing elements. Larger systems of 128 and 1024 PEs are available. A PE can communicate with other PEs by use of asynchronous DMA. The implementation distributes the neurons in each layer between the PEs of the hypercube. A comparison was made between single weight storage and double weight storage for the output layer. For the double weight storage, the forward pass uses neuron parallelism – Equation 3.1, while the backward pass uses synapse parallelism – Equation 3.3. The same double weight storage scheme is employed for MUSIC – see Section 3.3.6. The drawback of this method is that the weights are stored and updated twice. Thus, the results show roughly equivalent communication time, but less computation time for the single weight storage scheme. The peak performance is 0.19 MCUPS. For synapse parallelism the broadcast time has been shown to increase linearly for a NCube size from 1 to 128 [132]. It is concluded in the paper that the effect of reducing the training time by this parallel implementation is minimal.

The hypercube computer Intel iPSC/860 is capable of being configured with up to 128 Intel i860 processors. A node parallel implementation of BP on a 32 PE version of the iPSC/860 is given in [58]. The network is partitioned vertically, i.e. all input neurons and at least one hidden unit is mapped to each PE. Each PE computes the output value of

²A multistage switching network that can perform all possible connections between inputs and outputs. Conflicts in the use of switches or communication links are solved by rearranging connections [53].

its assigned hidden neurons (neuron parallelism). Then, each PE computes a part of the output value based on the assigned hidden units (synapse parallelism). Thus, the only communication required in the forward pass is to sum the partial sums of the output units. For the NETtalk application with 80 hidden units, 11 MCUPS was reported. The weights were updated for every 2000 training vectors.

Connection Machines

Singer gives a review of five different ANN implementations on the Connection Machine (CM-1 and CM-2³) [122]. The most sophisticated of those is developed by Zhang et al. [161] for the CM-2. This is a massively parallel SIMD computer with between 16K and 64K processors, each consisting of a one bit processing unit and 8KB or 32KB local memory. Moreover, every 32 processors share one floating point unit. The parallel BP algorithm combines neuron parallelism and training set parallelism. It will be fully described in the next chapter. An implementation published by Rosenberg and Blelloch assigns one processor for each neuron in the network and two processors for each weight [111]. This is an inefficient processor assignment and they suggest an improved algorithm using one processor per weight and no separate processor for each neuron. Further, to avoid only processors with weights from one layer to be active at each time, pipelined computation – as described in Section 3.1.2, of the training patterns is suggested.

BP for CM-5 is presented in [75]. The CM-5 consists of 544 PEs, each a 33 MHz SPARC-2 chip with its own 32 Mbyte memory. The mapping is based on node parallelism, where the input and output neurons are evenly distributed over all processors. To use the *Control Network* for adding operation and avoid message passing communication, *all* the hidden neurons are mapped to each processor. Each processor computes a part of each hidden neuron sum in parallel – synapse parallelism. The partial results are summed and the sum is sent to *all* PEs. Then, each processor performs the sigmoid function. This is said to be the only redundant computation in the scheme. For the output layer, the PEs compute the outputs in parallel. No communication is needed since all hidden layer outputs are readily available in every processor. The implementation was tested on protein tertiary structure prediction with a network of 257 input neurons, 256 hidden neurons and 131,072 output neurons. For 512 processors a performance of 76 MCUPS was obtained. For a CM-5 with 32 PEs, the NETtalk network (80 hidden units) learned at 18.33 MCUPS using a batch implementation [1].

Kumar et al. [65] propose a hybrid scheme using node and training set parallelism for hypercube architectures. The node parallel scheme is similar to the one described in Section 5.1.2 and called *checkerboarding*. This method makes all-to-all broadcast unnecessary, since only communication within rows or columns is needed. The processors are parti-

³CM-2 is the same architecture as CM-1 with some added features like floating point accelerators and larger local memory.[146]

tioned into clusters. A cluster forms a hypercube and contains one copy of the network. Each cluster contains different training patterns. The method makes it possible to vary the number of processors assigned to node parallelism against those assigned to training set parallelism. Theoretical expressions for the total training time were derived based on the cost of computation and communication. The computation cost was based on time per weight per pattern, while communication cost was based on startup time and per-word time for communication. The effect of changing weight update interval is not considered. The expressions are used to compare the model to simpler schemes. A non-optimized version of the hybrid method performed over 50 MCUPS for a 1024 x 256 x 64 neural network on a 256 processor CM-5.

Warp

The Warp machine consists of a linear array of 10 PEs called cells. Each cell can communicate with its left and right neighbors in the array. In a much cited paper, Pomerleau et al. propose several BP mapping algorithms onto the Warp computer [107]. In the first implementation, columns of weights were stored in each cell – synapse parallelism, whereas in the second implementation, training set parallelism was used. As only a 32 Kword⁴ memory is available in each cell – weights are in the latter implementation, pumped through the array from a central cluster memory. Nine cells compute the forward and backward pass, while the tenth is reserved for updating the weights. Each weight is used for all the training patterns in a cell, before the next weight is received. For the NETtalk network, the performance was measured to be 17 MCPS (CUPS measurement is not reported), which was the fastest performance reported at that time. The synapse parallel implementation was much slower.

Supercomputers

Supercomputers are an alternative to parallel computers in providing high speed computation. Supercomputers consist of one or a few complex processing elements. They usually consist of multiple functional units, vector registers, and a highly pipelined processor. BP training has been implemented on the Fujitsu VP-2400/10 vectorial supercomputer [113]. The computer was programmed using an extended FORTRAN77 compiler. A high degree of vectorization is reported. The NETtalk network trained at 60 MCUPS.

Symult S2010

Symult S2010 is a computer with 16 processors. It is a mesh connected message-passing parallel computer with dedicated hardware for wormhole routing. An interesting BP map-

⁴Each word is 4 byte.

ping is described in [3], which combines neuron parallelism and pipelining. The processors are connected by a ring bus as shown in Figure 3.7.

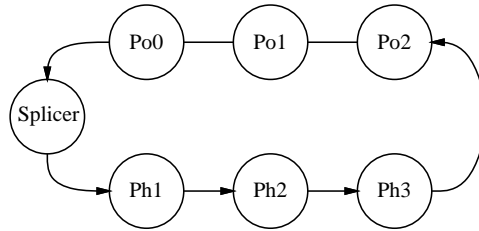


Figure 3.7: Combined neuron parallelism and pipelining in Symult S2010 (from [3]).

In this example the three lower processors are designated for the hidden layer, while the processors in the upper row are assigned to the output layer. The splicer processor composes messages of training vectors, one vector per message, which are sent to the first element of the pipeline – Ph1. Each message progresses through the pipeline and at each processor, activation values are computed and added to the received activation values which are sent on to the next processor on the ring. After a message has passed all Ph processors, each Po processor in the ring computes its output activation values based on the received hidden activation values. Moreover, the output processors compute the output error values which are also passed through the pipeline. The splicer includes the error values in the training vector messages to be used for computing weight change in each hidden processor. The weights are updated by *learning by epoch*. An expression for estimating the best number of processors for each layer is derived based on the number of neurons in each layer and the ratio between computing one hidden layer connection versus one output connection. The method has a reduced amount of communication. A disadvantage of this method is that the time for filling the pipeline is proportional to the number of processors. Moreover, to avoid using old weights after a weight update, the pipeline needs to be refilled after a weight update.

3.3.2 Interconnected Workstations

For mapping BP training onto workstations background workload has to be considered. A detailed study of both static and dynamic mapping algorithms is reported in [19]. In the dynamic case the neurons are re-mapped as the workload changes.

3.3.3 Research by Use of Models of Parallel Machines

In this section, schemes that have not been implemented on a real system are described.

Data-Driven Systems

Data-driven computation is based on asynchronous parallel execution of computations, which may be represented by directed graphs. A processor instruction is executed based on the availability of its data operands. Q-x is a floating point data-driven processor. A message-passing multiprocessor is able to be configured of up to 1024 processors. The processors are interconnected by a 2D-torus communication network. Processing and storage in the processor is carried out as packets flowing through elastic pipelines. A description of the mapping of BP onto the architecture is reported in [2]. The scheme is based on combining neuron parallelism and training set parallelism, similar to [161] – see Section 5.2.1. That is, one row of PEs compute one copy of the network. The neurons are distributed among the processors in a row by vertical slicing of the network. Performance is evaluated by using a system simulator for training an image compression 256 x 128 x 256 network. For 64 processors 44 MCUPS was predicted.

Mesh Topology

A theoretical study of implementing pipelining of training patterns on a mesh topology is given in [130]. The method is based on assigning a dedicated mesh to each layer, see Figure 3.8. The hidden layer is computed by $p \times q$ PEs and $q \times r$ PEs compute the output

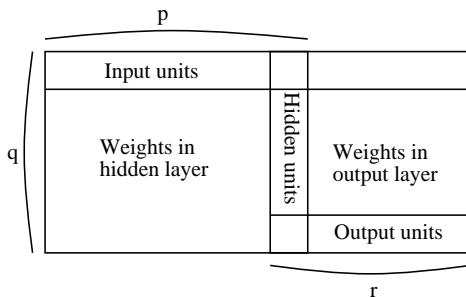


Figure 3.8: Piped-mesh implementation (from [130]).

layer as indicated in the figure. The data for the hidden units are transmitted from the left to the right mesh. Theoretical expressions for transmission count and computation count are derived and used to estimate the optimal number of neurons per PE. For a 1024 x 128 x 128 network, 6 neurons per PE (u) is found to be optimal and therefore

$$u = \left\lceil \frac{N_i}{p} \right\rceil = \left\lceil \frac{N_h}{q} \right\rceil = \left\lceil \frac{N_o}{r} \right\rceil \quad (3.4)$$

where N_i , N_h and N_o are the number of neurons in each layer as defined in Section 2.1.

A framework for implementing neural networks on massively parallel machines is proposed by Azema-Barac [7]. The model formalizes a speedup for the different degrees of neural network parallelism on the given parallel machine. Derived expressions for one training iteration are made based on the grain of parallelism⁵ and the communication overhead. It is not specified how to estimate or measure the communication overhead. Moreover, the accuracy of the method is not substantiated. Further, the effect of weight update interval on the convergence is not considered. The distribution scheme is designed for node parallelism, training set parallelism, and a combination of these two.

The target machine for implementing the scheme has been DAP 600 (Distributed Array Processor). This is a SIMD machine with 64x64 PEs, each interconnected to its four nearest neighbors. One of the few results given is based on a digit recognition application [9]. A speedup of 50 over a Sparc station is reported.

2D-Lattice Topology

A proposed reconfigurable machine, called Dynamically Reconfigurable Extended Array Multiprocessor (DREAM) is presented in [119] for implementation of neural networks. It is classified as a SIMD machine and the PEs are arranged on a 2D lattice, i.e. each PE is connected to eight of its closest neighbors through programmable switches. Multiple rings of various lengths can be mapped onto the machine. An *embedded* ring structure is proposed for the case of uneven neuron layers in a multilayer network. For a network of N_l and N_{l-1} neurons in layer l and $l-1$, respectively, the ring size is set to $\max\{N_l, N_{l-1}\}$. If $N_{l-1} > N_l$ the shorter ring of length N_l is embedded into the ring of length N_{l-1} as shown in Figure 3.9. The shorter ring of size N_2 is embedded into the ring of size N_1 . The weight matrices are partitioned vertically – synapse parallelism, and the output of a neuron is computed by accumulating partial sums from each PE. First, N_{l-1} cycles are used for computing the layer $l-1$ outputs. Secondly, a shortened version of the ring, equal to the N_l , is used for the next layer. Thus, the ring size is dynamically changed as the computation moves from layer to layer. If the number of neurons is larger than the number of PEs, time-multiplexing is used for creating virtual PEs.

The results presented in the paper are from analytical estimation and not real implementations.

Hypercube Topology

A theoretical study of the mapping of a tree structure onto a hypercube topology is proposed in [76]. The trees are used for communication of elements. One tree is used for broadcasting a value from a root to all the leaves and another tree is used to sum values from the leaves into a root. $\mathcal{O}(\log N)$ time is required, where N is the size of the

⁵The number of neurons assigned to each processor.

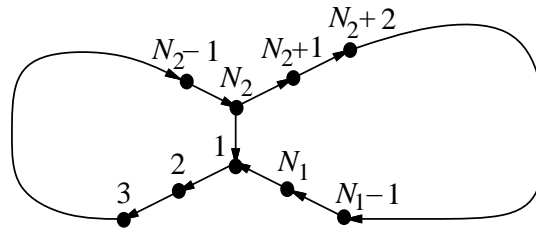


Figure 3.9: A ring of size N_2 embedded into a longer ring of size N_1 (from [119]).

largest layer. The scheme is based on node parallelism, similar to the one described in Section 5.1.2. Pipelining of the patterns is also suggested included.

Systolic Array Design

A programmable systolic array is proposed in [67] for a wide variety of artificial neural networks. VLSI is the target architecture for the mapping. A data *dependency graph* (DG) is used to express the recurrence and parallelism associated with the neural network algorithm. The graph is mapped onto a proposed ring systolic array – one ring for each weight layer. The weights for one output neuron, i.e. one row of the weight matrix, are mapped to one systolic element, see Figure 3.10. Thus, both the learning and recall phase share the same storage and processing hardware. Input elements are rotated across each ring array.

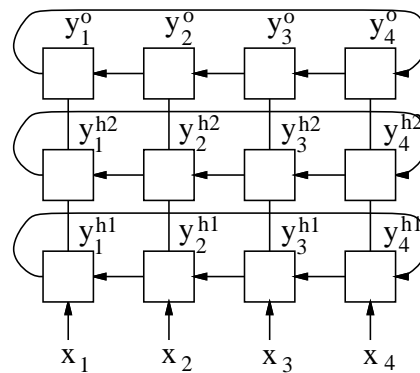


Figure 3.10: Cascaded systolic ANN (from [67]).

The proposed systolic design is efficient when the sizes of the different layers are approximately equal. To make the design applicable for multi layer networks of different sizes, the number of systolic elements assigned to each layer may be varied. Each weight layer is

mapped to a number of systolic elements equal to the number of rows or columns, depending of which is smallest in number. A two layer neural network with configuration (3-9-4) is used in the paper, and the design is shown in Figure 3.11. The hidden weight matrix is vertically partitioned, and one column of weights is stored in each systolic element of the lower PE row. The output weight matrix is partitioned horizontally and stored in the upper PE row. This approach minimizes the number of PEs needed.

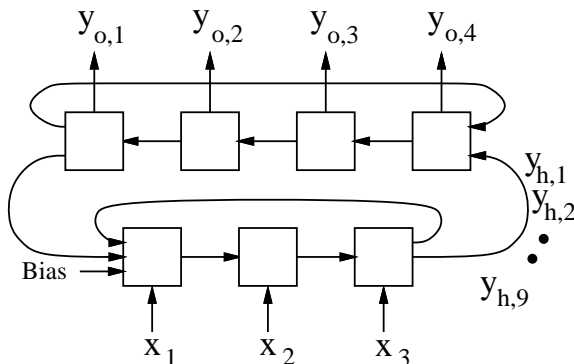


Figure 3.11: Systolic design for feed-forward networks with layers of different sizes (from [67]).

First, the forward pass starts by residing the inputs $\{x_i, i = 1, 2, 3\}$ at the corresponding PEs in the first layer. The bias values are pipelined one-by-one into the array from the leftmost PE. Each bias input initiates an accumulation operation where the product $w_{h,j}x_i$ is added at the i -th PE. The complete hidden neuron element $y_{h,j}$ is output at the rightmost PE after 3 accumulation operations. The 9 hidden neuron elements are sequentially pumped into the upper array of PEs and used for computing the output layer elements $y_{o,k}$. After the nine elements $y_{h,j}$ have passed the four upper PEs, the output $y_{o,k}$ is ready at the k -th PE. In a similar way, the backward phase is computed.

A more general mathematical formulation, covering mapping of a wide range of ANNs to ring systolic arrays, is presented by the same authors in [68, 69]. However, a single ring is used and one layer is computed at a time. The design assumes that all the neuron layers are of uniform size. Further, a systolic mapping where each PE stores a row of the hidden weight matrix and a column of the output weight matrix is presented in [70, 71]. The PEs are connected by a ring bus. Each input element is sequentially input to the upper PEs. The method is extended by using *learning by epoch*, which allows pipelining of the training patterns in rectangular arrays of PEs.

SIMD Mesh-connected Machine Implementations

A fine grain node parallel mapping is proposed in [72, 108]. Although claims are made that the method is applicable to various SIMD machines, the details are given for the Systolic/Cellular Array Processor (SCAP). However, no implementation results are reported. The scheme is based on a graph-theoretic approach, and there is no requirement for fully interconnected layers. If the the total number of neurons and weights in the network is less than or equal to the number of available processors, then each neuron and each weight is mapped to a separate processor. The input elements are then sent to the processors storing the weights to be multiplied. A transformation scheme is given for transforming the weight matrix between row and column major order. In the opposite case, where the number of available processors is less than the total number of neurons and weights, the data is stored in a global memory. The size of the local memory is very limited. Thus, the processor array is operating on sub-arrays from the global memory. The essential data movement operations are realized by embedding a *Benes* network.

3.3.4 FPGA Implementations

Field Programmable Gate Arrays (FPGAs) have recently become an alternative to full custom VLSI design. FPGA is reasonably priced and makes the development cycle shorter, in-comparison to VLSI. However, the flexibility leads to less performance and functionality per unit area of silicon. To fully utilize the flexibility of the FPGA technology it should be used with run-time reconfiguration. This is exploited in the RRANN (Run-time Reconfiguration ANN) system [28]. That is, the BP algorithm is divided into sequentially executed stages: At run-time only one stage is configured on the FPGA at a time. Since only one stage is used at any given time, more hardware resources are available for each stage. In the system, 6 neurons are mapped to each FPGA (XC3090). Results show that to achieve a higher performance, than that achieved by a non-run-time reconfigurable system, more than 22 FPGAs need be used. This is due to the re-configuration overhead.

A system for feed-forward recall-phase is presented in [13]. Each hidden and output neuron is implemented in 2 FPGAs (XC3042) connected to an EPROM. The EPROM contains a lookup table for the sigmoid activation function. A network with 5 input units, 4 hidden units and 2 output units executes at 4MCPS. They conclude that the size and speed of the system can be greatly improved by using higher density FPGAs. An experimental project called REMAP consists of building an entire computer using only FPGA and memory [73]. It is based on bit-serial processing elements with SIMD control. The PEs are designed for neural computation. A full-scale prototype (256PEs) can run in 10 MHz.

Since one of the main bottlenecks in implementing ANNs in FPGAs is the multiplication operation, the circuitry devoted to multiplication must be significantly reduced, according to Bade and Hutchings [10]. This can be accomplished in two ways. First, by reducing the

number of multipliers and/or secondly, by reducing the complexity of the multiplier. Bade and Hutchings propose to use bit-serial stochastic computing to reduce the circuitry necessary for multiplication. That is, to represent signal values as long bit-streams according to the statistical distribution of “1”s in the bit-stream. Then, multiplication can be implemented as a bit-wise Boolean (“and”) operation, which is implemented as a lookup-table. Since FPGAs are normally based on lookup-tables, this scheme is very efficient. The system can process 181K patterns per second (recall phase). This is regardless of the network size. The drawback of the method is that learning is difficult due to lack of precision.

3.3.5 Transputer Implementations

A transputers chip contains a microprocessor, memory, and ports for communication. The most commonly used transputer is the INMOS T800.

An analysis of a transputer system shows that for a fully or randomly connected neural network, the topology of the processor network only has a small, constant effect on the iteration time [92]. An expression for the optimal number of processors is derived. Communication overhead is shown to be a major limiting factor in the efficiency of implementation on transputers.

Yoon and Nang [156] have shown that hypercube networks have less communication time complexity than mesh networks for backpropagation training. Thus, as the number of processing elements involved increases, the performance of hypercube mapping outpace the mesh mapping. The model was verified by implementations on a transputer system.

In [20], a neural specification language is defined. The purpose is to parallelize neural networks automatically based on a high level language. The parallelizing scheme only considers neuron parallelism. The approach scales acceptably for a system of 13 transputers, when the network consists of a large number of neurons.

Transputers Interconnected by Mesh or 2D-Torus Network

Fujimoto et al. propose a load balanced mapping of the BP and Hopfield neural networks onto both a Toroidal Lattice Architecture (TLA) [40] and a Planar Lattice Architecture (PLA) [39]. A TLA is a one-way directed 2D-torus network, while a PLA is a bidirectional mesh network. Both schemes are based on mapping a single weight to a virtual processor. Thus, the method uses node parallelism, i.e. both neuron and synapse parallelism. The matrix of virtual processors is partitioned into sub-matrices, each of which is mapped to a real transputer processor. Before the partitioning, the rows and columns in the virtual matrix are permuted so that each sub-matrix contain an equal amount of computation (multiply-accumulate and sigmoid computation).

A feed-forward network of 256 x 64 x 256 neurons is used in the work. The performance

for a 16 transputer system interconnected by a PLA is estimated to be slightly higher than that achieved on a TLA interconnected system. An implementation on TLA learned at 0.6 MCUPS. The conversion rate from MFLOPS to MCUPS is roughly estimated to 1/40 in the worst case. The proposed schemes are proved to exhibit almost linear performance with respect to the number of processors when expanded for large scale neural networks. The processor topologies are planned implemented in wafer scale integration (WSI) technology.

The research reported in [106] considers each layer of the network as a stage of a pipeline. Theoretical expressions for the performance of an implementation of pipelining and node parallelism on a 2D-torus transputer system is given. The weight matrices are divided into $Q \times Q$ blocks, where Q represent the number of PEs in a row or column. The matrix-vector product is obtained by broadcasting the input elements along columns and accumulating sub-results in the rows. This is similar to the method presented in Section 5.1.2. However, in this work, little is said about the mapping of pipelining of the computation in the layers onto the computer.

Results from a 16 transputer system show that for a three neuron layer network, the time usage for the pipelined and the non-pipelined implementation was almost identical. Preliminary experiments show that the pipelining algorithm – using delayed weight update, seems to converge in the same way as the non-pipelined algorithm – updated after each training pattern. It is concluded that pipelining increases the computation load versus communication per processor and enables the computation to be more evenly distributed throughout the network. Thus, pipelining is preferable since the overhead of its operation is negligibly small.

Transputers Connected by a Ring

A thorough study of training set parallelism on a ring of 32 T800 transputers is undertaken by Paugam-Moisy [105]. Each transputer is assigned an equal number of training patterns, and different block sizes are tested for *learning by block*. For a classification application – described in Section 2.1.7, the total training time is computed for different weight update intervals – see Section 7.2. A block size in the range [192, 384] is found to be optimal. A model is also made for estimating the time for one training iteration and estimation of number of iterations needed for convergence is undertaken – see Section 7.2.3.

A transputer ring topology is presented in [162]. The idea is to combine training set parallelism and pipelining. A three layer network is mapped onto 15 PEs as shown in Figure 3.12. Each shaded box represents one network cluster. One PE is assigned for each layer within a cluster. No performance results are given.

The MEIKO Computing Surface is a reconfigurable parallel computer based on the T800 INMOS transputer. A mapping scheme of BP onto this system is given in [23]. The mapping scheme combines node and training set parallelism. The method implies that a subset of the rows of the hidden weight matrix and subset of the columns of the output weight

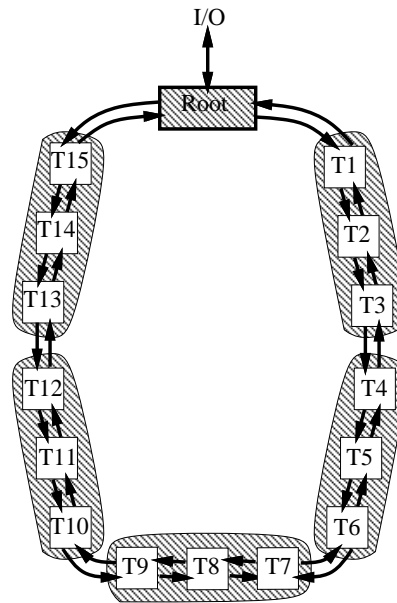


Figure 3.12: A transputer ring topology combining training set parallelism and pipelining.

matrix are stored in each processor. In this way, no communication between processors is needed until the partial results from each PE is to be summed to compute the activation of the output layer. This is done by a *master* processor. The number of processors that can be efficiently used, by using only node parallelism and *learning by pattern*, is 20. For a larger number of processors, the performance is only marginally increased. A performance of 8 MCUPS is reported for a 512 x 256 x 64 network, when training set parallelism is included and the PEs are connected by a 3x40 mesh. A similar mapping is used on the MP-1 [22].

Transputers connected in a pipelined ring topology are used for training set parallelism by Foo in [34, 36]. An administrative transputer distributes the training set evenly amongst the other processors in the pipe. For weight update, each pipe processor receives weight changes and errors from its upstream neighbor, adds its own weight changes and errors to these and passes the new values to the downstream neighbor. Thus, the last downstream processor will hold the overall weight change and the total error. After computing the new weight values it propagates them back through the pipe. Simulations show that for small networks the serial implementation out-performs the parallel implementations. An analytic expression for the training time per iteration is derived. However, the scheme does not consider the effect of the weight update frequency on the convergence. The weight change summing seems to be leaving processors idle, even though an improvement is proposed. In that case, the downstream transputers are assigned a larger number of training patterns [35]. Using this latter scheme, 1.54 MCUPS is reported for NETtalk using 53 T805-20

transputers.

Transputer Tree

Several implementations on 9 transputers are reported in [26]. Both fully connected network and the extension of shared weights–receptive fields (SW–RF) network are used. The latter network is beneficial for shift-invariant feature extraction and improves the generalization properties – see Section 2.1.5 and 2.1.6.

The best performance for the SW–RF network (almost linear) is seen when the transputers are connected in a tree and use training set parallelism. However, the problem of slow convergence due to *learning by epoch* is mentioned. Another implemented scheme combines node parallelism and pipelining. The parallel processing speedup approaches a value of six for the largest input image size. For a fully connected network the speedup is higher, which is explained by an increased utilization of the allocated processors. However, the SW–RF neural network showed better generalization properties.

Transputer Hypercubes

Training set parallelism for hypercubes made up of transputers is studied in [62]. An $\mathcal{O}(\log n)$ step algorithm for summing the weight change matrices is proposed. To be able to use all the communication channels in the cube concurrently, the weight matrix is split into $\log n$ parts, where n is the number of processors. Each part is summed along one axis at a time. The redundant summation in every two PEs is eliminated by doing one half of the summation in each PE and exchanging the results. For a small training set, the described optimizing methods improves the speedup. However, for a training set of 512 samples or more, the difference in speedup is marginal.

The study is restricted to hypercubes of dimension 1, 2, and 3. Moreover, the weight update interval is not considered. The application used was speech recognition.

A training set parallel scheme is reported in [150] for training a Time Delay Neural Network (TDNN) for speech recognition. The convergence in the number of iterations varied for different weight update frequency. However, with no favor for frequent weight updates. 16 transputers showed a speedup factor of 8.8 over a VAX 3600.

Summary of Transputer Techniques

Much published work has been undertaken in this area. Communication seems to be a bottleneck in large systems. Some of the implementations depend on using a master processor, which limits the scalability. In many of the projects, training set parallelism is

employed, see Table 3.3. However, with a few exceptions [105, 150], experiments on the effect of the weight update frequency on the total convergence time has not been reported.

3.3.6 Digital Signal Processor (DSP) Based Systems

MUSIC (MUlti Signal processor system with Intelligent Communication) is a DSP based parallel system with distributed memory. A system of 45 PEs with a peak performance of 2.7 GFLOPS [89] is implemented. Each PE consists of a DSP (Motorola 96002), up to 8Mbyte of Video RAM (VRAM), up to 1 Mbyte of SRAM and an FPGA communication controller. Each board consists of 3 PEs controlled by a transputer, which also interfaces the PEs to the host computer. Inter-PE communication is by a pipelined ring. A dynamic load balancing scheme is implemented, where the operating system distributes data according to each PE's computation in the previous step. An implementation of BP based on standard neuron parallelism is used – where one network layer is computed after the other. However, to be able to compute the hidden delta error without using partial sums, each PE stores a local copy of the weights needed to compute the error. For a network of two weight layers, this only applies to the output weight layer. This double weight storage was also used in the NCube/4+ mapping described earlier.

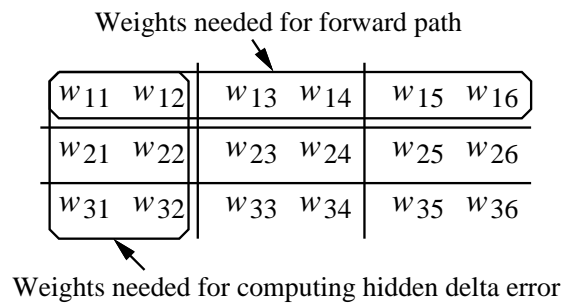


Figure 3.13: The output weight matrix, where the weights required by processing element 1 are circled.

Figure 3.13 shows the sub-matrices of the output weights stored in PE 1 for a network of 3 output neurons. The horizontal slice is used to compute the forward pass, while the vertical slice is used to compute the delta error for the hidden layer. To avoid communicating the weights, each processor updates both weight subsets individually. Thus, each weight will be updated twice. Muller et al. claim that this implementation is faster than computing partial sums of the hidden layer delta error. A performance of 203 MCUPS is reported. In [125], Solheim has compared several neuron based mapping schemes and found that storing the weights twice is not beneficial, except for small networks.

RAP (Ring Array Processor) is a neurocomputer designed for continuous speech recognition at the International Computer Science Institute, Berkeley, California [87]. The machine is

a DSP based system with a low-latency ring interconnecting the PEs. Each PE consists of a TMS 320C30 DSP, 4–16 Mbyte DRAM and 256 Kbyte of local memory. FPGAs were used for the interprocessor ring and the memory controllers. Four interconnected PEs are contained on each board. To speed up the computation, a library of matrix-oriented assembly language routines were written. A working system of 40 PEs was used in the experiments. For a network with 128 neurons in each layer, the learning speed for vertically sliced feed-forward network was 57.3 MCUPS. Recall speed was 211.1 MCPS.

Based on the experiences of designing RAP, the CNS-1 (Connectionist Network Supercomputer-1) was designed [5]. It is proposed to supply orders of magnitude more computing capability than RAP. Custom designed VLSI digital processing nodes for neural network calculations are connected in a mesh topology and operate independently in a MIMD style. This is different from RAP, which uses standard, commercially available components. The initial system will be built with 128 processing nodes and a total of 4 GB distributed storage. The processor consists of a scalar CPU, a vector coprocessor running at 125 MHz, and hardware for communication, which are all comprised in a single chip. A PE consists of the processor chip and DRAM memory, minimizing the board area required per processor. The goal is that the minimum configuration should perform recall at 100 GCPS and learning should in the worst case be 1/5 of the recall speed.

Sandy is a ring-based neurocomputer developed by Fujitsu [60, 158]. The system is based on TMS320C30, and the PEs are connected by a ring bus. The mapping of BP training is by neuron parallelism, i.e. vertical slicing. A prototype has been built, running NETtalk learning at 42 MCUPS.

RENNS (REconfigurable Neural Network Server) is a DSP based parallel computer designed at the Norwegian Institute of Technology (NTH). A description of which is given in Section 4.2.

3.3.7 Commercially Available Digital Neurocomputers

During recent years several neurocomputers have approached the market. They are all designed in digital technology⁶ and most of them are general purpose machines. To obtain high performance they consist of custom designed VLSI circuits and some of them with reduced floating point precision. The neurocomputers presented in this section all provide both learning and recall.

One of the first commercially available VLSI neural implementations which offered on-chip learning was the CNAPS-1064 (Connected Network of Adaptive Processors) chip from Adaptive Solutions, Inc [46, 47]. Low cost is emphasized for inclusion in commercial applications. Each PE was made as small as possible to maximize the number of PEs on a single chip. The chip contains a parallel array of 64 PEs – called processing nodes (PNs),

⁶Information about commercial machines using other technologies has not been seen by the author.

configured for SIMD operation, as depicted in Figure 3.14. The CNAPS Server II offers from 64 to 512 PNs.

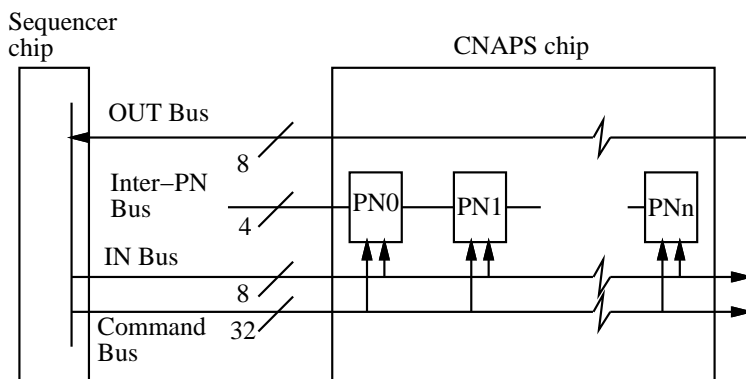


Figure 3.14: The CNAPS architecture consisting of processing nodes (PNs) connected together by three global buses. A sequencer chip controls the flow of data and commands on the buses.

Each processor is a simple digital signal processor with 4Kbyte of local memory. Each chip can compute 960 million multiply-accumulate operations per second. The weights can be 1, 8, or 16 bits. Multiple chips can be connected together and are controlled by a single sequencer chip. The mapping of the feed-forward network is by neuron parallelism, i.e. one or more neuron(s) is computed in each PN. Neurons from different layers may either be mapped to the same PN or separate PNs. The NETtalk network learned more than 40 times faster on a CNAPS chip than on a Sun SparcStation.

A similar chip is designed by Hitachi and included into their MY-NEUPOWER neurocomputer [115]. One chip consists of 8 digital micro-programmable PEs, each of which includes local memory for on-chip learning. The machine has been shown to learn at a maximum speed of 1.26 GCUPS. 32 neural chips are placed on each neural board. Two neural boards are connected together by a bus which interconnects all the processing elements. This makes a total of 512 processing elements. The weights are either 8 or 16 bits. The digital processing elements operate in SIMD mode, as depicted in Figure 3.15. Performance measurements show that the computer can learn 100 to 1,000 times faster than a workstation.

SYNAPSE-1 (SYnthesis of Neural Algorithms on a Parallel Systolic Engine) is a general purpose neurocomputer developed by Siemens AG [109, 110]. The design of the computer is shown in Figure 3.16.

It consists of four separate boards. The intensive parts of the neural computation are undertaken in the MA16 array, while non-intensive computation is performed by the Data Unit. The design differ from many others by storing the weights on a separate board. The MA16 array is a 2-dimensional systolic array of ASIC neural signal processors (MA16s),

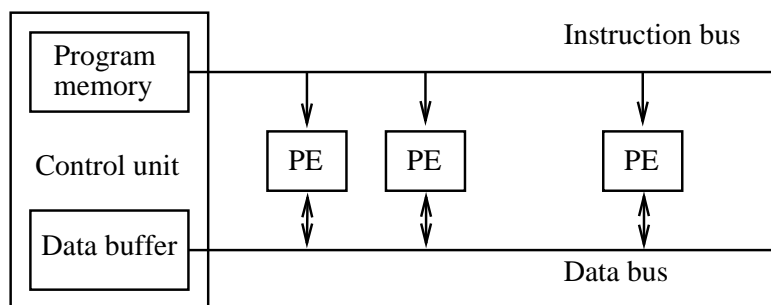


Figure 3.15: The MY-NEUPOWER computer architecture.

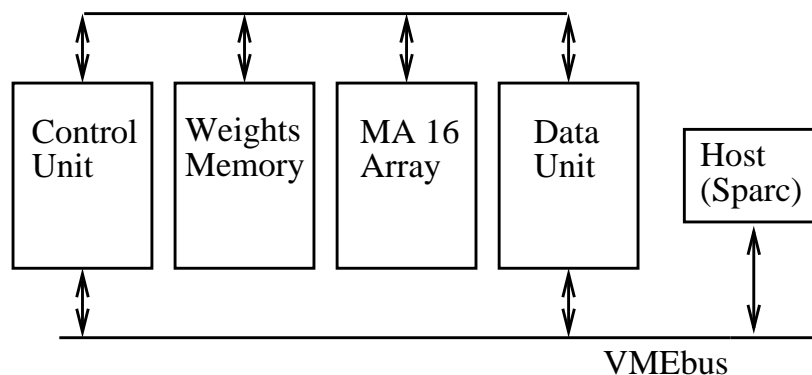


Figure 3.16: The SYNAPSE-1 neurocomputer.

arranged in two rows of four columns. Each MA16 itself consists of a linear systolic array of 4 processing modules. A local memory is connected to each MA16 for storing intermediate results. The weight resolution default is 16 bits. A programming language, called nAPL (neural Algorithms Programming Language), is defined for the computer. The SYNAPSE-1 has proved to be 8000 times faster than a SUN SparcStation 2 for computation of one layer of neurons. The peak performances of the CNAPS server and the SYNAPSE-1 are stated to be similar. However, SYNAPSE-1 needs less additional hardware for running large applications.

The SNAP (SIMD Numerical Array Processor) system is designed by HNC for speeding up neural network applications [81, 82]. It is based on a general purpose chip, HNC100, designed with a SIMD systolic linear array of 4 processors, each of which implements 32-bit floating-point arithmetic operations. Figure 3.17 shows the layout of the SNAP system. Each PE contains a local memory of 512 Kbyte. All the PEs are controlled by a single instruction from the external controller. Each PE is connected to a triple-ported global memory for communication to the host processor and/or peripherals. The global data bus allows data in the global memory to be broadcasted to all PEs at the same time. Alternatively, data in a single PE can be broadcasted. The SNAP system comprises of 1

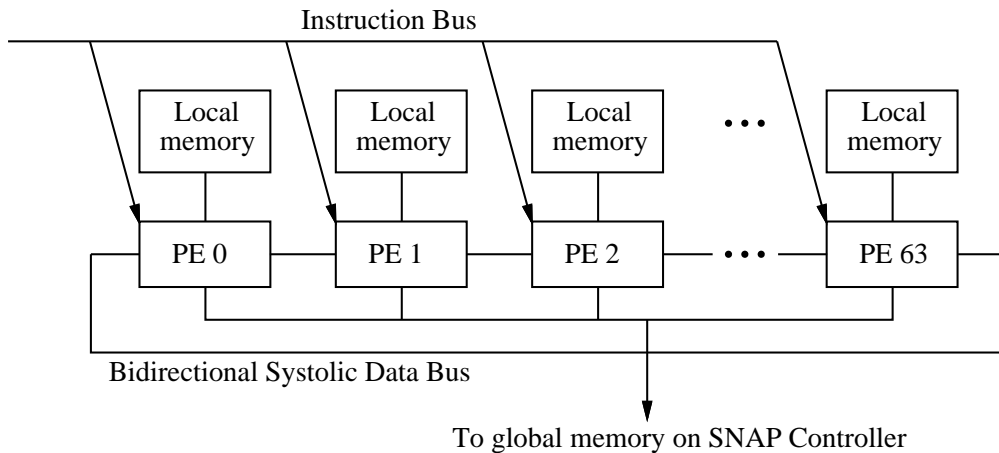


Figure 3.17: The SNAP-64 architecture.

to 4 boards, each with 4 HNC chips, making a total of 16 to 64 SNAP processing elements in the linear ring array. The largest system of 64 PEs provides 2.56 GFLOPS. A feed-forward network with 512 neurons in each layer learned at 302 MCUPS on the system of 64 PEs, while 64 neurons in each layer learned at 76.6 MCUPS [124].

General Remarks

Except for SYNAPSE-1, all the systems operate in SIMD mode. The PEs are made as simple as possible to include as many as possible on a single chip. The BP parallelization scheme is based on neuron parallelism. It is interesting to note that synapse parallelism has not been of interest so far.

An important issue when designing a neurocomputer is the I/O interface. Research applications on neurocomputers have in some cases performed a thousand times slower than peak performance due to I/O-limits [5].

3.3.8 Other Technologies

The survey of BP implementations have been for digital target architectures. However, work is also continuing into the use of other technologies like analog and optoelectronics. These are based respectively on analog voltage levels and light, for signal representation. Since these technologies are not yet commercialized and are outside the scope of this thesis, a survey is not included in this work. Description of the many proposals for VLSI implementations (both digital and analog) has also been omitted here. Many of these projects are only “paper-projects” that have not yet been implemented in silicon.

A survey of Nordström and Svensson on parallel implementation of neural network includes descriptions of some special-purpose neural hardware systems [101].

A survey of hardware implementations of neural networks in Japan is done by Hirai [51]. Several characteristics of neural chips are listed, which appear in Table 3.1.

	Advantages	Disadvantages
Analog	Compact High speed	Susceptible to process-parameter variation Susceptible to noise Lacks scalability The precision of synaptic weights is limited Difficult to make modifiable synapses
Opto-electronics	Large fan-in and fan-out Modifiable synapses possible	Opto-electronic and electro-optic transformation Susceptible to noise and parameter variation
Digital	High precision High scalability Modifiable synapses	Large circuit size

Table 3.1: Advantages and disadvantages of different chip technologies.

The digital approach is currently the best solution to implement neural networks with high precision, even though the circuit size for neural networks becomes larger than for the other approaches. The scalability of digital chips, allows many chips to be easily connected together. To attain scalability, many analog chips employ a 1 bit digital output. Hirai concludes that for special purpose use, which does not require high precision, analog internal circuits with digital output is the best solution. For general purpose use and where high precision is required, the digital approach is the best. Another opinion [91] is that often it is desirable to process analog sensor/actuator data without conversion to digital form. Moreover, a number of tasks, for reasons of speed or integration, require hardware (preferably analog) implementation [42].

One of the benefits of custom design is that the designer can limit the precision to that required by the neural algorithm. However it is limited how much the precision can be reduced. Franzi reports that for fixed point representation, the neural learning only converged in a few cases [37].

3.3.9 General versus Special Purpose Hardware

Garth [42] gives the following reasons for why so few commercial systems implementing neural networks in hardware have emerged:

- Hardware solutions are frequently poorly equipped to accept modification to algorithms as they arise, i.e. the trade-off between efficiency and flexibility.
- Difficulties in providing a complete solution. External pre- and post processing may obviate many of the advantages of the neural chip.
- Many chips achieve much of their speedup by severely restricting the precision in the number of bits. These restrictions may affect the performance of the network.

There seems to be an inverse proportional relation between performance speed and reconfigurability, as indicated in Figure 3.18. By making special-purpose chips it is impossible to gain both high speed and reconfigurability. Neural chips often lack flexibility and scalability, compared to general purpose computer simulation of neural networks.

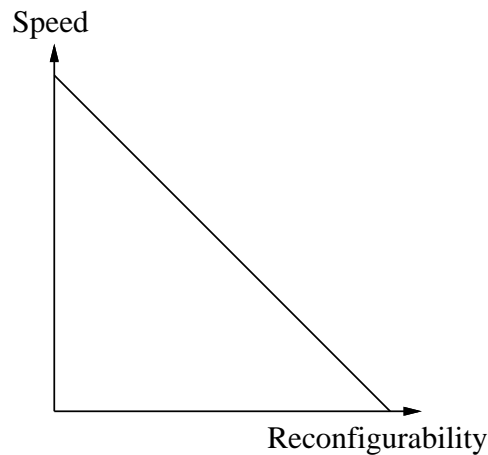


Figure 3.18: The relation between speed and reconfigurability of the hardware.

The design choice between analog and digital technology may depend on whether the research goal is biological modeling or making a programmable research tool [5]. Moreover, a theoretical high speed is useless if the machine is not flexible to do what the user requires.

3.3.10 Summary of the Parallel Mapping Schemes

The degrees of parallelism used parallel schemes on general purpose architectures are listed in Table 3.2. Neuron parallelism is most often used, while pipelining is rarely used. For

Computer	# PEs	Tr. set p.	Pipel.	Neuron p.	Synapse p.	Mapping
CM-2 [161]	64K	x		x		H
CM-2 [111]	16K			x	x	H
CM-5 [75]	512			x	x	H
CM-5 [1]	32	x		x	x	H
CM-5 [65]	256	x		x	x	A
DAP 600 [9]	4096	x		x	x	-
DREAM [119] (M)					x	A
GF-11 [154]	356					H
Hypercube (M) [76]			x	x	x	A
Intel iPSC/860 [58]	32			x	x	H
Mesh (M) [130]			x	x	x	A
MP-1 [22]	4096	x		x	x	H
MP-1 [17]				x	x	H
MP-1 [160]	16K	x		x		H
NCube/4 ⁺ [61]	16			x		H
Q-x system (M) [2]	64	x		x		H
SCAP (M) [72, 108]				x	x	A
Systolic array (M) [67]			x	x	x	A
Warp [107]	10	x				H

Table 3.2: Parallel general purpose computer mappings of BP training. A mapping is implemented on the computer unless “M” is indicated in the leftmost column. An “M” indicates that a theoretical model of the system is used.

Computer	# PEs	Tr. set p.	Pipel.	Neuron p.	Synapse p.	Mapping
TLA/PLA [40]	16			x	x	A
2D-torus [106]	16		x	x	x	H
Ring (M) [162]	15	x	x			A
Ring [23]	80	x		x	x	A
Ring (M) [34, 36]	-	x				H
Ring [105]	32	x				H
Tree [26]	9	x				H
Hypercube [62]	8	x				H
Hypercube [150]	16	x				H

Table 3.3: Transputer implementations of BP. In the leftmost column, “M” indicates that a theoretical model of the system is used.

transputer technologies, Table 3.3 indicates the parallel mapping schemes used. Training set parallelism is often used. This is probably to minimize the communication bottleneck of transputer systems. For neurocomputers, only implementations employing neuron parallelism are yet published, see Table 3.4. However, few implementations for these computer

Computer	# PEs	Tr. set p.	Pipel.	Neuron p.	Synapse p.	Mapping
CNAPS [47, 46]	512			x		-
MUSIC [89]	45			x		H
MY-NEUPOWER [115]	512			x		-
RAP [87]	40			x		H
Sandy [158]	256			x		H
SNAP [158]	256			x		H

Table 3.4: BP neurocomputer implementations.

architectures have been reported.

The majority of parallel *implementations* in the undertaken survey are of the heuristic kind, see Table 3.2. The assignment of processors to each degree of parallelism does not change according to the network size and training set size. This represents a problem since only a few neural network applications will train near optimally for fixed implementations.

The proposed algorithmic mappings are, with a few exceptions ([23, 40, 65]), only theoretical studies. Moreover, some sophisticated methods are suggested but only implemented on small systems, e.g. 16 transputers [40].

3.3.11 Summary of the Performance

The training performance of the parallel back propagation implementations presented in Section 3.3 are listed in Table 3.5. Performance is measured in CUPS.

The NETtalk network has been used in many of the reports. However, the commercial neurocomputer companies seem to avoid using this benchmark. Since different networks are used to measure the learning speed, direct comparison between them is not possible. Also, the comments in Section 2.1.4 regarding the CUPS measure should be borne in mind when comparing the results. The performance is highly dependent on the degree of optimization of the program code [4]. The double loops can be organized so that the data variables can be kept in fast memory. To optimize speed, the code must be written in assembly language.

Computer	No of PEs	Network size	μ	FP	MCUPS
Sun-3 [58]	1	NETtalk	1	32	0.034
NCube/4 ⁺ [61]	16	Optimal network	1	32	0.19
Sun SparcStation 10 [90]	1	-	1	32	1.1
Alpha Station [90]	1	-	1	32	3.2
CM-2 [12]	16K	NETtalk (60)	1	-	2.8
Cray 2 [75]	4	257 x 256 x 131,072	1	32	10 ¹
iPSC/860 [58]	32	NETtalk (80)	2000	32	11
MP-1 [22]	4096	128 x 64 x 16	1536	32	12
IBM RISC/6000 550 [4]	1	1000 x 1000 x 1	1000	32	17.6
Cray X-MP [75]	4	257 x 256 x 131,072	1	32	18 ¹
CM-5 [1]	32	NETtalk (80)	lbb	32	18.33
CM-2 [161]	65536	NETtalk (80)	4096	32	40
Cray Y-MP [75]	2	257 x 256 x 131,072	1	32	40 ¹
MP-1216 [160]	16384	NETtalk (120)	lbb	32	41
Q-x system [2]	64	256 x 128 x 256	8192	32	44 ¹
CM-5 [65]	256	1024 x 256 x 64	lbe	32	50
Fujitsu VP-2400/10 [113]	1	NETtalk (60)	1	32	60
CM-5 [75]	512	257 x 256 x 131,072	1	32	76
CM-2 [121]	64k	128 x 128 x 128	65536	32	350
TLA [40]	16	256 x 64 x 256	1	32	0.6
Transputers [35]	53	NETtalk (60)	1500	32	1.54
MEIKO [23]	120	512 x 256 x 64	30	32	8
Sandy [158]	32	NETtalk (60)	1	32	42
Cellular arch. [31]	4175	NETtalk (60)	1	16	51.5 ¹
SNAP [124]	64	64 x 64 x 64	-	32	76.6
RAP [87]	40	640 x 640 x 640	1	32	102
MUSIC [89]	45	-	1	32	203
SNAP [124]	64	512 x 512 x 512	-	32	302
Sandy [158]	256	Optimal network	1	32	567
MY-NEUPOWER [115]	512	Optimal network	-	16	1,260
CNAPS [47, 46]	512	1900 x 500 x 12	1	16	2,379 ¹

¹The performance has been estimated or simulated (not measured)

Table 3.5: Performance of neural network training. μ is the weight update interval, and FP is the floating point precision. The horizontal lines separate general purpose computers, transputer systems, and neurocomputers, respectively.

Chapter 4

Hardware for Running Neural Networks

In this chapter, the two parallel computers used in this study for implementing the back propagation training algorithm are presented. In addition, conditions for the experiments will be discussed. The first computer presented – Fujitsu AP1000, has been used for the major part of the implementations presented in this thesis. The second computer presented – RENNS, has been used in the final part of this thesis.

4.1 Fujitsu AP1000

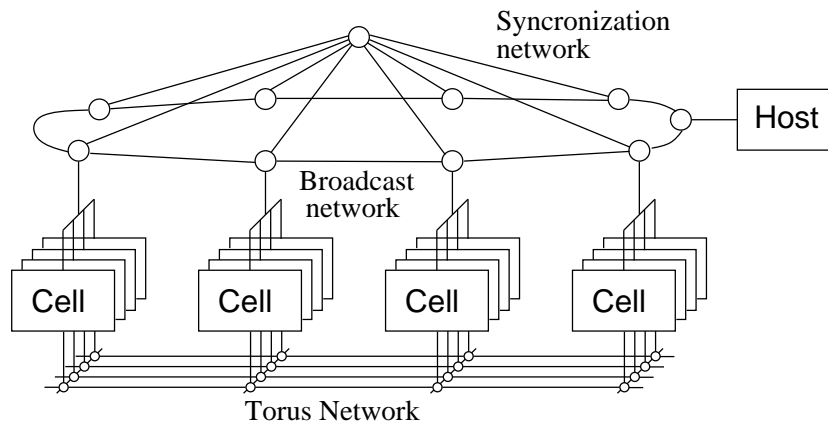


Figure 4.1: The AP1000 architecture.

Fujitsu AP1000 is a general purpose message passing MIMD computer with distributed memory [57]. Figure 4.1 depicts the architecture. The AP1000 processing element is

called a *cell* and consists of a Sparc CPU, an FPU, a message passing unit, 128 Kbyte cache and 16 Mbyte local memory. The largest available system consists of 512 cells¹. The cells are connected by three different communication networks as seen in the figure. First, the cells are interconnected by bidirectional channels in a two dimensional torus network. Communication is by packet switching and packets are routed from source to destination using the wormhole routing switching method. As such, the path length has little effect on communication time. Secondly, there is a broadcast network for bidirectional communication with the host – a Sun 4 workstation running the UNIX operating system. Thirdly, there is a network for fast synchronization. The computer can be programmed using either FORTRAN or C programming languages. The languages are enhanced for doing communication etc.

The AP1000 is accessible over the Internet, without any fee, to registered users in Japan and overseas from the *Fujitsu Parallel Computing Research Facilities* (FPCRf, Kawasaki, Japan). Several versions of the computer are available: 64, 128, 256, and 512 cells. Through an *Open Use* program, each user group can reserve any of the computers for up to 20 hours each month. In this research, I mainly used a 64 cell version located at Kyoto University, Japan and obtained the results for larger systems by running the programs at FPCRf. A similar facility with a 128 cell system, is available at the Imperial College, London.

4.1.1 Programming the System

To use the AP1000 computer, at least two programs are required: a host program running on the host computer and one or more cell program(s). The host program configures the cells by use of a command – a call to a library function, which specifies the number of cells in the x and y dimension. Cell *program(s)* are down-loaded to all cells by a single command. The host can communicate with cells by broadcasting to all or addressing a specific cell.

A library of global reduction functions is available in the programming language to calculate the absolute maximum, absolute minimum, or the sum of data in all cells or in cells in the same column or row of the 2D-torus network. The functions operate on single variables, not arrays. A scatter/gather data function can be used to communicate data between all cells and the host. Each cell receives or sends only the portion of the data specified.

A performance analysis tool is available which shows the activity of each cell during run-time. A trace function is also available to plot the activity of cells in a selected time interval. Support for more detailed timing information is also available.

¹Organized as 32 rows by 16 columns of cells.

4.1.2 Neural Network Implementations

Only one published work is known, except for this, on parallel neural network implementations on AP1000. Kuga et al. have implemented a neuron parallel implementation by vertical slicing of the feed-forward network [64]. Thus, each PE contains a slice of the network. Three different methods for all-to-all communication are compared. First one-to-one communication is used. Secondly, messages are rotated in horizontal and vertical rings. Thirdly, the sending PE is located in the center of the torus and messages are propagated through parallel routes to the other PEs. The methods are compared for the recall phase for the case of an equal number of neurons in each layer. Results are expressed in terms of speed-up for each of the methods. The second method, broadcasting by rotation, obtained the highest speedup. The difference in speedup for the methods increases as the number of PEs increases (256 neurons per layer are used).

4.2 The RENNS Computer System

RENNS (REconfigurable Neural Network Server) has been designed by the Group for Computer Architecture and Design at the Norwegian Institute of Technology. It is a flexible general purpose neurocomputer designed to experiment with adapting the hardware to the problem at hand. One of the differences to general purpose parallel computers is the flexibility of the communication network. An in-depth description of the system is given in [93, 125, 147]. The computer is operative with 15 processing elements – called Processing Modules (PMs). Each consists of a Processor Module and a Communication Subsystem. A block diagram of the RENNS module is shown in Figure 4.2.

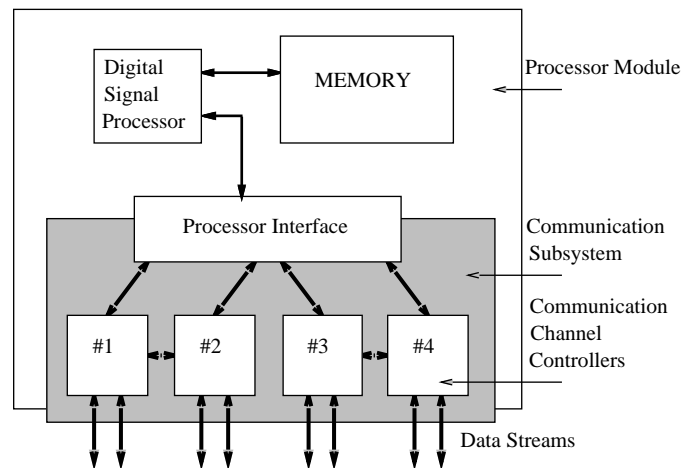


Figure 4.2: Block diagram of the RENNS Module

The Processor Module is built around a digital signal processor, the TMS320C30. It is

a 32 bit processor running at 32 MHz and has a maximum performance of 32 MFLOPS for multiply-add operations. The memory requirements of an ANN-algorithm tend to be very large. To hold connection-matrices and input/output-data, the Processor Modules is equipped with 4 – 16 Mbyte of DRAM, and 256 Kbyte – 1 Mbyte of SRAM. The present system contains 4 Mbyte DRAM and 512 Kbyte SRAM. In addition, the processor has 8 Kbyte of fast, internal SRAM. A VME-interface² gives access to a Sun SparcStation compatible host, connected to the local area network. Each processing module can be accessed over a serial RS-232C connector. This was used in the described experiments to access the RENNS system.

The RENNS Communication Subsystem is implemented in field programmable gate arrays (FPGAs), thus it can be reconfigured in-system to meet different requirements from ANN algorithms. A block diagram of the RENNS Communication Subsystem is given in Figure 4.3. Each subsystem has eight individually programmable channels connected to other subsystems. The channels are all 12 bits wide, normally used for 8 bits of data and 4 bits of control information. The current version has a communication clock of 10MHz, giving a communication bandwidth of 5 Mbyte/s on each channel.

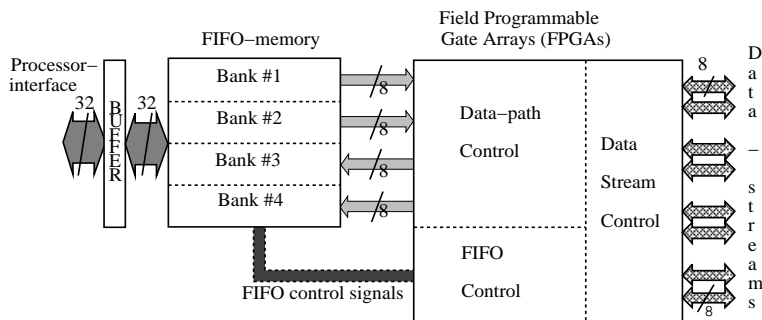


Figure 4.3: A block diagram of the RENNS Communication Subsystem

The processor accesses the Communication Subsystem through memory-mapped FIFO-banks and I/O-mapped setup and control registers implemented in FPGAs. The four FIFO-banks, two for outgoing and two for incoming data, are all 1024 words³ deep.

RENNS has so far been used for only neural network simulations. However, the machine is of a general nature that should make it suitable to many other real world computing problems.

²This is not yet operative.

³All data types for the TMS320C30 are 32 bit wide.

4.2.1 Programming the System

The system is programmed using the C programming language. Program development is undertaken on a UNIX workstation. No debugging tools are available for the system. Therefore, the output from one or a few modules (connected by the serial interface) and 7-segment displays on each module are used to debug programs.

Writing efficient programs is easier for RENNS than for AP1000, as more documentation about the system is available. The effect is most significant with respect to the information available about the communication protocol.

4.2.2 Neural Network Implementations

Several training algorithms for different neural networks models have been implemented on the RENNS system. Several different parallel implementations of Self-Organizing Feature Maps (SOM) have been constructed by Myklebust [93, 94, 95, 96, 97, 98]. Solheim has implemented several backpropagation learning programs based on neuron parallelism [125, 126, 127]. Utne has implemented variants of the Hopfield network [147].

4.3 Applications that Need Parallel Hardware

In Section 2.2.6 it was reported that few researchers studying neural network applications were using parallel processing. It was also noted in Section 3.3 that researchers making parallel programs and hardware for neural computing have mainly been running one or a few benchmark programs, e.g. NETtalk. In addition, for some instances only the maximum obtainable performance is reported. This is usually based on a large non-realistic sized network.

To make parallel computers more accessible to application developers there ought to be a standardized neural network language. Several languages have been proposed, e.g. PYGMALION [143], without leading to frequent employment.

It would seem that parallel processing will be employed in the future for applications that need fast real-time performance like speech recognition and image processing applications. A market for special purpose hardware for sensor and control applications will also exist. These assumptions are supported by the fact that the number of *hardware* related papers submitted to neural network conferences is said to be dropping. However, the remaining papers are principally about sensor and control applications.

Much *research* is continuing at the application level and since some parallel computers are available for free (like Fujitsu AP1000) this ought to be an interesting alternative to be able to experiment with a larger number of parameters in the same frame of time. Furthermore,

it allows investigation into the performance rates obtainable for larger or more complex neural networks.

4.3.1 Neural Network Applications Used in this Work

In this work realistic figures for network size are used by analysis of real world applications. The applications used in the experiments in the following chapters are listed in Table 4.1.

Application	Network size	# Training patterns
NETtalk [118]	203x30–240x26	5438
Character recognition (a X11 font set)	40x30x26	26
Image compression [18]	64 x 4–16 x 64	4096
Speech Recognition [86]	256x64–1024x64	4096
Image Recognition/OCR	1024x512x64	4096
Sonar Target Classification [43, 44]	60 x 24 x 2	208

Table 4.1: BP applications used for measuring training performance.

NETtalk, which has been used in performance testing of both general purpose computers [1, 12, 58, 107, 113, 160, 161] and neurocomputers [31, 158], is regarded as a benchmark for comparing computers running the BP algorithm. However, some differences in the number of hidden units and weight updating frequency exist. In this thesis, NETtalk is used as the main benchmark. An increased variation in the number of hidden units has been included in the experiments to obtain more general results.

4.4 Experimental Conditions in this Work

4.4.1 AP1000

The implementations were first debugged and tested for a small character recognition problem. The training performances were then measured using randomly generated training patterns on networks with size equal to real applications. Bias was not explicitly implemented, but assumed to be one of the input units and one of the hidden units set to “1”. Due to the large cell memory, the *whole* training set can be stored in the cells, thus downloading is only needed initially. This means that the down-loading time is very short compared to the total training time and is omitted from the time measurements.

The weight matrices are updated according to Equation 2.27 and 2.28. The smoothing term is beneficial for *learning by block/epoch*, as the best learning rate is increased [118].

As the main purpose of these experiments has not been to show the computer speed, little time has been used on program code optimization.

4.4.2 RENNS

Due to limited communication bandwidth between the host and the modules, it was not possible to download training patterns to each cell. Downloading time has been omitted even though a large training set could slow down the performance. This would occur in the case when it is not possible to store all training patterns in local memory. The bias was not explicitly implemented.

The weight matrices are updated according to Equation 2.18 and 2.19.

Chapter 5

Fixed Mappings of BP Onto 2D-torus MIMD Architectures

This chapter describes four implementations of BP training on AP1000, each using a fixed assignment of each of the degrees of parallelism. That is, the mapping can not be modified according to each neural network application. The algorithms were designed for two weight layer networks. However, they can easily be extended for a larger number of layers. The description of the parallel mappings apply to 2D-torus MIMD architectures in general. However, the pseudo-code listed may need small changes for use in other multiprocessor systems.

First two methods exploiting a single degree of parallelism, training set parallelism and node parallelism, respectively, are explained. Then, two approaches combining several degrees of parallelism are described. The latter approach is proposed in this work [135, 139].

5.1 Single Parallel Degree Solutions

5.1.1 Training Set Parallelism

The easiest way to make a serial program execute in parallel is usually by coarse grain splitting. Most obviously in BP this means splitting the training set across the available processors. Each cell will therefore store a subset of the training patterns – as indicated in Figure 5.1. Let C denote the total number of cells i.e. PEs and P the total number of training patterns. Then P_c , where $c \in \{0, \dots, C - 1\}$, the number of training patterns in cell c , is given by

$$P_c = \left\lfloor \frac{P}{C} \right\rfloor + k \quad \text{where} \quad k = \begin{cases} 0 & \text{if } c \geq (P \bmod C) \\ 1 & \text{if } c < (P \bmod C) \end{cases} \quad (5.1)$$

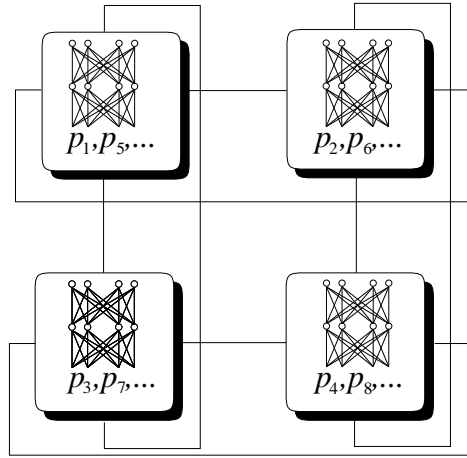


Figure 5.1: Training set parallelism on a computer with 2D-torus interconnections.

If the number of cells is increased, the number of training patterns per cell decreases according to Equation 5.1. Thus, fewer training patterns are computed by each cell between weight updates. This means the computation/communication ratio decreases and limits the speedup. Another problem that may arise if the neural network is large, is that a large local memory is necessary in each PE to store the weight matrices. In the AP1000, cell memory is large enough to store most of the networks for real applications.

Arbitrary and not-fully connected networks will execute with even load balance among the PEs, since the whole network is stored in each PE.

5.1.2 Node Parallelism

A fine grain BP algorithm using node parallelism for mesh-connected multiprocessors is proposed in [159]. It combines neuron and synapse parallelism, i.e. each cell computes a partial sum for a single neuron. The weight matrices are divided into rectangular blocks or sub-matrices that are distributed among the cells, thus the method is often called checkerboard partitioning [66]. The algorithm can be explained by using a simple fully connected network of 4 input, 4 hidden, and 4 output neurons and a system of 4 cells. The weight matrices are partitioned into sub-matrices and multiplied by the input vectors \mathbf{X}_L as follows:

$$\begin{bmatrix} y_{L,1} \\ y_{L,2} \\ y_{L,3} \\ y_{L,4} \end{bmatrix} = \begin{bmatrix} w_{L,11} & w_{L,12} & w_{L,13} & w_{L,14} \\ w_{L,21} & w_{L,22} & w_{L,23} & w_{L,24} \\ w_{L,31} & w_{L,32} & w_{L,33} & w_{L,34} \\ w_{L,41} & w_{L,42} & w_{L,43} & w_{L,44} \end{bmatrix} \begin{bmatrix} x_{L,1} \\ x_{L,2} \\ x_{L,3} \\ x_{L,4} \end{bmatrix} = \begin{bmatrix} \mathbf{W}_{L,11} & \mathbf{W}_{L,12} \\ \mathbf{W}_{L,21} & \mathbf{W}_{L,22} \end{bmatrix} \begin{bmatrix} \mathbf{X}_{L,1} \\ \mathbf{X}_{L,2} \end{bmatrix} \quad (5.2)$$

where

$$L = \{\text{Layer} \mid \text{Layer} \in \{h, o\}\}$$

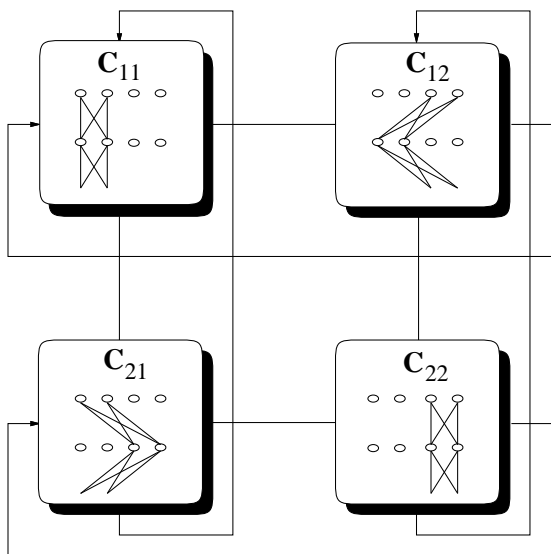


Figure 5.2: Mapping of the weights for the node parallel implementation described in Equation 5.2.

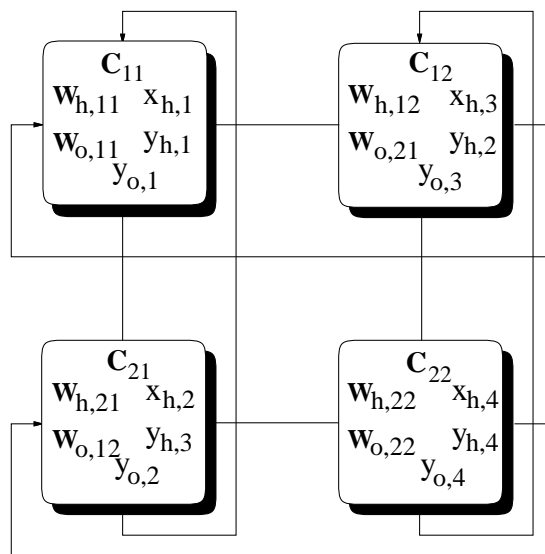


Figure 5.3: Mapping of weight matrices, input- and output vectors of Equation 5.2.

Each cell stores sub-matrices *statically* for the hidden layer ($\mathbf{W}_{h,yx}$) and the output layer ($\mathbf{W}_{o,yx}$) – see Figure 5.2. Each value, $x_{h,i}$, of the input elements are mapped to only one cell¹ as shown in Figure 5.3. If the number of values in the input vectors is larger than the number of cells, each cell will keep more than one value of the vector. During matrix-vector multiplication the input $x_{h,i}$ is transferred *vertically* between the cells – see Figure 5.4. The weights in the sub-matrix are multiplied by the value $x_{h,i}$ and the result for each multiplication is added to the partial sum of the corresponding output, as shown in the right-hand part of Figure 5.4.

Partial results are accumulated *horizontally* in each row of cells, to calculate the output of the hidden layer, $y_{h,j}$ – see Figure 5.5. This output becomes the input $x_{o,j}$ to the output layer, which is multiplied to the output weight to compute $y_{o,k}$. This is calculated in a similar way, but the input vector is now located in different cells. Thus, the sub-matrix $\mathbf{W}_{o,xy}$ is mapped to cell C_{yx} to avoid extra communication – see Figure 5.2. The input elements $x_{o,j}$ are propagated horizontally and the partial results are accumulated vertically. The weights are updated for every pattern and the backward phase is executed in a similar way as in the forward phase by transferring values 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

¹To reduce the redundant communication, the input values can be stored in all the cells in a column during the first iteration to be re-used in the following training iterations.

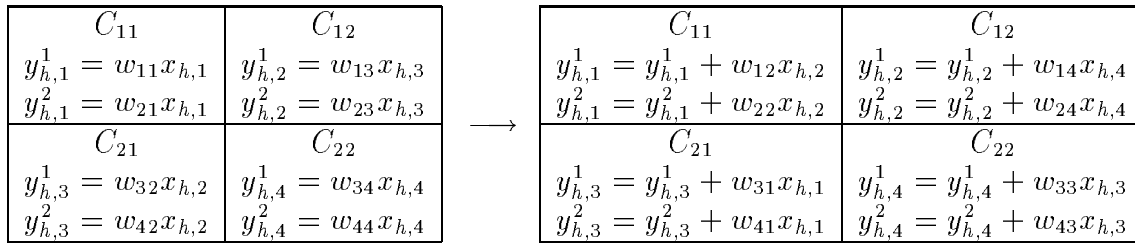


Figure 5.4: Partial hidden layer outputs $y_{h,j}$ are calculated by transferring the inputs $x_{h,i}$ vertically.

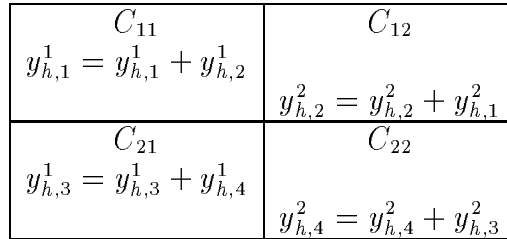


Figure 5.5: Hidden layer outputs $y_{h,j}$ are calculated by horizontal accumulation.

the processors may become idle. Although communication is minimized in the algorithm, the communication overhead will still be large for most real neural networks and parallel machines because the computation grains become too small.

5.2 Combined Solutions

The two former schemes are based on a *single* degree of parallelism (node parallelism contain two sub-degrees of parallelism). The two next implementations combine several degrees of parallelism to improve the scalability to a large number of cells.

5.2.1 Combining Two Degrees of Parallelism – Training Set and Neuron Parallelism (2APC)

The many interconnections in a fully connected feed-forward neural network lead to much communication between cells, which increases for larger number of cells. The fixed structure of the target machine makes it impossible to increase the number of cell interconnections. When the neural network/training set is dispersed over many cells, the granularity becomes finer 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 neuron parallelism for the Connection Machine is proposed in [161]. The network is replicated in each row of cells as shown in Figure 5.6. Each gray box is one processing element (cell).

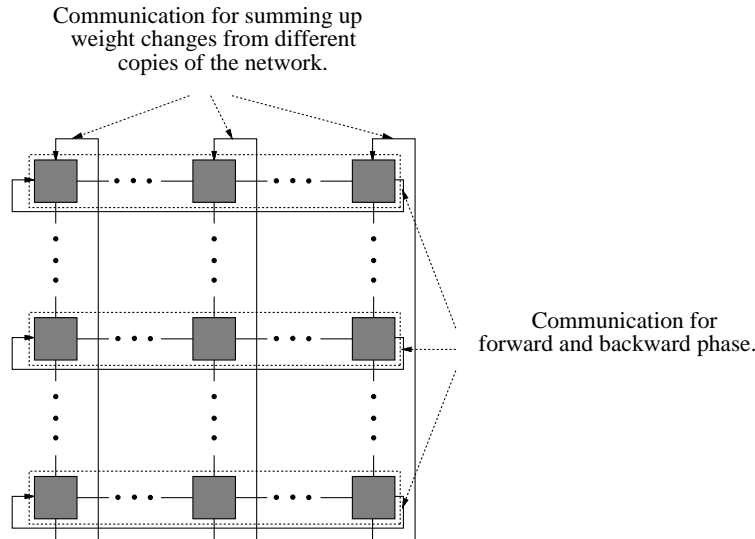


Figure 5.6: Combination of training set and neuron parallelism.

Each row contains different training patterns, thus weight update is performed by communication along columns to sum up weight changes (*learning by block/epoch*) for the different copies of the network. Each cell in a row computes the output of certain neurons, thus the weight matrices are partitioned horizontally (vertical slicing of the feed-forward network) and mapped statically.

As an example, if a row consists of two cells the weight matrix partitioning and multiplication in the forward pass is as follows

$$\begin{bmatrix} y_{L,1} \\ y_{L,2} \\ y_{L,3} \\ y_{L,4} \end{bmatrix} = \begin{bmatrix} w_{L,11} & w_{L,12} & w_{L,13} & w_{L,14} \\ w_{L,21} & w_{L,22} & w_{L,23} & w_{L,24} \\ w_{L,31} & w_{L,32} & w_{L,33} & w_{L,34} \\ w_{L,41} & w_{L,42} & w_{L,43} & w_{L,44} \end{bmatrix} \begin{bmatrix} x_{L,1} \\ x_{L,2} \\ x_{L,3} \\ x_{L,4} \end{bmatrix}$$

Each cell in a row holds all input training pattern elements. This allows the hidden outputs to be computed directly without inter-cell communication. Each cell then contains a disjoint subset of the hidden neuron outputs. These values have to be exchanged between the cells in a row to compute the output layer neurons.

Pseudo-code for the implementation named 2APC is given in Figure 5.7. This is the program running in each cell, which is downloaded by the host.

Each cell is assigned an equivalent number of neurons. If the number of neurons in an applications is not divisible by the number of cells, the additional neurons can be masked.

```

BackPropagation_2APC()
begin
   $n_h = \left\lfloor \frac{N_h}{C_x} \right\rfloor$ ; { Number of hidden neurons in a cell }
   $n_o = \left\lfloor \frac{N_o}{C_x} \right\rfloor$ ; { Number of output neurons in a cell }
   $\mu_c = \left\lfloor \frac{\mu}{C_y} \right\rfloor$ ; { Weight update interval }
  LookUp(MyCellId_x MyCellId_y);
  Read learning parameters from the host;
  Initialize weights;
  if MyCellId_y < (P mod C_y) then
    MaxPattern =  $\left\lfloor \frac{P}{C_y} \right\rfloor + 1$ ;
  else
    MaxPattern =  $\left\lfloor \frac{P}{C_y} \right\rfloor$ ;
  Read training patterns from the host;
  while trainend = 0 do begin
    totalerror = 0;
    for p = 1 to MaxPattern do begin
      { Forward phase }
      Compute  $n_h$  hidden outputs,  $\mathbf{y}_h$ ;
      Multiply  $\mathbf{y}_h$  by the corresponding weights  $\mathbf{W}_o$  for  $\mathbf{y}'_o$  partial output values;
      for c = 1 to  $C_x - 1$  do
        Send_receive_multiply_add( $n_h$  hidden elements);
      Compute sigmoid function for the  $n_o$  outputs,  $\mathbf{y}_o$ ;
      { Backward phase }
      Compute  $n_o$  elements of delta error  $\delta_o$  and sum of error from the  $n_o$  outputs;
      error = SumError(x-dimension, error); { Global reduction function }
      if error > e then
        totalerror = totalerror + 1;
      Compute  $N_h$  partial  $\delta_h$  values; { Hidden delta error }
      for c = 1 to  $C_x - 1$  do
        Send_receive_add( $n_h$  elements of  $\delta_h$ );
      Accumulate_weight changes;
      if (p mod  $\mu_c$ ) = 0 and p > 0 then
        Update_weights;
    end;
    SetStatus(totalerror > 0);
    Update_weights;
    if GetStatus = 0 then
      trainend = 1;
  end;
  Check_network;
end;

```

Figure 5.7: Parallel BP training algorithm running in each cell, combining training set parallelism and neuron parallelism (2APC). C_x and C_y are the number of cells in horizontal and vertical dimension, respectively.

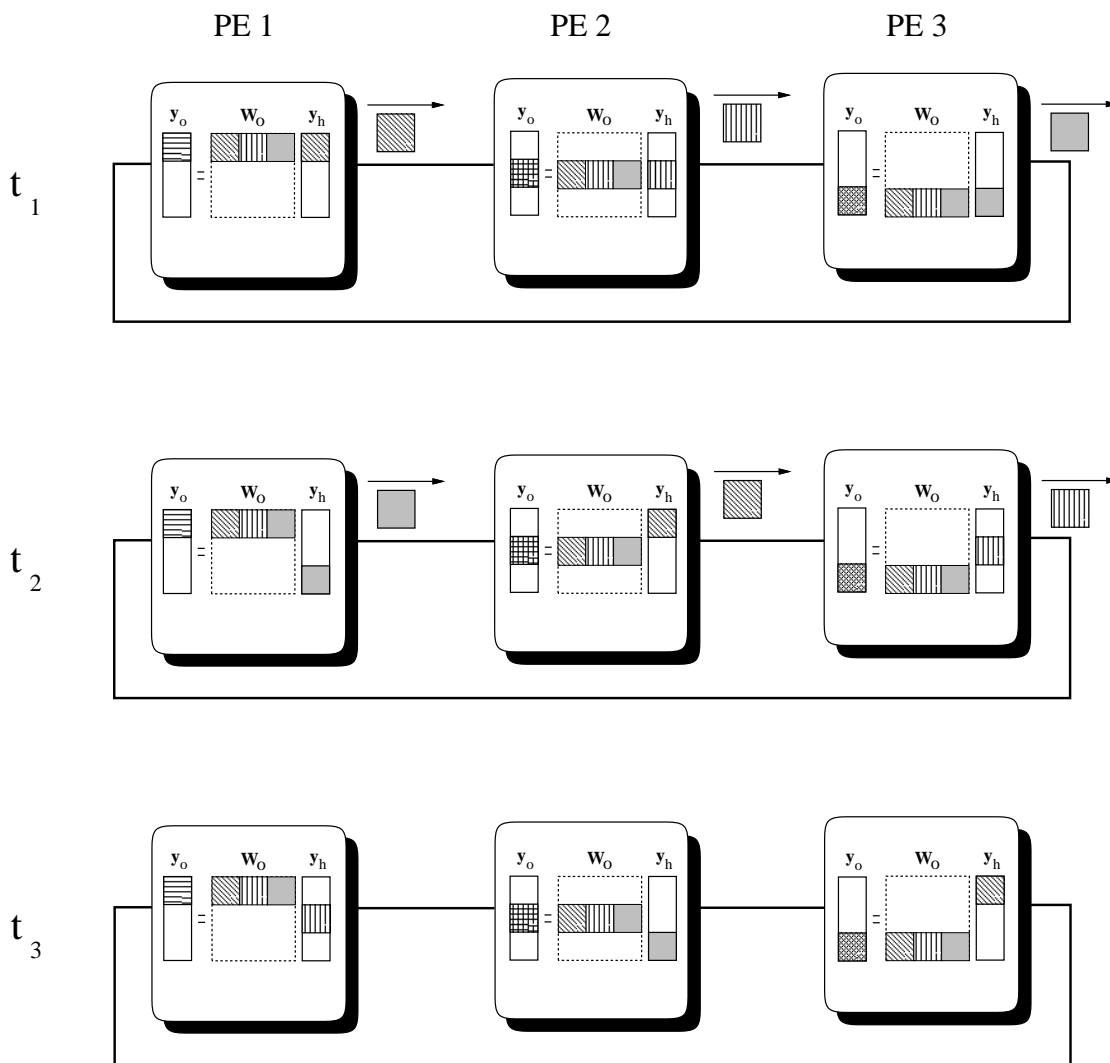


Figure 5.8: Exchange of the hidden output values for $C_x = 3$. Two communication steps are required for broadcasting the hidden outputs to the other cells.

After initialization, the training starts by computing the forward phase. The exchange of the values of the hidden units and computation of output layer units is done by the `Send_receive_multiply_add()` function. It is explained by an example in Figure 5.8. In the first step, after multiplying weights and the locally computed hidden units, each cell sends the hidden elements to their right-hand neighbors. Then, the received elements are multiplied by the corresponding weights and the results are added to the output elements. The received package is sent further thus after C_x-1 propagations, where C_x is the number of cells in a row, all hidden elements have visited all cells.

At the start of the backward phase the error and output delta error δ_o is computed locally

in each cell. The hidden delta error δ_h cannot be directly calculated as explained in Section 3.1.3 for neuron parallelism. The accumulation of partial hidden delta error δ_h , which is done by the `Send_receive_add()` function, is shown by an example in Figure 5.9. The method is presented in [159]. After two send-receive-add operations each cell has received its part of δ_h to be used for hidden weight update.

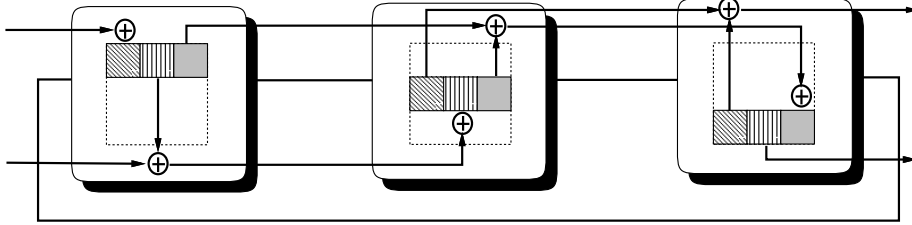


Figure 5.9: Accumulation of the hidden delta error.

After one training iteration, a *status* is set by all cells according to the accumulated error for the training patterns. If patterns in all rows have been learned, the training is stopped.

In the `Check_network()` function the weights from two rows² are sent to the host for verification. The host then computes the forward phase of all the training patterns and counts the number of patterns below the error threshold – defined in Equation 2.23. This number should equal that obtained by the serial implementation, if the same initial training parameters and weights are used. This function therefore ensures that the parallel algorithm trains correctly.

5.2.2 Combining Three Degrees of Parallelism – Pipelining, Training Set Parallelism and Neuron Parallelism (3APC)

Although the multiple degrees solution (2APC) will give better speedup than the previously described single parallel degree algorithms, it does not benefit from pipelining the training patterns. It is therefore here proposed to combine three degrees of parallelism in the BP algorithm to obtain a higher speedup on a computer with a *large* number of cells. As seen in Section 3.3.10, few implementations combine more than two degrees of parallelism.

The partitioning of the network, which extends the 2APC by pipelining, is shown in Figure 5.10. Notice that *all* input training pattern elements are stored in *each* hidden layer cell to reduce communication. The output layer cells *must* propagate their input elements (hidden layer output elements) along the row of cells. As before, training set parallelism means that several copies of the network are made. The mapping can be explained by the example of the 2-by-4 cell topology given in Figure 5.11. Each degree of parallelism

²Hidden weights from row number 2 and output weights from row number 3 are tested.

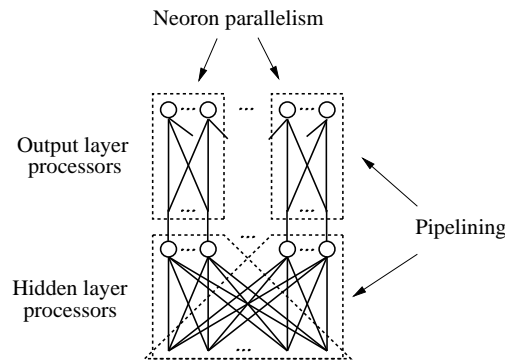


Figure 5.10: Partitioning of the network when pipelining and neuron parallelism are combined. The dotted lines show which *neurons* are mapped to each cell.

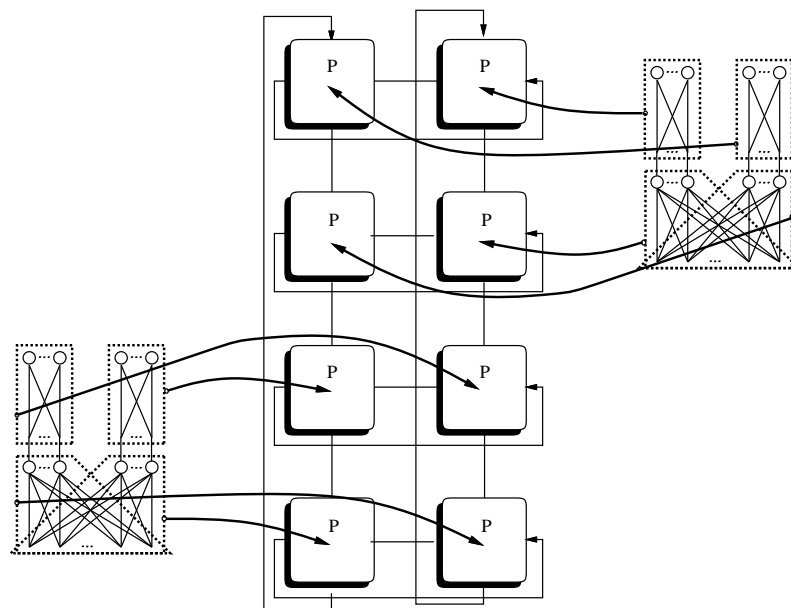


Figure 5.11: Example for combining three degrees of BP parallelism (3APC).

is assigned to two subgroups of cell resources. That is, each copy of the network is sliced both vertically and horizontally into two partitions. Further, the training set is partitioned into two disjoint subsets.

A suggested cell assignment for the general case is shown in Figure 5.12. Two processor rows compute the weight changes for one training subset. The **H** row acts as the hidden layer, and the **O** row acts as the output layer. Interlayer communication is column-wise, because the training set parallelism only requires communication when weights are updated. This layout reduces the degree of training set parallelism, thus it is required to double the

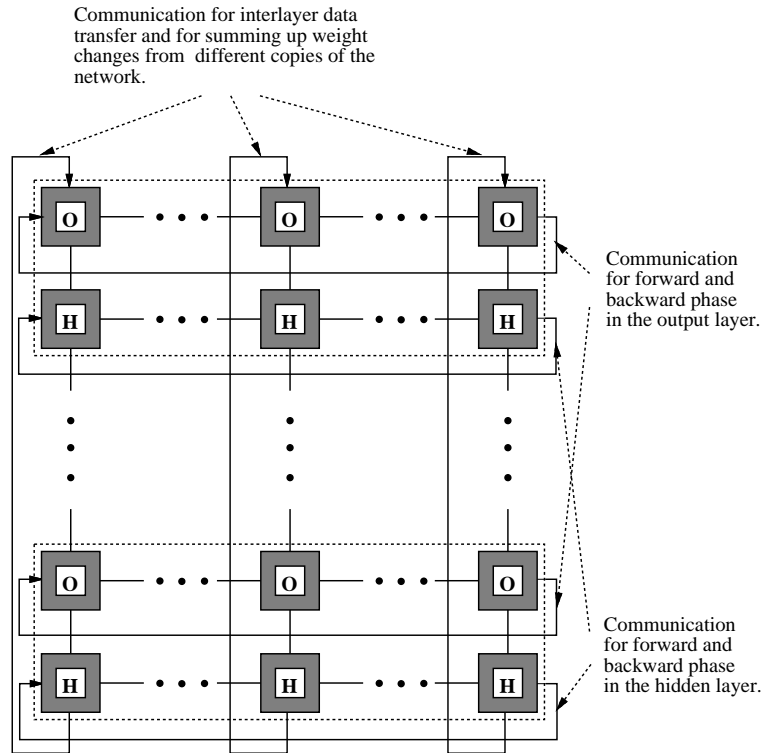


Figure 5.12: Combination of three types of parallelism in the BP algorithm; Training set parallelism, neuron parallelism and pipelining of the training patterns are combined in a way that minimizes communication conflicts.

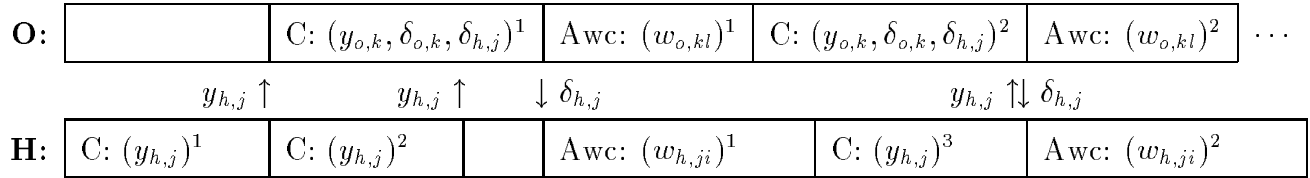
number of training patterns assigned to each row. The problem of grain size reduction for a large number of cells is therefore reduced. The advantages of the pipelining method may be summarized as:

- Twice as many patterns are computed on each cell between weight updates.
- Reduced memory requirement, since
 - only one weight matrix has to be stored in each processing element.
 - program code size can be reduced.

Disadvantages may be:

- The hidden layer PEs have to store twice as many input patterns, leading to larger memory requirements. However, no output vectors need to be stored in these PEs.
- The performance is dependent on the computation load balance between the layers.

When the weights are updated, communication is between every other row of cells. Since inter-cell communication time on AP1000 is independent of distance, the updating time of weights will not essentially increase compared to direct communication between two rows.



where:

H: Hidden layer	C: $(z)^p$: Compute z for pattern p
O Output layer	Awc: $(w)^p$: Accumulate weight change values for pattern p
$1 \leq i \leq \text{number_of_input_neurons}$	
$1 \leq j \leq \text{number_of_hidden_neurons_in_cell}$,	$1 \leq l \leq \text{number_of_hidden_neurons}$
$1 \leq k \leq \text{number_of_output_neurons_in_cell}$	

Figure 5.13: Pipelining of hidden and output layer calculation.

The computation in one cell of each of the layers is shown in Figure 5.13, which details the principle presented in Figure 3.3. The hidden layer is represented in the lower row, while the upper row indicates the output layer. Communication between the layers is only needed to transfer $y_{h,j}$ to the output layer and the hidden layer error $\delta_{h,j}$ to the hidden layer.

Pseudo-code for the implementation named 3APC is given in Figures 5.14 and 5.15 for the hidden and output layer, respectively. The program is similar to the 2APC but includes modifications for the distributed computation of the two weight layers. In the hidden layer program, concerns has to be shown for weight updating, so that weight change values for some patterns are not missed. The output layer cells test if training should stop and set their status accordingly. The hidden cells read this status to test if learning can stop.

The balance of computation load between the layers depends on the number of neurons in each layer. The output layer cells do most of the backward phase computation. However, the hidden layer is usually larger – see Section 2.2, and therefore *more* time is used to compute outputs and accumulate weight change values. Parts of the $\delta_{h,j}$ computation may be moved to the layer having least computation, as shown in the output cell pseudo-code in Figure 5.16.

Three different placements of program code are possible. The code *after* the selected alternative (A, B, or C) is moved to the hidden layer cell. The code in front of the alternatives is executed on the output layer cells. In alternative A, each output cell sends the partial sums of hidden delta error to its corresponding hidden cell. Then, the hidden cells sum the partial sums. In alternative B and C this summing is done by the output cells. The y_h elements are also stored in the output cells, since they were used in the forward

```

Hidden_3APC()
begin
   $n_h = \lfloor \frac{N_h}{C_x} \rfloor$ ;
   $\mu_c = \lfloor \frac{\mu}{C_y/2} \rfloor$ ; { Weight update interval }
  LookUp(MyCellId_x MyCellId_y);
  Read learning parameters from the host;
  Initialize weights;
  if MyCellId_y < (P mod C_y/2) then
    MaxPattern =  $\lfloor \frac{P}{C_y/2} \rfloor + 1$ ;
  else
    MaxPattern =  $\lfloor \frac{P}{C_y/2} \rfloor$ ;
  Read training input patterns from the host;
  while trainend=0 do begin
    Compute  $n_h$  hidden outputs for pattern 1;
    Send hidden outputs elements NORTH; { Send to output layer cell }
    for p= 2 to MaxPattern do begin
      { Forward phase }
      Compute  $n_h$  hidden outputs for pattern p;
      Send hidden outputs elements NORTH; { Send to output layer cell}
      { Backward phase }
      Receive  $n_h$  elements of  $\delta_h$  from NORTH;
      Accumulate_weights for pattern p - 1;
      if (p mod  $\mu_c$ ) = 0 and p > 0 then begin
        Receive  $\delta_h$  from NORTH and accumulate weight changes for pattern p;
        Update_weights;
        p = p + 1;
        Compute  $n_h$  hidden outputs for pattern p;
        Send hidden outputs elements NORTH;
      end;
    end;
    Receive  $\delta_h$  from NORTH and accumulate weight changes for pattern MaxPattern;
    Update_weights;
    if GetStatus = 0 then
      trainend= 1;
  end;
end;

```

Figure 5.14: Parallel BP training algorithm running in a hidden layer cell for 3APC.

```

Output_3APC()
begin
   $n_h = \left\lfloor \frac{N_h}{C_x} \right\rfloor$ ;
   $n_o = \left\lfloor \frac{N_o}{C_x} \right\rfloor$ ;
   $\mu_c = \left\lfloor \frac{\mu}{C_y/2} \right\rfloor$ ; { Weight update interval }
  LookUp(MyCellId_x MyCellId_y);
  Read learning parameters from the host;
  Initialize weights;
  if MyCellId_y < (P mod C_y/2) then
    MaxPattern =  $\left\lfloor \frac{P}{C_y/2} \right\rfloor + 1$ ;
  else
    MaxPattern =  $\left\lfloor \frac{P}{C_y/2} \right\rfloor$ ;
  Read training output patterns from the host;
  while trainend=0 do begin
    totalerror = 0;
    for p = 1 to MaxPattern do begin
      { Forward phase }
      Read  $n_h$  hidden output elements  $\mathbf{y}_h$  from SOUTH;
      Multiply  $\mathbf{y}_h$  by the corresponding weights  $\mathbf{W}_o$  for  $\mathbf{y}'_o$  partial output values;
      for c = 1 to  $C_x - 1$  do
        Send_receive_multiply_add( $n_h$  hidden elements);
      Compute sigmoid function for the  $n_o$  outputs,  $\mathbf{y}_o$ ;
      { Backward phase }
      Compute  $n_o$  elements of delta error  $\delta_o$  and sum of error from the  $n_o$  outputs;
      error = SumError(x-dimension, error); { Global reduction function }
      if error > e then
        totalerror = totalerror + 1;
      Compute  $N_h$  partial  $\delta_h$  values; { Hidden delta error }
      for c = 1 to  $C_x - 1$  do
        Send_receive_add( $n_h$  elements of  $\delta_h$ );
      Send  $n_h$  elements of  $\delta_h$  SOUTH;
      Accumulate_weights;
      if (p mod  $\mu_c$ ) = 0 and p > 0 then
        Update_weights;
    end;
    SetStatus(totalerror > 0);
    Update_weights;
    if GetStatus = 0 then
      trainend = 1;
  end;
end;

```

Figure 5.15: Parallel BP training algorithm running in a output layer cell for 3APC.

```

for  $j = 1$  to  $N_h$ 
  for  $k = 1$  to  $n_o$ 
     $\delta_h[j] = \delta_h[j] + w_o[k][j]\delta_o[k];$ 
  Send_to_hidden_cell;                                     (Alt. A)
for  $c = 1$  to  $(C_x - 1)$ 
  Send_receive_add( $n_h$  elements of  $\delta_h$ );
  Send_to_hidden_cell;                                     (Alt. B)
for  $j = 1$  to  $n_h$ 
   $\delta_h[c_x n_h + j] = y_h[c_x n_h + j] (1 - y_h[c_x n_h + j]) \delta_h[c_x n_h + j];$ 
  Send_to_hidden_cell;                                     (Alt. C)

```

Figure 5.16: Hidden delta error computation.

phase. Thus, no additional communication is needed for alternative B or C. Moreover, in alternative A each output cell has to send one message of size N_h – the total number of hidden neurons, to the corresponding hidden cell. However, for the two other alternatives each output cell has to send a message of size only n_h – the number of hidden neurons assigned to a cell. In this work, alternative C has been used, because $N_i \gg N_o$ for most applications. The Send_receive_add() function is based on sequential summing of partial sums (as illustrated in Figure 5.9) and not a $C_x - 1$ message broadcast of sums from each output cell *directly* to the target cells. This technique is implemented on RENNS and is described in Chapter 9.

Optimizing the Weight Update

As described earlier, convergence rate improves when weights are updated frequently and as such it is important to minimize the weight update time. To avoid weight updating becoming a bottleneck in large systems, a $(\log_2 C_y - 1)$ step summing technique is proposed for summing weight change matrices. C_y is the number of cells in y dimension and for AP1000 $C_y \in \{4, 8, 16, 32\}$. The summing algorithm is illustrated in Figure 5.17a) [139].

The summing starts in the leftmost column in the figure. Hidden layer cells send their matrices to cells south of themselves, while output layer cells send to cells to their north. The solution is called *edge summing*, since the matrices are summed 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 8 weight change matrices. Then, the result is sent back to each hidden and output layer cell – see Figure 5.17b). This is done in a similar way as in the summing, but the opposite direction is used for sending the data.

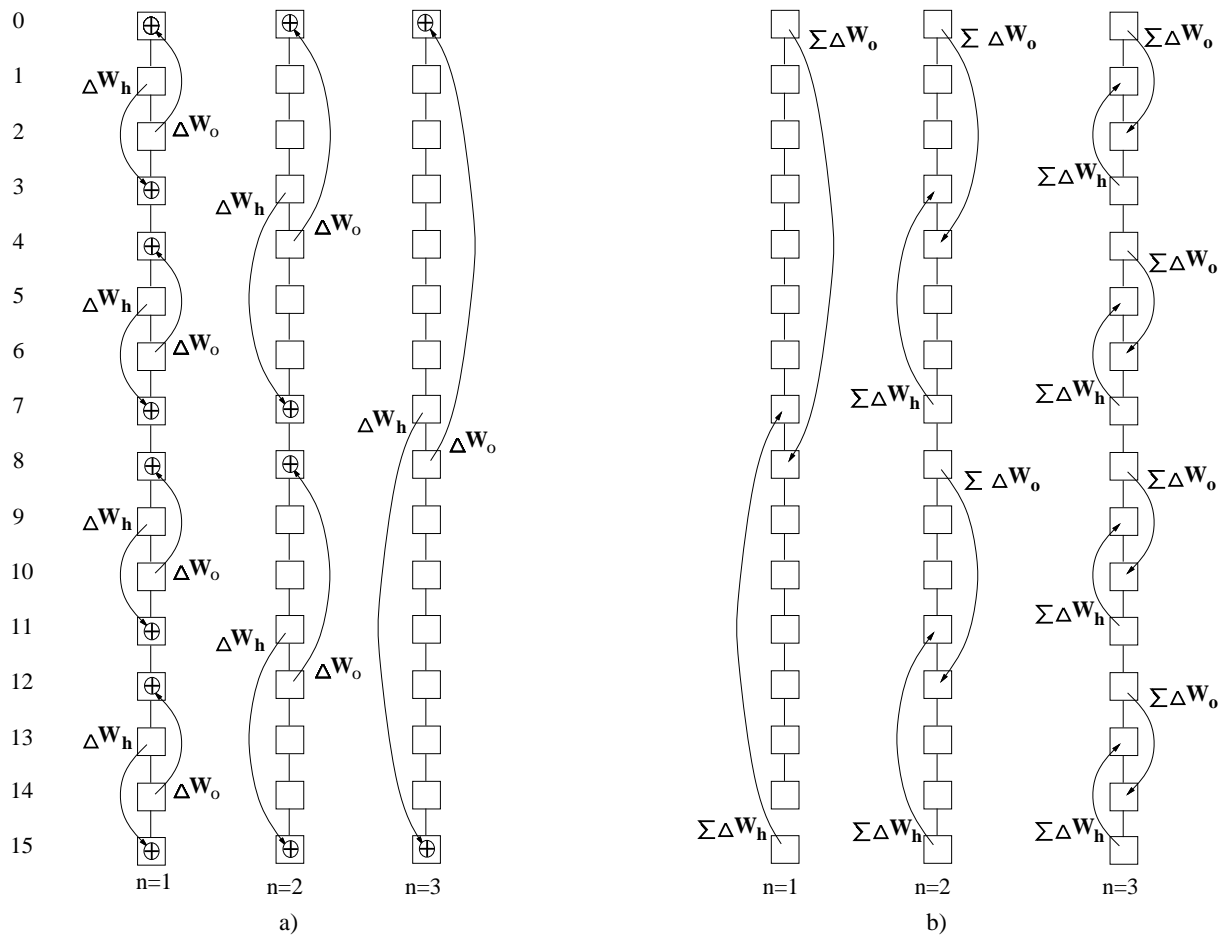


Figure 5.17: a) Summing the weight change matrices before weight update in a system of 16 vertical cells. b) Broadcasting the summed weight change matrices back to each cell.

Pseudo-code for the summing and distribution of the results is given in Figure 5.18. During summing the receiving cell adds its local weight change matrix to the received matrix and then forwards the summed matrix. After summing, the summed matrices are broadcasted, in the opposite direction of summing, to the cells. After each cell receives the summed weight matrix, they *update* their local weight matrices.

One problem of this scheme is network contention, i.e. several cells send messages concurrently to the same link. However, the communication time on AP1000 is reasonable also when network contention exists. It is approximately doubled if five messages are on the same channel, indicating that much time is spent within each cell for communication overhead and less on the physical transfer of data [56].

The broadcasting part of the scheme shown in Figure 5.17b) can be omitted if we make

```

SumHiddenWeightChanges()
begin
  NumberSummingSteps =  $\log_2(C_y) - 1$ ;
  for s = 1 to NumberSummingSteps do begin { Sum weights }
    if  $((\text{MyCellId}_y - 2^s + 1) \bmod 2^{s+1}) = 0$  then
      SendWeights(MyCellId_y +  $2^s$ );
    if  $((\text{MyCellId}_y - 2^{s+1} + 1) \bmod 2^{s+1}) = 0$  then
      ReceiveAndSumWeights(MyCellId_y -  $2^s$ );
  end;
  for s = NumberSummingSteps to 1 do begin { Broadcast summed weights }
    if  $((\text{MyCellId}_y - 2^{s+1} + 1) \bmod 2^{s+1}) = 0$  then
      SendWeights(MyCellId_y -  $2^s$ );
    if  $((\text{MyCellId}_y - 2^s + 1) \bmod 2^{s+1}) = 0$  then
      ReceiveWeights(MyCellId_y +  $2^s$ );
  end;
end;

SumOutputWeightChanges()
begin
  NumberSummingSteps =  $\log_2(C_y) - 1$ ;
  for s = 1 to NumberSummingSteps do begin { Sum weights }
    if  $((\text{MyCellId}_y - 2^s) \bmod 2^{s+1}) = 0$  then
      SendWeights(MyCellId_y -  $2^s$ );
    if  $(\text{MyCellId}_y \bmod 2^{s+1}) = 0$  then
      ReceiveAndSumWeights(MyCellId_y +  $2^s$ );
  end;
  for s = NumberSummingSteps to 1 do begin { Broadcast summed weights }
    if  $(\text{MyCellId}_y \bmod 2^{s+1}) = 0$  then
      SendWeights(MyCellId_y +  $2^s$ );
    if  $((\text{MyCellId}_y - 2^s) \bmod 2^{s+1}) = 0$  then
      ReceiveWeights(MyCellId_y -  $2^s$ );
  end;
end;
end;

```

Figure 5.18: Function for summing the weight change matrices for 3APC.

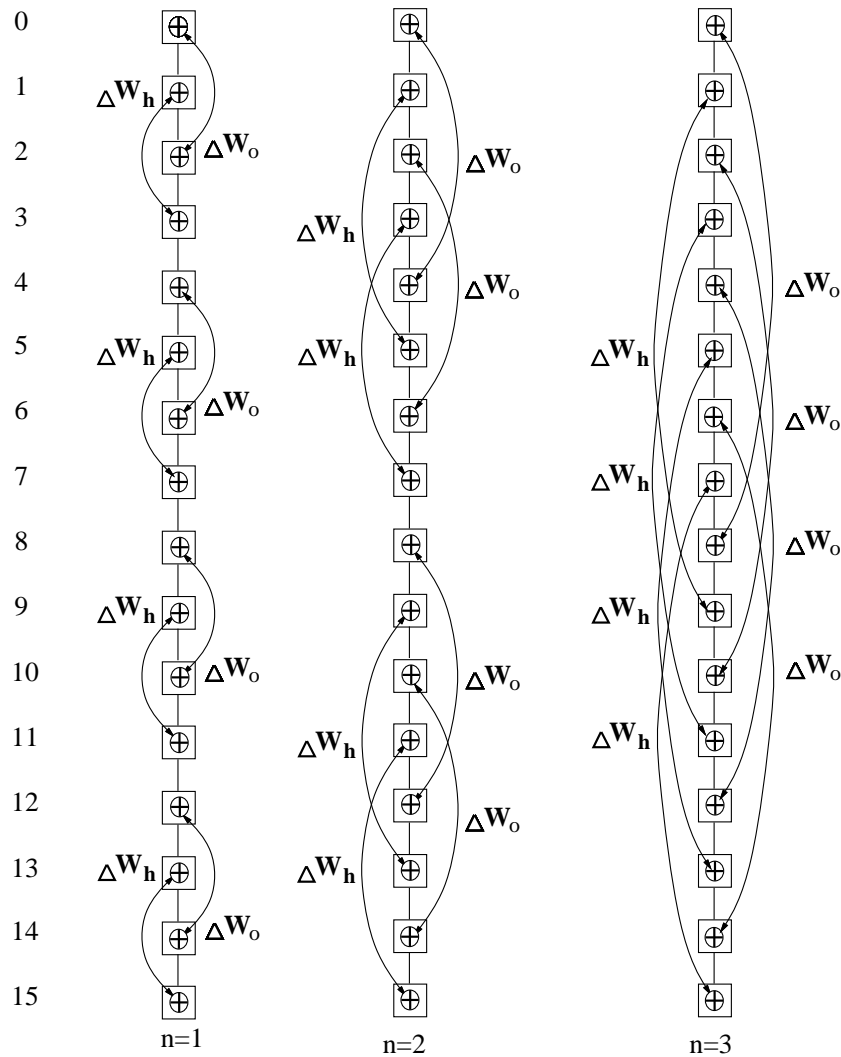


Figure 5.19: Summing the weight matrices before weight update using bi-directional communication and duplicated summing.

the communication in part Figure 5.17a) bi-directional and duplicate the summing – see Figure 5.19. However, as seen in the figure, this introduces more communication conflict and redundant computation overheads.

The same summing method can be used for the mapping without pipelining [154]. The number of summing steps becomes $(\log_2 C_y)$. Thus, one less step of summing is needed for 3APC compared to 2APC. However, for 3APC there are additional overheads incurred due to filling the pipeline after new weights are computed.

5.3 Summary

A description of how to exploit multiple degrees of BP parallelism on a 2D-torus MIMD computer has been given. Three techniques suggested by other researchers are described, followed by an outline of the mapping proposed in this work. A fast method for summing the weight change matrices is also proposed to reduce the weight update time.

In the next chapter, the performance of these mappings on the AP1000 computer is reported.

Chapter 6

Comparing Fixed Parallel Implementations on AP1000

This chapter reports the performance results for the four parallel implementations described in the previous chapter. First the NETtalk network is used to compare the different implementations. This includes results on the computational load balance for the scheme using pipelining. Then, the implementations are tested using several other applications. Most of the results are published in [135, 139].

Except for the node parallel algorithm, *learning by block* was used. Few guidelines for choosing proper weight updating frequency exists. In [58], the weights were updated after 2000 training patterns for NETtalk. For the GF11 implementation [154], the weights were updated only after the whole training set of 12022 patterns was input. In the experiments described below, it was chosen to update the weights after 1024 patterns for NETtalk. For the speech and image recognition applications the weights were updated after 512 patterns. The results presented in this chapter are also included in tables in Appendix B.1.

6.1 NETtalk

Figure 6.1 shows the performance of NETtalk network training for the four implementations and with between 1 and 64 processors. 80 hidden neurons are used. As shown, the combined solutions give rise to increasing CUPS values for larger number of processors. In the following sections, the performance of each implementation is discussed.

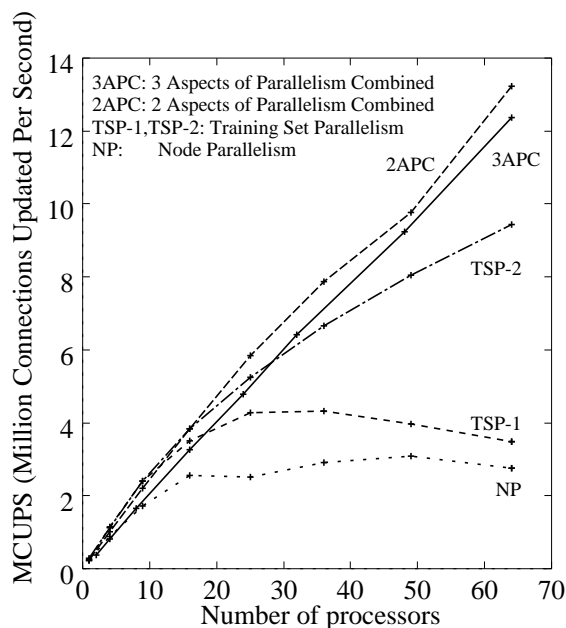


Figure 6.1: CUPS performance for the implemented algorithms running NETtalk with 80 hidden neurons.

6.1.1 Training Set Parallelism

First a version (TSP-1) that used the host for summing the weight-change matrices was implemented. The algorithm performs well for a *small* number of processors, but if more than 36 processors are used – 28 training patterns computed per cell between weight updates, the training time *increases*. This can be explained due to the large weight update overhead caused by the training patterns being distributed over many cells. Later, a version (TSP-2) was implemented that used *global reduction functions* for summing the weight-change matrices. The performance improved, but for the case of a large system, it is not as high as in the combined solutions. Although these training set parallel implementations give better performance than node parallelism, *learning by block* leads to slower convergence rates and therefore requires more training iterations.

6.1.2 Node Parallelism

The main problem with the node parallel algorithm is that the computation granularity becomes too fine, such that it is impossible to obtain high speedup for a large number of processors. From the nonlinearity in the result curves, it can be estimated 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. The algorithm benefits from using *learning by pattern*.

6.1.3 Combined Solutions

The results support the proposal that there is a 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, probably because the latter suffers from an uneven load balance between the hidden and output layer. Both algorithms are sensitive to how well the perceptron network/training set is mapped to the architecture. Thus, the performance is not strictly linear.

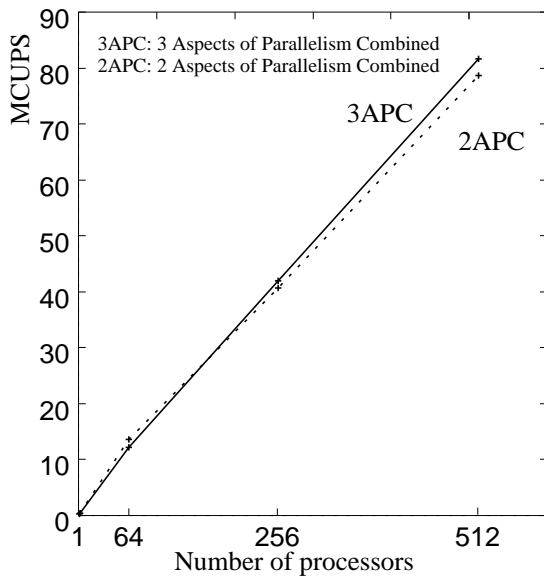


Figure 6.2: Performance for the two algorithms combining different parallel aspects, running NETtalk with 80 hidden neurons.

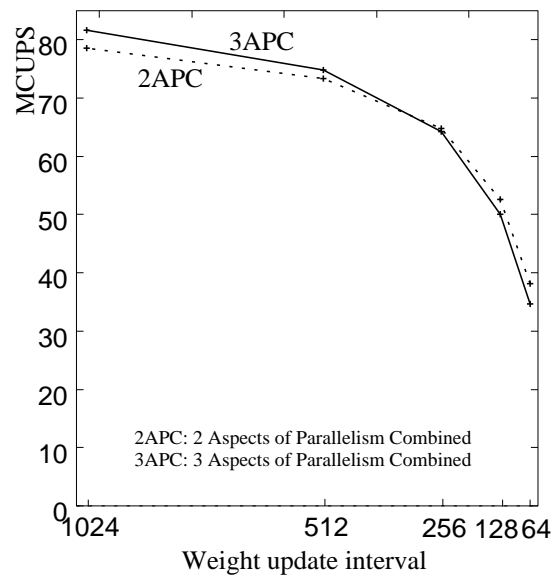


Figure 6.3: Performance for different numbers of weight updates per iteration in the NETtalk experiment on a 512 cell system.

Figure 6.2 illustrates the performance for the two implementations for a 80 hidden neurons network on 1 to 512 processors. 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 be explained in part by the small computational grain for computing 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 that for 2APC. The reason for this is that, 3APC uses *two* rows for each network copy, while 2APC needs only one row. In comparison, the NETtalk application on a Sparc 10 workstation¹ reached a performance of 0.6 MCUPS, using *learning by pattern*. A NETtalk performance of 81 MCUPS is faster than

¹The parts of the program required for parallel execution on AP1000 were removed to make the program executable on serial computer.

results reported from other general purpose computers – see Table 3.5. The serial-parallel speedup ratio is 135. However, the real difference in total training time will be less due to slower convergence for the parallel training scheme. This topic will be investigated in chapter 8.

Figure 6.3 illustrates the performance decrease with 512 processors 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 the hidden and output layer weights are updated concurrently for 3APC.

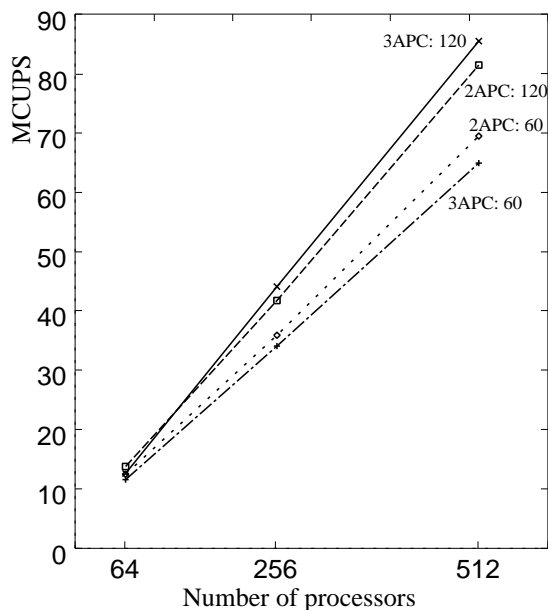


Figure 6.4: Performance for the 2APC and 3APC algorithms for the NETtalk network with 60 and 120 hidden neurons.

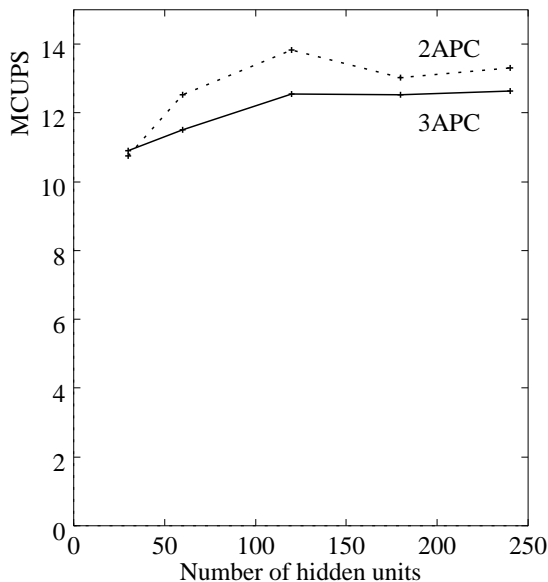


Figure 6.5: NETtalk running on 64 cells.

In Figure 6.4, the NETtalk performance for 60 and 120 hidden neurons is given. 3APC performs better than 2APC for 120 hidden neurons. Both programs have better scaling for 120 hidden neurons than for 60. When the number of hidden neurons is increased, both the number of hidden and output *forward* computations increases. The hidden layer cells get a larger number of neurons to process, while the output layer cells get a larger number of inputs per neuron. The increase for each of the layers is of equal proportion, e.g. for 120 neurons, the number is twice that for 60 neurons. The hidden layer cells are slightly more loaded because of the computation of the sigmoid function. However, the *backward* phase requires more computation by the output layer processors. This is due to the main part of the backward pass being undertaken by the output layer processors. A large number of hidden neurons seems to be beneficial for the hidden-output load balance in this application. According to these observations, a decrease in the number of input

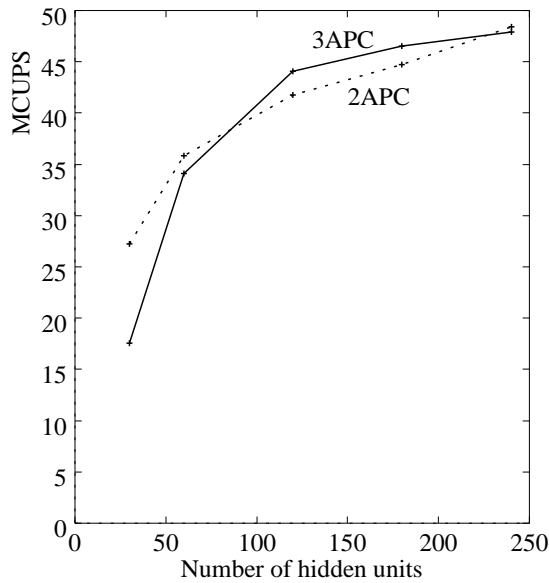


Figure 6.6: NETtalk running on 256 cells.

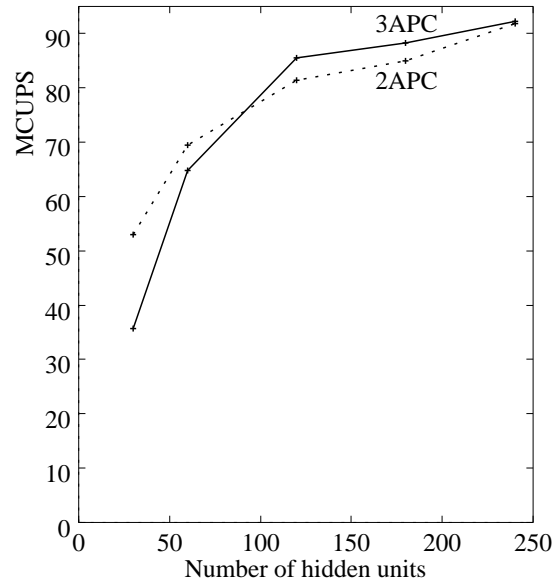


Figure 6.7: NETtalk running on 512 cells.

neurons or an increase in the number of hidden or output neurons moves computation load from the hidden to the output processors. If the network becomes small the communication overhead may change this interpretation – as shown in Section 9.4.

Figure 6.5 – 6.7 present the performance as a function of the number of hidden units for 64, 256, and 512 cells, respectively. For 64 cells, the learning speed is less variable compared to that in larger systems, and 2APC is faster than 3APC except for 30 hidden units. On large systems, the performance is highly dependent on the number of hidden units. For 120 and 180 hidden units, 3APC gains higher speed than 2APC. Thus, the load assignment is better for larger systems than smaller systems for these numbers of hidden units.

6.1.4 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, the results are shown for two performance analyses, using an AP1000 performance analyzer tool.

Figure 6.8 presents the processor utilization in a 16 processor (4x4) configuration, running NETtalk (80 hidden neurons), for the first 4 training patterns. The time axis is horizontal. The upper part of the figure shows the number of active cells. The lower part shows the activity for the hidden processors 4 and 12, and the output processors 0 and 8. This picture corresponds to the earlier illustration of pipelining in Figure 3.3. Dark gray areas indicate

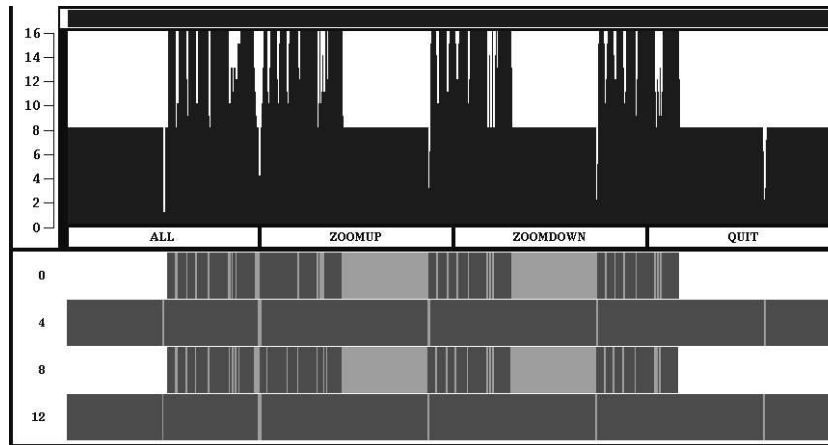


Figure 6.8: Performance on a 16 processor configuration for the first 4 training patterns of NETtalk. Dark gray areas indicate active processor, while light gray areas indicate that the processor is waiting for data.

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 compute almost all 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 white thin columns in the upper part of the performance plot arise from the communication between the output layer processors.

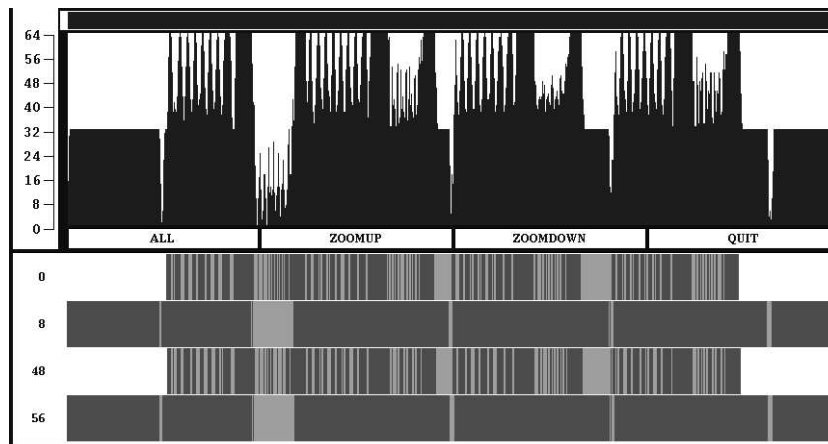


Figure 6.9: Performance on a 64 processor configuration for the first 4 training patterns of NETtalk.

Figure 6.9 shows that the idle time for the output layer processors is reduced, when the number of processors in the system increases. The output cells become more active, since

they do more communication than the hidden layer cells. Exchanging values between 8 processors needs more time than between 4. The hidden layer processors become idle while waiting for the error of the first pattern. Usually there will be many training patterns between weight updating, thus this idle time will not be a major problem. In conclusion, the NETtalk application – with relatively few output neurons compared to input neurons, execute more efficiently i.e. obtain better load balance, on a large system than on a small system.

6.2 Other Applications

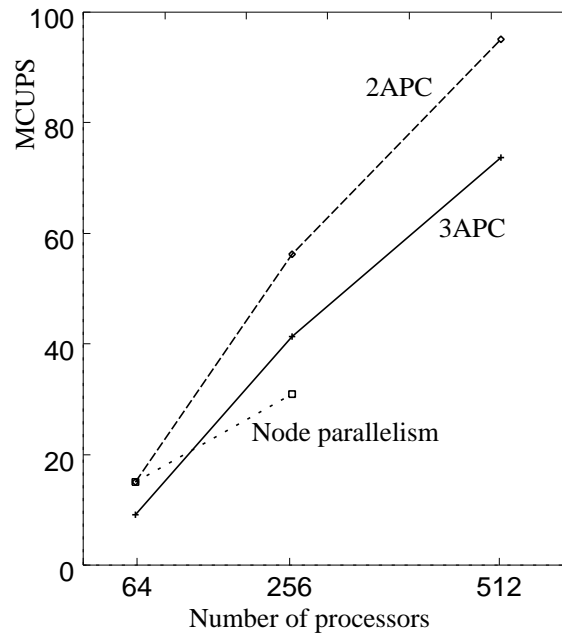


Figure 6.10: Performance for a large feed-forward network with 1024 input, 512 hidden and 64 output neurons.

Performance for a large neural network, to be used for e.g. image recognition, is given in Figure 6.10. The network consists of 1024 input, 512 hidden and 64 output neurons. 4096 training patterns were used, and the weight updates occurred every 512 patterns. 2APC outpaces 3APC, which is understandable from the difference in number of neurons 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 parallelism². For 64 processors the performance is equal to 2APC. For 256 processors the performance is almost half that of 2APC. This indicates that the computation grain becomes small, even for a relatively large neural network. However, the algorithm may need less training iterations since *learning by pattern* is used.

²It was impossible to run the node parallel implementation on the 512 cell configuration.

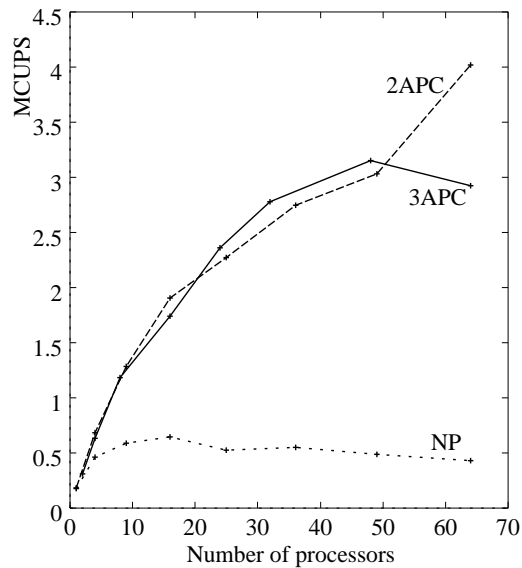


Figure 6.11: Performance for a character recognition network.

Figure 6.11 presents the learning speed for a small network learning to recognize 26 characters. For the training set parallel implementations, weights were updated after each presentation of the 26 characters. The application was used as a test program during development of the implementations for AP1000. The small network limits the performance for all the mapping methods, compared to the NETtalk application. However, the combined schemes scale well compared to NP. For 20 to 50 cells the 3APC performs better than 2APC. In this range, the load balance between the layers seems to even out.

Speech recognition requires a large network/training set. Thus, parallel processing is required. A network with 256 input units and 64 output units is used in Figure 6.12–6.14 for 64, 256, and 512 PEs, respectively. A weight update interval of 512 patterns was used. The performance is plotted as a function of the number of hidden units. The figures indicate that 3APC trains faster than 2APC for a certain range of hidden units. For a small number of cells, a small number of hidden units makes 3APC better than 2APC – see Figure 6.12. For 256 cells and 512 cells the number of hidden units, which makes 3APC better than 2APC, is larger. The *difference* in performance is also larger for these systems. Thus, 3APC is beneficial for large systems, when load balance is present. The shift in the best performance for 3APC indicates what number of hidden units results in the best load balance for the three different processor configurations. As seen for the NETtalk network, a larger number of cells needs a larger number of hidden units for obtaining load balance. This can be explained by the increase in the communication load for the output layer cells in a larger system. It is interesting to note that the largest number of hidden units does not give the best performance. This can be explained by increased communication overhead.

The performance plotted as a function of the number of processors appear in Figure 6.15

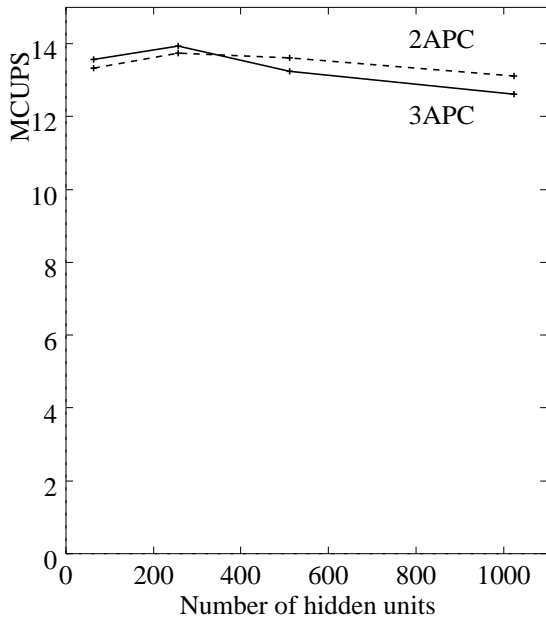


Figure 6.12: Speech recognition network on 64 cells.

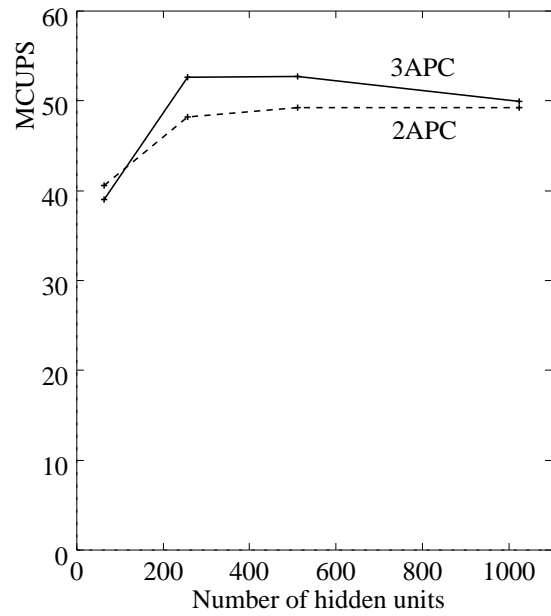


Figure 6.13: Speech recognition network on 256 cells.

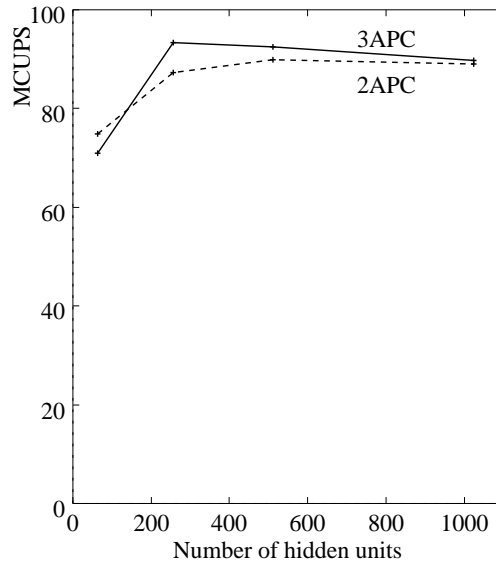


Figure 6.14: Speech recognition network on 512 cells.

and 6.16 for 2APC and 3 APC, respectively. For 64 hidden units, the performance and corresponding scaling is less than that for a larger number of hidden units. For 2APC, 512 hidden units gives the best performance, while 256 hidden units performs best for 3APC. This is for the case of a 512 cell system.

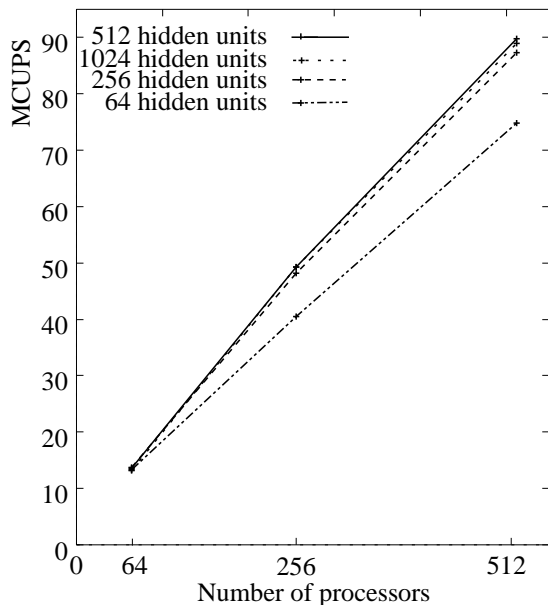


Figure 6.15: 2APC training speech recognition network.

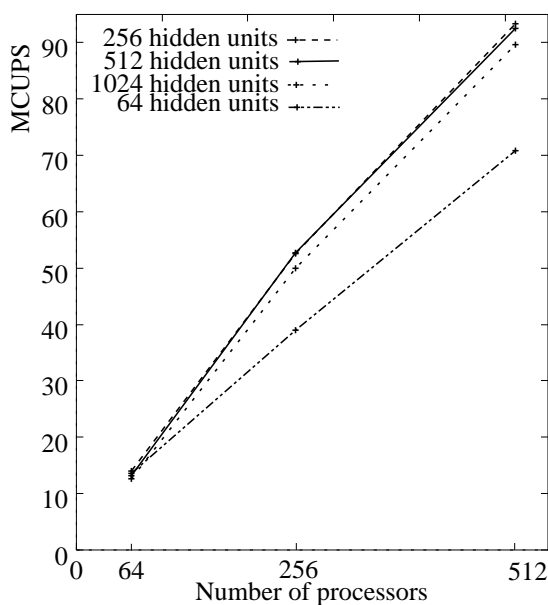


Figure 6.16: 3APC training speech recognition network.

6.3 Minimizing the Total Training Time

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, prior to start of training, compare the time used for 1 iteration of the various parallel implementations on the target machine. The times can be found either by running one iteration of each or by using performance estimation. The convergence rates should also be considered. Many parameters affect the convergence rate, like the number of hidden neurons, initial weight values, learning rate and the weight update frequency. Therefore it is difficult to estimate how often the weights should be updated to minimize the total training time. This topic is considered in the following chapters.

6.4 Summary and Discussion

In this chapter, results from execution of four parallel implementations on AP1000 are reported. The performance varies for different neural network sizes and number of processors in the system. NETtalk network with 80 hidden units is shown to train at a maximum speed of 81 MCUPS, using the proposed pipelining scheme. This combines three degrees of parallelism and the speedup is faster than results reported from other general purpose computers.

The results prove that it is necessary to combine multiple parallel degrees to obtain the highest possible performance on a large number of processors. By combining the degrees of BP parallelism, computational granularity can remain coarse even when using a large number of processors. Node parallelism has the advantage of *learning by pattern* and for large neural networks it may be an alternative.

The 3APC was not always faster than 2APC and thus it is shown that efficiency is dependent on network size and the number of processors used. For the NETtalk application, it was shown that the load balance between the hidden and output layer computation becomes more even as the number of processors is increased. The limitations of the mapping schemes are seen for small networks, where performance is limited. Thus, more flexibility should be introduced in the mappings to better adapt these to a given neural application.

The results are not limited to the parallel computer used but are relevant to implementations on other large general purpose computers.

Although the performance improvement by including pipelining is rather small, the inclusion can be of more importance if we assign a *different* number of cells to compute the different layers. This will be studied in more detail in Chapter 9.

Chapter 7

General Mapping onto 2D-torus MIMD Computers

The topic of this chapter is to make a mapping of BP onto a parallel processor, which minimizes the total training time of *any* BP application. As found in Section 3.3, most real implementations of BP are based on a heuristic with a fixed assignment of processors to each involved degree of parallelism. However, the different kinds of applications – neural network structure and training set, require different mappings in order to obtain minimum training time. Thus, the performance of these fixed implementations are highly application dependent. In many publications on parallel BP mappings, performance is only measured for at most one real application. For instance, some select a non realistic network, usually large, that runs near optimally on the parallel machine, without reporting the performance of real applications. Also, when training set parallelism is used, the weight update frequency is usually not considered.

In this chapter a general mapping scheme, which includes all degrees of parallelism, is proposed. It is a heuristic for mapping feed-forward neural networks near optimally onto a 2D-torus MIMD computer. The assignment of cells to each degree of parallelism is done according to the given neural network and training set. That is, the method selects the appropriate mapping according to the given application. The method will run on any number of processors, available in todays parallel computers. However, the mapping's main advantage is for the case of a large number of processors. Small networks and training sets in particular can run more efficiently on larger parallel systems.

Today's technology allows more logic to be placed on a given circuit area. It is therefore expected that tomorrows technology will allow a larger number of processors on a single circuit and that parallel systems will consist a of larger number of processors. Massively parallel computers will be more available than the few presently available machines. These massively parallel systems can overcome the limit of sequential program execution by several orders of magnitude if the parallel algorithms can scale efficiently.

The main interest here is to minimize the *total* training time. Thus, it is impossible to omit the selection of a proper weight update interval when training set parallelism is used. Few of the published mapping methods address the issue of total training time, which is a topic of this chapter. To obtain high speedup on a highly parallel computer, it was shown in the previous chapter that the issue is to combine *multiple* degrees of parallelism. While the previous work was on fixed combinations of the degrees of parallelism, the new scheme allows arbitrary combinations of training set parallelism, pipelining and node parallelism.

In the Section 3.1.4, the dimension of each parallel degrees was listed. Some mapping strategies (e.g. [67, 160]) are based on assigning one processing element to each neuron. In this way, the utilization of the parallel computer depends on the neural network. This assignment requires a certain number of available processors and there is a possibility of processors being idle. A better approach is to base the mapping on the number of available processors and then ask how the different degrees should be combined to in the best way utilize the available processors. Thus, to combine the different parallel BP training dimensionalities in the way that minimizes the total training time. This is the underlying idea of the mapping that will be presented below. The name *application adaptable* mapping is given to this scheme. Parts of this work is published in [140, 142].

7.1 The Proposed Mapping Scheme

The proposed mapping aim at obtaining the best possible load balance between the processing cells for *any* neural network application. Hence, the number of processors assigned for each degree of parallelism is made to be changeable. The mapping will be presented first in a version without pipelining and secondly in a version including pipelining. Further, some improvements will be given.

The first part of the proposed mapping is shown in Figure 7.1 and consists of a set of possible system configurations. Each configuration combines node and training set parallelism. Within each dotted rectangle node parallelism is used. The box within the figure indicates the mapping of the elements for node parallelism. Neuron parallelism is represented within each row. Synapse parallelism is achieved in the vertical dimension. This scheme is based on an ordinary parallelization of matrix-vector multiplication, as described in Section 5.1.2. The lower part of the figure gives a selection of the number of training set partitions versus processors assigned to neuron parallelism. Correspondingly in the left-hand part of the figure, the choice in ratio between training set parallelism and synapse parallelism is shown. This illustrates the flexibility to adjust the parallel configuration to both the neural network size and training set size.

The mapping can be extended by including pipelining, i.e., make one processing element box in the figure into one O–H pair as shown in Figure 7.2. The processors marked **H** compute the hidden layer part, while the output layer part is computed by the processors

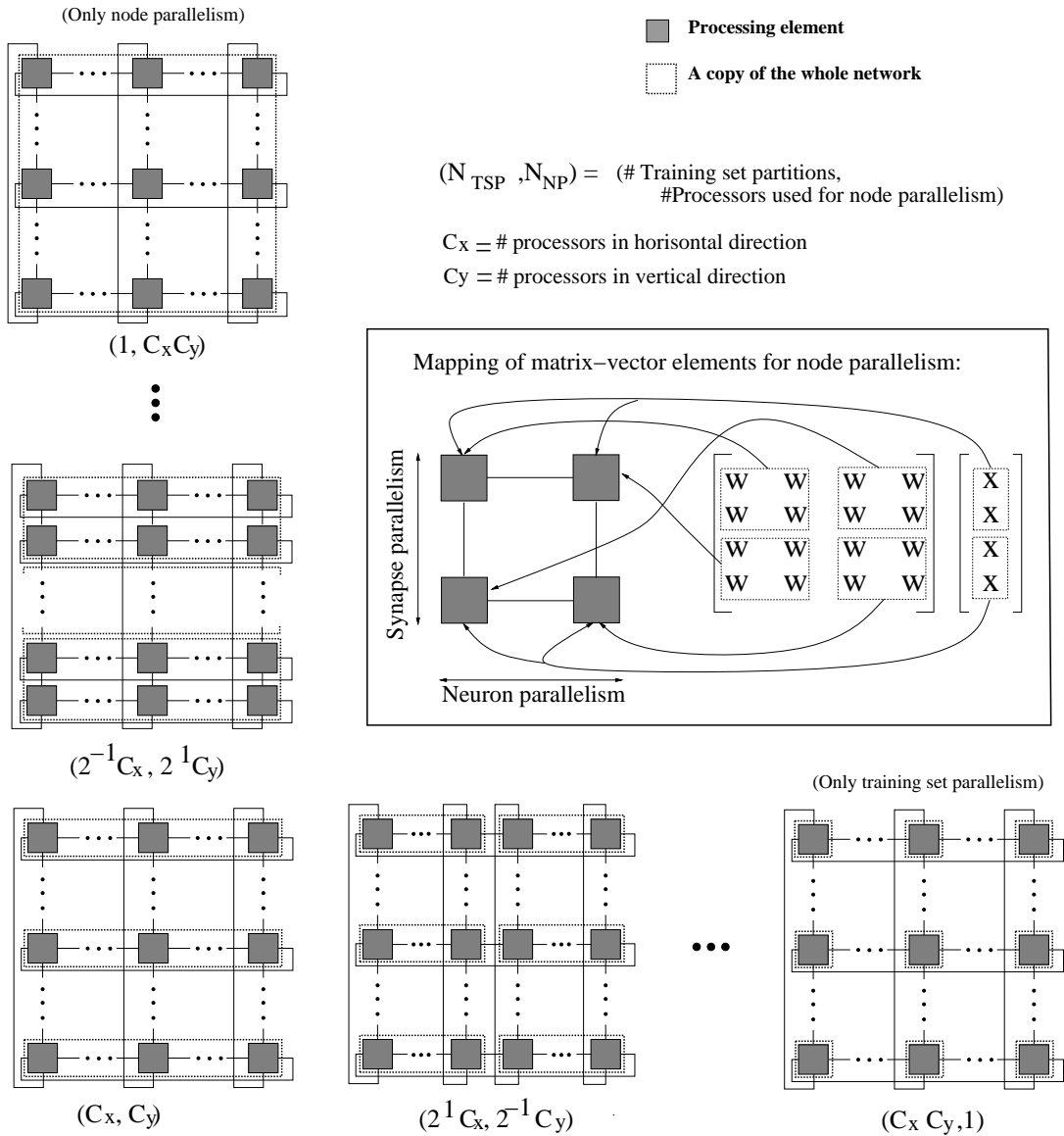


Figure 7.1: BP mappings with training set and node parallelism.

marked **O**. This scheme combines all degrees of BP parallelism except training session parallelism. The efficiency of including pipelining depends on the number of processors and the number of neurons in each layer. This was illustrated in Section 6.1.4.

Below each processor configuration in the figures, the number of training set partitions N_{TSP} and the number of processors for node parallelism N_{NP} are given. For Figure 7.1 the relation between them can be expressed by

$$N_{TSP} = \frac{C_x C_y}{N_{NP}} \tag{7.1}$$

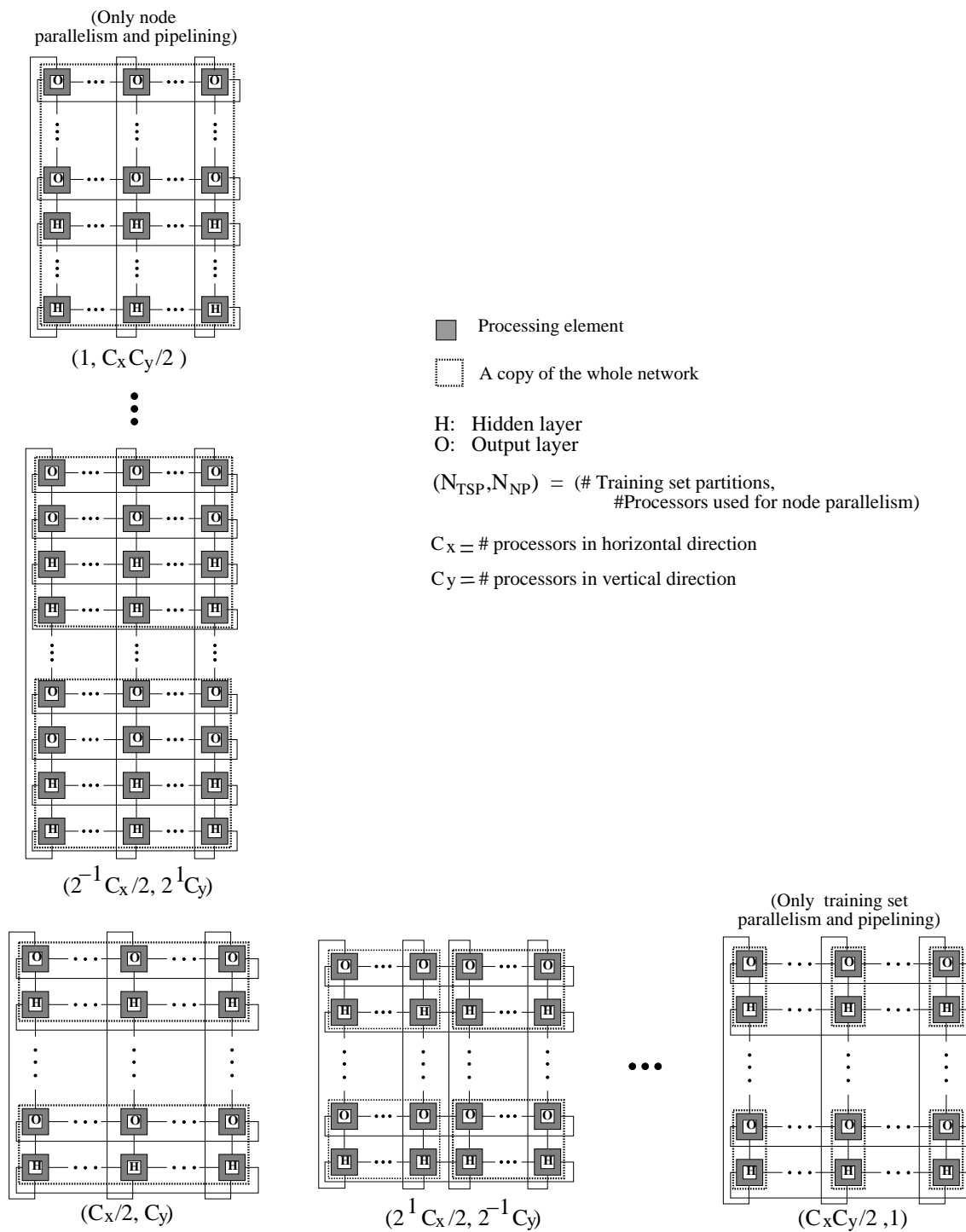


Figure 7.2: BP mappings including training set parallelism, pipelining and node parallelism. H indicate processors training the hidden layer, while O indicate processors training the output layer.

where C_x and C_y are the number of cells in the horizontal and vertical dimension respectively in the 2D-torus array. If pipelining is included we get

$$N_{TSP} = \frac{C_x C_y}{2N_{NP}} \quad (7.2)$$

For a given weight update interval, a larger number of training set partitions imply fewer training pattern computations on each processor between weight updates.

7.1.1 The Implemented Mapping

To be able to evaluate the benefit of a parallel back propagation algorithm that can vary the amount of each degree of parallelism, the lower part of Figure 7.1 – configuration (C_x, C_y) to $(C_x C_y, 1)$, was implemented on AP1000. Hence, the system can be configured to an even amount of neuron and training set parallelism, just training set parallelism, or an *intermediate* configuration. The latter implies that there will be conflicts in the communication.

The main training algorithm is quite similar to 2APC. However, the definition of some variables are different, as shown in the pseudo-code of Figure 7.3. The source code for this implementation is included in Appendix A.

The distribution of the training patterns is as indicated in Figure 7.4. The number of training vectors assigned to each network partition is calculated as shown in Figure 7.3 and stored in the variable MaxPattern.

For communication within each copy of the network (neuron parallelism) the calculation of the *id* of the communicating cell is shown in Figure 7.5. Sending is in the EAST direction and receiving is from the WEST direction.

The summing of the weights are as for 2APC in the vertical dimension, while propagation is used in the horizontal dimension – see Figure 7.6. The weight change matrices are pipelined in the row and summed to the local weight change matrices $N_{TSP} - 1$ times. This operation is done sequentially for the hidden and output layer.

7.1.2 Improvements of the Mapping

According to the survey of neural applications in Section 2.2, the neural network is often irregular in number of neurons in each layer. Further, the number of neurons in the output layer is often small. In this section, it is suggested how the mapping can be further improved with regard to these circumstances.

```

BackProagation_GeneralMap()
begin
   $n_h = \left\lfloor \frac{N_h}{N_{NP}} \right\rfloor$ ;
   $n_o = \left\lfloor \frac{N_o}{N_{NP}} \right\rfloor$ ;
   $\mu_c = \left\lfloor \frac{\mu}{N_{TSP}} \right\rfloor$ ; { Weight update interval }
  LookUp(MyCellId_x MyCellId_y);
  Read learning parameters from the host;
  Initialize weights;
  if (MyCellId_y  $\frac{C_x}{N_{NP}} + \frac{\text{MyCellId}_x}{N_{NP}}$ ) < (P mod  $N_{TSP}$ ) then
    MaxPattern =  $\left\lfloor \frac{P}{N_{TSP}} \right\rfloor + 1$ ;
  else
    MaxPattern =  $\left\lfloor \frac{P}{N_{TSP}} \right\rfloor$ ;
  Read training patterns from the host;
  while trainend = 0 do begin
    PSEUDOCODE AS FOR 2APC – given in Figure 5.7
  end;
end;

```

Figure 7.3: Parallel BP training algorithm running in each cell, combining training set parallelism and neuron parallelism.

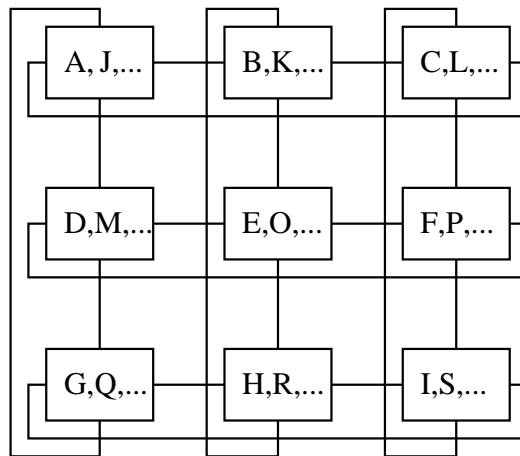


Figure 7.4: Distribution of training pattern A, B, C,...

```

if (MyCellIdx mod  $N_{NP}$ ) =  $N_{NP} - 1$  then
    SendTo = MyCellIdx -  $N_{NP} + 1$ ;
else
    SendTo = MyCellIdx + 1;
if (MyCellIdx mod  $N_{NP}$ ) = 0 then
    ReceiveFrom = MyCellIdx +  $N_{NP} - 1$ ;
else
    ReceiveFrom = MyCellIdx - 1;

```

Figure 7.5: The sending and receiving indices for neuron parallelism.

```

SendTo = MyCellIdx +  $N_{NP}$  { Send to EASTern cell }
if (SendTo  $\geq C_x$ ) then
    SendTo = SendTo -  $C_x$ 
ReceiveFrom = MyCellIdx -  $N_{NP}$  { Receive from WESTern cell }
if (ReceiveFrom < 0) then
    ReceiveFrom = ReceiveFrom +  $C_x$ 
 $\Delta \mathbf{W}_{acc} = \Delta \mathbf{W}$ ;
for  $t = 1$  to  $N_{TSP} - 1$  do begin
    SendWeights( $\Delta \mathbf{W}$ , SendTo);
    ReceiveWeights( $\Delta \mathbf{W}$ , ReceiveFrom);
     $\Delta \mathbf{W}_{acc} = \Delta \mathbf{W}_{acc} + \Delta \mathbf{W}$ ;
end;

```

Figure 7.6: The summing of weight change matrices in horizontal dimension.

Few Output Units

To train a network efficiently when it consists of only a few output layer units, the number of processors assigned to the output layer computation should be reduced.

Figure 7.7 shows a possible pipelined mapping. This scheme can be combined with the one presented in Figure 7.1 to include pipelining. The dotted rectangle in Figure 7.7 represents one dotted rectangle in Figure 7.1. 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. This scheme will be tested on RENNS, see Chapter 9.

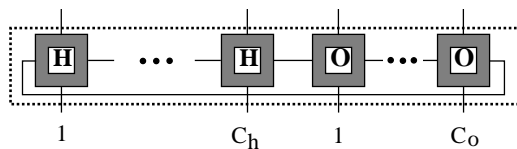


Figure 7.7: Pipelined mapping of BP with flexible assignment of cells to each layer.

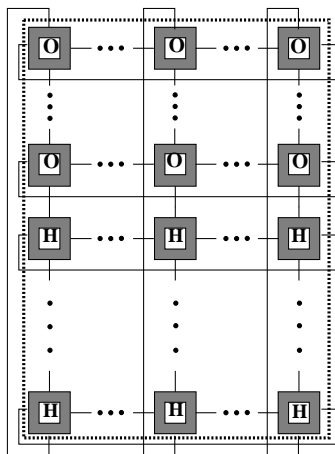


Figure 7.8: An extension of the mapping in Figure 7.2, where the number of cells used for synapse parallelism for each layer is assigned according to the computation load.

Improving Load Balance

The scheme in the left-hand part of Figure 7.2 can be improved on load balance, as depicted in Figure 7.8. A larger number of rows is assigned (synapse parallelism) to the hidden layer with respect to the output layer. This scheme maps *large* feed-forward networks efficiently, while *small* networks should be trained by the method presented in the previous section. Both methods make the implementation adjustable to the computation load in the different layers.

7.1.3 Training Session Parallel Scheme

The convergence of BP is highly dependent on the various parameters like learning rate, initial weights, and number of hidden units. Moreover, hierarchical network may need training of several different networks. Thus, a training session parallel scheme can be an interesting alternative, where networks train in parallel with different initial training pa-

rameters. In Figure 7.9 a combination of training session parallelism and neuron parallelism is proposed.

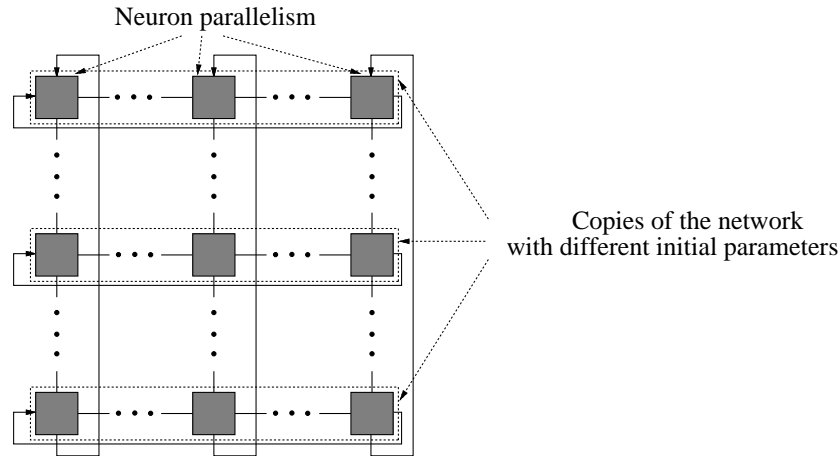


Figure 7.9: Training session parallelism and neuron parallelism combined.

This approach can train using *learning by pattern*. The method can be implemented as an iterative improvement scheme, where results from using different learning parameters are compared. The best ones are selected and combined for the next training session.

7.2 Heuristic for Selection of the Best Mapping

Before running a neural application, the mapping giving the shortest possible total training time has to be selected. First, the best initial training parameters – weights, learning rate, etc. have to be found. Then, for each mapping the total training time for a set of weight update intervals has to be predicted. The total training time is given by

$$T_{total}(\mu) = T_{1it}(\mu)N(\mu) \quad (7.3)$$

where $T_{1it}(\mu)$ is the time for one training iteration and $N(\mu)$ is the number of iterations needed for convergence, given a weight update interval μ . T_{1it} can be found either by estimation or by running each of the possible mappings for one training iteration. $N(\mu)$ can be estimated based on the error after a few iterations [142]. The mapping giving the smallest possible T_{total} is selected and the corresponding μ -value is used.

The heuristic is formulated in pseudo-code in Figure 7.10. It covers all steps to make a near optimal mapping of BP onto a 2D-torus computer. It evaluates all possible mappings and picks the best one for a set of μ -values. First, some initialization of the parameters has to be done. In the second step, the performance of the mappings in Figure 7.1 and 7.2 is estimated for some given weight update intervals. The *best* configuration for each

-
1. Find optimal initial weights, number of hidden neurons, learning rate and momentum value.
 2. For the given neural network, training set and computer architecture, estimate the performance for 1 iteration using the algorithm *Best_Map()*. Find the best mapping $M[\mu_i]$ for each of the weight update intervals $\mu_1, \mu_2, \dots, \mu_n$, and store the time for 1 iteration, $T_{1it}[\mu_i]$.

```

Best_Map()
begin
for  $\mu = \mu_1, \dots, \mu_n$  do
  Min_time_1it[ $\mu$ ] =  $\infty$ ;
for pipeline= 0 to 1 do { Pipeline not included/ included }
  for config= 0 to ( $\log_2 (C_x \cdot C_y) - \text{pipeline}$ ) do { Examine each mapping }
  begin
    nodepar=  $2^{\text{config}}$ ;
    trainpar=  $(C_x \cdot C_y) \text{ div } (\text{nodepar} \cdot 2^{\text{pipeline}})$ ; { From Eq. (7.1) and (7.2) }
    for  $\mu = \mu_1, \dots, \mu_n$  do begin
      time = Estimate_time_1_iteration(pipeline,nodepar,trainpar, $\mu$ );
      while (time can be reduced) do { Perf. impr. by reducing no of cells? }
        try reducing number of cells, update time;
      if (time < Min_time_1it[ $\mu$ ]) then
        update_M(pipeline,nodepar,trainpar, $\mu$ ); { Store new map. as the best }
    end;
  end;
end;
end;

```

3. Estimate the number of iterations needed, $N[\mu_i]$, for $\mu_1, \mu_2, \dots, \mu_n$ equal to those in step 2.
4. Pick μ_m so that

$$T_{total}[\mu_m] = T_{1it}[\mu_m]N[\mu_m] \quad m = 1, \dots, n$$

becomes minimized.

5. Configure the system by $M[\mu_m]$ and update the weights for every μ_m pattern.
-

Figure 7.10: Heuristic in pseudo-code for selecting the best parallel mapping of BP onto a 2D-torus computer.

given weight update interval is found. In step 3, the number of training iterations needed is estimated for the given weight update intervals. Combining the results of steps 2 and 3 gives us in step 4 the best weight update interval and the corresponding best configuration. That is, the total training time is minimized. The heuristic is described in more detail in the next section.

The heuristic is based on the one given in [105] for selecting the best weight update interval according to the total training time. However, the mapping used is a fixed implementation based on training set parallelism. Also, experimental rules on convergence rates are deduced from a small application using synthetic data.

7.2.1 Selection of Back Propagation Parameters

Several papers have shown that back propagation parameter setting is of major importance to improve the convergence rate. The number of hidden neurons, initial weight values, the learning rate (η), a momentum term (α) and the weight update interval all influence the convergence rate. Both Sato [114] and Higashino [50] found from experiments that the number of training iterations N is proportional to $(1 - \alpha)/\eta$ within the region in which a stable convergence is obtained. Thus, it is desirable that η takes as large a value as possible and α is as near to 1 as possible. The convergence rate can be accelerated by adaptively adjusting η and α during training as shown in [11, 15, 148].

One of the problems of back propagation is that it has a tendency to get trapped in the local minima. Different approaches have been suggested to overcome this problem. One is to use Genetic Algorithms (GA), which search a large parameter space. However, this means longer training time. GA has been used in experiments for improving BP in several ways:

- Avoid local minima by adjusting the weights by GA, either initially [63] or during training [79].
- Adjust learning rate and momentum term during training [79].
- Find the optimal neuron interconnections by adding and deleting connections between neurons in the same and different layers [103].

Since the topic of initial parameters is covered in many papers, it is not included in this work. Still, the learning parameters are very important for the convergence rate. For each applications used in this research, the best parameters have been searched for.

7.2.2 Estimation of Execution Time

The execution time can be estimated based on the count of floating point instructions and time for communication. A similar approach is used in [161] to estimate neural network performance on CM-2. The accuracy of the method is dependent on various aspects as idle time and cache misses. However if the accuracy is acceptable, the method of estimation is to be preferred instead of measuring real execution time, because less computation is needed.

The *Best_Map()* algorithm in Figure 7.10 examines all the possible mappings described. First, the performance using all cells available is estimated. For large systems, using fewer cells may give better performance. Thus, a test on this issue has been included. The *Estimate_time_1_iteration()*-function will be explained below.

Timing Data

In Table 7.1, the floating point performance and communication timing of the AP1000 is presented [56]. The given timing is for basic operations like adding floating point variables. In a real program, overhead will be added for additional instructions for example for addressing. Thus, when estimating the performance based on the number of floating point operations this overhead is omitted. The compiler is assumed to generate code that optimally utilizes the arithmetic units in the processor. However, the result can still be of interest to estimate the theoretically maximum obtainable performance.

In the result section (Section 8.1.2), average timing for floating point operations measured in the real program will be used for estimation. In addition to predicting execution time, the estimation method makes it possible to determine the optimal number of processors. This is probably much larger than the number of PEs in any of the available systems.

t_{add}	442	Time to add floats sequentially.
t_{mul}	442	Time to multiply floats sequentially.
$t_{mul+add}$	646	Time for alternating multiply and add of floats.
$t_{load/store}$	880	Time for load/store when cache miss occurs.
$t_{sigmoid}$	7500	Time to compute $1/(1 + e^{-x})$.
$t_{comm}(B)$	$16700 + 175B$	Communication time for sending B bytes between cells.
$t_g(C)$	$6000 + 16000 \log_2(C)$	Time for global reduction function (e.g. x_fsum) for C cells.
$t_{cont}(B, n)$	$T_{cont}(B, n)$	Network contention; Communication time when n messages of size B are sent between cells. See [56].

Table 7.1: Timing on AP1000 (ns) including loading from and storing to cache [56].

Time Estimation

Below, the time used for one iteration will be estimated for the implementation described in Section 7.1.1. First the basic time units are computed.

Time to compute output of the hidden layer

$$T_{Fh} = n_h(N_i t_{mul+add} + t_{sigmoid}) \quad (7.4)$$

Time to compute output of the output layer

$$T_{Fo} = n_h n_o t_{mul+add} + (N_{NP} - 1)(t_{comm}(n_h) + n_h n_o t_{mul+add}) + n_o t_{sigmoid} \quad (7.5)$$

Time to compute the output layer error

$$T_{Boe} = n_o(t_{add} + t_{mul+add} + t_{mul} + t_{mul+add}) + (N_{NP} - 1)(t_{comm}(1) + t_{add}) \quad (7.6)$$

Time to compute the hidden layer error

$$T_{Bhe} = N_h n_o t_{mul+add} + (N_{NP} - 1)(t_{comm}(n_h) + n_h t_{add}) + n_h(t_{mul+add} + t_{mul}) \quad (7.7)$$

Time to accumulate the output layer weight change

$$T_{Woa} = N_h n_o t_{mul+add} \quad (7.8)$$

Time to accumulate the hidden layer weight change

$$T_{Wha} = N_i n_h t_{mul+add} \quad (7.9)$$

Time to update the hidden and output layer weights

$$T_{Wupd} = \log_2(C_y)(\max\{t_{comm}(N_i n_h), t_{comm}(N_h n_o)\}) \quad (7.10)$$

$$+ \max\{N_i n_h t_{add}, N_h n_o t_{add}\} + t_{comm}(N_i n_h) + t_{comm}(N_h n_o) \quad (7.11)$$

$$+ k\left(\frac{C_x}{N_{NP}} - 1\right)(t_{comm}(N_i n_h) + N_i n_h t_{add} + t_{comm}(N_h n_o) + N_h n_o t_{add}) \quad (7.12)$$

$$+ 3N_i n_h t_{mul+add} + 3N_h n_o t_{mul+add} \quad (7.13)$$

Expressions 7.10 and 7.11 represent the vertical summing of the cell array, followed by the horizontal summing in Expression 7.12. The constant k compensates for the additional overhead, due to communication conflicts (contention). Its value will be given in the next chapter. The final Expression 7.13 adds the weight update time. Except for T_{Wupd} , the units are computed for *one* training pattern.

Based on these units it is possible to get an estimate of the time needed for one whole training iteration

$$T_{it}(\mu) = P_c(T_{Fh} + T_{Fo} + T_{Boe} + T_{Bhe} + T_{Wha} + T_{Woa}) + \left\lceil \frac{P}{\mu} \right\rceil T_{Wupd}$$

where P is the total number of training patterns and P_c the number of patterns in each cell.

7.2.3 Convergence Estimation

One of the major problems of BP training is to decrease the number of iterations needed to train an application. Much work is published on this topic. However, estimation for training time is seldom reported.

At least two approaches are possible. First, to use existing formulas [105, 114] for estimation of $N(\mu)$. These are based on specific applications and compute the number of iterations needed based on the learning rate. Since these have been tested only for a few application, they may be inaccurate in general. A bound on the probabilistic rate of convergence of feed-forward networks trained to estimate a smooth function is detailed in [77].

A second approach is outlined in [142] and explained in the following chapter. It is based on running the algorithm for a few iterations – for various weight update intervals, and based on the initial error, the total number of iterations $N(\mu)$ used for convergence is estimated. A formula is derived for the convergence of NETtalk, based on the convergence after 25 initial iterations. By estimating the convergence based on the initial error, any techniques for improving the convergence rate is reflected in the estimation.

7.2.4 Modeling the Memory Usage

In this section, general expressions for the memory needed for the implemented mapping is derived. The memory allocated in each cell for the implementation described in Section 7.1.1 is given by

Weight matrices:	$M_w = 4 \cdot 4(N_i n_h + N_h n_o) = 16(N_i \lceil \frac{N_h}{N_{NP}} \rceil + N_h \lceil \frac{N_o}{N_{NP}} \rceil)$ byte
Training patterns:	$M_p = 4P_c(N_i + n_o) = 4\lceil \frac{P}{N_{TSP}} \rceil(N_i + \lceil \frac{N_o}{N_{NP}} \rceil)$ byte
Array and data variables:	$M_d < 10$ Kbyte
Program memory:	$M_p = 250$ Kbyte

The factor of 4 in the expressions is due to floating point variables requiring 4 bytes of memory. The expressions are given as maximum values. In the case of non-even distribution of the neurons to cells some cells may allocate less memory. These expressions can be used to show how memory requirement for each cell is reduced for an increasing number of cells.

7.3 Summary

This chapter has described a general mapping scheme, which includes all the inherent parallel degrees of the backpropagation algorithm. The mapping is flexible and combines the different degrees so that the total training time can be minimized.

To select the best implementation, a heuristic is given. This estimates the training time for the possible configurations. These estimates are used together with convergence estimates for different weight update intervals to select the best configuration and weight update interval.

In the next chapter, a subset of the mapping scheme will be tested. Moreover, the estimation of execution time and convergence of real neural applications will be studied.

Chapter 8

Results on the General and Flexible BP Mapping

In this chapter, one part of the mapping proposed in the previous chapter is evaluated through experiments on AP1000. The implementation is described in Section 7.1.1. The applications NETtalk, sonar classification, speech recognition, and image compression are used in the experiments. Training speed for different network sizes and processor configurations are given. For the two first applications the convergence for different weight update frequencies are studied. The mapping that minimizes the total training time for the parallel program is determined for NETtalk. Results from estimation of execution time will also be given. Due to the limit of time, it was not possible to undertake more experiments on estimation and convergence than reported. The results presented in this chapter are also included in tables in Appendix B.2.

8.1 NETtalk - Results and Discussion

In this section the NETtalk neural network application is used in the experiments. The section consists of four parts. First, the training speed is evaluated from measurements on the AP1000. Secondly, the performance is estimated, and thirdly, the convergence is tested and analyzed. Finally, the minimum total training time is determined. The training set consists of 1000 of the most common English words (a total of 5438 characters). Experiments have been conducted for $\mu = 63, 259, 494, 906, 1360, 2719, 5438$. The best performance reported by Sejnowski and Rosenberg [118] were by using 120 hidden units, and as such, is chosen for the experiments in this section. Bias was implemented using one of the idle input units (to be used for punctuation and word boundaries for continuous speech) and one of the 120 hidden units.

8.1.1 Training Speed

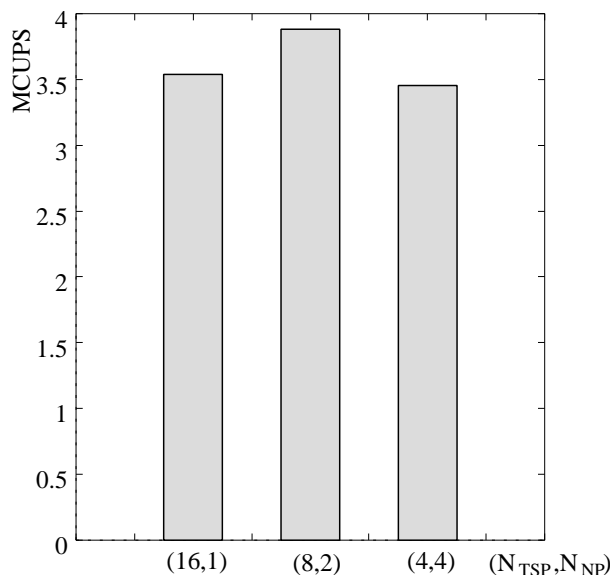


Figure 8.1: MCUPS performance for different combinations of neuron and training set parallelism running the NETtalk application with 120 hidden neurons. 4 x 4 cell configuration for $\mu = 906$.

Figure 8.1 illustrates the performance of the algorithm running the NETtalk application on 16 processors. The weights are updated after 906 patterns i.e. $\mu = 906$. Performance is measured in MCUPS. The configuration for each bar is indicated by (N_{TSP}, N_{NP}) – as explained in Section 7.1. The fastest configuration is *not* the even combination – 2APC, given in the rightmost column, but $(N_{TSP}, N_{NP}) = (8, 2)$. This shows that using 8 training sub-sets are better than using 4, even though network contention occurs. One reason for this is that mapping the 26 output neurons to 4 processors gives rise to an uneven load balance, while this is not the case for 2 processors. However, the difference in performance is not significant in comparison to the other configurations. That is, on a small number of processors the computation grains are quite large. Therefore, the way BP is parallelized is not that critical on a small system.

When the number of processors increases, as in Figure 8.2, the difference in performance becomes larger. The figure illustrates the comparison between the training speed for 4 different weight update intervals on the 64 cell system. The combined solutions achieve better performance than using only training set parallelism – $N_{NP} = 1$. However, for large weight update intervals e.g. $\mu = 906$ and $\mu = 5438$, smaller N_{NP} values (and thus larger N_{TSP} values) result in performance improvements. This is as expected since infrequent weight updates reduce the total weight update time for an iteration. Thus, a larger number of training set partitions can train efficiently. Therefore, the fastest solution is for $\mu = 906$, as in Figure 8.1, not the even combination $N_{NP} = 8$, but $N_{NP} = 4$. When the weights are

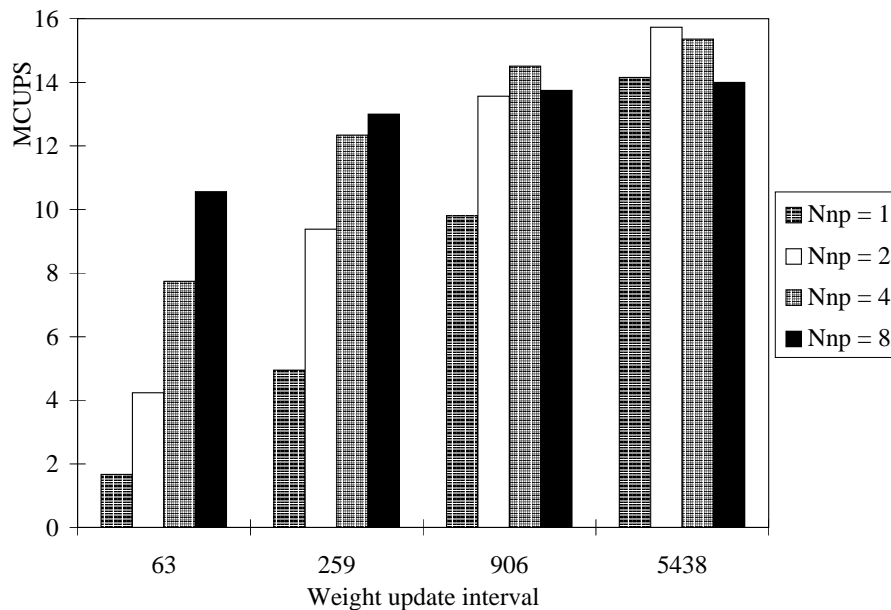


Figure 8.2: MCUPS performance for different combinations of neuron and training set parallelism on a 8 x 8 cell configuration. $N_{NP} = 1, 2, 4,$ and 8 .

more frequently updated, a larger N_{NP} -value will be more beneficial, as seen in the bar chart.

Results for the 256 cell system are displayed in Figure 8.3. A similar performance as in the 64 cell system is seen, but the weight update interval becomes more significant. Thus, as the number of processors increases, the difference in the fastest training speed for frequent and infrequent weight updates increases. The best performance for $\mu = 5438$ is more than twice the performance for $\mu = 63$. For the largest available system consisting of 512 processing elements – see Figure 8.4, this difference is almost three. These results are understandable, since the number of training set partitions is large on a system of many processors. Thus, less training patterns are trained between weight updates. Further, more processors are assigned to each partition leading to shorter computation time for each training pattern. Hence, the impact of the weight update becomes more dominant.

A summary of the results above are plotted in Figure 8.5, where the maximum performance for 64, 256, and 512 processors is given. The performance on one cell is less than 1 MCUPS, thus the performance scaling is reasonable. This is especially true for large weight update intervals.

In Figure 8.6, the performance of the best configuration is compared to the system with the largest N_{NP} value – equal to the 2APC-implementation. The measure of the speedup between the two configurations is given, i.e. the ratio between CUPS performances. For

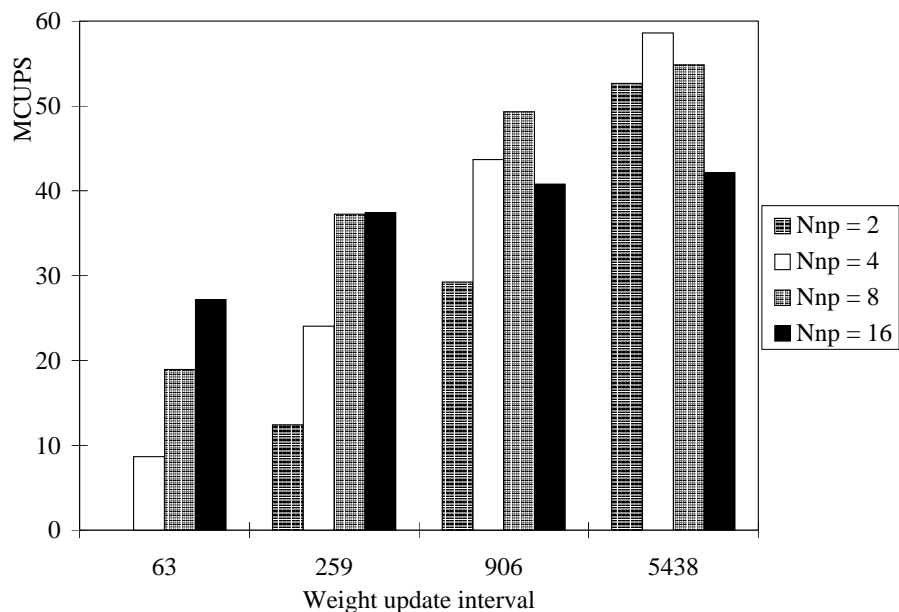


Figure 8.3: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 cell configuration. $N_{NP} = 2, 4, 8,$ and 16.

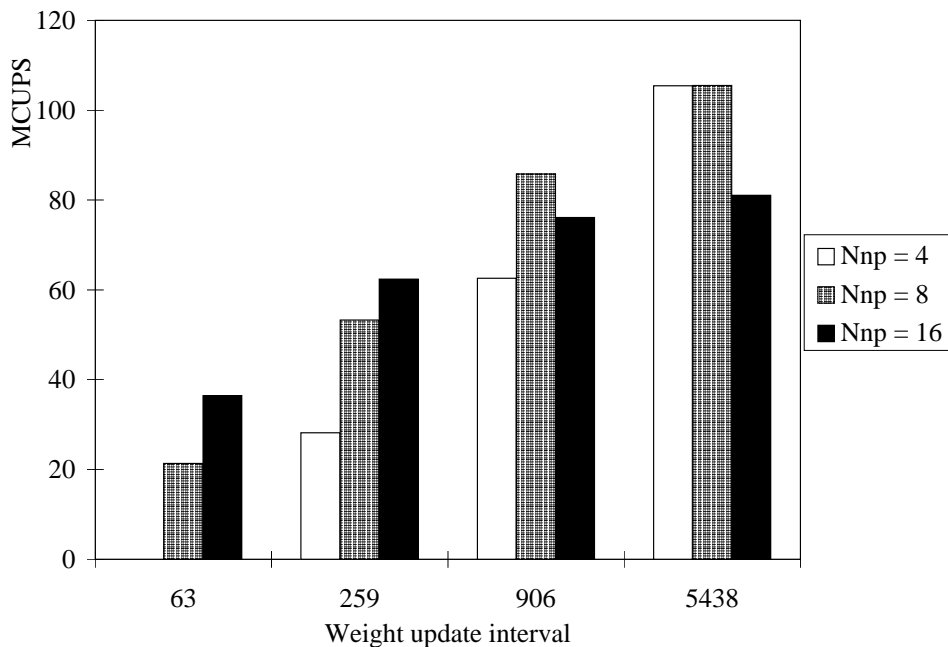


Figure 8.4: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 cell configuration. $N_{NP} = 4, 8,$ and 16.

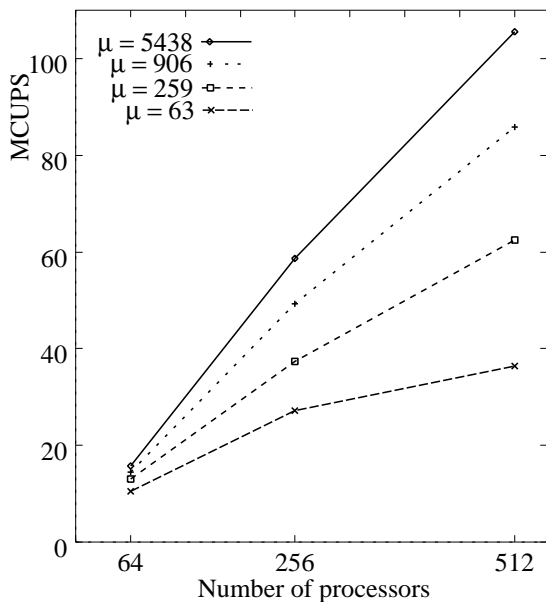


Figure 8.5: Training speed for NETtalk network plotted as a function of the number of processors.

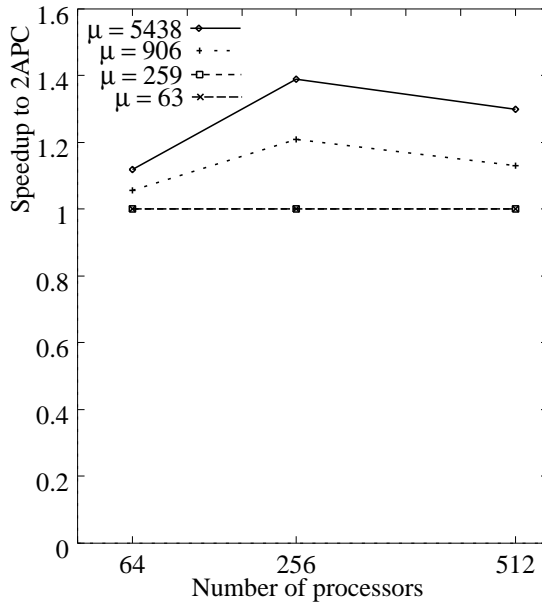


Figure 8.6: Speedup of the best configuration compared to the implementation with the largest N_{NP} value, i.e. the 2APC configuration.

$\mu = 63$ and $\mu = 259$ the 2APC configuration is the fastest, and thus there is no speedup. For larger μ the speedup reaches 1.4 in the best case. This means that the performance of the best mapping is 40% faster than that of the 2APC mapping.

8.1.2 Estimation of the Performance

The accuracy of the estimation model presented in Section 7.2.2 is investigated in this section. The training performance of the flexible mapping is tested using the NETtalk application with 120 hidden units.

The results will state if the estimation method is accurate enough to predict the execution time – compared to using real measurements, for selecting the best mapping for a given neural network application. If the estimation method is accurate, it can also be used to predict the possible performance for larger non-existing configurations of the parallel machine.

To measure the average time of various operations, several tests were undertaken in the implemented programs. The average timing used for the following estimations are given in Table 8.1.

Early experiments revealed underestimated weight update time, and therefore, the constant

t_{add}	1280	Time to add floats sequentially.
t_{mul}	1400	Time to multiply floats sequentially.
$t_{mul+add}$	1770	Time for alternating multiply and add of floats.
$t_{sigmoid}$	7500	Time to compute $1/(1 + e^{-x})$.
$t_{comm}(B)$	$60000 + 250B$	Communication time for sending B bytes between cells.

Table 8.1: Time measured for operations in implemented programs on AP1000 (*ns*).

k (in Expression 7.12) was introduced and set equal to 1.5 in the following experiments. A cache hit prediction has not been implemented in the model as the cache performance analysis on AP1000 showed that the average hit-rate was about $99\% \pm 0.5\%$.

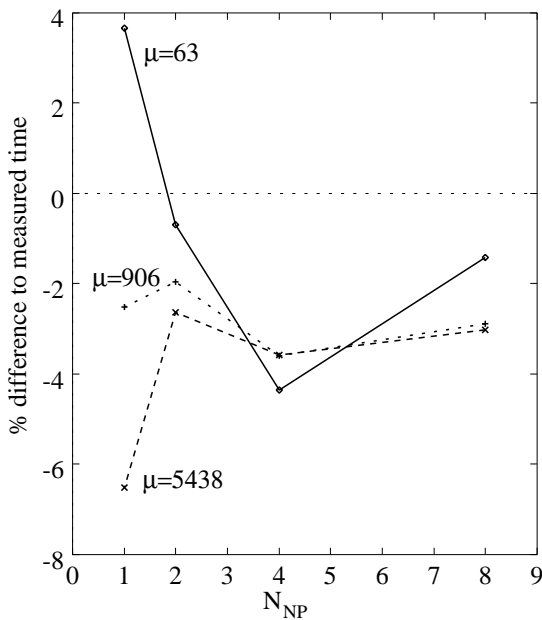


Figure 8.7: Deviation of estimate against measured time for the 64 cell configuration for different weight update intervals.

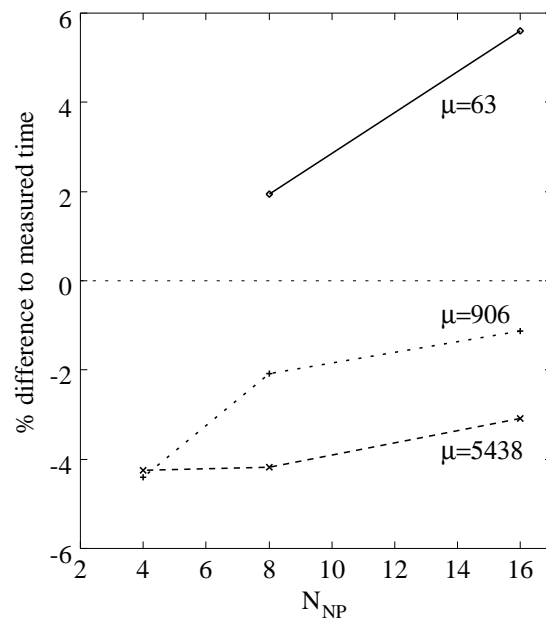


Figure 8.8: Deviation of estimate against measured time for the 512 cell configuration for different weight update intervals.

Figure 8.7 and Figure 8.8 show the error profile for the execution time estimates for 64 PEs and 512 PEs, respectively. The estimates are seen to be quite accurate, with an average error less than 5%. For 512 cells the estimated time for the small weight update interval $\mu = 63$ is greater than that measured. For $\mu = 906$ and $\mu = 5438$, on the other hand, the estimates are less than the measured values. On 64 cells, μ equal to 64 is also underestimated for some configurations. This is due to more training patterns – compared on 512 cells, computed between weight updates. Hence, the impact of weight update is less for the small system. Therefore, to improve the model, the training time without weight

updating must be increased and the weight update time decreased.

The time estimates can be used to calculate the performance in MCUPS. Figure 8.9 compares the estimated and measured performance for the fastest configurations for 64, 256, and 512 PEs and $\mu = 906$. The N_{NP} values for the fastest mappings are listed in Table 8.2. The estimated performance is slightly higher than the measured performance. Considering the simplicity of the estimation model i.e. only based on floating point operation count and a simple expression for communication time, the estimation method is fairly good. With this level of accuracy, it is interesting to estimate the performance for larger systems.

Number of processors	N_{NP}
64	4
256	8
512	8

Table 8.2: The number of N_{NP} values used in Figure 8.9.

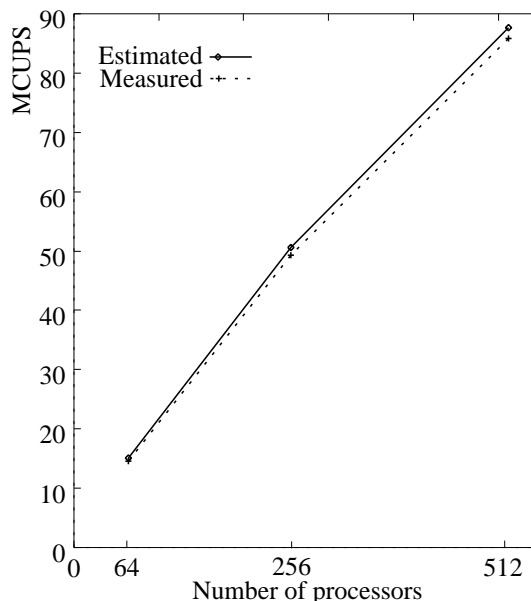


Figure 8.9: Comparing the fastest configuration for 64, 256 and 512 processing elements to estimated values for a weight update interval equal to 906.

Figure 8.10 shows the estimated training performance for systems larger than 512 PEs. The configurations with the highest performance are used in the figure. The weight update interval highly influences the obtainable performance. The real difference in performance for the given weight update intervals is probably smaller, according to the estimation error in Figure 8.8. For $\mu = 63$ the MCUPS value is probably underestimated, while for the other it is probably overestimated.

The performance for even larger systems are shown in Figure 8.11. *Learning by epoch* gives a maximum training speed of 3.6 GCUPS, while the weight update interval of $\mu = 906$ shows a maximum of 1.3 GCUPS. Both are for 500K cells. Thus, the performance could be at least ten times faster than that for 512 cells if there were no limitations on the number of cells. This shows the properties of the mapping on a massively parallel computer.

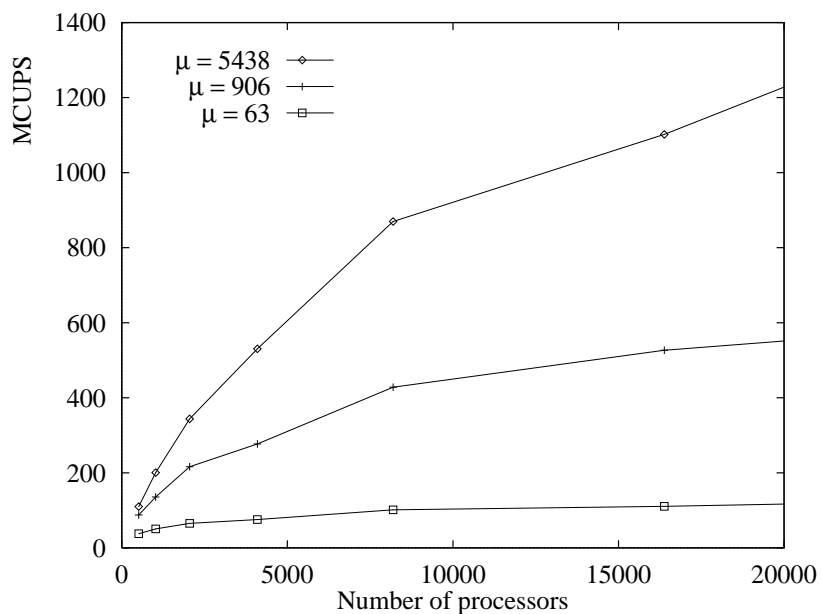


Figure 8.10: Estimated training performance for the NETtalk application for three different weight update intervals, μ .

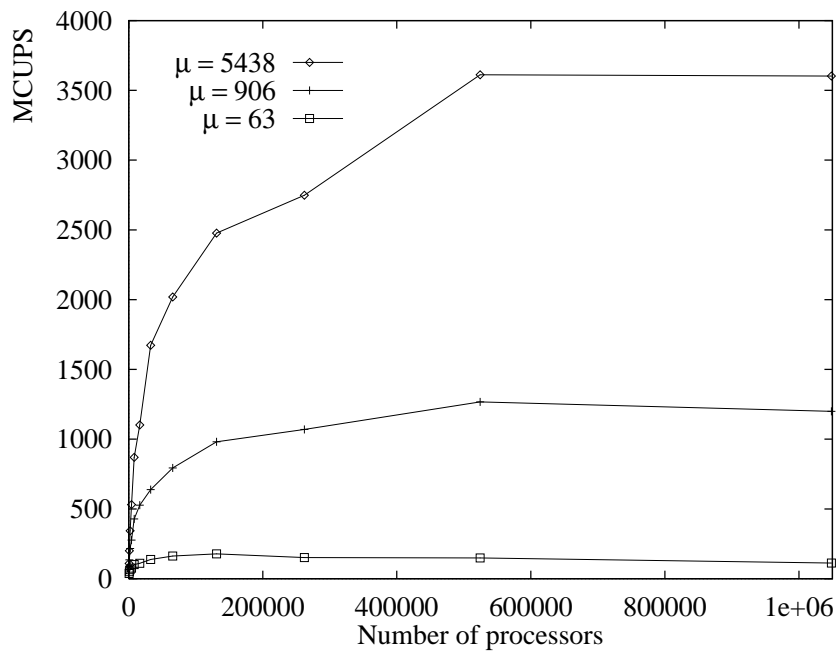


Figure 8.11: Estimated training performance for NETtalk application for three different weight update intervals, μ .

The estimations do not take into account the additional overhead incurred when *few* connections and training vectors are computed by a cell. Thus, the estimates are probably slightly too high for large systems.

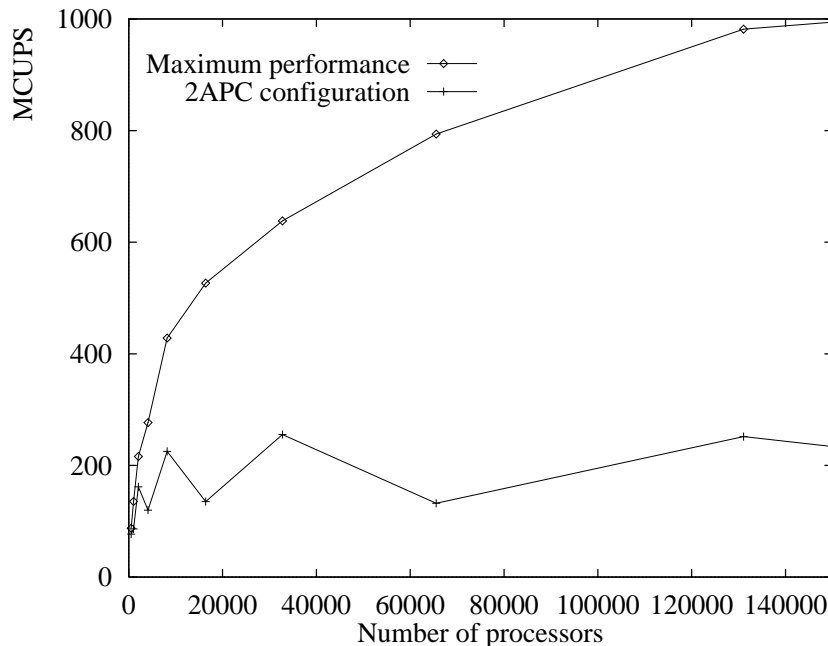


Figure 8.12: Comparison of the maximum performance of the flexible configuration to the 2APC configuration, for $\mu = 906$.

Figure 8.12 presents a performance comparison of the fixed configuration 2APC (given by $N_{NP} = C_x$, i.e. no training set parallelism in the vertical dimension) with the fastest possible training speed. As the number of PEs increases the gap between the two increases. This emphasizes the importance of *application adaptable* mapping on large parallel systems.

Using the technical data for AP1000 given in Section 7.2.2 we can estimate the maximum theoretical training speed of the available systems – as shown in Figure 8.13. For 64 cells the measured performance is 36.4% of the maximum performance, where $\mu = 906$. The difference in performance on 512 cells is 38.5%. Therefore, the implementation could, in the best case, be optimized by a factor of slightly less than three. However, it is anticipated this would require some assembly programming in the code.

This simulation has been tuned to the NETtalk application. A few experiments were undertaken with an image compression and a speech recognition networks. To obtain accurate estimates, k had to be set to a specific value for each application. Thus for making a general estimation method, an expression for computing k ought to be developed. This should be based on the network size, since its purpose is to compensate for added overhead (communication conflicts) when large weight change matrices are to be added.

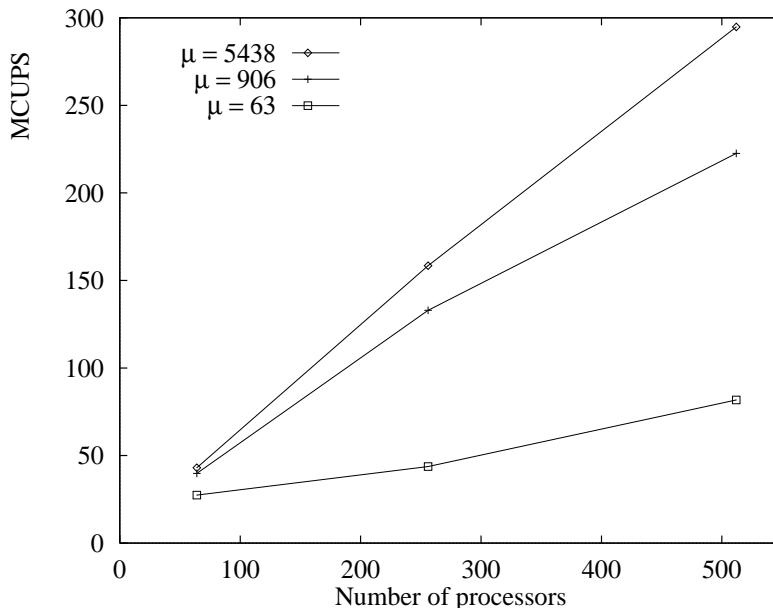


Figure 8.13: Maximum possible training speed on AP1000.

In conclusion, the experiments are promising for using this estimation technique in picking the best parallel configuration for a neural network application.

8.1.3 The Convergence of NETtalk Learning

To investigate the convergence rate for different weight update intervals, 1000 of the most common English words (a total of 5438 characters) were used. The error function for a training pattern p is equal to Equation 2.4 and is given by

$$E_p(n) = \frac{1}{2} \sum_k (d_{p,k} - y_{p,o,k})^2$$

where n is the training iteration number, $d_{p,k}$ is the target output and $y_{p,o,k}$ is the output layer output. Rumelhart et al. use the values 0.1 and 0.9 as target values, since the extreme values of 0 or 1 can never be reached [112]. If the error is 0.1 per output unit the pattern error becomes

$$E_p(n) = \frac{1}{2} \sum_{k=1}^{26} (0.1)^2 = 0.13$$

and this value has been used as a threshold for determining if a pattern is trained or not. Weight change values were accumulated for a pattern p having $E_p(n) > 0.13$, while a pattern is trained when $E_p(n) \leq 0.13$. This is a slightly different error measure than reported by Sejnowski and Rosenberg, who measured the error at each individual output

neuron. They used a “best guess” which compared the output values to a subset of the training set (52 target vectors). If the vector that had the smallest output error was equal to the training vector then the vector was considered correctly trained. Both methods decrease the sum of square error function

$$E(n) = \sum_{p=1}^{5438} \sum_{k=1}^{26} (d_{p,k} - y_{p,o,k})^2 \quad (8.1)$$

The error measure used in this research avoids over-training without a large loss in the load balance of the computation. Since the purpose of these experiments has been to study the relation between weight update interval and convergence and not the performance of the application, a separate test set has not been used. Testing convergence on the training set requires less time than using a test set. Thus, more tests for each weight update interval could in this way be undertaken.

For each weight update interval investigated, the best learning rate η was searched for. Between 10 and 15 runs of each weight update interval were performed. These experiments in total required several hundred CPU hours. For a small μ , the best learning rate was equal to 1.5. It had to be decreased for larger values of μ . For larger μ , the convergence was more sensitive to the selection of η . The best value for the smoothing term α was, after some initial experiments, found to be 0.9. This value was used for all the experiments. Due to this smoothing term, it was possible to keep a fairly large learning rate value even with infrequent weight updates. Bias was also used in the experiments.

Experiments with adaptive learning rate during training and other similar methods to speed up the convergence are not included. This is mainly because it would have required a *much* larger number of experiments to make a fair comparison of the convergence for different weight update intervals.

Figure 8.14 shows the results of the convergence as a percentage (called $E\%$) of the characters that have not been trained. Each curve represents one weight update interval. Almost without exception, a larger μ implies a slower reduction in number of non-trained patterns. The percentage of characters not trained went down and stabilized slightly below 5%. This is due to the representation limitation of the network as mentioned in [118]. Therefore, a network is here regarded as converged, when $E\% < 5\%$.

Figure 8.15 shows the optimal learning rate – the value resulting in the smallest number of iterations for convergence, for each of the μ used in the experiments. Based on these values the following rule is established

$$\eta(\mu) = 6.0\mu^{-0.5} \quad (8.2)$$

This is comparable to the rule derived by Paugam-Moisy in [105]: $\eta(\mu) = k_1\mu^\alpha$ for the application of classifying patterns into three classes – see Section 2.1.7.

The number of iterations required for convergence is given in Figure 8.16. Due to the fairly linear distribution of the points, the least squares method is used to establish the

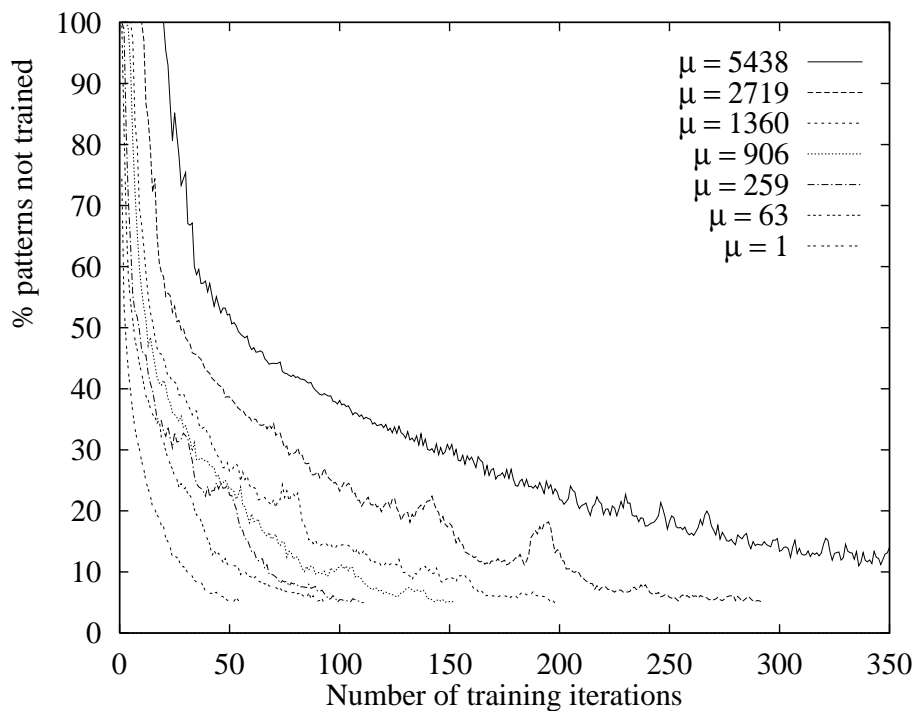


Figure 8.14: Percentage of characters that are not trained for various weight update intervals.

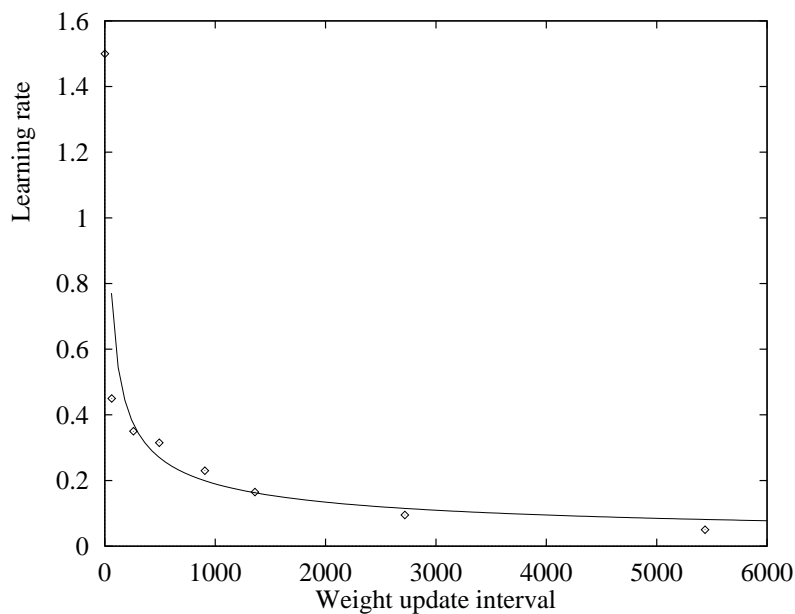


Figure 8.15: The best learning rate (η) for each investigated weight update interval. The curve plots the derived rule.

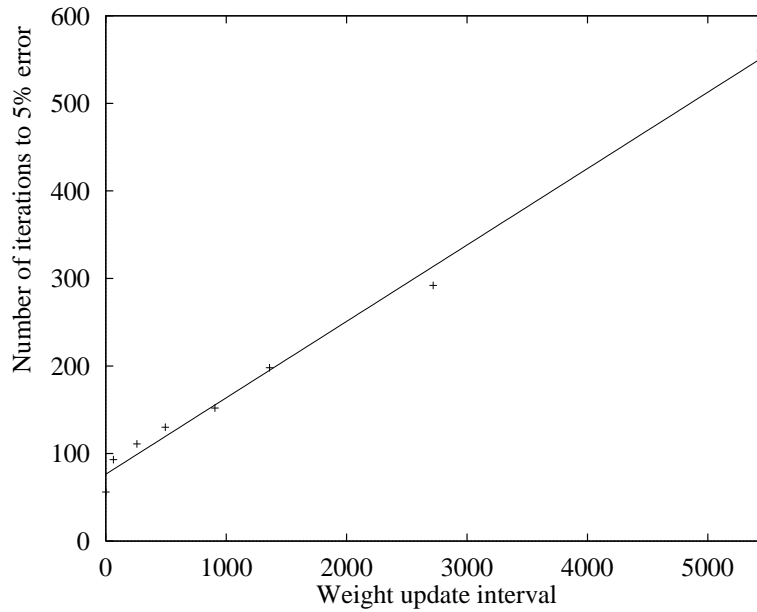


Figure 8.16: Number of iterations needed to obtain convergence with $E_{\%} < 5\%$ as the stopping criteria.

experimental rule

$$N(\mu) = \frac{\mu}{11.5} + 76.4 \quad (8.3)$$

This linear relation between number of training iterations and the weight update interval was also found by Paugam-Moisy. The linear distribution of points tells us that for predicting $N(\mu)$ it is sufficient to estimate the total number of iterations for a few (minimum 2) weight update intervals. These can then be used to find a linear formula for $N(\mu)$.

As mentioned in Section 7.2.1 several researchers have found that N is proportional to $(1 - \alpha)/\eta$. Figure 8.17 plots this expression for the learning rates in Figure 8.15. The points are linearly distributed along the least squares fit line. By scaling by the number of iteration for $\mu = 906$ we get

$$N(\mu) = \frac{\mu}{8.43} + 51.2 \quad (8.4)$$

This gives a good estimation for $\mu = 1$ with a number of 5 iterations less than measured. However, the estimates are less precise for larger weight update intervals, e.g. for $\mu = 5438$ a number of 85 iterations more than measured is estimated.

In the following it is shown that the total number of iterations needed can be estimated based on the error after a small number of training iterations. By looking at the curves in Figure 8.14, we see that they are of similar form, but have different curvature. To find a curve that estimates each error curve, logarithmic regression can be used to find an

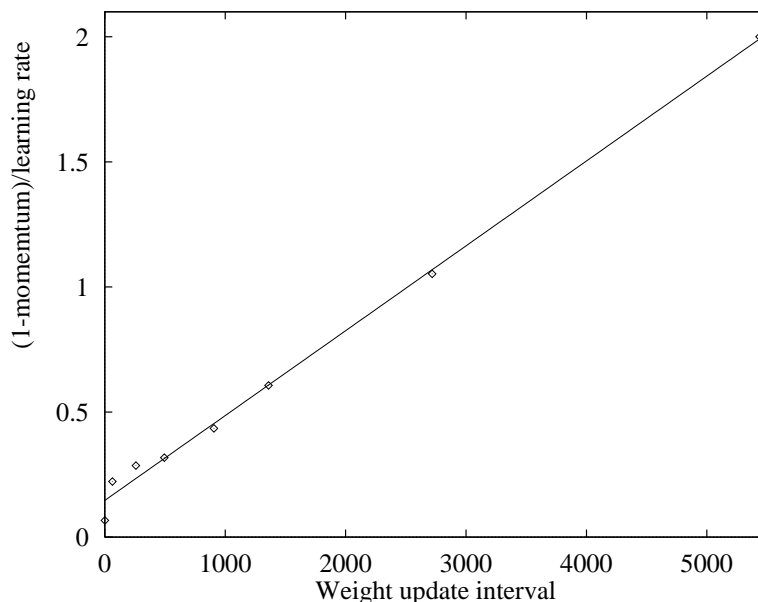


Figure 8.17: Plotting of $(1 - \alpha)/\eta$.

expression for each μ_i of the form

$$E_{\%}(n) = k_1 n^{k_2} \quad (8.5)$$

where n is the iteration index and k_1 and k_2 are parameters to be estimated. This requires that the points $(\log n_i, \log E_{\%}(n_i)) \forall i$ are situated on a straight line. Figure 8.18 shows a logarithmic plot of Figure 8.14.

Above 20 iterations – see on the right-hand side of the vertical line in the figure, the curves are becoming more linear for increasing iterations. Thus, regression should be used on this linear part. Several different regression analyses were made to search for a curve describing the error convergence. Equation 8.5 was extended

$$E_{\%}(n) = k_1(n + k_3)^{k_2} \quad (8.6)$$

by k_3 , which may be called the offset of the error curve, equal to twice¹ the number of iterations from start of training until $E_{\%} \leq 95\%$. Equation 8.6 was solved for each μ for the initial point $(n = 25, E_{\%}(25))$ and the convergence point $(n$ when $E_{\%}$ becomes less than 5%, 5) with respect to k_1 and k_2 . The points were given by the registered values during training – see Table 8.3.

The value of k_2 was found to be fairly constant for all the weight update intervals and averaged to approximately -1.25. Now k_1 can be computed based on *only* the error at

¹Experiments showed a better approximation when using twice the number, compared to using the explicit number.

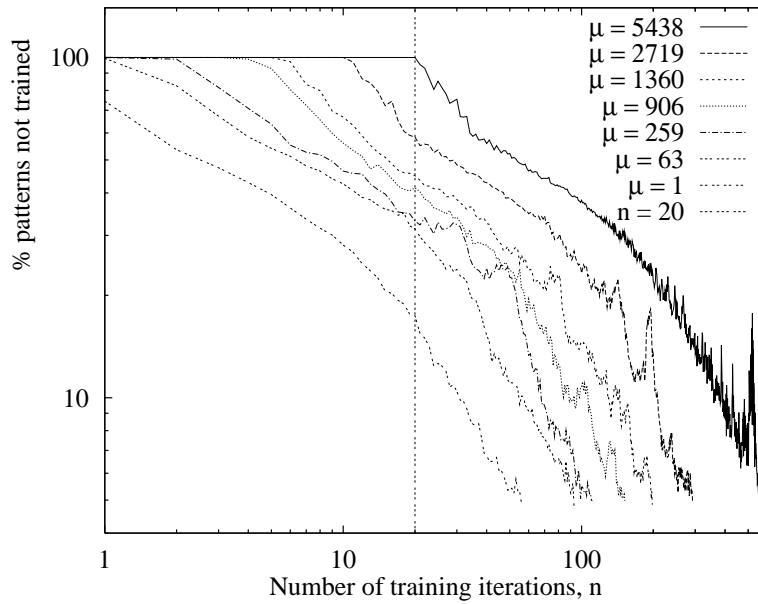


Figure 8.18: Convergence of NETtalk, logarithmic plot. A vertical line is plotted for $n = 20$.

Weight update interval (μ)	k_3	$E_{\%}(25)$	n when $E_{\%}(n) < 5$
1	0	12.61	56
63	0	25.65	93
259	4	32.86	111
494	6	32.38	130
906	8	35.77	152
1360	12	41.12	198
2719	22	53.51	292
5438	42	85.20	560

Table 8.3: The data on NETtalk convergence used for estimation.

$n = 25$ iterations

$$k_1 = \frac{E_{\%}(n)}{(n + k_3)^{-1.25}} \tag{8.7}$$

Furthermore, the total number of iterations is given from Equation 8.6 by

$$N = \left(\frac{5}{k_1}\right)^{1/-1.25} - k_3 \tag{8.8}$$

Figure 8.19, which shows both measured and estimated values of $N(\mu)$, proves the accuracy of the estimation model for this application. However in general this accuracy may not be

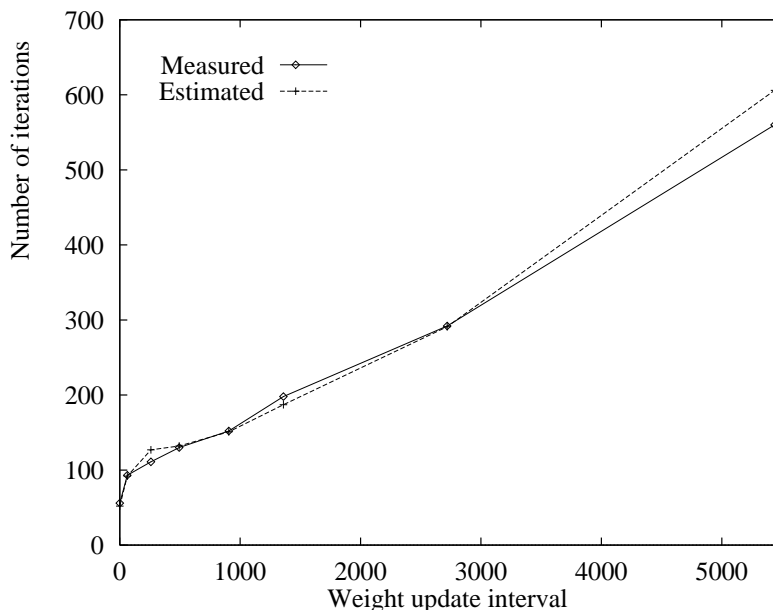


Figure 8.19: Comparing measured and estimated values of $N(\mu)$.

the case. This is a field of research where very little published work using real applications exists. Thus, more studies are required to form a more general framework and to see if it is possible to estimate the convergence in general.

Sensitivity and Convergence Aspects

The BP algorithm is more sensitive to the selection of the learning rate when the weights are infrequently updated. Figure 8.20 shows the convergence for $\mu = 1$ (*learning by pattern*) and $\mu = 5438$ (*learning by epoch*). The error $E_{\%}$ is plotted for the iteration number n_c , which for the best learning rate gives convergence – i.e. $E_{\%}(n_c)$ is less than 5%. The x-axis is logarithmic, and still the epoch learning has a narrower valley in the error curve.

In the training scheme used, a pattern is said to be trained if the error is below a certain threshold. Since the weights are changed during an iteration, there is a risk of trained patterns becoming de-trained, i.e. have an error larger than the threshold after the training iteration. Thus, the number of patterns trained correctly may be less than the number counted during an iteration. Figure 8.21 compares the $E_{\%}$ accumulated *during* each training iteration against $E_{\%}$ computed for the whole training set *after* each training iteration for $\mu = 906$. Even though the epoch computed error is less stable, the two convergence criteria harmonize well with little difference in the final convergence. More unstable result is seen in Figure 8.22 where $\mu = 1$. However, only two extra training iterations are required before convergence is reached.

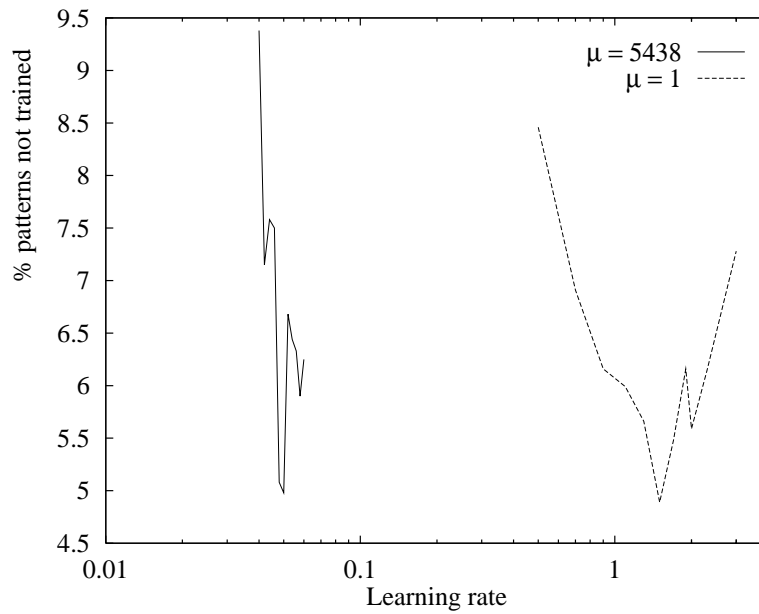


Figure 8.20: The sensitivity of the learning rate (η) on the convergence for epoch versus pattern learning.

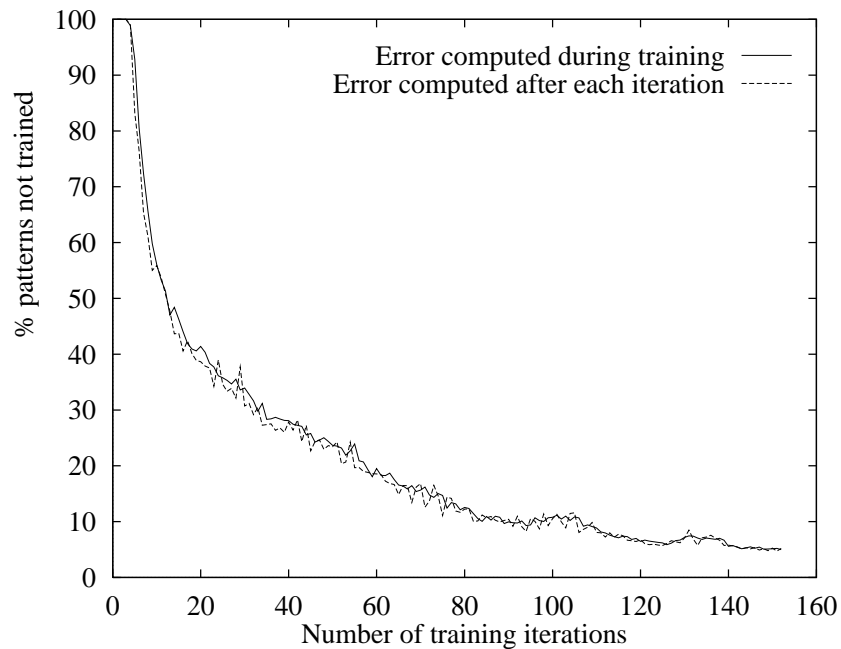


Figure 8.21: Error computed during training compared to error computed after each iteration (epoch) for $\mu = 906$.

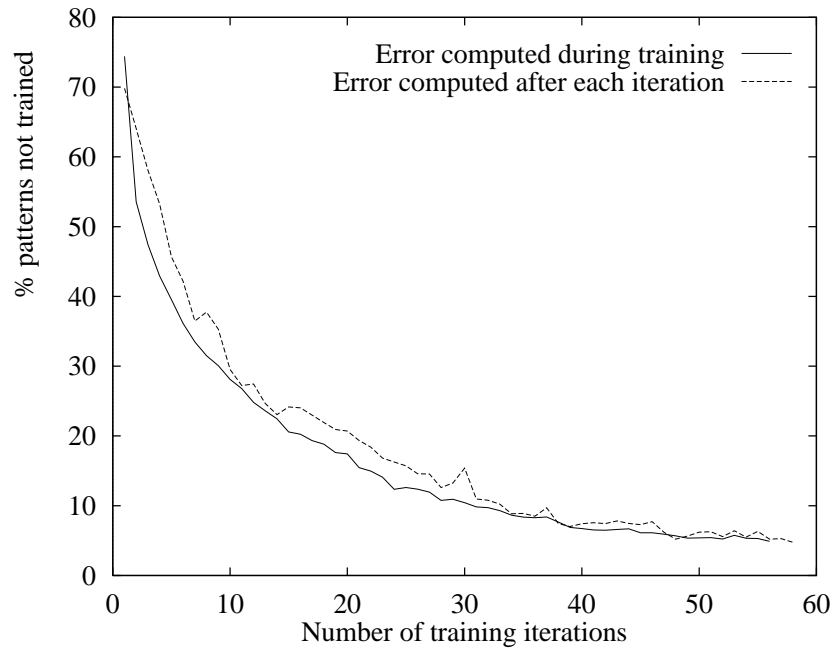


Figure 8.22: Error computed during training compared to error computed after each iteration (epoch) for $\mu = 1$, i.e. *learning by pattern*.

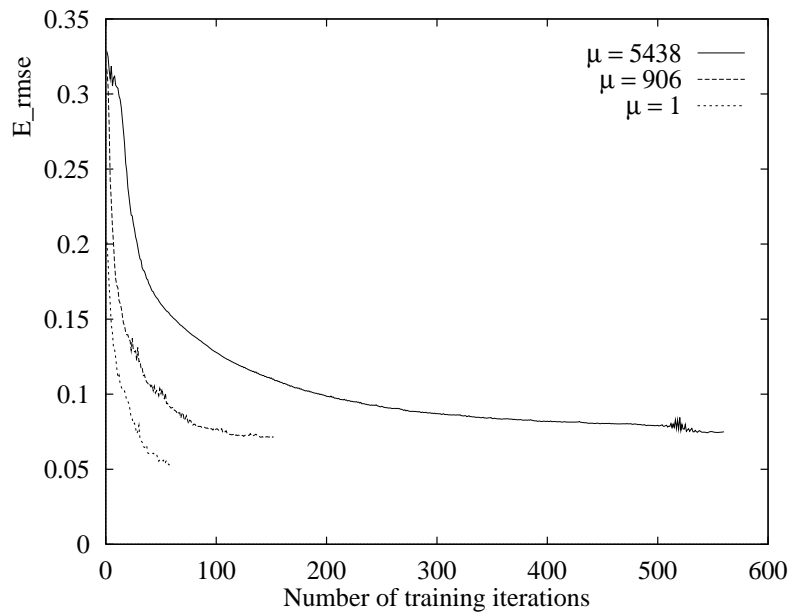


Figure 8.23: The E_{RMSE} plotted for three weight update intervals.

Another convergence measure is the root mean square error, E_{RMSE} . Figure 8.23 plots the E_{RMSE} for three weight update intervals. An interesting observation is that less frequent weight updates lead to a larger error value. This may imply that for infrequent weight updates the patterns become less “overtrained”, which lead to better generalization according to Section 2.1.5.

In [32], Fjerdingstad and Greve report experiments on NETtalk convergence. Only $\mu = 1$, 90 and 190 were tested for the 1000 word corpus, by using a momentum equal to 0.9. For weight updates, after each *word* was presented, they applied a learning rate of 0.2. After 50 iterations $E(n) = 675$ – see Equation 8.1, was obtained. This is equivalent to $E_{RMSE} = 0.0691$. In the above experiments where $\mu = 1$, a lower error is shown – $E_{RMSE} = 0.0567$.

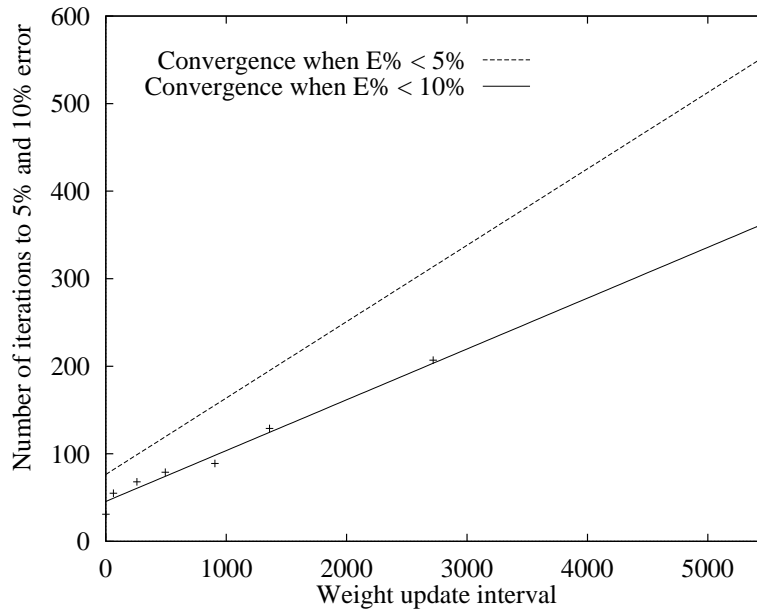


Figure 8.24: Comparing $E_{\%} < 5\%$ and $E_{\%} < 10\%$ error criteria.

Figure 8.24 shows the convergence when $E_{\%} < 10\%$. 31 iterations are required before convergence using *learning by pattern*. In the NETtalk paper [118], the training continued for 30 iterations. As for the $E_{\%} < 5\%$ convergence criteria, there is a linear relation between the weight update interval and the number of iterations required before convergence.

An interesting aspect is the ratio

$$k_e = \frac{N(P)}{N(1)}, \quad (8.9)$$

where P is the number of patterns in the training set. For NETtalk training with 5% error threshold

$$k_e = \frac{560}{56} = 10 \quad (8.10)$$

For a 10% error threshold this becomes

$$k_e = \frac{359}{31} = 11.58 \quad (8.11)$$

For the least strict threshold (10%), there is a larger difference in number of iterations for *learning by pattern* compared to *learning by epoch*.

8.1.4 Minimizing the Total Training Time

In this section, the best weight update intervals for running NETtalk on AP1000 is determined. By applying Equation 7.3, the total training time is found as the time for one iteration multiplied by the number of iterations used for convergence. Total time calculations are conducted for $\mu = 63, 259, 494, 906, 1360, 2719, \text{ and } 5438$.

The time for one iteration, T_{it} , can be computed from Figures 8.2 – 8.4 (not all μ -values are shown). The number of iterations needed for convergence, $N(\mu)$, is given in Figure 8.16.

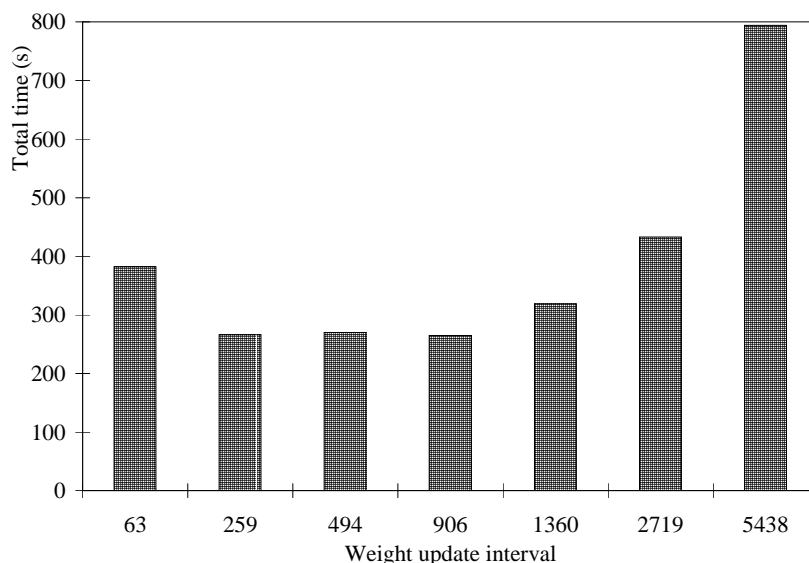


Figure 8.25: Total training time for NETtalk running on 512 cells, using 120 hidden units.

For the 512 cell system, the total training time is plotted in Figure 8.25. The best weight update interval is for $\mu = 906$ using the configuration $(N_{TSP}, N_{NP}) = (64, 8)$. Convergence is obtained after 265 seconds. However, there is only a marginal differences in the time for $\mu \in \{259, 494, 906\}$. The slowest training time is for $\mu = 5438$. Thus, infrequent weight update is not as promising as the MCUPS measure indicates.

Weight changes have not been accumulated for a pattern where the error is sufficiently small, to avoid overtraining. Thus, T_{1it} decreases during training as the number of patterns trained increases. As a constant T_{1it} is used, the T_{total} is larger than the real execution time.

Running the same program – with parallel parts removed, on a Sparc10 workstation using *learning by pattern*, reached convergence after 56 iterations and 115 minutes. This was measured when no other users were active on the computer. AP1000 speedup based on total execution time ($T_{1it}N$) is therefore shown to be 26 times. This is a conservative measure, since the decreasing T_{1it} for AP1000 is not accounted for. When the serial program updated the weights for every pattern, the total training time was 236 minutes. This is 54 times slower than training on AP1000. Thus, total training time can be reduced from hours to minutes by using parallel processing.

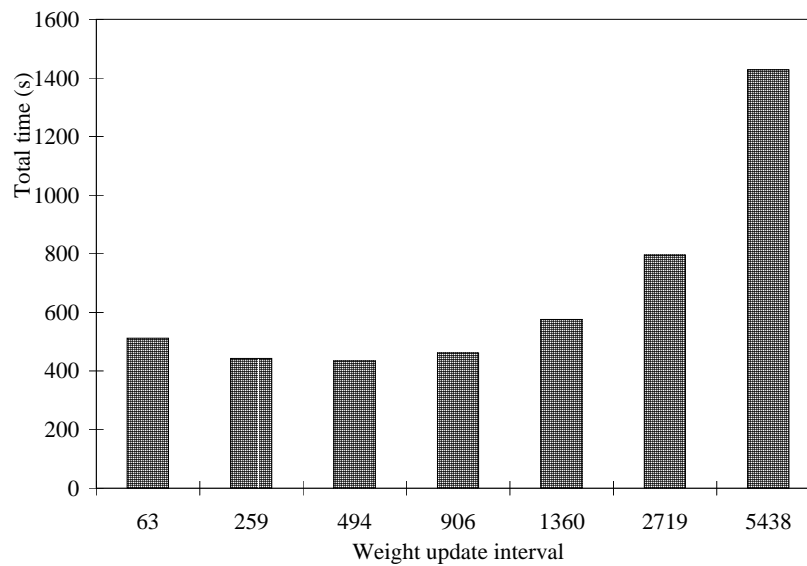


Figure 8.26: Total training time for NETtalk running on 256 cells, using 120 hidden units.

Total training time for 256 cells and 64 cells are given in Figures 8.26 and 8.27, respectively. As the system gets smaller, more frequent weight updates lead to the shortest training time. This can partly be understood by the number of training set partitions, as explained in Section 8.1.1. In Figure 8.2 the difference in maximum performance for the various μ values is small compared to Figure 8.4. Combined with the few iterations required for convergence for frequent weight updating, this leads to the shortest total training time for small systems.

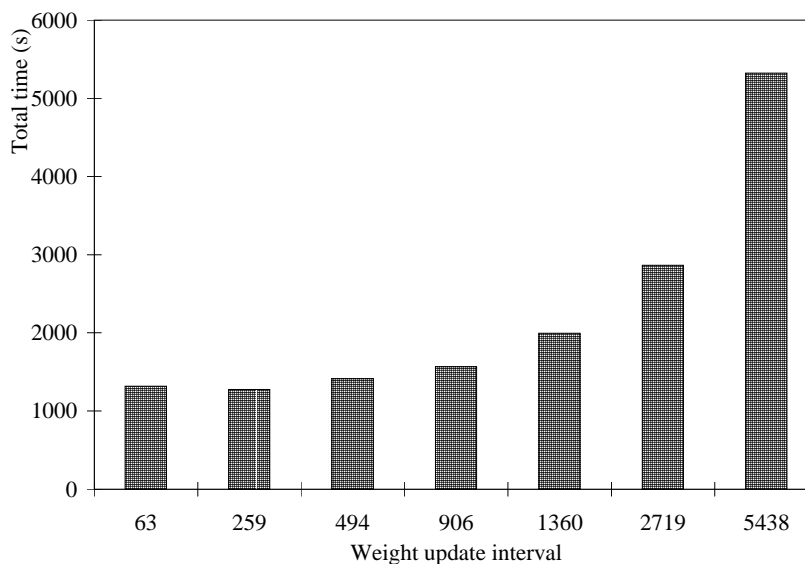


Figure 8.27: Total training time for NETtalk running on 64 cells, using 120 hidden units.

8.2 Neural Network Trained to Classify Sonar Targets

In this section, the results on the convergence of the sonar return application, described in Section 2.2.3, are given. In the experiments, all the available patterns (208) have been used as a training set. The weights are updated – or accumulated for *learning by block*, for a pattern where the error is larger than 0.04. That is

$$E_p(n) > \frac{1}{2} \sum_{k=1}^2 (0.2)^2 = 0.04$$

This threshold is based on an error of 0.2 on each of the output units, which was used by Gorman et al. [43, 44]. As in their work, the smoothing term was set to 0. The number of hidden units was set to 24, which was reported as giving the best recognition rate on the training set.

8.2.1 Results on Sonar Target Classification

The convergence for different weight update intervals is given in Figure 8.28. The y-axis represents the percentage of training patterns with $E_p > 0.04$. The results are very similar to NETtalk – a larger weight update interval leads to slower learning. However, the

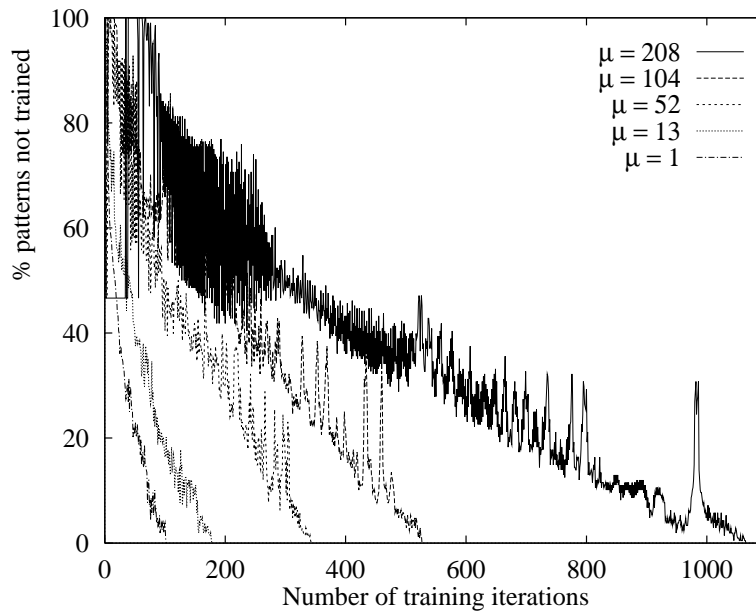


Figure 8.28: The percentage of patterns that have $E_p > 0.1$, for learning sonar return classification.

convergence curves are steeper and it is not possible to use the same logarithmic regression as for NETtalk to estimate the number of training iterations.

The convergence was very sensitive to small changes in initial weights and other parameters. It was seen that the resolution of floating point variables influenced the convergence rate. Continuous values in the input units, instead of two-level values as in the case of NETtalk, probably make the algorithm more sensitive to the accuracy of the variables.

The best learning rate for each weight update interval is plotted in Figure 8.29. A similar experimental rule to Equation 8.2 is established

$$\eta(\mu) = 3.0\mu^{-0.5} \quad (8.12)$$

The total number of iterations until *all* training patterns have reached $E_p \leq 0.04$ during a learning cycle is plotted in Figure 8.30 together with the derived least squares fit of the points

$$N(\mu) = 106.66 + 4.51\mu \quad (8.13)$$

The number of iterations required for an error threshold equal to 5% not training patterns is plotted in Figure 8.31. This is equivalent to the error threshold used for NETtalk. The least squares error equation is given by

$$N(\mu) = 97.04 + 3.88\mu \quad (8.14)$$

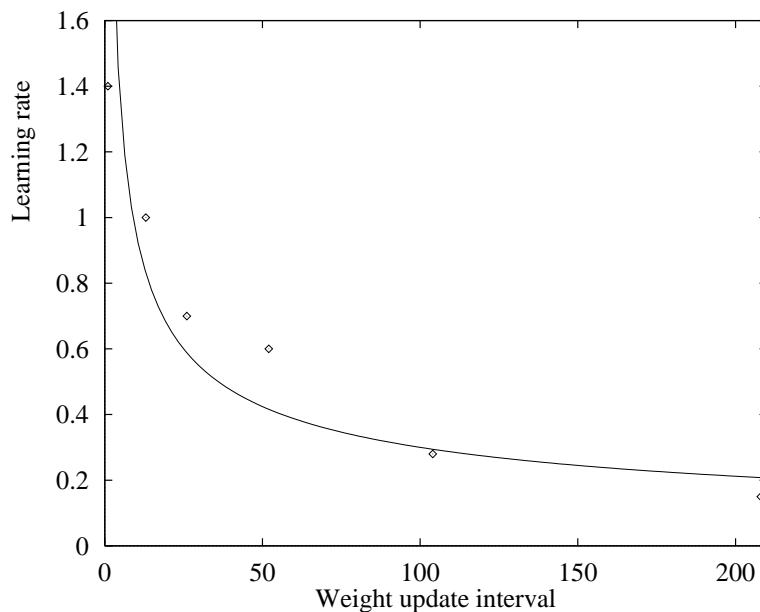


Figure 8.29: The best learning rate η for each of the investigated weight update intervals. The curve plots the derived rule.

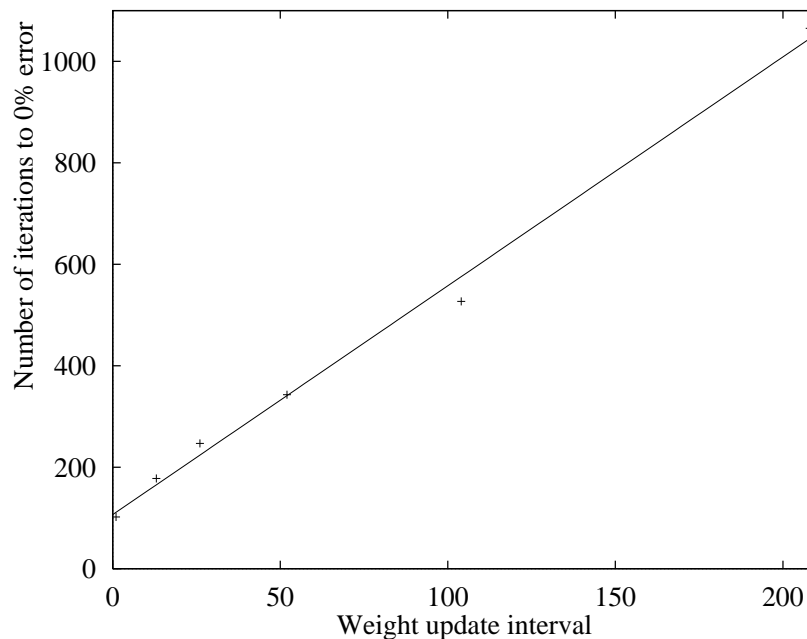


Figure 8.30: Number of iterations required to obtain convergence, i.e. $E_p \leq 0.04$ for all training patterns.

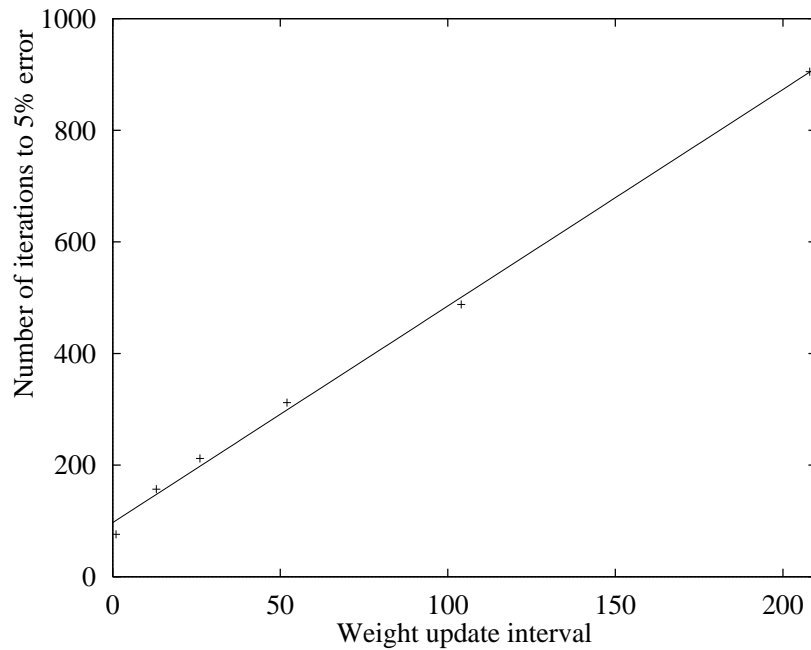


Figure 8.31: Number of iterations needed to reach a stopping criteria of 5% error.

8.2.2 Comparing Sonar Results to NETtalk Results

In this section, common properties of the two applications are highlighted. The idea is to search for a general way of estimating the number of iterations needed for convergence, N , as a function of the weight update interval.

For NETtalk, the learning ratio was found to be $k_e = 10$ for a 5% error threshold and $k_e = 11.58$ for a 10% error threshold. Correspondingly for the sonar target classification (all of the patterns trained) we have

$$k_e = \frac{1065}{102} = 10.44 \quad (8.15)$$

For the less strict stopping criteria of 5%, $k_e = 11.9$. Thus, k_e is in the 10-12 range for both applications. Thus, *learning by pattern* trains 10-12 times faster than *learning by epoch*. Moreover, a less strict error threshold increases k_e . The resemblance between these two applications was not expected, since they are different in both network size and number of training patterns.

This result, of a near constant k_e ratio, will now be used in estimating $N(\mu)$ based on a single result for convergence, $N(1)$. The relation between μ and the k -ratio is shown in Figure 8.32. The equation for the line passing through the points (1,1) – *learning by*

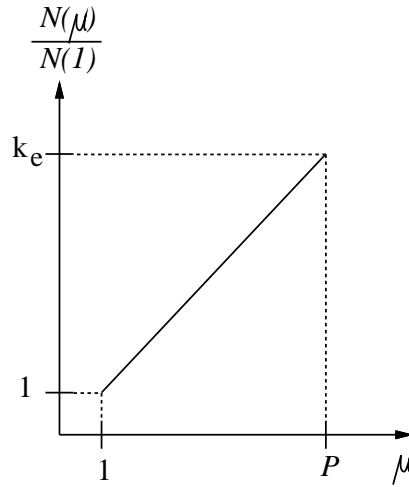


Figure 8.32: The relation between μ and k_e , where P is the total number of patterns in the training set.

pattern and (P, k_e) – learning by epoch is given by

$$k_\mu - 1 = \frac{k_e - 1}{P - 1}(\mu - P) \Leftrightarrow k_\mu = \frac{k_e - 1}{P - 1}(\mu - P) + 1 \quad (8.16)$$

where k_μ also can be understood as $N(\mu)/N(1)$. An estimate of the $N(\mu)$ is then given by

$$N(\mu) = k_\mu N(1) \quad (8.17)$$

For neural training, the application is usually trained several times with different parameters. Therefore, Equation 8.17 can be used for the approximation of $N(\mu)$, after some initial convergence tests, in the heuristic shown in Figure 7.10. To confirm the estimation method, other applications should be tested.

To set the appropriate learning rate, the previous experimental rules (Equation 8.2 and 8.14) can be formulated in a general approximation rule

$$\eta(\mu) = k\mu^{-0.5} \quad (8.18)$$

where k is a positive constant less than 10. As the training set gets smaller, the appropriate k value decreases. This is in accordance to the two experiments reported in this thesis.

8.3 Speech Recognition Network

Figure 8.33 shows the performance for a speech recognition network on 512 processing elements. It was not possible to execute all the configurations due to communication

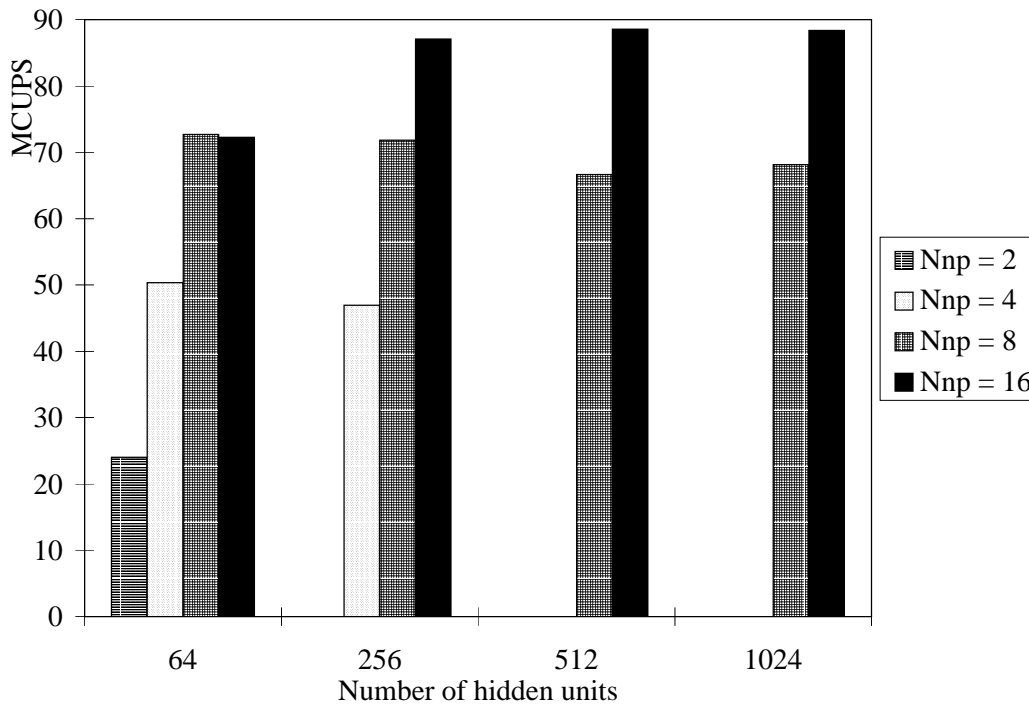


Figure 8.33: Training performance on 512 processors for speech recognition network. $N_{NP} = 2, 4, 8,$ and 16 .

limitations (*ring buffer overflow*). Except for 64 hidden units, the highest training speed is obtained for $N_{NP} = 16$ i.e. 16 cells assigned to each copy of the network. For this configuration there is no training set partition in the horizontal dimension and therefore, the training speed does not exceed that for the fixed 2APC implementation. Further, since 3APC trains faster than 2APC as indicated in Figure 6.14, 3APC should be used instead of this flexible implementation – except for 64 hidden units. To further improve the performance, synapse parallelism should probably be introduced due to the size of the network.

8.4 Image Compression

Image compression requires a small network [18], but implies a large amount of data, both for training and for real-time compression. The performance for a 64 input and 64 output network and 512 cells is given in Figure 8.34 for three different compression ratios. The number of hidden units varied between 4, 8, and 16. A small number of hidden units implies a large compression ratio. The weights were updated for every 512 training patterns. For the smallest network (4 hidden units) the performance of the non-contention mapping

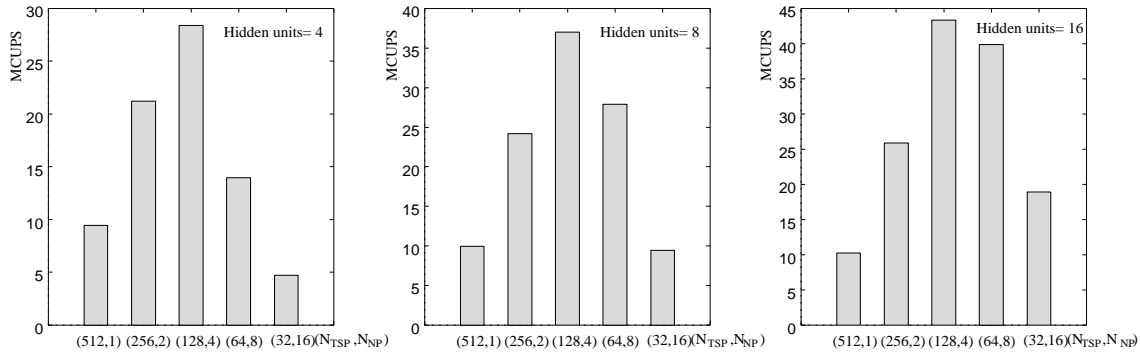


Figure 8.34: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 processor configuration training a compression network.

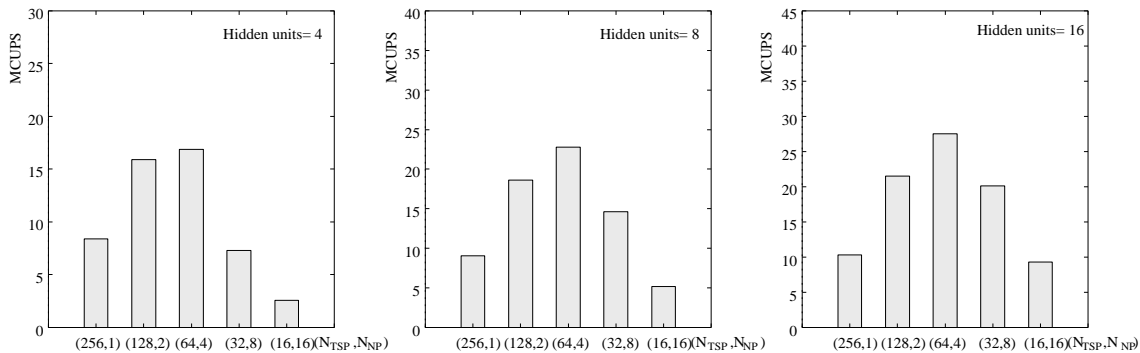


Figure 8.35: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 processor configuration training a compression network.

$(N_{TSP}, N_{NP}) = (32, 16)$ is approximately 6 times less than that for the best mapping $(N_{TSP}, N_{NP}) = (128, 4)$. Thus, for small networks a large degree of training set parallelism is beneficial.

Figure 8.35 shows the performance for 256 cells. In this configuration, $N_{NP} = 4$ also gives the best overall performance. However, for $N_{NP} = 2$ the performance is higher than for $N_{NP} = 8$ – for 8 and 16 hidden units, which is opposite from Figure 8.34.

The performance on 128 processors appears in Figure 8.36. The number of cells in the y-dimension (16) and x-dimension (8) is half of that in the 512 cell system. The difference between the best and worst results is lower than for the larger systems. For 4 and 8 hidden units the maximum learning speed is reached when $N_{NP} = 2$. Thus, assigning fewer processors to each copy of the network is important as the number of cells is reduced. In this way, a large number of training set partitions can be maintained. The training speed on 64 cells is given in Figure 8.37.

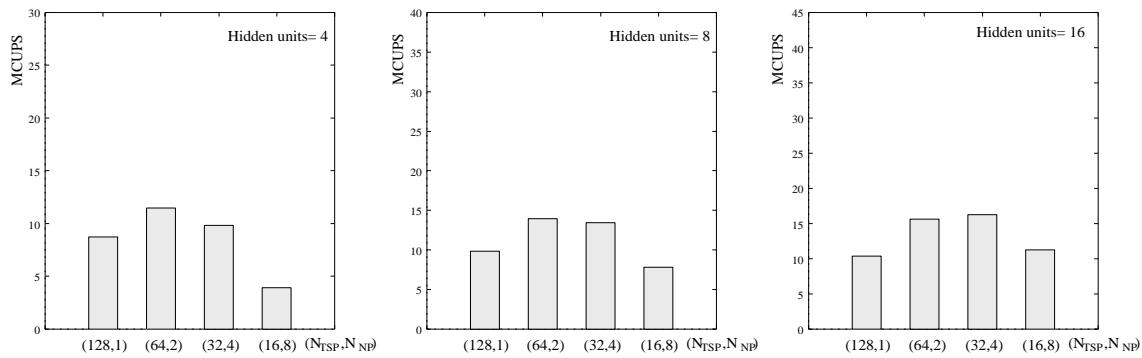


Figure 8.36: MCUPS performance for different combinations of neuron and training set parallelism on an 8 x 16 processor configuration training a compression network.

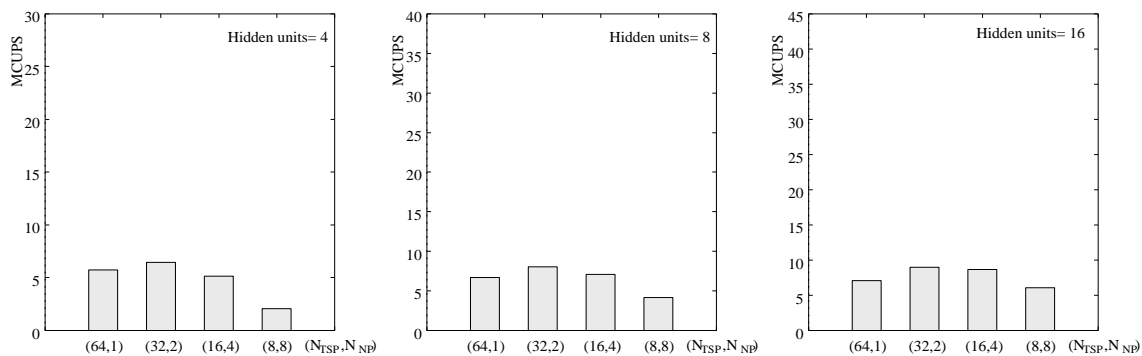


Figure 8.37: MCUPS performance for different combinations of neuron and training set parallelism on an 8 x 8 processor configuration training a compression network.

The highest speeds in each of the above experiments are plotted in Figure 8.38. The ratio between the highest performance achieved and the performance of the 2APC configuration is plotted in Figure 8.39. As shown, a reduction in the number of hidden units leads to larger ratios. For 4 hidden units and 256 processors the difference is a factor of 6.5 – 550% a considerable difference. The nonlinearity in the curves in the figure can be explained by the difference in configuration. For 128 and 512 processors $C_y = 2C_x$, while for 64 and 256 processors $C_y = C_x$. Thus, when there are twice the number of processors in the vertical dimension, additional training set partitions in the horizontal dimension do not give rise to the same increase the performance as for 64 and 256 processors. It can be seen that the scaling in Figure 8.38 would drop dramatically if the 2APC-configuration had been used. This is especially true for networks with a small number of hidden units, for which the flexible mapping already has a limited scaling.

The primary goal of designing the parallel mapping scheme has been to reduce the neural

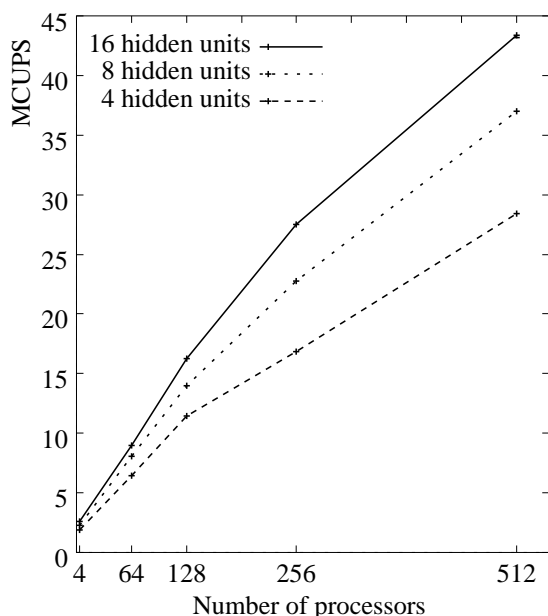


Figure 8.38: Training speed for image compression network plotted as function of the number of processors.

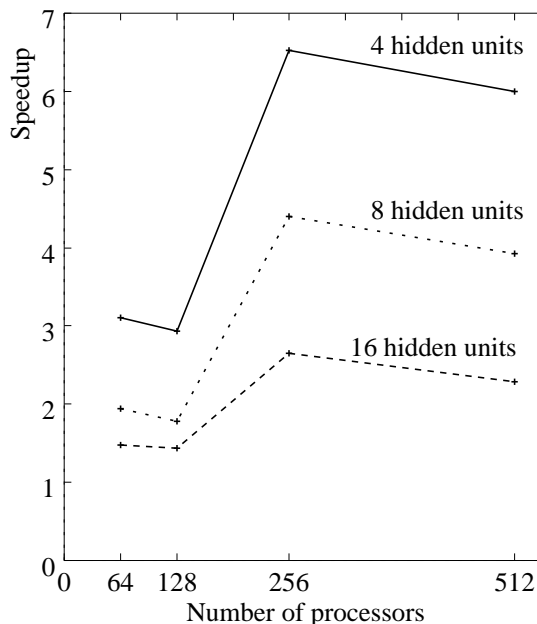


Figure 8.39: Speedup of the best configuration compared to the 2APC configuration.

network training time. However, for image compression there will be a large amount of data to be processed for real-time compression of video frames. The proposed mapping will allow fast compression by compressing different parts of each frame on different copies of the neural network.

8.5 Summary of the Application Adaptable Mapping

In this chapter, a near optimal mapping of the back propagation algorithm onto a parallel computer has been proposed. Multiple degrees of parallelism have been exploited and the influence of the weight update interval on convergence has been included in the mapping strategy.

The importance of the flexible implementation depends on the neural application. For NETtalk, 40% higher performance than for the 2APC configuration was achieved in the best case. For the smaller networks like image compression network the difference in performance was larger – 550% in the best case. This highlights the advantages of an *application adaptable* mapping algorithm to reduce training time.

The results also indicate that the flexibility becomes more important for parallel computers

with a large number of processing elements. This is because most neural network applications require a relatively small neural network size. However, the number of training patterns for training or the amount of data to be processed in the recall-phase may be very large. Therefore on large systems, the number of training set partitions should be large rather than the number of processors assigned to each subset.

Estimation of execution time, based on the number of floating point operations and communication time estimates, was tested. For the NETtalk application it turned out to be accurate and was used to show that the number of processors for the flexible mapping can be increased almost without limit resulting in increased performance.

Most other BP implementations published do not consider the weight update interval or the mapping strategy according to the *given* neural network application. The convergence tests of two applications in this chapter indicate a linear relation between the weight update interval and the number of iterations to reach convergence. *Learning by epoch* is a factor of 10-12 times slower than *learning by pattern*. Even though the speedups indicated by the MCUPS figures are not practically obtainable, the computation of total execution time undertaken shows that parallel processing can drastically reduce training time. This requires that the parallel implementation is flexible and adaptable to the given neural network and training set.

Chapter 9

Implementation of BP on RENNS

The results from the different implementations on AP1000 showed that for a small number of processors the difference in performance between them was small. Thus, many different implementations on RENNS should result in similar performance. However, training set parallelism imply slow convergence and therefore, other parallel degrees should be used instead.

In this chapter, two implementations of backpropagation training on RENNS are described and the results on performance are given. Furthermore, the communication time for the two implementations is compared by time estimates.

First, a solution using neuron parallelism is briefly described. Then, a solution which rearranges the processor assignments according to the given neural network application is proposed. This method combines neuron parallelism and pipelining.

9.1 Neuron Parallelism

One or two ring buses¹ connects the processing modules, see Figure 9.1. Each module contains one slice of the network. For each pattern presented during training, two communication phases are required. First, all modules must broadcast their local hidden layer output values to all other modules. Second, partial sums of the error to be used for updating the hidden layer weights have to be distributed to the dedicated processors. Since the two steps are different, separate ring buses were used for each of these two communication steps. This reduces the total communication overhead and is therefore faster than using a single ring bus solution. The neuron parallel implementation is written by Solheim [125].

¹The token based ring protocol called Vector was used.

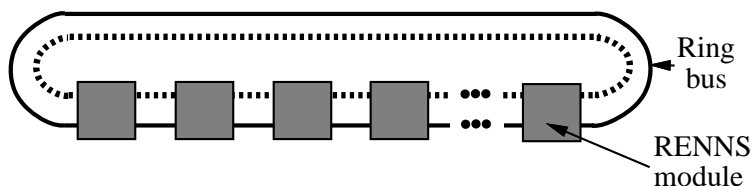


Figure 9.1: The RENNS modules connected by a single ring bus. An optional bus is shown by the dotted line.

9.2 Neuron Parallelism and Pipelining Combined

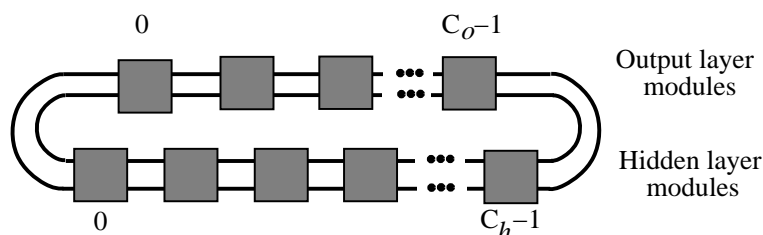


Figure 9.2: The RENNS modules connected by two ring buses for combining neuron parallelism and pipelining.

Based on the neuron parallel implementation, a combined solution is proposed here using neuron parallelism and pipelining of the training patterns, as shown in Figure 9.2. The upper RENNS modules compute the output layer, while the lower row computes the hidden layer. To avoid an interleaving of hidden-to-output communication with output error communication two ring buses should be used. C_h RENNS modules are used for the hidden layer and C_o for the output layer. The number of available modules is $C = C_h + C_o$. The method is similar to the proposal for AP1000 in Section 7.1.2, where the number of processors assigned to each layer is flexible. The motivation for this method is that

- More computation (in double loops) is done between communication steps, since less processors are used for each layer. This implies reduced communication (see below).
- Reduction of the program code size implies more space for data in fast memory.

This implementation is similar to the implementation on Symult S2010 – described in Section 3.3.1 and in [3], but does not use a separate processor for message handling. Moreover, delayed weight update is used instead of *learning by block/epoch*. The communication strategy is also different and based on equal sized packages sent around the ring. The token ring protocol implemented on RENNS enables efficient broadcasting of packages, which is more efficient than accumulation of values along the ring as used on Symult S2010.

```

Forward(integer pat) { Forward phase }
begin
  for j = 1 to  $n_h$  do begin
    for i = 1 to  $N_i$  do begin
      compute hidden output j for pattern pat;
      broadcast element to output PEs;
    end;
  end;
end;

Update(array  $\delta_h$ ) { Update the hidden weights }
begin
  for j = 1 to  $n_h$  do
    for i = 1 to  $N_i$  do
      Update weight  $w_{h,ji}$ ;
    end;
  end;
end;

Hidden()
begin
  Initialize the communication system;
  Allocate memory;
   $E = \infty$ ;
  while  $E > limit$  do begin
    Forward(1);
    for p = 2 to  $P_c$  do begin { Training pattern index }
      Forward(p);
      Receive  $\delta_h$  values from output PEs;
      Update( $\delta_h$ ); { Update weights for pattern p - 1 }
    end;
    Receive array of  $\delta_h$  values from output PEs; { Hidden delta error for pattern p }
    Update( $\delta_h$ );
    Receive  $E$ ;
  end;
end;
end;

```

Figure 9.3: Pseudo-code for the hidden layer program.

```

float Forward(integer pat) begin { Forward phase }
  Read  $n_h$  input elements from  $C_h$  hidden modules;
  for  $k = 1$  to  $n_o$  do begin
    for  $j = 1$  to  $N_h$  do
      compute output,  $\delta_o$ , and accumulate error for unit  $k$  for pattern pat;
      Broadcast error for pattern pat to output PEs;
      for  $c = 1$  to  $C_o$  do
        Read and accumulate error;
      return error;
    end;
  end;

ComputerDeltaError() begin
  for  $c = C_h + 1$  to  $C_h + C_o$  do
    if  $c = \text{MyId}$  then
      Compute partial  $\delta_h$  to be used locally;
    else
      Compute partial  $\delta_h$  and send them to output PE  $c$ ;
  for  $c = 1$  to  $C_o - 1$  do
    Read partial  $\delta_h$  and add to local array of  $\delta_h$ ;
  for  $j = 1$  to  $n_h$  do
    Multiply  $\delta_h$  by the derivate of the sigmoid function;
    SendToHiddenModules( $\delta_h$ );
end;

Update(array  $\delta_o$ ) begin { Update the hidden weights }
  for  $k = 1$  to  $n_o$  do
    for  $j = 1$  to  $N_h$  do
      Update weight  $w_{o,kj}$ ;
  end;

Output()
begin
  Initialize the communication system;
  Allocate memory;
   $E = \infty$ ;
  while  $E > \text{limit}$  do
    for  $p = 1$  to  $P_c$  do begin { Training pattern index  $p$  }
      error = Forward( $p$ );
      Update  $E$  by error;
      if  $\text{MyId} = C_h + 1$  then { Output module 0 }
        BroadcastToHiddenModules(error);
        ComputerDeltaError();
        Update();
      end;
  end;
end;

```

Figure 9.4: Pseudo-code for the output layer program.

Pseudo-code respectively for the hidden and output layer program are given in Figure 9.3 and 9.4. The pseudo-code is similar to that for the implementation on AP1000 described in Section 5.2.2. The hidden layer program `Hidden()` computes the forward pass for two patterns before it starts updating the weights. Hence, a delayed weight update scheme is employed (described in Section 2.1.7). Since training set parallelism is not employed in this implementation, the momentum has been omitted – see the smoothing term description in Section 2.1.3. The output layer program `Output()` computes the output and hidden delta error in addition to the forward pass, leading to more code than that for the hidden layer program. In the forward pass, all output PEs receive hidden output values from each of the hidden PEs. In the backward pass, the hidden delta error, δ_h , is first computed and accumulated among the output modules. Then, the result is sent from the output PEs to the hidden PEs as illustrated in Figure 9.5.

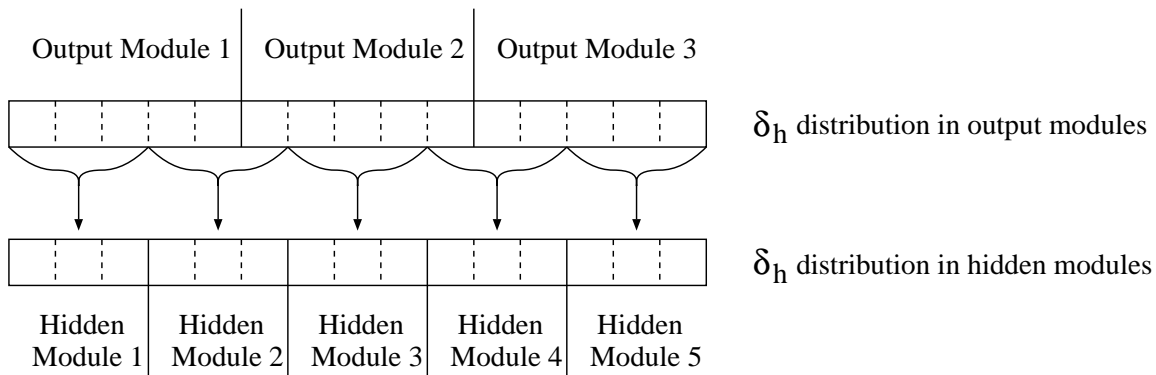


Figure 9.5: Example of the distribution and communication of the δ_h among hidden and output modules.

In this example the output modules send their δ_h to the corresponding hidden module(s) as indicated by arrows. Similarly, if $C_h < C_o$ the vector from each output module is partitioned and each part is sent to a single hidden module.

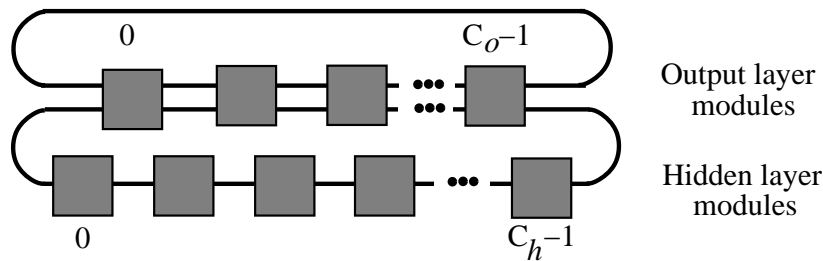


Figure 9.6: Shortened output module ring.

The communication in the pipelined mapping scheme can be optimized by using the shortened ring approach in Figure 9.6. By this configuration the output modules propagate

packages on their local ring for summing the partial results of the hidden delta error.

```

 $T_h(C_h)$ : Time for computing hidden layer for 1 pattern
 $T_o(C_o)$ : Time for computing output layer for 1 pattern
 $C_h = C/2$ ;
 $C_o = C/2$ ;
 $T_m = \max(T_h(C_h), T_o(C_o))$ ;
if ( $T_h(C_h) > T_o(C_o)$ ) then
    while ( $(t = \max(T_h(C_h + 1), T_o(C_o - 1))) < T_m$ ) do begin
         $C_h = C_h + 1$ ;
         $C_o = C_o - 1$ ;
         $T_m = t$ ;
    end;
else
    while ( $(t = \max(T_h(C_h - 1), T_o(C_o + 1))) < T_m$ ) do begin
         $C_h = C_h - 1$ ;
         $C_o = C_o + 1$ ;
         $T_m = t$ ;
    end;
ConfigureSystem( $C_h, C_o$ );

```

Figure 9.7: Heuristic for selecting the best assignment of modules for each neural network layer.

To obtain the best distribution of modules, the algorithm in Figure 9.7 can be used. The algorithm starts with an equal number ($C/2$) of hidden and output modules. Then, it alters the number of modules assigned to each layer by comparing $T_h(C_h)$ and $T_o(C_o)$. If the hidden layer modules need most time, it increases the number of hidden modules. In many cases this will make a better load balance, since most ANN applications require a much larger number of input neurons than output neurons. However, if this is not the case, a larger number of number of modules assigned to the output layer is tested. The timing required (T_h and T_o) is measured by execution on RENNS. Training a neural network usually requires several hundred iterations, each consisting of presenting the whole training set. Thus, the initial time measurements to find the best configuration do not add much to the total execution time. The algorithm could be improved by use of binary search.

It is possible to execute the forward and backward phase in parallel [111]. This creates a three stage pipeline for a two layer network. However, the output weights would have

to be stored and updated twice, since they are necessary for both the forward output and hidden delta error computations. This increases the intermodule communication.

9.3 Comparison of Communication

In this section, the communication time for the two major communication steps of the parallel BP training algorithm are compared. First, the time for exchanging the hidden output values is considered. Secondly, the communication for the hidden delta error computation is investigated.

To compare the communication, a model can be made to estimate the time used for communication. The cycle time of the communication subsystem is $t_{cc} = 100ns$ and the propagation time between two data stream controllers has been measured to $t_{sdel} = 60ns$. Solheim has derived the time for communication operations on a ring bus using the vector communication protocol² [125]

- sending overhead (move data from FIFO to the data stream controller)

$$t_{sendov} = \frac{7}{2}t_{cc} = 350ns$$

- pass an intermediate module $t_{byp} = \frac{5}{2}t_{cc} + t_{sdel} = 310ns$

- throughput in time per word $t_{word} = 8t_{cc} = 800ns$

- receiving overhead (synchronization and moving data to FIFO)

$$t_{recvov} = 2t_{cc} + t_{sdel} + 9t_{cc} = 1160ns$$

- remove address $t_{remaddr} = 7t_{cc} = 700ns$

These figures were used in Myklebust's thesis [93]. He further used them to derive expressions for various types of communication. For instance, if C modules each want to send a vector of L words³ to the other modules ("all to all" communication) the time is given by

$$T_{allall}(C, L) = (C(310C + 800L) + 2210)ns \quad (9.1)$$

Hidden Output Communication

For the simple neuron parallel scheme, all C modules send their hidden output vector of $n_h = N_h/C$ words to all other modules. Therefore, Equation 9.1 can be applied for $L = n_h$

²This was used in the following experiments.

³One word is 32 bits and contains one floating point value.

to compute the communication time

$$\begin{aligned} T_F^{neur}(C, N_h) &= (C(310C + 800\frac{N_h}{C}) + 2210)ns \\ &= (310C^2 + 800N_h + 2210)ns \end{aligned} \quad (9.2)$$

For the pipelined mapping, C_h hidden modules each send $n_h = N_h/C_h$ words to the output modules. By assuming⁴ that the token is in hidden module 0, no time is added to wait for the token to arrive. Since the hidden modules are not receiving, the next hidden module can remove the token and use it for sending its vector. This is similar to “all to one” communication time derived by Myklebust. In the neuron scheme all modules were both sending and receiving. Thus, the token had to propagate around the ring before it could be used for sending by the next module. The number of intermediate modules to the final receiving module is

$$C_{int} = C_h - 1 + C_o - 1 = C - 2 \quad (9.3)$$

The vector is sent to all output modules, hence the vector propagates through $C_o - 1$ modules. The initial latency before the first word is received is $t_{sendov} + C_{int}t_{byp} + t_{recov} + t_{remaddr}$. Since the operations in the communication system are highly parallelized in a pipeline, the latency is only included for the first vector to be sent. The C_h vectors are transferred, by each using the time $n_h t_{word}$. The time for changing sending module is approximated to t_{byp} and is undertaken $C_h - 1$ times. The overall time is then

$$\begin{aligned} T_F^{pipe}(C, C_h, N_h) &= t_{sendov} + C_{int}t_{byp} + t_{recov} + t_{remaddr} + C_h n_h t_{word} + (C_h - 1)t_{byp} \\ &= (350 + (C - 2)310 + 1160 + 700 + C_h \frac{N_h}{C_h} 800 + (C_h - 1)310)ns \\ &= (310(C + C_h) + 800N_h + 1280)ns \end{aligned} \quad (9.4)$$

In Equation 9.4, $C + C_h < 2C$. Thus, if $2C < C^2$ the pipeline scheme uses less time for communication.

$$2C < C^2 \Rightarrow C > 2 \quad (9.5)$$

Hence, for parallel systems with more than 2 processing modules, the pipelining scheme is in the forward phase faster than the neuron parallel scheme. For two modules, the pipelining scheme may as well be the fastest.

Delta Error Communication

For the neuron parallel method, C modules send $C - 1$ vectors of length $n_h = N_h/C$. All modules are both sending and receiving, and Equation 9.1 is applied to compute the communication time

$$\begin{aligned} T_B^{neur}(C, N_h) &= (C(C - 1)(310C + 800\frac{N_h}{C}) + 2210)ns \\ &= ((C - 1)(310C^2 + 800N_h) + 2210)ns \end{aligned} \quad (9.6)$$

⁴This is possible by a small modification to the communication protocol.

For pipelining, C_o modules send $C_o - 1$ vectors of length $n_h = N_h/C_o$. Applying the same equation here leads to a communication time

$$\begin{aligned} T_B^{pipe}(C, C_o, N_h) &= (C_o(C_o - 1)(310C + 800\frac{N_h}{C_o}) + 2210)ns + T_{outtohid}() \\ &= ((C_o - 1)(310C_oC + 800N_h) + 2210)ns + T_{outtohid}() \end{aligned} \quad (9.7)$$

where $t_{outtohid}$ is the time for sending the summed hidden delta error vectors from the output modules to the hidden modules. Considering $C_h = C_o$, C_o modules have to send a vector of size n_h to one hidden module. Further, the first output module on the ring is assumed to have the token. This leads to an initial number of intermediate modules

$$C_{int} = C_o - 1 + C_h - 1 = C - 2 \quad (9.8)$$

This gives a communication time

$$\begin{aligned} T_{outtohid}(C, C_o, N_h) &= t_{sendov} + C_{int}t_{byp} + t_{recvov} + t_{remaddr} + C_on_h t_{word} + (C_o - 1)t_{byp} \\ &= (350 + (C - 2)310 + 1160 + 700 + 800N_h + (C_o - 1)310)ns \\ &= (310(C + C_o) + 800N_h + 1280)ns \end{aligned} \quad (9.9)$$

As the number of modules increases

$$310(C + C_o) + 800N_h \ll (C_o - 1)(310C_oC + 800N_h) \quad (9.10)$$

and $T_{outtohid}()$ do not have to be considered. Further, since always

$$(C_o - 1)(310C_oC + 800N_h) < (C - 1)(310C^2 + 800N_h) \quad (9.11)$$

the pipelining scheme will require least time for computing the hidden delta error. The computation intensive part is among the output modules. Thus, the pipelining scheme benefits from communication between fewer modules.

The present overlap of computation and communication is not considered in the estimates above. However, for plain neuron parallelism it is not possible to overlap as much as for the pipelined scheme. This is because communication takes more time and less computation is needed in between communication.

9.4 Results

The performance on RENNS of the implementations described in the previous sections is given below. The code has been more optimized than that for AP1000. Among other things, a look-up table for the sigmoid function is implemented. Thus, the results in the earlier chapters of this thesis are not directly comparable to the following results. The results presented in this section are also included in tables in Appendix B.3.

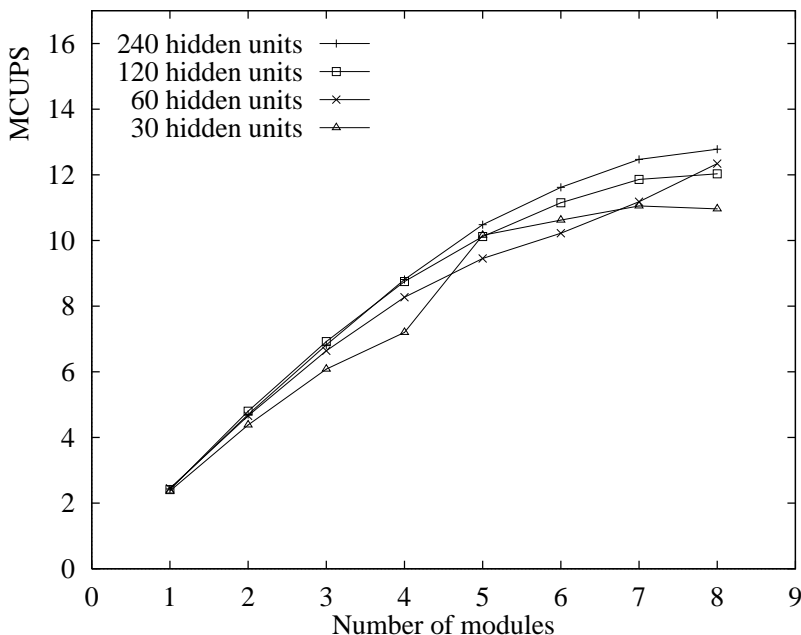


Figure 9.8: Neuron parallel NETtalk training on 1 to 8 processing modules.

Figures 9.8 – 9.10 show the training performance of the neuron parallel implementation for the different applications. These are equivalent to those used to measure performance on the AP1000, as described earlier. For these experiments, 8 modules were connected in a ring. If less than 8 modules are used, the message packets still propagate around the ring of 8 modules. Thus, the communication time could have been reduced by connecting only those modules being tested. However, this would have been cumbersome when doing the experiments and with small improvements in the performance – compare the results for 8 modules in Figure 9.8 and 9.9 to Figure 9.19.

For NETtalk and the speech recognition networks, training performance increases when more processors are used, except for the NETtalk network with 30 hidden units. For this network the performance decreases after 7 modules. The leap in the measures between 4 and 5 modules for a NETtalk network with 30 hidden units (Figure 9.8) is due to the ability to store more of the data arrays in fast SRAM. On 5 modules the data arrays are smaller than when 4 modules is used. A similar leap can be observed for the speech recognition network (Figure 9.9).

As seen in Figure 9.10, speedup is very limited for small networks. For the image compression networks, the optimal number of processing modules is 6, while for the sonar classification network the number is 3.

Figures 9.11 – 9.14 present the performance of the pipeline implementation on 8 modules. The x-axis represents the number of hidden modules employed, C_h . The corresponding number of output modules is given by $C_o = 8 - C_h$.

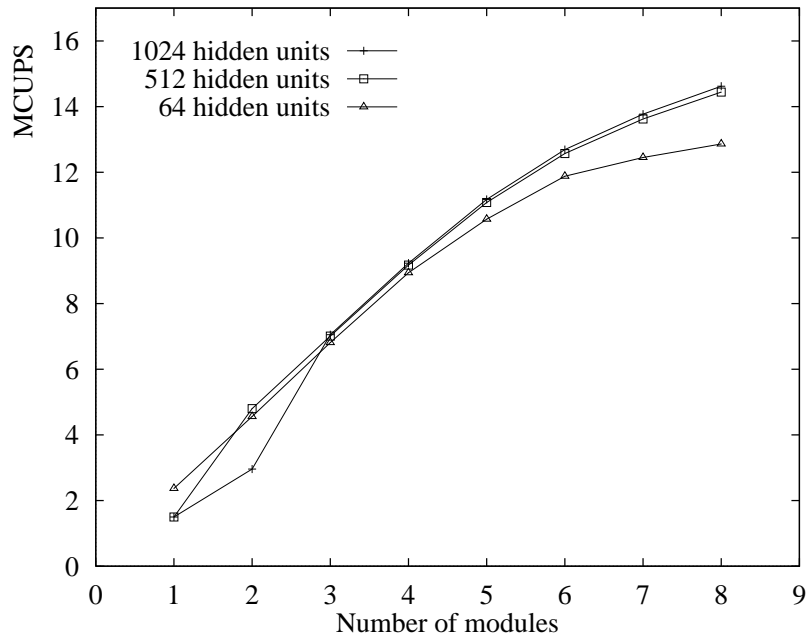


Figure 9.9: Neuron parallel speech recognition network training on 1 to 8 processing modules.

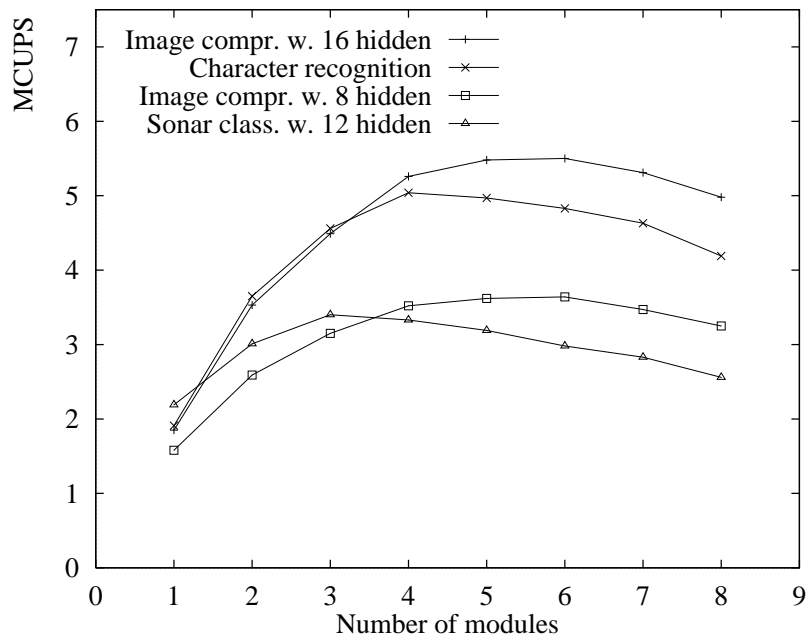


Figure 9.10: Neuron parallel training of small networks on 1 to 8 processing modules.

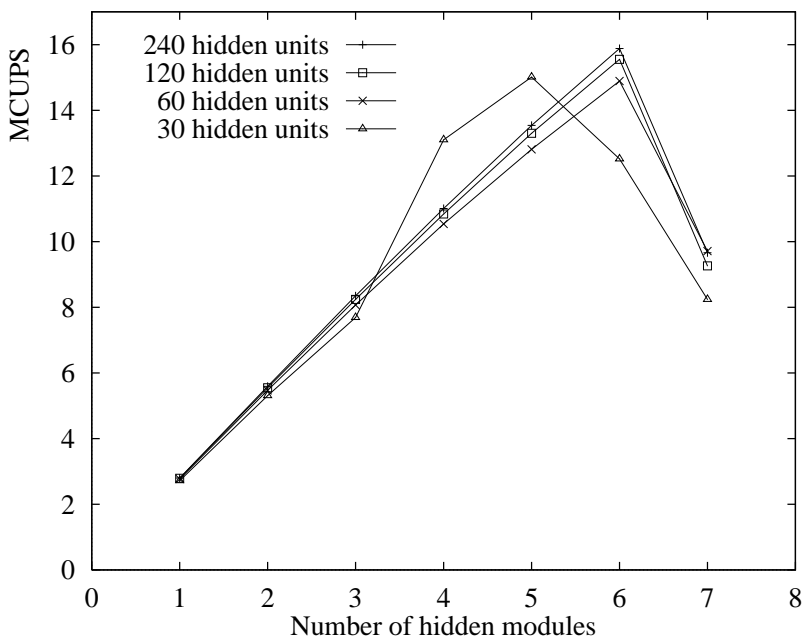


Figure 9.11: Pipelined training of NETtalk. The horizontal axis shows the number of modules involved for hidden layer computation. The total number of modules is 8 for all the experiments.

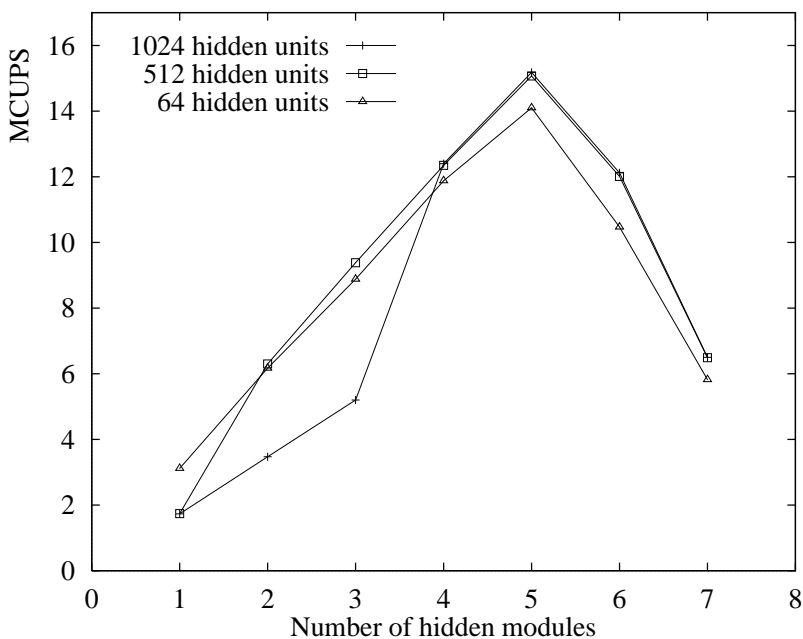


Figure 9.12: Pipelined training of speech recognition network on 8 modules.

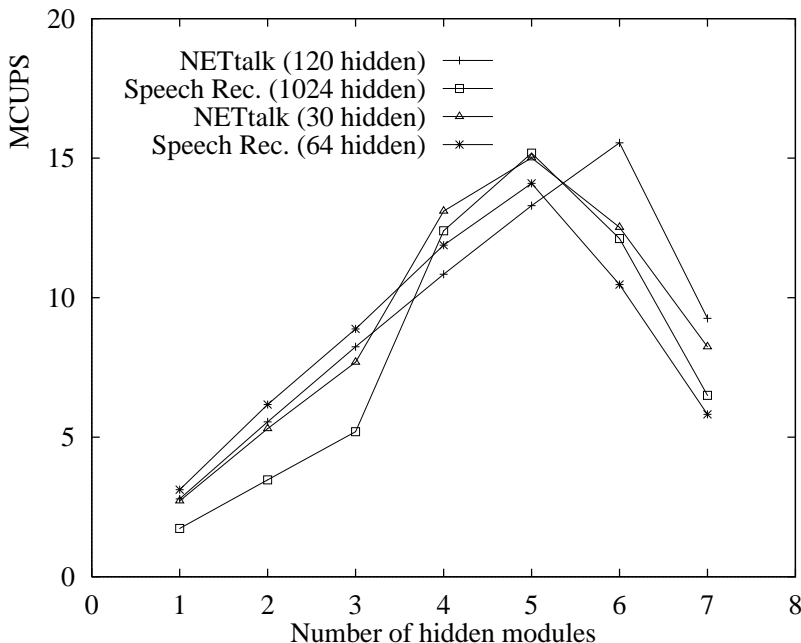


Figure 9.13: Pipelined training of large networks on 8 modules.

The best performance of NETtalk networks (Figure 9.11) and speech recognition networks (Figure 9.12) is achieved when $C_h > C_o$. This is due much more input and hidden neurons than output neurons. All these neural networks trains faster than the plain neuron parallel mapping using the best pipeline configuration. The NETtalk network with 120 hidden units runs 29% faster for the combined algorithm, when using a total of 8 modules. This illustrates the benefit of using many (6) processors in the hidden layer (computation intensive) and few (2) in the output layer (communication intensive). When the NETtalk network has a smaller number of hidden neurons (30), the optimal number of hidden modules is reduced to 5. This is in opposition to the explanation given in Section 6.1.3 that computation load is moved to the output layer when the number of hidden neurons is increased. However, for this network the number of hidden neurons (30) almost equals the number of output neurons (26). Also, since the network is relatively small, the communication overhead becomes more prominent. To compensate for this, more modules are needed to process the output layer in parallel.

For the speech recognition networks the best performance is achieved when 5 modules are assigned to the hidden layer and 3 to the output layer. The performance of the 1024 hidden units network is 4% higher than for the non-pipelined version. This is less than for the other applications. One explanation for this is that large networks do not obtain as even load balance between each layer as smaller networks due to larger computation grain level.

Figures 9.11 and 9.12 are combined in Figure 9.13 to compare training of the two different applications. Both applications have a maximum performance of approximately 15 MCUPS.

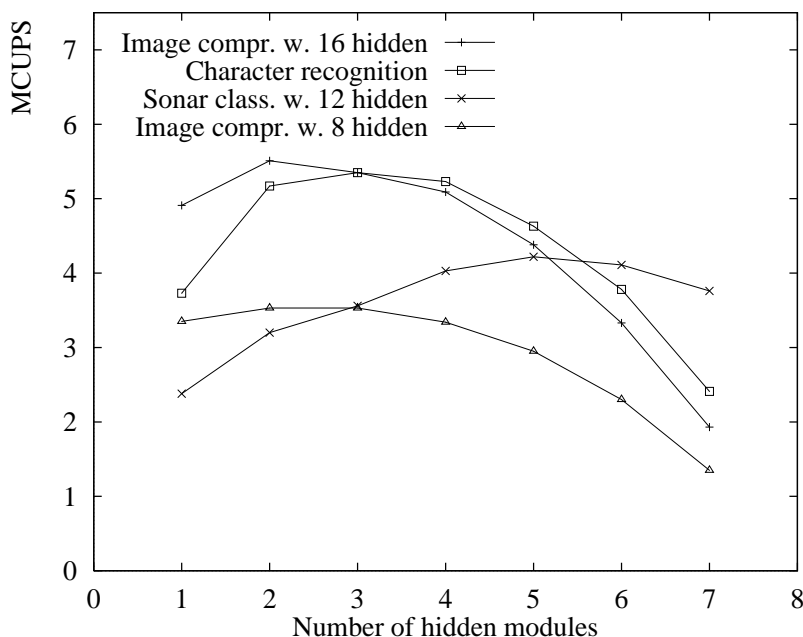


Figure 9.14: Pipelined training of small networks on 8 modules.

For small networks like image compression (Figure 9.14), the best assignments of modules have $C_h < C_o$. Only 2 out of 8 modules are used for the hidden layer in the best case. Except for the image compression network with 8 hidden units, the pipelining method trains faster than the neuron parallel method and the difference is largest for sonar target classification with 24% higher performance.

Figure 9.15 and 9.16 show the pipelined training on four processing modules for large networks and small networks, respectively. For most of the networks, the performance is lower for the pipelined mapping than for the neuron parallel mapping. This may arise for several reasons. On 4 processors, there is little communication and plain neuron parallelism can run efficiently. Further, the partitioning of the feed-forward network may not obtain proper computation load balance. This can either be due to the size of the network (it is not the largest networks that get the highest performance in Figure 9.15) or because the network contains uneven number of neurons in each layer (some of the graphs are linearly increasing in Figure 9.15).

For small networks the training speed for sonar target classification is 35 % higher for pipelining than for neuron parallelism. The other small networks train slower. While all the small applications for neuron parallelism trains slower on 8 modules than on 4 modules, this only apply to the sonar classification for pipelined training. This indicates the strength of the pipelined mapping when the number of processors increases.

Figure 9.17 presents the performance on 15 processing modules. Compared to Figure 9.13

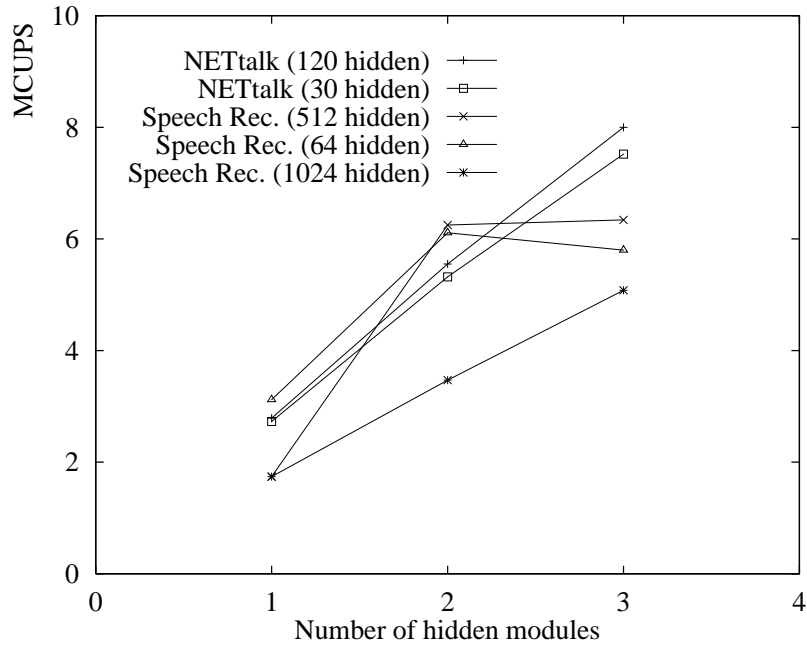


Figure 9.15: Pipelined training of NETtalk and speech recognition networks on 4 modules.

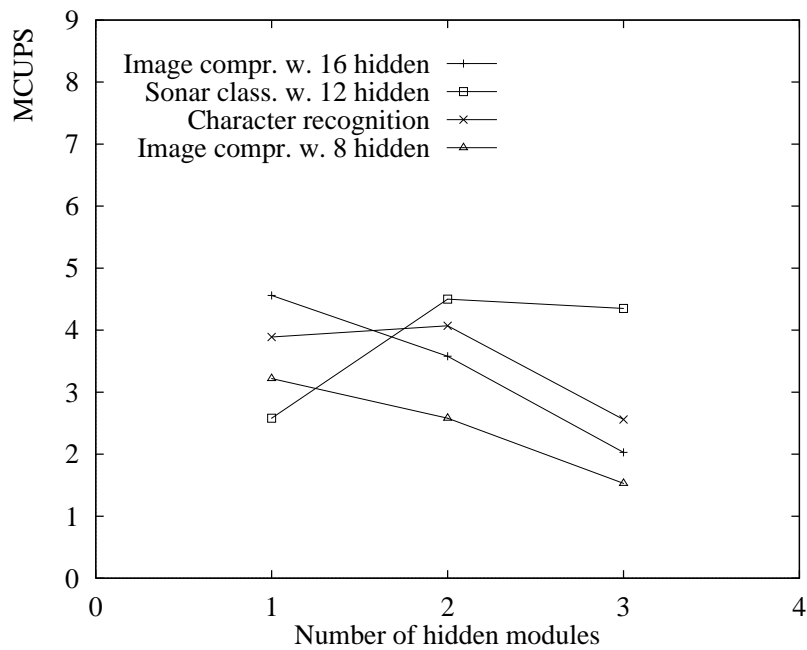


Figure 9.16: Pipelined training of small networks on 4 modules.

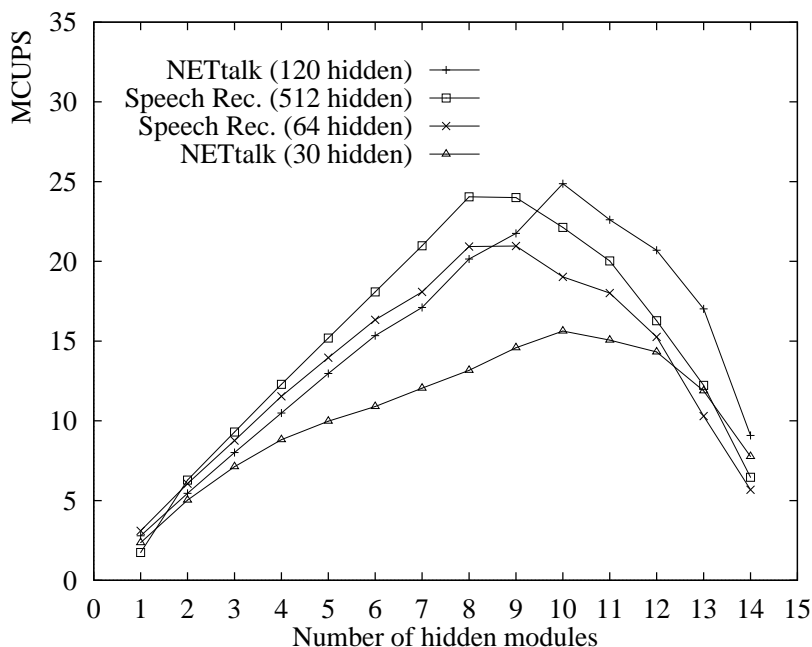


Figure 9.17: Pipelined training of NETtalk and speech recognition networks on 15 modules.

the number of modules assigned to the hidden layer versus to the output layer for the best performance is less for 15 modules. For example for NETtalk with 120 hidden units, twice (10/5) as many modules are assigned to the hidden layer as to the output layer using 15 modules, while this number is three (6/2) for 8 modules. This can be explained by the output layer spending more time on communication. Thus, a larger number of modules is needed to reduce the computation time to compensate for the added communication time.

Figure 9.18 shows the best training speed for four different networks as a function of the total number of modules ($C_o + C_h$). That is, the performance of the pipelined configuration with highest training speed for each total number of modules is plotted in the figure.

Except for NETtalk with 30 hidden units the scaling is promising and better than for only neuron parallelism, as seen in Figure 9.19. The difference in performance between the two implementations increases as the number of modules increases. The results emphasize the importance of including pipelining for a large number of modules. On 15 modules, pipelined training of NETtalk network for 120 hidden units is 131% faster than non-pipelined training. The difference was only 29% for 8 modules.

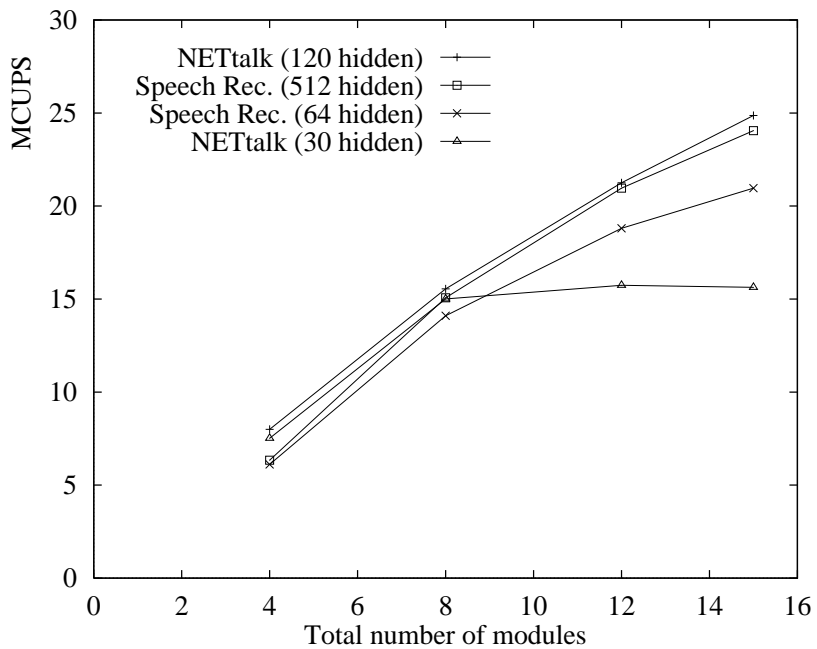


Figure 9.18: Maximum pipelined training speed.

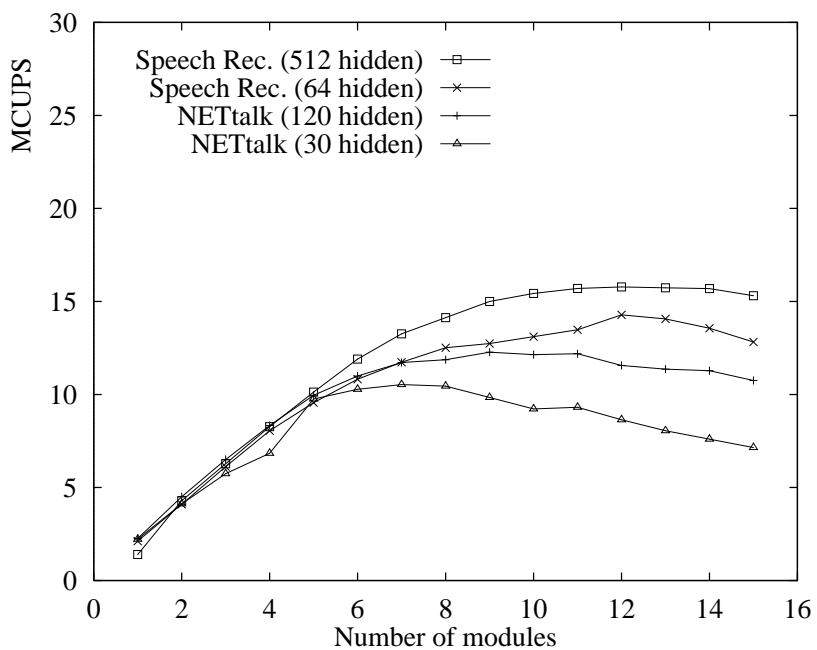


Figure 9.19: Neuron parallel training speed.

9.5 Summary

A description of two mappings of BP on the reconfigurable neurocomputer RENNS has been given. The architecture makes it possible to connect the modules by one or more ring buses in a reconfigurable way. A method has been proposed which divides the processors into two groups. Each of which computes different parts of the backpropagation training algorithm.

For a small number of processors the partitioning of the computation of the different layers onto different subsets of the PEs did not significantly improve the performance for 4 PEs. For 8 modules the performance was on average improved, e.g. NETtalk with 120 hidden neurons trained 29% faster.

The performance was tested for up to 15 modules. The training speed for the pipelining method was monotonely increasing for the NETtalk (except for 30 hidden units) and speech recognition networks. The scaling was much better than for the non-pipelined method.

In conclusion, it has been shown that the pipelined partitioning is beneficial for BP networks of different sizes if more than 4 processors are used.

Chapter 10

Concluding Remarks and Further Work

This thesis reports the results from implementing the back propagation training algorithm on a large parallel computer containing 512 processing elements (AP1000) and on a small parallel computer of 16 processing elements (RENNS). The survey of neural applications indicates that much work is based on *small* neural networks. Thus, as the number of processors available in a parallel system becomes larger, the demand on the mapping algorithm becomes higher. The main goal of this work has been to design an *application adaptable* mapping that trains different kinds of neural applications efficiently.

10.1 AP1000 Implementation

First, some fixed implementations are compared. The speedup is very limited if the implementation is based only on a single degree of BP parallelism. Better performance figures are achieved when multiple degrees of parallelism are combined. Including pipelining for some networks improves the performance. However, the computation load balance between the layers is shown to highly influence the training speed.

Second, a flexible mapping heuristic is proposed, of which a subset is implemented. It is shown that various neural network applications require a specific configuration to run at the highest possible training speed. This emphasizes the importance of flexibility and adaptability of the implementation.

Estimation of execution time was undertaken for the flexible implementation. For the NETtalk application, the estimates turned out to be accurate and estimates were used to show that the number of processors for the flexible mapping can be increased almost without limit resulting in increased performance.

Results show that training speed can be drastically increased by using parallel processing. However, the effect of the weight update interval on the convergence for the two applications tested (NETtalk and sonar target classification) reduce the apparent obtainable speedup.

Even though a general purpose parallel computer is not able to optimize the hardware in the same way as special purpose hardware, experiments undertaken show that substantial reduction in total training time is possible compared to single processor machines.

10.2 RENNS Implementation

Two different BP mapping schemes were compared on RENNS. One is based exclusively on neuron parallelism, another combines neuron and layer parallelism. The latter pipelines the training patterns from hidden layer processors to output layer processors. The error is propagated back to the hidden layer processors. When more than 4 processors were employed, the pipelined mapping performed faster, with a few exceptions, than the non-pipelined. Therefore, the partitioning of the computation in neural layers lead to a better utilization of the processor resources.

10.3 General Comments

The difference in the best performance is generally larger for the pipelined versus the non-pipelined implementation on RENNS compared to for the 3APC versus the 2APC implementation on AP1000. The major reason for this is that for RENNS, the number of processing elements assigned to each layer is flexible, while for AP1000 it is equal for each layer. The results show that the pipelining scheme 3APC suffered from unequal load balance for some networks and training sets. This becomes amplified in the RENNS scheme, where we see that for most experiments the best performance is given by $C_h \neq C_o$. According to the results from implementing the flexible implementation on RENNS, it would have been interesting to have used a similar approach on AP1000.

This improved implementation for the AP1000 is described in Section 7.1.2. Efficiency is dependent on the overhead generated by network contention, which was one of the reasons that the mapping was not implemented in the first place. Since the fixed pipelined mapping for many experiments performed faster than the non-pipelined method, a flexible pipelined mapping on AP1000 could possibly be faster in general than a non-pipelined mapping. Combined with the flexible implementation (Section 7.1.1) this would be a promising next step in making an optimal *application adaptable* mapping.

10.4 Future Work

During this study it was not possible to investigate all the topics that arose. Thus, the following list are the omissions and recommendations for further work:

- To fully exploit the parallel machine, the proposed mappings should be fully implemented. It was estimated that the code for AP1000 could run nearly three times faster in the best case. Thus, effort should be made on writing more efficient code.
- Further experiments should be undertaken using other applications to verify the relation between the weight update interval and the number of iterations needed for convergence. Also, more investigation should be made to determine if the initial training can be used in general for estimating the convergence.
- Test if it is favorable to change the weight update interval during training.
- Estimation of execution time should be studied for other applications to verify that the method can be used in the heuristic which selects the appropriate mapping.
- Test if a delayed weight update affects the convergence.
- The memory requirements could be studied to see how the size of the memory needed on each PE is reduced for larger systems.
- The utilization of the target architecture could be studied by comparing the FLOPS number given by the manufacturer to the FLOPS measured during program execution.
- Parallel processing of preprocessing techniques could be investigated
- Extend the parallelization schemes for other neural networks e.g. not fully connected networks and recurrent networks.
- Determine if better results (e.g. recognition rate) can be obtained by using more hidden layers, larger number of neurons per layer and larger training set. If this is possible, it will emphasize the importance of using parallel processing of neural network computation.

Appendix A

Source code

This appendix contains the source code for the implementation described in Section 7.1.1. Due to lack of space/similarity of programs, source code for the other implemented programs have not been included here.

The host and cell programs are contained in the following files:

<code>Makefile</code>	Used to compile the programs
<code>global.defs.bp</code>	Training parameter file
<code>hostmix.c</code>	Host program file
<code>cellhmix.h</code>	Header file for cell program
<code>cmainmix.c</code>	Cell program file
<code>trainmix.c</code>	Cell program file
<code>forwrdmix.c</code>	Cell program file
<code>backwrdmix.c</code>	Cell program file

Makefile

```
#Makefile for mix of neuron and training set parallelism
dummy: ap cell;

ap:      hostmix.c global.defs.bp;
        cc.hc7 -o ap -O3 hostmix.c

cell:    cmainmix.o trainmix.o forwrdmix.o backwrdmix.o;
        cc.cc7 -stack 128 -o cell -O3 cmainmix.o trainmix.o forwrdmix.o
        backwrdmix.o -lm

cmainmix.o:    cmainmix.c trainmix.o cellhmix.h;
               cc.cc7 -c -O3 cmainmix.c

trainmix.o:    trainmix.c forwrdmix.o backwrdmix.o cellhmix.h;
               cc.cc7 -c -O3 trainmix.c

forwrdmix.o:   forwrdmix.c cellhmix.h global.defs.bp;
               cc.cc7 -c -O3 forwrdmix.c

backwrdmix.o:  backwrdmix.c cellhmix.h global.defs.bp ;
               cc.cc7 -c -O3 backwrdmix.c
```

global.defs.bp

```

/* The file contains constant definitions for the back propagation training
   for mix of neuron and training set parallelism.
*/

/* Definitions for the training set 26x5x8char.ascii
   Common for both cell and host
*/

/* To change number of cells: Change NUM_PROC_X and NUM_PROC_Y

   To change application:
       Change NUM_INPUTS, NUM_HIDDEN_REAL, NUM_OUTPUTS_REAL
       Change NUM_PAT_TOT. Also change host and cell program.
*/

/* Change wupd: PAT_BETWEEN_WUP, LEARNING_RATE, ALPHA_INIT, MAX_IT
*/

#define NUM_PROC_X 6  /* Number of processors in x-dimension – horisontal
                       MUST BE EVEN number */
#define NUM_PROC_Y 8  /* Number of processors in y-dimension */

#define NUM_NP_X 3

#define NUM_INPUTS 40
#define NUM_HIDDEN_REAL 30
#define NUM_OUTPUTS_REAL 26

#define NUM_PAT_TOT 26

#define LEARNING_RATE 0.20
#define ALPHA 0.0
#define MAX_IT 1000
#define PAT_BETWEEN_WUP 26
                                     /* Training pattern file */
#define PATTERN_FILE "../trainpat/26x5x8char.ascii"

#define TIMING_ON 0  /* TIMING_ON= 0 The alg. run until convergence
                       or MAX_IT
                       TIMING_ON= 1 Continue training untill MAX_IT */
#define PRINT_ERROR 1 /* If active, the number of pattern not

```

*properly trained is printed for every iteration */*

hostmix.c

```

/* Main host program for back propagation training.
   Implementation mix of training pattern and neuron parallelism
   This version allows to vary the amount of training set parallelism
   in the x-dimension, in addition to the y-dimension. */

/* To change number of processors:
   Change #define NUM_PROC_X
   #define NUM_PROC_Y
*/

/* To change application:
   Change NUM_INPUTS, NUM_HIDDEN_REAL and NUM_OUTPUTS_REAL,
   NUM_PATTERNS and filename
   Also change cell program
*/

#include<stdio.h>
#include<chost.c7.h>
#include"global.defs.bp"
#define PROGRAMME "cell"      /* Cell program name */
#define TID 30                /* task ID */

#define NUM_TSP_X (NUM_PROC_X/NUM_NP_X)
#define NUM_PROC (NUM_PROC_X * NUM_PROC_Y)

#define USTAT_MAX 8
#define NULL 0
#define MAX_FILENAME_LENGTH 30

#define NUM_OUTPUTS_CALC
      (((NUM_OUTPUTS_REAL - 1)/(NUM_NP_X) + 1) * NUM_NP_X)

float x[NUM_PAT_TOT][NUM_INPUTS]; /* Input vector */

float d[NUM_PAT_TOT][NUM_OUTPUTS_CALC]; /* Target vector */

extern double dgettime();
double t1, t2, time;

read_network_files(argc, argv) /* Read network training parameters */
int argc; char *argv[];

```

```

{
  int c;
  char netfile[MAX_FILENAME_LENGTH];
  FILE *fp;

  struct learn_parameters {
    float ny;
    float alpha;
    int wupd;
  };
  struct learn_parameters lp;
  lp.ny= LEARNING_RATE;
  lp.alpha= ALPHA;
  lp.wupd= PAT_BETWEEN_WUP;
  while (--argc > 0 && **++argv == '-') {
    c = **++argv[0];
    --argc;
    switch (c) {
      case 'l':
        lp.ny = (float) atof(**++argv);
        break;
      case 'a':
        lp.alpha = (float) atof(**++argv);
        break;
      case 'w':
        lp.wupd = (float) atoi(**++argv);
        break;
      case 'h':
        printf("Usage: bp [-l learning rate] [-a alpha]
                [-w # patt_betw_weight_update] [network_file]\n");
        exit(0);
        break;
      default:
        printf("bp: illegal option %c\n",c);
        break;
    }
  }
  if (argc != 1){
    printf("No netfile specified, definitions in 'global.defs.bp' is used\n");
  }
  else {
    strcpy(netfile, *argv);
    if ((fp= fopen(netfile, "r")) == NULL) {

```

```

        printf("bp: can't open %s\n", netfile);
        exit(1);
    } else {
        printf("Reading parameter settings from: %s\n",netfile);
        /* To be extended */
    }
}
printf("Ny (learning rate)= %f\n",lp.ny);
printf("Alpha (moment term)= %f\n",lp.alpha);
printf("Weights upd for every %d pattern\n",lp.wupd);

cbroad(30, 0, &lp, sizeof(lp));
}

check_parameters() /* Check that the configuration is valid */
{
    int tspno;
    int nrp= NUM_PAT_TOT;
    int ncelx= NUM_PROC_X;
    int npx= NUM_NP_X;

    tspno= NUM_TSP_X * NUM_PROC_Y;

    if (tspno > nrp) {
        printf("conflmix2bp: More training set groups than cells\n");
        exit(1);
    }

    if ((ncelx % npx)!= 0) {
        printf("conflmix2bp: Mismatch for training set par. in x-dimension\n");
        exit(1);
    }
}

void reset_array()
/* Reset the target output d. */
{
    int nri, nro, nrp;
    int p, i,k;

    nro = NUM_OUTPUTS_CALC;
    nrp = NUM_PAT_TOT;

```

```

    for (p=0; p< nrp; p++) {
        for (k=0; k<nro; k++)
            d[p][k]= 0.0;
    }
}

void readpatt(pattfile) /* Read character patterns from ascii file */
char *pattfile;
{
    FILE *fp;

    int c;

    int k,j,i;

    if ((fp= fopen(pattfile, "r")) == NULL) {
        printf("fileread: can't open %s\n", pattfile);
        exit(1);
    } else {

        for (k=0; k< NUM_PAT_TOT; k++) {
            c = 0;
            i= 0;
            while ((c != '\n') && (c != EOF)){ /* Remove the character number */
                i++;
                c = getc(fp);
            }
            for (i=0; i< NUM_INPUTS; i++) {
                c= getc(fp);

                if (c == '\n') {
                    /* True if multiline pattern is used and removes newline*/
                    c= getc(fp);
                }
                if (c== EOF) {
                    printf("fileread: file is too short\n");
                    exit(1);
                }
                if (c == '-') {
                    x[k][i]= 0.0;

```

```

    }
    else if (c == '#') {
        x[k][i]= 1.0;
    }
    else printf("Error in file, char= %d\n", c);
}
getc(fp); /* Remove the newline character after each pattern*/
d[k][k%NUM_OUTPUTS_REAL]= 1.0; /* One output is set high */
}
}
fclose(fp);

}

host_main(argc, argv)
int argc;
char *argv[];
{
    char *filename = PATTERN_FILE;
    int it,tspno;

    double nw;
    float cups;

    double nri = NUM_INPUTS;
    double nrh = NUM_HIDDEN_REAL;
    double nro = NUM_OUTPUTS_REAL;
    double nrp = NUM_PAT_TOT;

    printf("** Mixed parallelism: Neuron & training set\n");
    printf("nri= %d\nnrh= %d\nnro= %d\nnrpatt= %d\n",(int) nri,
        (int) nrh, (int)nro, (int) nrp);
    nw= (nri + nro) * nrh * nrp;
    printf("# Weights-upd/it= %e\n",nw);
    printf("NUM_NP_X= %d\n",NUM_NP_X);

    check_parameters();

    /* Set up a x,y cell configuration */
    cconfxy(NUM_PROC_X,NUM_PROC_Y);

    /* Creates a task with task TID from the cell program module PROGNAME
       in the host */

```

```
ccreat(PROGNAME,TID,NULL);

read_network_files(argc, argv);

reset_array(); /* Reset d[][] */

/* Read training set from file and broadcast to cells */
readpatt(filename);

printf("Scatter training set (x) to cells\n");
h_sscatter(TID, x, sizeof(x));

printf("Scatter training set (d) to cells\n");
h_sscatter(TID, d, sizeof(d));

printf("BP starts\n");
initstat(1);
it = MAX_IT;
t1= dgettime();

while (cstat() == 1) {

}
t2= dgettime();

time= t2 - t1;
printf("Time %dit training= %f s\n", it, time);
printf("Time 1 it training= %f s\n", time/it);

cups= ((float) nw * it)/ time;
printf("CUPS= %f\n", cups);
}
```

cellhmix.h

```

/* This is a header file for the back propagation training
   for mix of neuron and training set parallelism.
   This version allows to vary the amount of training set parallelism
   in the x-dimension, in addition to the y-dimension.*/

/* To change number of procs: Change NUM_PROC_X and NUM_PROC_Y */

/* To change network: Change NUM_INPUTS, NUM_HIDDEN_REAL,
   NUM_OUTPUTS_REAL
   Change NUM_PAT_TOT. Also change host program*/

#define NUM_TSP_X (NUM_PROC_X/NUM_NP_X)

#define NUM_PROC (NUM_PROC_X * NUM_PROC_Y)

#define NUM_PATTERNS
    ((NUM_PAT_TOT - 1)/(NUM_PROC_Y * NUM_TSP_X) + 1)

#define NUM_HIDDEN
    (((NUM_HIDDEN_REAL - 1)/(NUM_NP_X) + 1) * NUM_NP_X)
#define NUM_OUTPUTS
    (((NUM_OUTPUTS_REAL - 1)/(NUM_NP_X) + 1) * NUM_NP_X)

#define NUMBER_OF_ROW_IN_ONE_PROC_KR_H1 NUM_HIDDEN/NUM_NP_X
#define NUMBER_OF_ROW_IN_ONE_PROC_KR_O NUM_OUTPUTS/NUM_NP_X
/* The above assignment leads to an equal number of neurons in each cell. */

extern int tid;
extern int cid;
extern int ncidx;
extern int ncidy;
extern int ncel;
extern int ncelx;
extern int ncely;

extern int xperc, hperc, yperc;
extern int npx, tspx;

extern train();
extern calc_yh1();
extern calc_yo();

```

```
extern float calc_delth();
extern adjust_weights();

/* Weight matrix for hidden layer 1 */
extern float wh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

extern float dwh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

/* Weight matrix for output layer*/
extern float wo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];

extern float dwo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];

/* Input vector */
extern float x[NUM_PATTERNS * NUM_INPUTS];

/* Target vector */
extern float d[NUM_PATTERNS][NUMBER_OF_ROW_IN_ONE_PROC_KR_O];

/* Output vector from hidden layer*/
extern float yh1[NUM_HIDDEN];

/* Output vector */
extern float yo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O];

/* Delta vectors */
extern float delth1[NUM_HIDDEN];

extern float delto[NUMBER_OF_ROW_IN_ONE_PROC_KR_O];
```

cmainmix.c

```

/* Main cell program for back propagation.
   Implementation of training pattern and neuron parallelism.
   This version allows to vary the amount of training set parallelism
   in the x-dimension, in addition to the y-dimension.
*/

#include<stdio.h>
#include<ccell.c7.h>
#include"global.defs.bp"
#include"cellhmix.h"

int ncel;          /* The total number of cells */
int ncelx;        /* The number of x-cell */
int ncely;        /* The number of y-cell */
int cid;          /* ID of running cell */
int ncidx;        /* ID of running x-cell */
int ncidy;        /* ID of running y-cell */
int tid;          /* ID of running task */

int xperc, hperc, yperc;
int npx = NUM_NP_X;
int tspx = NUM_TSP_X;
float alpha, ny;
int pat_betw_wupd;

/* Weight matrix hidden layer 1 */
float wh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

float dwh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

/* Weight matrix output layer */
float wo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];

float dwo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];

/* Input vector */
float x[NUM_PATTERNS * NUM_INPUTS];

/* Target vector */

```

```

float d[NUM_PATTERNS][NUMBER_OF_ROW_IN_ONE_PROC_KR_O];

/* Output vector from hidden layer*/
float yh1[NUM_HIDDEN];

/* Output vector */
float yo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O];

/* Delta vectors */
float delth1[NUM_HIDDEN];

float delto[NUMBER_OF_ROW_IN_ONE_PROC_KR_O];

read_parameters() /* Read network training parameters from the host */
{
    struct learn_parameters {
        float ny;
        float alpha;
        int wupd;
    };
    struct learn_parameters lp;
    h_recv(0);
    readmsg(&lp, sizeof(lp));

    ny= lp.ny;
    alpha= lp.alpha;
    pat_betw_wupd= lp.wupd;
    printf("ny= %f\n", ny);
    printf("alpha= %f\n", alpha);
    printf("pat_betw_wupd= %d\n", pat_betw_wupd);
}

read_train_patt(pmax) /* Read training patterns from the host */
int pmax;
/* Patterns are scattered one to each row of cells. Each x vector is sent
to all cells in a row while output vector d is partitioned and each part
is sent to a unique cell. */
{
    int i,j,k,iter, no;
    int nrproc, nri, nro, nrpattot;
    int xinit, xskip, xsize, xspace, yinit, yskip, ysize, yspace;

```

```
float sizepatt;

nrproc= NUM_PROC;
nri = NUM_INPUTS;
nro = NUM_OUTPUTS;
nrpattot= NUM_PAT_TOT;

sizepatt= nri * sizeof(float);

/* Reading training input patterns from the host */
xinit= 0;
xskip= 0;
xsize= sizepatt;
xspace= sizepatt;
yinit= ncidy * tspx + ncidx/npix;
yskip= ncely * tspx - 1;
ysize= 1;
yspace= nrpattot;

h_rscatter(x, xinit, xskip, xsize, xspace, yinit, yskip, ysize, yspace);

printf(" No pattern= %d\n No of inputs in cell= %d\n",pmax, nri);

xperc= (NUM_INPUTS);
hperc = (NUM_HIDDEN - 1)/(npix) + 1;
yperc = (NUM_OUTPUTS - 1)/(npix) + 1;
printf("xperc= %d\n", xperc);
printf("hperc= %d\n", hperc);
printf("yperc= %d\n", yperc);

/* Reading target output patterns from host */
xinit= (ncidx % npix)*yperc*sizeof(float);
xskip= (nro - yperc)*sizeof(float);
xsize= yperc*sizeof(float);
xspace= nro*sizeof(float);
yinit= ncidy * tspx + ncidx/npix;
yskip= ncely * tspx - 1;
ysize= 1;
yspace= nrpattot;

h_rscatter(d, xinit, xskip, xsize, xspace, yinit, yskip, ysize, yspace);
}
```

```

cell_main() /* First level BP training function for cell */
{
    int modpatt, pattmax;
    int ntsp;

    printf("Back-propagation\n");

    /* Get cell ID & no of cells */
    ncel = getncel(); /* Get total no of cells */
    ncelx = getncelx(); /* Get no of cells for x-dimension */
    ncely = getncely(); /* Get no of cells for y-dimension */
    cid = getcid(); /* Get the own cell ID */
    tid = gettid(); /* Get the own task ID */
    ncidx = getcidx(); /* Get the own x-cell ID */
    ncidy = getcidy(); /* Get the own y-cell ID */

    ntsp= ncely * tspx;

    printf("Node_par_x= %d\nTSP_x= %d\n\n",npx, tspx);

    modpatt= NUM_PAT_TOT % ntsp;
    if (modpatt && (ncidy * tspx + ncidx/npx) < modpatt)
        pattmax= NUM_PAT_TOT/ntsp + 1;
    else
        pattmax= NUM_PAT_TOT/ntsp;

    printf("Pattmax= %d\n", pattmax);

    read_parameters();
    read_train_patt(pattmax); /* Read training set from host */

    printf("BP training starts\n");
    train(ny,alpha,pat_betw_wupd,pattmax, ntsp); /* BP training algorithm */
}

```

trainmix.c

```

/* The file contains functions to do back propagation training for mix of
   neuron and training set parallelism. This version allows to vary the
   amount of training set parallelism in the x-dimension, in addition to
   the y-dimension.
*/

#define USTAT_MAX 8
#include<stdio.h>
#include<ccell.c7.h>
#include<math.h>
#include"global.defs.bp"
#include"cellhmix.h"

int nsumstep;

unsigned long int n1,n2, n3, next;
int xinit, xskip, xsize, xspace, yinit, yskip, ysize, yspace;

float upd_dwh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

float upd_dwo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];

float random()
/* Returns a random float in [-0.5,0.5> */
{
    float rfl;

    next = next*112411 + 3777271;
    rfl = ( (float) (next % 32011)) / 32011.0) - 0.5;
    return rfl;
}

initialize_w()
/* Initialize weight matrix */
{
    int i,j;
    for (j=0; j < hperc; j++)
        for (i=0; i < (NUM_INPUTS); i++) {
            wh1[j][i]= random();
        }
}

```

```
    for (j=0; j < yperc; j++)
        for (i=0; i < (NUM_HIDDEN); i++) {
            wo[j][i]= random();
        }
}

int find_nsum() /* Determine the number of weight summing steps */
{
    int nosum;
    switch (ncely) {
        case 2: nosum= 0; break;
        case 4: nosum= 1; break;
        case 8: nosum= 2; break;
        case 16: nosum= 3; break;
        case 32: nosum= 4; break;
        default:
            printf("Not valid ncely for weight updating function\n");
            exit(1);
            break;
    }
    return nosum;
}

int power(base, n)
int base, n;
{
    int i,p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p= p * base;
    return p;
}

reset_dw() /* Reset the weight change matrices */
{
    int i,j,k;

    for (k=0; k < yperc; k++) {
        for (j=0; j < (NUM_HIDDEN); j++)
```

```

        dwo[k][j]= 0.0;
    }

    for (j=0; j < hperc; j++)
        for (i=0; i < (NUM_INPUTS); i++) {
            dwh1[j][i]= 0.0;
        }
}

update_w_alt(dwh1p,wh1p,dwop,wop,ny,alpha) /* Update the weights */
/* The weight change matrices are first summed in vertical dimension. Then
if tspan > 1, the weight change matrices are summed in the horizontal
dimension.
*/

float *dwh1p, *wh1p, *dwop, *wop;
float ny, alpha;
{
    int i,j,k,t,tpe, sdir, rdir, sind, rind;
    int index, nrh= NUM_HIDDEN, nri= NUM_INPUTS;
    int noproc = NUM_PROC_Y;
    int windx;

    int q= NUM_NP_X;

    float acc_dwo[NUMBER_OF_ROW_IN_ONE_PROC_KR_O][NUM_HIDDEN];
    float acc_dwh1[NUMBER_OF_ROW_IN_ONE_PROC_KR_H1][NUM_INPUTS];

    float *acc_dwop;
    float *acc_dwh1p;
    acc_dwop= &acc_dwo[0][0];
    acc_dwh1p= &acc_dwh1[0][0];

    for (k=0; k < yperc; k++) { /* OUTPUT layer */
        windx= k*nrh;
        for (j=0; j < nrh; j++) {
            acc_dwo[k][j]= *(dwop + windx + j);
        }
    }

    for (j=0; j < hperc; j++) { /* HIDDEN layer */
        windx= j*nri;

```

```

    for (i=0; i < nri; i++) {
        acc_dwh1[j][i]= *(dwh1p + windx + i);
    }
}

/* Determine if the present cell is to send (vertically) weight change matrix */
for (t= 0; t <= nsumstep; t++) {
    tpe= t+3;

    if (((ncidy - (power(2,t))) % (power(2,t+1))) == 0) {
        sind= ncidy - power(2,t);
        r_asend(ncidx, sind, tid, tpe, acc_dwo, nrh*yperc*sizeof(float));
    }
    else
        if (((ncidy - (power(2,t)-1)) % (power(2,t+1))) == 0) {
            sind= ncidy + power(2,t);
            r_asend(ncidx, sind, tid, t, acc_dwh1, nri*hperc*sizeof(float));
        }
}

/* Determine if the present cell is to receive and sum weight change matrix. */
if ((ncidy % (power(2,t+1))) == 0) {
    rind= ncidy + power(2,t);
    r_arecv(ncidx,rind, tid,tpe);
    readmsg(dwop, nrh*yperc*sizeof(float));

    for (k=0; k < yperc; k++) {
        windx= k*nrh;
        for (j=0; j < nrh; j++) {
            *(acc_dwop + windx + j)= *(acc_dwop+windx+j) + *(dwop+windx+j);
        }
    }
}
else
    if (((ncidy - (power(2,t+1)-1)) % (power(2,t+1))) == 0) {
        rind= ncidy - power(2,t);
        r_arecv(ncidx,rind, tid,t);
        readmsg(dwh1p, nri*hperc*sizeof(float));

        for (j=0; j < hperc; j++) {
            windx= j*nri;
            for (i=0; i < nri; i++) {
                *(acc_dwh1p + windx + i)= *(acc_dwh1p+windx+i) + *(dwh1p+windx+i);
            }
        }
    }
}

```

```

    }
  }
}

/* Broadcast vertically, the summed weight change matrices, back to each cell */
y_brd(0, acc_dwo, nrh*yperc*sizeof(float)); /* Output weights */

y_brd(ncely-1, acc_dwh1, nri*hperc*sizeof(float)); /* Hidden weights */

/* Addressing for horizontal summing */
sdir = ncidx + npx;          /* Send to EASTern npx-block*/
if (sdir > (ncelx-1))
  sdir = sdir - ncelx;

rdir = ncidx - npx;          /* Receive from WESTern npx block */
if (rdir < 0)
  rdir = rdir + ncelx;

/* Horizontal summing of output weight change matrices */
for (j=0; j < nrh; j++) {
  for (k=0; k < yperc; k++) {
    *(dwop+k*nrh+j)= acc_dwo[k][j];
  }
}

for (t=1; t < tspx; t++)
{
  tpe= t+3*q;
  r_asend(sdir, ncidy, tid, tpe, dwop, nrh*yperc*sizeof(float));

  r_arecv(rdir, ncidy, tid,tpe);
  readmsg(dwop, nrh*yperc*sizeof(float));

  for (j=0; j < nrh; j++) {
    for (k=0; k < yperc; k++) {
      acc_dwo[k][j]= acc_dwo[k][j] + *(dwop+k*nrh+j);
    }
  }
}

/* Update output weights */
for (k=0; k < yperc; k++) {

```

```

    windx= k*nrh;
    for (j=0; j < nrh; j++) {
        upd_dwo[k][j]= alpha * upd_dwo[k][j] + (1.0 - alpha) * acc_dwo[k][j];
        *(wop + windx +j)= *(wop + windx +j) + ny * upd_dwo[k][j];
    }
}

/* Horizontal summing of hidden weight change matrices */
for (j=0; j < hperc; j++) {
    for (i=0; i < nri; i++) {
        *(dwh1p + j*nri +i)= acc_dwh1[j][i];
    }
}

for (t=1; t < tspx; t++)
{
    r_asend(sdir, ncidy, tid, t, dwh1p, nri*hperc*sizeof(float));

    r_arecv(rdir, ncidy, tid,t);
    readmsg(dwh1p, nri*hperc*sizeof(float));

    for (j=0; j < hperc; j++) {
        for (i=0; i < nri; i++) {
            acc_dwh1[j][i]= acc_dwh1[j][i] + *(dwh1p+j*nri+i);
        }
    }
}

/* Update hidden weights */
for (j=0; j < hperc; j++) {
    windx= j*nri;
    for (i=0; i < nri; i++) {
        upd_dwh1[j][i]= alpha * upd_dwh1[j][i]+(1.0-alpha)* acc_dwh1[j][i];
        *(wh1p + windx +i)= *(wh1p + windx +i) + ny * upd_dwh1[j][i];
    }
}

}

/* Second level BP training function for cell */
train(ny,alpha,pat_betw_wupd,pmax, ntsp)

float ny,alpha;

```

```

int pat_betw_wupd;
int pmax, ntsp;
{
    int it,tp,error;
    int pbw, pupdmax;
    int stat,trainend;
    float paterror;

    int j;

    pbw= pat_betw_wupd/ntsp;
    pupdmax= NUM_PAT_TOT/ntsp;
    printf("Pat between upd. in cells= %d\n",pbw);

    nsumstep= find_nsum();
    printf("Number of wupd summings= %d\n",nsumstep+1);

    /* Initializing weights */
    next= 1 + 13 * (ncidx % npx);
    /* Must have _the same_ initial weights in all cells in a column. */
    initialize_w();

    /* Training starts */
    it= 0;
    printf("ny= %f\nalpha= %f\n",ny, alpha);
    error = 0;
    trainend = 0;

    while (trainend == 0) {
        error = 0;
        reset_dw();
        for (tp=0;tp< pmax; tp++) {
            /* Calculate hidden layer */
            calc_yh1(wh1, x, yh1, tp, ny);

            /* Calculate output layer */
            calc_yo(wo, yh1, yo, tp, ny);

            /* Calculate delta term */
            if (calc_delth(d, yo, wo, yh1, delth1, delto, tp) > 0.1)
                error++;

            /* Accumulate weights */

```

```
    acc_weights(delth1,dwh1,delto,dwo, x, yh1,tp);

    /* Check if weights are to be updated */
    if (((tp % pbw) == 0) && (tp > 0) && (tp < pupdmax)){
        update_w_alt(dwh1,wh1,dwo,wo,ny,alpha);
        reset_dw();
    }
}

it++;

if ((error == 0 && !TIMING_ON) || it==MAX_IT)
    pstat(0);
else
    pstat(1);

/* Update weights */
update_w_alt(dwh1,wh1,dwo,wo,ny,alpha);

if(PRINT_ERROR)
    printf("It%d: #Err= %d \n", it, error);

if (cstat() == 0)
    trainend = 1;
}
}
```

forwrdmix.c

```
/* The file contains functions to do the forward phase of the BP training.
*/
```

```
#include<stdio.h>
#include<ccell.c7.h>
#include"global.defs.bp"
#include"cellhmix.h"
#include<math.h>
```

```
calc_yh1(wh1p, xp, yh1p, patnr, ny) /* Calculate hidden layer outputs */
```

```
float *wh1p, *xp, *yh1p;
```

```
int patnr;
```

```
float ny;
```

```
{
```

```
int i,j,k,index, yindx, xindx, pindx;
```

```
int nri = NUM_INPUTS;
```

```
k= (ncidx % npx);
```

```
for (j=0; j< hperc; j++) {
```

```
    yindx= k*hperc + j;
```

```
    *(yh1p+yindx)= 0.0;
```

```
    for (i=0; i < nri; i++) {
```

```
        index= i + nri*patnr;
```

```
        *(yh1p+yindx)= *(yh1p+yindx) + *(wh1p+j*nri+i) * (*(xp+index));
```

```
    }
```

```
    *(yh1p+yindx)= 1.0/(1.0 + exp(-*(yh1p+yindx)));
```

```
    }
```

```
}
```

```
calc_yo(wop, yh1p, yop, patnr, ny) /* Calculate output layer outputs*/
```

```
float *wop, *yh1p, *yop;
```

```
int patnr;
```

```
float ny;
```

```
{
```

```
int t,k,j,i, sdir,rdir, index, yindx, max_pnr;
```

```
int nrh;
```

```
    nrh = NUM_HIDDEN;
```

```
    max_pnr = npx;
```

```

if ((ncidx % npx) == (npx-1))
    sdir = ncidx - npx + 1;          /* Must rotate to the leftmost cell in npx*/
else                                /* = comm. conflict */
    sdir = ncidx + 1;              /* Send EAST */

if ((ncidx % npx) == 0)
    rdir = ncidx + npx - 1;        /* Must rotate from the rightmost cell */
else                                /* = comm. conflict */
    rdir = ncidx - 1;             /* Receive from WEST */

k = ncidx % npx;

for (j=0; j < yperc; j++) {
    *(yop+j) = 0.0;
    for (i=0; i < hperc; i++) {
        index = i + hperc*k;
        *(yop+j) = *(yop+j) + *(wop+j*nrh+i+hperc*k) * (*(yh1p+index));
    }
}

for (t=1; t < max_pnr; t++) /* Multiplies submatrix w with subvector x */
{
    index = hperc*k;
    r_asend(sdir, ncidy, tid, t + 1, yh1p+index, hperc*sizeof(float));

    k = (ncidx % npx) - t;
    if (k < 0)
        k = k + max_pnr;
    index = hperc*k;

    r_arecv(rdir, ncidy, tid, t + 1);
    readmsg(yh1p+index, hperc*sizeof(float));

    for (i=0; i < hperc; i++) {
        index = i + hperc*k;
        for (j=0; j < yperc; j++) {
            *(yop+j) = *(yop+j) + *(wop+j*nrh+i+hperc*k) * (*(yh1p+index));
        }
    }
}

k = ncidx % npx;

```

```
for (j=0; j< yperc; j++) {  
    *(yop+j)=1.0/(1.0+exp(-*(yop+j)));  
}  
}
```

backwrdmix.c

```

/* The file contains functions to do the backward phase of the BP training.
*/
#include<stdio.h>
#include<ccell.c7.h>
#include"global.defs.bp"
#include"cellhmix.h"

float calc_delth(dp, yop, wop, yh1p, delth1p, deltop, pattnr)
float *dp, *yop, *wop, *yh1p, *delth1p, *deltop;
int pattnr;
/* The function calculates the output error, delta hidden error, and delta
   output error. */
{
  int q,nri,nrh,nro,t,k,l,j, elemsize, index, tpe;
  int sdir, rdir;

  float diff, error, tot_err;
  float deltemp[NUM_HIDDEN/NUM_NP_X];

  q= NUM_NP_X;

  nri = NUM_INPUTS;
  nrh = NUM_HIDDEN;
  nro= NUM_OUTPUTS;

  /* Calculate sub results of deltah1 */
  error = 0.0;

  for (k=0; k < yperc; k++) {
    diff= *(dp + yperc*pattnr + k) - *(yop + k); /* Output error */
    *(deltop + k)= *(yop + k) * (1.0 - *(yop + k)) /* Delta output error */
    * diff;
    error= error + diff*diff;
  }

  tot_err = error;          /* Sum the output error for one pattern */

  if ((ncidx % npx) == (npx-1))
    sdir = ncidx - npx + 1;      /* Must rotate to the leftmost cell */
  else
    sdir= ncidx + 1;           /* = comm. conflict */
                                /* Send EAST */
}

```

```

if ((ncidx % npx) == 0)
    rdir = ncidx + npx - 1;          /* Must rotate from the rightmost cell */
else
    rdir = ncidx - 1;              /* = comm. conflict */
    /* Receive from WEST */

for (t=1; t < q; t++)
{
    r_asend(sdir, ncidy, tid, t+q, &error,sizeof(float));
    r_arecv(rdir, ncidy, tid, t+q);
    readmsg(&error, sizeof(float));
    tot_err = tot_err + error;
}

/* Delta hidden error calculation */
for (j=0; j < nrh; j++) {
    *(delth1p + j) = 0.0;
    for (k=0; k < yperc; k++) {
        *(delth1p + j) = *(delth1p + j) + *(wop + k*nrh + j) * *(deltop + k);
    }
}

/* Accumulate the sub-results of delta hidden error */
l = (ncidx % npx) - 1;
if (l < 0)
    l = l + q;

for (t=1; t < q; t++)
{
    tpe = t+2*q;
    r_asend(sdir, ncidy, tid, tpe, delth1p + l*hperc, hperc*sizeof(float));

    r_arecv(rdir, ncidy, tid, tpe);
    readmsg(deltemp, hperc*sizeof(float));

    l = (ncidx % npx) - t - 1;
    if (l < 0)
        l = l + q;

    for (j=0; j < hperc; j++)
        *(delth1p+l*hperc+j) = *(delth1p+l*hperc+j) + *(deltemp +j);
}

```

```

    for (j=0; j< hperc; j++){
        index= hperc*l + j;
        *(delth1p+index)= *(yh1p+index) * (1.0 - *(yh1p+index))*
            *(delth1p+index);
    }

    return (0.5*tot_err);
}

acc_weights(delth1p,dwh1p,deltop,dwop, xp, yh1p,pat)
    /* Accumulate weight changes */
float *delth1p, *dwh1p, *deltop, *dwop, *xp, *yh1p;
int pat;
{
    int i,j,k;
    int index, xindx, windx, rindx, dindx, nri = NUM_INPUTS, nrh= NUM_HIDDEN;

    for (k=0; k < yperc; k++) { /* Output layer */
        for (j=0; j < nrh; j++) {
            *(dwop + k*nrh +j)= *(dwop + k*nrh +j) + *(deltop+k) * *(yh1p+j);
        }
    }

    xindx= pat*nri;
    dindx= hperc * (ncidx % npx);
    for (j=0; j < hperc; j++){ /* Hidden layer */
        rindx= j*nri;
        for (i=0; i < nri; i++) {
            index= xindx + i;
            windx = rindx + i;
            *(dwh1p + windx)= *(dwh1p + windx) +
                *(delth1p + dindx + j) * *(xp + index);
        }
    }
}

```

26x5x8char.ascii

Character 1

```
-----  
--##--  
-#--#  
-#--#  
-####  
-#--#  
-#--#  
-----
```

Character 2

```
-----  
-###-  
-#--#  
-###-  
-#--#  
-#--#  
-###-  
-----
```

Character 3

```
-----  
--##--  
-#--#  
-#---  
-#---  
-#--#  
--##--  
-----
```

Character 4

```
-----  
-###-  
-#--#  
-#--#  
-#--#  
-#--#  
-###-  
-----
```

.
. .
.

Appendix B

Tables

This appendix contains tables with the results presented in Chapter 6, 8, and 9. To make reading easier, the table numbers have been adjusted to match the figure numbers in the different chapters.

B.1 Results for the Fixed Implementations On AP1000

Number of processors	NETtalk				
	NP	TSP-1	TSP-2	2APC	3APC ¹
1	0.240000	0.2287142	0.287685	0.240788	
2					0.369350
4	0.889753	1.135451	1.126180	1.018509	0.814868
9	1.727374	2.409358	2.393630	2.207790	1.644742
16	2.546687	3.508640	3.840101	3.843802	3.265587
25	2.507338	4.274064	5.245777	5.844287	4.792791
36	2.899817	4.312684	6.664512	7.871402	6.407494
49	3.078626	3.964320	8.052412	9.777560	9.238712
64	2.752892	3.477013	9.439815	13.228067	12.365695
Training speed (MCUPS)					

Table B.1: CUPS for the implemented algorithms running NETtalk with 80 hidden neurons. ¹The number of processors is not always as given in the leftmost column.

Number of processors	NETtalk	
	2APC	3APC
1	0.240790	
2		0.369358
64	13.665979	12.145721
256	40.666964	41.948244
512	78.626320	81.692512
Training speed (MCUPS)		

Table B.2: Performance for the two algorithms combining different parallel aspects, running NETtalk with 80 hidden neurons.

Weight update interval	NETtalk	
	2APC	3APC
64	38.117740	34.685548
128	52.541604	50.040576
256	64.821076	64.235356
512	73.373616	74.853440
1024	78.626320	81.692512
Training speed (MCUPS)		

Table B.3: Performance for different numbers of weight updates per iteration in the NETtalk experiment on a 512 cells system.

Number of processors	NETtalk			
	2APC (60 hid.)	2APC (120 hid.)	3APC (60 hid.)	3APC (120 hid.)
64	12.523283	13.830035	11.517567	12.560177
256	35.851444	41.773228	34.104088	44.040596
512	69.490552	81.464344	64.848048	85.500464
Training speed (MCUPS)				

Table B.4: Performance for the 2APC and 3 APC algorithms for the NETtalk network with 60 and 120 hidden neurons.

Number of hidden units	NETtalk	
	2APC	3APC
30	10.746275	10.907035
60	12.523283	11.517567
120	13.830035	12.560177
180	13.019945	12.537019
240	13.301820	12.633351
Training speed (MCUPS)		

Table B.5: NETtalk running on 64 cells.

Number of hidden units	NETtalk	
	2APC	3APC
30	27.256988	17.554100
60	35.851444	34.104088
120	41.773228	44.040596
180	44.746932	46.557576
240	48.389036	47.914476
Training speed (MCUPS)		

Table B.6: NETtalk running on 256 cells.

Number of hidden units	NETtalk	
	2APC	3APC
30	52.970436	35.659968
60	69.490552	64.848048
120	81.464344	85.500464
180	84.944208	88.285032
240	91.836424	92.276456
Training speed (MCUPS)		

Table B.7: NETtalk running on 512 cells.

Number of processors	Large feed-forward network		
	NP	2APC	3APC
64	14.961165	15.051761	9.066978
256	30.958746	56.187252	41.313628
512		95.124992	73.763992
Training speed (MCUPS)			

Table B.10: Performance for a large MLP network with 1024 input, 512 hidden and 64 output neurons.

Number of processors	Character Recognition		
	NP	2APC	3APC ²
1	0.174257	0.183174	
2			0.308264
4	0.463464	0.685887	0.629750
9	0.590519	1.282996	1.185694
16	0.648562	1.906802	1.743034
25	0.527041	2.270126	2.359148
36	0.547576	2.747519	2.775919
49	0.490397	3.031005	3.149996
64	0.430708	4.018293	2.925465
Training speed (MCUPS)			

Table B.11: Performance for a character recognition network. ²The number of processors is not always as given in the leftmost column.

Number of hidden units	Speech Recognition	
	2APC	3APC
64	13.330362	13.569155
256	13.747552	13.947401
512	13.618308	13.241242
1024	13.121106	12.618181
Training speed (MCUPS)		

Table B.12: Speech recognition network on 64 cells.

Number of hidden units	Speech Recognition	
	2APC	3APC
64	40.541604	39.055920
256	48.243500	52.598764
512	49.245232	52.721832
1024	49.222964	49.906364
Training speed (MCUPS)		

Table B.13: Speech recognition network on 256 cells.

Number of hidden units	Speech Recognition	
	2APC	3APC
64	74.797400	70.901528
256	87.327464	93.332224
512	89.818200	92.516624
1024	89.021744	89.704984
Training speed (MCUPS)		

Table B.14: Speech recognition network on 512 cells.

Number of processors	Speech Recognition			
	64 hidden	256 hidden	512 hidden	1024 hidden
64	13.330362	13.747552	13.618308	13.121106
256	40.541604	48.243500	49.245232	49.222964
512	74.797400	87.327464	89.818200	89.021744
Training speed (MCUPS)				

Table B.15: 2APC training speech recognition network.

Number of processors	Speech Recognition			
	64 hidden	256 hidden	512 hidden	1024 hidden
64	13.569155	13.947401	13.241242	12.618181
256	39.055920	52.598764	52.721832	49.906364
512	70.901528	93.332224	92.516624	89.704984
Training speed (MCUPS)				

Table B.16: 3APC training speech recognition network..

B.2 Results for the Flexible Implementations On AP1000

N_{NP}	N_{TSP}	NETtalk
		Weight update interval $\mu = 906$
1	16	3.540779
2	8	3.883230
4	4	3.456344
Training speed (MCUPS)		

Table B.1: MCUPS performance for different combinations of neuron and training set parallelism running the NETtalk application with 120 hidden neurons. 4 x 4 cell configuration for $\mu = 906$.

N_{NP}	N_{TSP}	NETtalk			
		$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 5438$
1	64	1.669289	4.951153	9.796948	14.148427
2	32	4.225292	9.381534	13.555270	15.729647
4	16	7.738127	12.337932	14.512598	15.363544
8	8	10.564511	12.996162	13.738988	14.008380
Training speed (MCUPS)					

Table B.2: MCUPS performance for different combinations of neuron and training set parallelism on a 8 x 8 cell configuration. $N_{NP} = 1, 2, 4,$ and 8.

N_{NP}	N_{TSP}	NETtalk			
		$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 5438$
2	128		12.425559	29.243890	52.659640
4	64	8.654276	24.019442	43.691348	58.609504
8	32	18.927082	37.244660	49.302208	54.835848
16	16	27.139308	37.432508	40.772984	42.133564
Training speed (MCUPS)					

Table B.3: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 cell configuration. $N_{NP} = 2, 4, 8,$ and 16.

N_{NP}	N_{TSP}	NETtalk			
		$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 5438$
4	128		28.189354	62.564108	105.407528
8	64	21.357012	53.304368	85.803440	105.490848
16	32	36.412524	62.393204	76.133720	81.044680
Training speed (MCUPS)					

Table B.4: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 cell configuration. $N_{NP} = 4, 8,$ and 16.

Number of processors	NETtalk			
	$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 5438$
64	10.564511	12.996162	14.512598	15.729647
256	27.139308	37.432508	49.302208	58.609504
512	36.412524	62.393204	85.803440	105.490848
Training speed (MCUPS)				

Table B.5: Training speed for NETtalk network plotted as a function of the number of processors.

Number of processors	NETtalk			
	$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 5438$
64	1.0	1.0	1.056	1.12
256	1.0	1.0	1.21	1.39
512	1.0	1.0	1.13	1.30
Speed up ratio				

Table B.6: Speedup of the best configuration compared to the implementation with the largest N_{NP} value, i.e. the 2APC configuration.

N_{NP}	N_{TSP}	NETtalk		
		$\mu = 63$	$\mu = 906$	$\mu = 5438$
1	64	3.659816	-2.522785	-6.522661
2	32	-0.693577	-1.956498	-2.634715
4	16	-4.359198	-3.593698	-3.571020
8	8	-1.419386	-2.887957	-3.015853
% difference to measured time				

Table B.7: Deviation in estimation to measured time for 64 cell configuration for different weight update intervals.

N_{NP}	N_{TSP}	NETtalk		
		$\mu = 63$	$\mu = 906$	$\mu = 5438$
4	128		-4.407855	-4.245820
8	64	1.94	-2.082901	-4.177302
16	32	5.60	-1.119922	-3.089645
		% difference to measured time		

Table B.8: Deviation in estimation to measured time for 512 cell configuration for different weight update intervals.

Number of processors	NETtalk	
	Measured	Estimated
64	14.512598	15.048044
256	49.302208	50.617948
512	85.803440	87.596536
		Training speed (MCUPS)

Table B.9: Comparison of the fastest configuration for 64, 256 and 512 processing elements to estimated values for a weight update interval equal to 906.

Number of processors	NETtalk		
	$\mu = 63$	$\mu = 906$	$\mu = 5438$
512	37.658308	87.596536	110.049160
1024	50.667136	135.503840	200.368496
2048	65.233648	216.144816	343.906784
4096	75.522224	276.797632	530.415744
8192	101.374912	428.224032	869.706496
16384	110.782256	526.743200	1101.859456
32768	137.877728	638.255104	1672.510848
65536	162.331648	793.698624	2018.951424
131072	177.864112	981.617472	2476.685056
262144	151.557312	1069.794944	2748.180992
524288	148.683680	1266.816384	3611.647232
1048576	112.077088	1199.436416	3602.352384
		Training speed (MCUPS)	

Table B.10: Estimated training performance for NETtalk application for three different weight update intervals, μ .

Number of processors	NETtalk	
	2APC configuration	Maximum performance
512	76.967696	87.596536
1024	86.282952	135.503840
2048	161.682224	216.144816
4096	120.007600	276.797632
8192	225.146304	428.224032
16384	135.325968	526.743200
32768	255.312240	638.255104
65536	132.170992	793.698624
131072	251.714592	981.617472
Training speed (MCUPS)		

Table B.12: Comparing the maximum performance of the flexible configuration to the 2APC configuration, for $\mu = 906$.

Number of processors	NETtalk		
	$\mu = 63$	$\mu = 906$	$\mu = 5438$
64	27.388560	39.826436	43.035364
256	43.726324	132.909800	158.419472
512	81.764912	222.592928	294.823264
Training speed (MCUPS)			

Table B.13: Maximum possible training speed on a 512 PE version of AP1000.

Number of iterations	NETtalk						
	$\mu = 1$	$\mu = 63$	$\mu = 259$	$\mu = 906$	$\mu = 1360$	$\mu = 2719$	$\mu = 5438$
1	74.40	99.49	100.00	100.00	100.00	100.00	100.00
10	28.10	42.55	46.34	56.07	66.81	100.00	100.00
20	17.43	31.39	33.05	41.39	44.92	58.33	100.00
30	10.43	23.81	32.31	33.95	38.25	48.34	75.41
40	6.73	15.28	24.05	28.10	32.51	42.83	55.87
50	5.39	11.05	22.60	23.67	26.48	38.64	52.21
56	4.89	10.10	16.79	20.89	26.35	36.48	48.62
60		9.14	13.33	19.51	23.65	34.74	46.41
70		7.54	9.45	15.65	21.86	34.28	44.17
80		6.66	8.13	12.54	22.21	29.57	41.85
90		5.24	6.88	9.95	14.47	26.39	39.44
93		4.73	6.49	10.13	14.78	26.98	39.22
100			5.46	10.79	14.56	23.23	38.07
110			5.00	8.86	12.76	20.76	35.73
111			4.98	8.15	13.57	21.35	35.07
120				6.47	10.96	20.72	33.63
130				7.17	9.99	19.79	31.91
140				5.57	10.10	21.61	30.07
150				5.13	8.83	17.95	30.03
152				4.98	8.83	16.07	29.13
160					7.91	12.76	28.80
170					6.11	11.79	26.92
180					6.09	12.32	24.16
190					6.11	16.24	23.65
198					4.85	13.17	23.61
200						13.44	21.85
210						9.01	20.87
220						7.39	19.40
230						6.71	22.71
240						7.26	17.56
250						5.85	18.24
260						6.22	16.35
270						5.44	16.86
280						5.30	14.47
290						5.33	14.55
292						4.98	14.40
300							13.39
310							12.95
320							14.53
330							14.55
340							11.64
350							14.03
% patterns not trained							

Table B.14: Percentage of characters that are not trained for various weight update intervals.

Weight update interval	NETtalk
	Learning rate
1	1.5
63	0.45
259	0.35
494	0.315
906	0.23
1360	0.165
2719	0.095
5438	0.050

Table B.15: The best learning rate (η) for each investigated weight update interval. The curve plots the derived rule.

Weight update interval	NETtalk
	Number of iterations
1	56
63	93
259	111
494	130
906	152
1360	198
2719	292
5438	560
Number of iterations to 5% error	

Table B.16: Number of iterations needed to obtain convergence with $E_{\%} < 5\%$ as the stopping criteria.

Weight update interval	NETtalk
	$(1 - \alpha)/\eta$
1	0.0667
63	0.2222
259	0.2857
494	0.3175
906	0.4348
1360	0.6061
2719	1.0526
5438	2.0

Table B.17: Plotting of $(1 - \alpha)/\eta$.

Weight update interval	NETtalk	
	Estimated	Measured
1	52	56
63	92	93
259	127	111
494	132	130
906	151	152
1360	187	198
2719	291	292
5438	606	560
Number of iterations		

Table B.19: Comparing measured and estimated values of $N(\mu)$.

NETtalk			
Learning rate	$E\%$ for $\mu = 1$	Learning rate	$E\%$ for $\mu = 5438$
0.5	8.46	0.04	9.38
0.7	6.91	0.042	7.15
0.9	6.16	0.044	7.58
1.1	5.99	0.046	7.50
1.3	5.66	0.048	5.08
1.5	4.89	0.050	4.98
1.7	5.48	0.052	6.68
1.9	6.16	0.054	6.44
2.0	5.59	0.056	6.33
2.3	6.14	0.058	5.90
3.0	7.28	0.060	6.25

Table B.20: The sensitivity of the learning rate (η) on the convergence for epoch vs. pattern learning.

Number of training iterations	NETtalk	
	Epoch error	Error during training
1	100.00	100.00
10	55.87	56.07
20	38.65	41.39
30	30.73	33.95
40	27.84	28.10
50	23.43	23.67
60	18.54	19.51
70	16.81	15.65
80	12.25	12.54
90	10.41	9.95
100	10.52	10.79
110	8.07	8.86
120	6.73	6.47
130	7.08	7.17
140	5.77	5.57
150	5.26	5.13
152	5.20	4.98
	% patterns not trained	

Table B.21: Error computed during training compared to error computed after each iteration (epoch) for $\mu = 906$.

Number of training iterations	NETtalk	
	Epoch error	Error during training
1	69.82	74.40
5	45.68	39.55
10	29.53	28.10
15	24.16	20.58
20	20.71	17.43
25	15.72	12.61
30	15.39	10.43
35	8.90	8.37
40	7.41	6.73
45	7.30	6.12
50	6.22	5.39
55	6.33	5.30
56	5.20	4.89
57	5.28	
58	4.76	
% patterns not trained		

Table B.22: Error computed during training compared to error computed after each iteration (epoch) for $\mu = 1$.

Number of training iterations	NETtalk		
	$\mu = 63$	$\mu = 906$	$\mu = 5438$
1	0.2013	0.3164	0.3284
10	0.1168	0.1723	0.3046
20	0.0952	0.1396	0.2378
30	0.0792	0.1217	0.1935
40	0.0606	0.1063	0.1724
50	0.0567	0.1005	0.1599
58	0.0529	0.0917	0.1532
60		0.0910	0.1517
70		0.0877	0.1447
80		0.0801	0.1386
90		0.0776	0.1332
100		0.0768	0.1278
110		0.0737	0.1232
120		0.0727	0.1196
130		0.0726	0.1162
140		0.0714	0.1130
150		0.0719	0.1108
152		0.0717	0.1100
160			0.1075
170			0.1047
180			0.1024
190			0.1006
200			0.0986
250			0.0919
300			0.0871
350			0.0838
400			0.0820
450			0.0806
500			0.0792
550			0.0749
560			0.0749
E_{RMSE}			

Table B.23: The E_{RMSE} plotted for three weight update intervals.

Weight update interval	NETtalk
	Number of iterations to 10% error
1	31
63	55
259	68
494	79
906	89
1360	129
2719	207
5438	359

Table B.24: Comparing $E_{\%} < 5\%$ and $E_{\%} < 10\%$ error criterias.

Weight update interval	NETtalk
	Total training time (s)
63	382
259	266
494	270
906	265
1360	319
2719	433
5438	794

Table B.25: Total training time for NETtalk running on 512 cells, using 120 hidden units.

Weight update interval	NETtalk
	Total training time (s)
63	512
259	443
494	434
906	461
1360	576
2719	796
5438	1428

Table B.26: Total training time for NETtalk running on 256 cells, using 120 hidden units.

Weight update interval	NETtalk
	Total training time (s)
63	1316
259	1277
494	1413
906	1566
1360	1995
2719	2864
5438	5322

Table B.27: Total training time for NETtalk running on 64 cells, using 120 hidden units.

Number of iterations	Sonar classification				
	$\mu = 1$	$\mu = 13$	$\mu = 52$	$\mu = 104$	$\mu = 208$
1	95.673077	88.942308	59.615385	70.673077	100.000000
10	58.653846	75.000000	100.000000	100.000000	46.634615
20	46.153846	55.769231	84.134615	98.076923	46.634615
30	34.615385	54.326923	74.519231	74.038462	46.634615
40	31.250000	50.961538	64.423077	73.076923	100.000000
50	20.673077	39.903846	84.615385	73.076923	100.000000
60	19.711538	36.538462	60.576923	67.788462	100.000000
70	13.461538	36.057692	53.846154	63.942308	95.673077
80	9.615385	28.846154	57.692308	64.423077	95.192308
90	2.884615	19.711538	53.365385	66.826923	93.269231
100	3.846154	16.346154	40.384615	66.826923	65.865385
102	0.000000	20.192308	50.000000	67.788462	69.230769
110		14.903846	42.788462	57.211538	66.346154
120		13.461538	49.038462	70.192308	52.403846
130		9.615385	45.192308	59.615385	52.884615
140		10.576923	40.384615	55.769231	48.557692
150		9.134615	40.865385	51.923077	48.557692
160		4.807692	36.538462	68.750000	47.115385
170		2.884615	39.423077	65.865385	46.634615
178		0.000000	33.173077	51.923077	45.673077
180			35.096154	61.538462	48.557692
190			28.846154	62.980769	41.826923
200			30.288462	50.480769	41.826923
250			15.865385	43.750000	50.000000
300			15.384615	32.211538	48.076923
343			0.000000	23.557692	45.673077
350				27.884615	45.192308
400				21.634615	36.057692
450				8.173077	32.211538
500				4.326923	31.730769
527				0.000000	45.192308
550					27.884615
600					24.519231
700					25.480769
800					29.326923
900					10.096154
1000					5.769231
1050					1.442308
1065					0.000000
% patterns not trained					

Table B.28: The number of patterns that have $E_p > 0.1$ in percentage, for learning sonar return classification.

Weight update interval	Sonar classification
	Learning rate
1	1.4
13	1.0
26	0.7
52	0.6
104	0.28
208	0.15

Table B.29: The best learning rate η for each of the investigated weight update intervals. The curve plots the derived rule.

Weight update interval	Sonar classification
	Number of iterations to 0% error
1	102
13	178
26	247
52	343
104	527
208	1065

Table B.30: Number of iterations needed to obtain convergence, i.e. $E_p \leq 0.04$ for all training patterns.

Weight update interval	Sonar classification
	Number of iterations to 5% error
1	76
13	157
26	212
52	312
104	488
208	905

Table B.31: Number of iterations needed to reach a stopping criteria of 5 % error.

N_{NP}	N_{TSP}	Speech recognition			
		64 hidden	256 hidden	512 hidden	1024 hidden
2	256	23.995732			
4	128	50.359140	46.921784		
8	64	72.677672	71.820384	66.632164	68.130040
16	32	72.262920	87.061584	88.556072	88.347664
Training speed (MCUPS)					

Table B.33: Training performance on 512 processors for speech recognition network.

N_{NP}	N_{TSP}	Image compression		
		4 hidden units	8 hidden units	16 hidden units
1	512	9.442108	9.984266	10.235404
2	256	21.240164	24.172498	25.881068
4	128	28.393238	37.027588	43.371164
8	64	13.954845	27.908680	39.882572
16	32	4.723187	9.446930	18.893342
Training speed (MCUPS)				

Table B.34: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 processor configuration running the compression sized network.

N_{NP}	N_{TSP}	Image compression		
		4 hidden units	8 hidden units	16 hidden units
256	1	8.393737	9.045833	10.339386
128	2	15.889891	18.595438	21.495230
64	4	16.869354	22.767360	27.503804
32	8	7.313240	14.626005	20.150200
16	16	2.584710	5.169584	9.290658
Training speed (MCUPS)				

Table B.35: MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 processor configuration running the compression sized network.

N_{NP}	N_{TSP}	Image compression		
		4 hidden units	8 hidden units	16 hidden units
128	1	8.725476	9.841765	10.383337
64	2	11.467651	13.953114	15.614437
32	4	9.844131	13.421539	16.273424
16	8	3.911333	7.822622	11.282639
Training speed (MCUPS)				

Table B.36: MCUPS performance for different combinations of neuron and training set parallelism on a 8 x 16 processor configuration running the compression sized network.

N_{NP}	N_{TSP}	Image compression		
		4 hidden units	8 hidden units	16 hidden units
1	64	5.726517	6.669064	7.107607
2	32	6.449299	8.035000	8.962044
4	16	5.135462	7.086301	8.665471
8	8	2.073115	4.146292	6.066419
Training speed (MCUPS)				

Table B.37: MCUPS performance for different combinations of neuron and training set parallelism on a 8 x 8 processor configuration running the compression sized network.

Number of processors	Image compression		
	4 hidden units	8 hidden units	16 hidden units
4	1.870214	2.279415	2.569448
64	6.449299	8.035000	8.962044
128	11.467651	13.953114	16.273424
256	16.869354	22.767360	27.503804
512	28.393238	37.027588	43.371164
Training speed (MCUPS)			

Table B.38: Training speed for image compression network plotted as function of the number of processors.

Number of procesors	Image compression		
	4 hidden units	8 hidden units	16 hidden units
64	3.11	1.94	1.48
128	2.93	1.78	1.44
256	6.527	4.40	2.65
512	6.001	3.92	2.29
Speedup ratio			

Table B.39: Speedup of the best configuration compared to the 2APC configuration.

B.3 Implementation On RENNS

Number of processors	NETtalk			
	30 hidden	60 hidden	120 hidden	240 hidden
1	2.37	2.45	2.41	2.43
2	4.38	4.66	4.80	4.70
3	6.08	6.64	6.92	6.82
4	7.20	8.27	8.75	8.81
5	10.16	9.45	10.12	10.48
6	10.62	10.22	11.15	11.62
7	11.05	11.18	11.86	12.47
8	10.96	12.34	12.03	12.78
Training speed (MCUPS)				

Table B.8: Neuron parallel NETtalk training on 1 to 8 processing modules.

Number of processors	Speech recognition		
	64 hidden	512 hidden	1024 hidden
1	2.37	1.50	1.50
2	4.56	4.80	2.96
3	6.81	7.01	7.05
4	8.94	9.17	9.23
5	10.57	11.08	11.17
6	11.88	12.57	12.69
7	12.45	13.62	13.77
8	12.86	14.44	14.62
Training speed (MCUPS)			

Table B.9: Neuron parallel speech recognition network training on 1 to 8 processing modules.

Number of processors	Sonar classification	Image compression 8 hidden	Character recognition	Image compression 16 hidden
1	2.19	1.58	1.91	1.85
2	3.01	2.59	3.65	3.53
3	3.40	3.15	4.56	4.49
4	3.33	3.52	5.04	5.26
5	3.19	3.62	4.97	5.48
6	2.98	3.64	4.83	5.50
7	2.83	3.47	4.63	5.31
8	2.56	3.25	4.19	4.98
Training speed (MCUPS)				

Table B.10: Neuron parallel training of small networks on 1 to 8 processing modules.

Number of hidden processors	NETtalk			
	30 hidden	60 hidden	120 hidden	240 hidden
1	2.72	2.77	2.79	2.80
2	5.31	5.47	5.55	5.59
3	7.69	8.07	8.24	8.35
4	13.10	10.54	10.84	11.01
5	15.01	12.81	13.30	13.54
6	12.52	14.89	15.55	15.88
7	8.24	9.71	9.26	9.66
Training speed (MCUPS)				

Table B.11: Pipelined training of NETtalk. The horizontal axis shows the number of modules involved for hidden layer computation. The total number of modules is 8 for all the experiments.

Number of hidden processors	Speech Recognition		
	64 hidden	512 hidden	1024 hidden
1	3.12	1.74	1.73
2	6.17	6.30	3.47
3	8.88	9.38	5.20
4	11.88	12.35	12.40
5	14.10	15.07	15.18
6	10.47	12.01	12.12
7	5.82	6.49	6.50
Training speed (MCUPS)			

Table B.12: Pipelined training of speech recognition network on 8 modules.

Number of hidden processors	Speech Recognition 64 hidden	NETtalk 30 hidden	Speech Recognition 1024 hidden	NETtalk 120 hidden
1	3.12	2.72	1.73	2.79
2	6.17	5.31	3.47	5.55
3	8.88	7.69	5.20	8.24
4	11.88	13.10	12.40	10.84
5	14.10	15.01	15.18	13.30
6	10.47	12.52	12.12	15.55
7	5.82	8.24	6.50	9.26
Training speed (MCUPS)				

Table B.13: Pipelined training for large networks on 8 modules.

Number of processors	Image compression 8 hidden	Sonar classification	Character recognition	Image compression 16 hidden
1	3.35	2.38	3.73	4.91
2	3.53	3.20	5.17	5.51
3	3.53	3.56	5.35	5.35
4	3.34	4.03	5.23	5.09
5	2.95	4.22	4.63	4.38
6	2.30	4.11	3.78	3.33
7	1.35	3.76	2.41	1.93
Training speed (MCUPS)				

Table B.14: Pipelined training of small networks on 8 modules.

Number of processors	Speech Recognition			NETtalk	
	1024 hidden	64 hidden	512 hidden	30 hidden	120 hidden
1	1.74	3.12	1.74	2.73	2.79
2	3.47	6.11	6.25	5.32	5.55
3	5.08	5.80	6.34	7.52	8.00
Training speed (MCUPS)					

Table B.15: Pipelined training of NETtalk and speech recognition networks on 4 modules.

Number of processors	Image compression 8 hidden	Character recognition	Sonar classification	Image compression 16 hidden
1	3.22	3.89	2.58	4.56
2	2.58	4.07	4.50	3.58
3	1.53	2.56	4.35	2.03
Training speed (MCUPS)				

Table B.16: Pipelined training of small networks on 4 modules.

Number of hidden processors	NETtalk 30 hidden	Speech Recognition 64 hidden	Speech Recognition 512 hidden	NETtalk 120 hidden
1	2.36	3.10	1.74	2.79
2	5.03	6.06	6.28	5.45
3	7.12	8.76	9.30	8.01
4	8.82	11.54	12.29	10.49
5	9.98	13.96	15.19	12.97
6	10.90	16.33	18.08	15.34
7	12.05	18.08	20.98	17.11
8	13.17	20.93	24.05	20.15
9	14.58	20.96	24.00	21.75
10	15.63	19.03	22.13	24.87
11	15.06	18.01	20.03	22.61
12	14.32	15.26	16.28	20.70
13	11.88	10.30	12.22	17.02
14	7.76	5.68	6.46	9.08
Training speed (MCUPS)				

Table B.17: Pipelined training of NETtalk and speech recognition networks on 15 modules.

Number of hidden processors	NETtalk 30 hidden	Speech Recognition 64 hidden	Speech Recognition 512 hidden	NETtalk 120 hidden
4	7.52	6.11	6.34	8.00
8	15.01	14.10	15.07	15.55
12	15.74	18.80	20.96	21.25
15	15.63	20.96	24.05	24.87
Training speed (MCUPS)				

Table B.18: Maximum pipelined training speed.

Number of processors	NETtalk 30 hidden	NETtalk 120 hidden	Speech Recognition 64 hidden	Speech Recognition 512 hidden
1	2.22	2.26	2.12	1.40
2	4.12	4.50	4.10	4.29
3	5.74	6.50	6.12	6.28
4	6.84	8.33	8.05	8.27
5	9.76	9.96	9.56	10.13
6	10.28	10.99	10.82	11.90
7	10.53	11.72	11.73	13.26
8	10.45	11.87	12.51	14.13
9	9.84	12.27	12.74	15.00
10	9.22	12.14	13.11	15.43
11	9.31	12.19	13.48	15.70
12	8.64	11.56	14.28	15.78
13	8.05	11.36	14.06	15.73
14	7.60	11.28	13.56	15.69
15	7.15	10.75	12.82	15.31
Training speed (MCUPS)				

Table B.19: Neuron parallel training speed.

Bibliography

- [1] J.M. Adamo and D. Anguita. Object oriented design of a BP neural network simulator and implementation on the Connection Machine (CM-5). Technical report, International Computer Science Institute, September 1994. TR-94-46.
- [2] Ali M. Alhaj and Hiroaki Terada. Exploiting parallelism in neural networks on a dynamic data-driven system. *IEICE Trans. on fundamentals*, E76-A(10):1804–1811, October 1993.
- [3] Wayne Allen and Avijit Saha. Parallel neural-network simulation using back-propagation for the ES-kit environment. In *Proc. of 1989 Conf. Hypercubes, Concurrent Computers and Application*, pages 1097–1102, 1989.
- [4] D. Anguita, G. Parodi, and R. Zunino. An efficient implementation of BP on RISC-based workstations. *Neurocomputing*, 6:57–65, 1994.
- [5] K. Asanovic, J. Beck, J. Feldman, N. Morgan, and J. Wawrzynek. Designing a connectionist network supercomputer. *International Journal of Neural Systems*, 4(4):317–326, December 1993. ISSN: 0129-0657.
- [6] Les E. Atlas and Yoshitake Suzuki. Digital systems for artificial neural networks. *IEEE Circuits and Devices Magazine*, pages 20–24, November 1989.
- [7] Magali E. Azema-Barac. A conceptual framework for implementing neural networks onto massively parallel machines. In *6th International Symposium on Parallel Processing (IPPS)*, pages 527–530, IEEE Press, 1992.
- [8] Magali E. Azema-Barac. *A generic strategy for mapping neural network models on transputer-based machines*, pages 244–249. IOS Press, 1992. In: *Transputing in Numerical and neural network applications*, G.L. Reijns and J. Luo, Eds.
- [9] Magali E. Azema-Barac. Neural network implementations and speed-up on massively parallel machines. *Microprocessing and microprogramming*, 35:747–754, 1992.
- [10] Stephen L. Bade and Brad L. Hutchings. FPGA-based stochastic neural networks – Implementation. In *Proc. of IEEE Workshop on FPGAs for custom computing machines*, pages 189–198, 1994.

- [11] Roberto Battiti. Optimizing methods for back propagation: Automatic parameter tuning and faster convergence. In *Proc. of Int. Joint Conference on Neural Networks*, volume I, pages 593–596, WASH.DC, 1990.
- [12] G. Blelloch and C. R. Rosenberg. Network learning on the Connection Machine. In *Proc. of IJCAI87*, pages 323–326, 1987.
- [13] Nazeih M. Botros and M. Abdul-Aziz. Hardware implementation of an artificial neural network using field programmable gate arrays. *IEEE Trans. on Industrial Electronics*, 41(6):665–667, December 1994.
- [14] A. Carpintero et al. Weather forecasting with adaptive time-delay neural networks: A case study. In *Proc. of Int. Conference On Neural Network Processing*, pages 842–846, 1994.
- [15] J.R. Chen and P. Mars. Stepsize variation methods for accelerating the back propagation algorithm. In *Proc. of Int. Joint Conference on Neural Networks*, volume I, pages 601–604, WASH.DC, 1990.
- [16] Tao Chen and Mikio Takagi. Rainfall prediction of geostationary meteorological satellite images using artificial neural network. In *Int. Geoscience and Remote Sensing Symposium*, volume 3, pages 1247–1249, 1993.
- [17] 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.
- [18] 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.
- [19] Lon-Chan Chu and Benjamin W. Wah. Optimal mapping of neural-network learning on message-passing multicomputers. *Journal of Parallel and Distributed Computing*, 14:319–339, 1992.
- [20] I. Cloete and J. Ludik. Parallelization of artificial neural networks on transputers. In *Proc. of the International Conference on Parallel Computing '91*, pages 367–373. Elsevier Science Publishers B.V., 1992.
- [21] Lawrence A. Crowl. How to measure, present and compare parallel performance. *IEEE Parallel & Distributed Technology*, 2(1):9–25, 1994.
- [22] Antonio d' Acierno and Roberto Vaccaro. A parallel implementation of the back-propagation or errors learning algorithm on a SIMD parallel computer. In *Proc. of Int. Conference on Artificial Neural Networks*, pages 1074–1077, 1993.

- [23] Antonio d' Acierno and Roberto Vaccaro. The back-propagation learning algorithm on parallel computers: A mapping scheme. In *Proc. of Sixth Italian Workshop Neural Nets WIRN VIETRE - 93*, pages 249–254, Salerno, 1994.
- [24] J. G. Daugman. Complete discrete 2-D Gabor transforms by neural networks for image analysis and compression. *IEEE Trans. Acoustics, Speech and Signal Processing*, 36(7):1169–1179, 1988.
- [25] David B. Davidson. A parallel processing tutorial. *IEEE Antennas and Propagation Society Magazine*, pages 6–19, April 1990.
- [26] K. Diakonikolaou, S. Kollias, D. Kontoravdis, and A. Stafylopatis. Implementation of neural network learning strategies on a transputer-based parallel architecture. In *Parallel and Distributed Computing in Engineering Systems. Proc. of the IMACS/IFAC International Symposium*, pages 365–370. North-Holland, June 1991. ISBN: 0 444 89276 1.
- [27] Thanh A. Diep and Hadar I. Avi-Itzhak. A neural network approach to high accuracy optical character recognition. In *Proc. of World Congress on Neural Networks*, volume 3, pages 76–86, 1994.
- [28] James G. Eldredge and Brad L. Hutchings. RRANN: The run-time reconfiguration artificial neural network. In *Proc. of the IEEE Custom Intergrated Circuits Conf.*, pages 77–80, 1994.
- [29] Martin D. Emmerson and Robert I. Damper. Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *IEEE Trans. on Neural Networks*, 4(5):788–793, September 1993.
- [30] S.E. Fahlman. Faster-learning variations on back-propagation. In *Proc. of the 1988 Connectionist Models Summer School*. Carnegie-Mellom University, 1988.
- [31] Bernard Faure and Guy Mazare. A cellular architecture dedicated to neural net emulation. *Microprocessing and Microprogramming*, 30(1-5):249–256, August 1990.
- [32] S.J. Fjerdingsstad and C.N. Greve. Parallelizing feed-forward artificial neural networks on transputers. Master's thesis, Aarhus University, DK, September 1991.
- [33] Rafal Foltyniewicz and Slawomir Skoneczny. An improved high order neural network for invariant recognition of human faces in gray scale. In *Proc. of World Congress on Neural Networks*, volume 1, pages 587–592, 1994.
- [34] Shou King Foo et al. A theoretical study of training set parallelism for backpropagation networks on a transputer array. In *Proc. of World Congress on Neural Networks*, volume 2, pages 519–524, 1994.

- [35] Shou King Foo, P. Saratchandran, and N. Sundararajan. Analysis of training set parallelism for backpropagation neural networks. *Int. Journal of Neural Systems*, 6(1):61–78, March 1995.
- [36] Shou King Foo, P. Saratchandran, and N. Sundararajan. Comparison of parallel and serial implementation of feedforward neural networks. *Journal of Microcomputer Applications*, 17(1):83–94, January 1995.
- [37] Edoardo Franzini. Neural accelerator for parallelization of back-propagation algorithm. *Microprocessing and Microprogramming*, 38:689–696, 1993.
- [38] James A. Freeman and David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques*. Addison-Wesley Publishing Company, Inc., 1991. ISBN 0-201-51376-5.
- [39] Yoshiji Fujimoto. An enhanced parallel planar lattice architecture for large scale neural network simulation. In *Proc. of Int. Joint Conference on Neural Networks*, volume 2, pages 581–586, San Diego, 1990.
- [40] Yoshiji Fujimoto, Naoyuki Fukuda, and Toshio Akabane. Massively parallel architecture for large scale neural network simulation. *IEEE Trans. on Neural Networks*, 3(6):876–887, November 1992.
- [41] M. D. Garris. Methods for enhancing neural network handwritten character recognition. In *Proc. of Int. Joint Conference on Neural Networks*, volume 1, pages 695–700, July 1991.
- [42] SCJ Garth. To simulate or not to simulate... In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks*, volume 2, pages 1397–1403. Elsevier Science Publishers B.V., 1992.
- [43] R. Paul Gorman and Terrence J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural networks*, 1:75–89, 1988.
- [44] R. Paul Gorman and Terrence J. Sejnowski. Learned classification of sonar targets using a massively parallel network. *IEEE Trans. on acoustics, speech and signal processing*, 36(7):1135–1140, 1988.
- [45] Kamil A. Grajski. *Parallel Digital Implementations of Neural Networks*, pages 51–76. PTR Prentice Hall, 1993. In Neurocomputing using the MasPar MP-1, K. Wojtek Przytula and Viktor K. Prasanna editors.
- [46] Dan Hammerstrom. A VLSI architecture for high-performance, low cost, on-chip learning. In *Proc. of Int. Joint Conference on Neural Networks*, volume 2, pages 537–542, 1990.

- [47] Dan Hammerstrom. CNAPS - 1064 Digital parallel processor, 1992. Product information AR800A, Adaptive solutions, Inc.
- [48] Dan Hammerstrom. Neural networks at work. *IEEE Spectrum*, 30(6):26–32, 1993.
- [49] Robert Hecht-Nielsen. *Neurocomputing*, pages 115–119. Addison-Wesley, 1990.
- [50] Junichi Higashino et al. Numerical analysis and adaption method for learning rate of back propagation. In *Proc. of Int. Joint Conference on Neural Networks*, volume I, pages 627–630, WASH.DC, 1990.
- [51] Yuzo Hirai. Hardware implementations of neural networks in Japan. *Neurocomputing*, 5:3–16, 1993.
- [52] Y. Hu et al. Neural network based cancer cell classification. In *Proc. of World Congress on Neural Networks*, volume 1, pages 416–421, 1994.
- [53] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.
- [54] Kai Hwang and Faye A. Briggs. *Computer architecture and parallel processing*. McGraw-Hill Book Company, 5th edition, 1989.
- [55] Paolo Ienne. Quantitative comparison of architectures for digital neuro-computers. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 1987–1990, 1993.
- [56] Hiroaki Ishihata. Performance evaluation of the AP1000. In *Proc. of CAP workshop*, pages N–1–8, 1991.
- [57] Hiroaki Ishihata et al. Third generation message passing computer AP1000. In *Proc. of the International Symposium on Supercomputing*, pages 46–55, Nov. 1991.
- [58] 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.
- [59] Young-Im Kang. Creation of an inpection system combining backpropagation networks and image processing system. In *Proc. of Int. Conference On Neural Network Processing*, pages 174–179, 1994.
- [60] 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.
- [61] E.J.H. Kerckhoffs, F.W. Wedman, and E.E.E. Frietman. Speeding up backpropagation training on a hypercube computer. *Neurocomputing*, 4:43–63, 1992.

- [62] Willis K. King, Goffredo Pieroni, Daniele Micci Barreca, and Blaž Zupan. A fast implementation of the back-propagation algorithm. In *Proc. of Sixth Italian Workshop Neural Nets WIRN VIETRE - 93*, pages 218–226, Salerno, 1994.
- [63] H. Kitano. Empirical studies on the speed of convergence of neural network training using genetic algorithms. In *Proceedings 8th JMIT national conference in artificial intelligence*, volume 2, pages 789–796, Boston MASS, 1990.
- [64] M. Kuga, Y. Namiuchi, B.O. Apduhan, and T. Sueyoshi. Implementation and performance evaluation of a neural network simulator on highly parallel computer AP1000. In *ICPAS*, 1993.
- [65] 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.
- [66] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, chapter 5. Benjamin-Cummings, 1993.
- [67] S. Y. Kung and J. N. Hwang. Parallel architectures for artificial neural nets. In *Proc. of IEEE Int. Conference on Neural Networks*, volume 2, pages 165–172, San Diego, July 24-27 1988.
- [68] S. Y. Kung and J. N. Hwang. A unified systolic architecture for artificial neural networks. *Journal of Parallel and Distributed Computing*, 6:358–387, 1989.
- [69] S. Y. Kung and J. N. Hwang. A unifying algorithm/architectures for artificial neural networks. In *Proc. of International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2505–2508, 1989.
- [70] Sun-Yuan Kung and Wei-Hsin Chou. Parallel Digital Implementations of Neural Networks. In K. Wojtek Przytula and Viktor K. Prasanna, editors, *Mapping of neural networks onto VLSI array processors*, chapter 1, pages 3–49. PTR Prentice Hall, 1993.
- [71] S.Y. Kung. *Digital Neural Networks*. PTR Prentice Hall, 1993.
- [72] Wei-Ming Lin, Viktor K. Prasanna, and K. Wojtek Przytula. Algorithmic mapping of neural network models onto parallel SIMD machines. *IEEE Transactions on Computers*, 40(12):1390–1401, December 1991. ISSN: 0018-9340.
- [73] Arne Linde, Tomas Nordstrom, and Mikael Taveniku. Using FPGAs to implement a reconfigurable highly parallel computer. In *Selected papers from: Second International workshop on Field-Programmable Logic and Applications (FPL'92)*, pages 199–210. Springer-Verlag, 1992.

- [74] Richard P. Lippmann. An introduction to computing with neural nets. *IEEE Acoustics, Speech, and Signal Processing Magazine*, 4:4–22, 1987.
- [75] Xiao Liu and George L. Wilcox. Benchmarking of the CM-5 and the Cray machines with a very large backpropagation neural network. In *Proc. of IEEE Int. Conference on Neural Networks*, volume 1, pages 22–27, 1994.
- [76] Q.M. Malluhi, M.A. Bayoumi, and T.R.N. Rao. A parallel algorithm for neural computing. In *Proc. of World Congress on Neural Networks*, volume 2, pages 563–569, 1994.
- [77] Daniel F. McCaffrey and A. Ronald Gallant. Convergence rates for single hidden layer feedforward networks. *Neural Networks*, 7(1):147–158, 1994.
- [78] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. In Anderson and Rosenberg, editors, *Neurocomputing: Foundations and research*, chapter 2, pages 18–28. The MIT Press, 1988. Reprint of the "Bulletin of Mathematical Biophysics", 1943.
- [79] Michael McInerney and Atam P. Dhawan. Use of genetic algorithms with back propagation in training of feed-forward neural networks. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 203–208, 1993.
- [80] S. J. McKenna et al. High-resolution classification of Papanicolaou smear cells using back-propagation neural networks. In *Proc. of Int. Conference on Artificial Neural Networks*, pages 907–910, 1993.
- [81] Robert W. Means. High speed parallel hardware performance issues for neural network applications. In *Proc. of IEEE Int. Conference on Neural Networks*, 1994.
- [82] Robert W. Means and Kent Pu Qing. Parallel image, signal and neural network processing with HNC's vision processor (ViP) and HNC's SIMD numerical array processor (SNAP). In *Int. parallel processing symposium (IPPS)*, 1993.
- [83] Marvin Minsky and Seymour Papert. *Perceptrons*. The MIT Press, 1969.
- [84] Martin Møller. Supervised learning on large redundant training sets. *Int. Journal of Neural Systems*, 4(1):15–25, 1993. World Scientific Publishing Company.
- [85] J. Moody and C .J. Darken. Fast learning in networks of locally tuned processing units. *Neural Computation*, 1:281–294, 1989.
- [86] Nelson Morgan. Using a million connections for continuous speech recognition. In *Proc. of Int. Conference On Neural Network Processing*, pages 1439–1444, October 1994.

- [87] Nelson Morgan et al. The ring array processor: A multiprocessing peripheral for connectionist applications. *Journal of Parallel and Distributed Computing*, pages 248–259, April 1992.
- [88] B. Mulgrew and C.F. Cowan. *Adaptive Filters and Equalisers*. Kluwer, 1988.
- [89] U.A. Muller, B. Baumle, P. Kohler, A. Gunzinger, and W. Guggenbuhl. Achieving supercomputer performance for neural net simulation with an array of digital signal processors. *IEEE Micro*, pages 55–65, October 1992.
- [90] Urs A. Muller. A high performance neural net simulation environment. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 1–4, Orlando, 1994.
- [91] Alan F Murray. Analogue neural VLSI: Issues, trends and pulses. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks*, volume 2, pages 35–43. Elsevier Science Publishers B.V., 1992.
- [92] Jacob M. J. Murre. Transputers and neural networks: An analysis of implementation constraints and performance. *IEEE Trans. on Neural Networks*, 4(2):284–292, March 1993.
- [93] Gaute Myklebust. *Implementations of an unsupervised neural network model on an experimental multiprocessor system*. PhD thesis, Norwegian Institute of Technology, 1996. ISBN 82-7119-893-9.
- [94] Gaute Myklebust, Jon Solheim, and Jarle Greipsland. Self organizing maps on a reconfigurable computer. In *Proceedings of the Sixth International Conference on Signal Processing Applications and Technology*, October 1995.
- [95] Gaute Myklebust and Jon Gunnar Solheim. Parallel Self-organizing Maps for actual applications. In *Proceedings of the IEEE International Conference on Neural Networks*, December 1995.
- [96] Gaute Myklebust, Jon Gunnar Solheim, and Erik Steen. Speeding up small sized Self Organizing Maps for use in visualization of multispectral medical images. In *Eighth IEEE Symposium on Computer-Based Medical Systems*, pages 103–110, Lubbock, Texas, June 1995. IEEE Computer Society Press.
- [97] Gaute Myklebust, Jon Gunnar Solheim, and Erik Steen. Wavefront implementation of Self Organizing Maps on RENNS. In *International Conference on Digital Signal Processing*, pages 268–273, Limassol, Cyprus, 1995.
- [98] Gaute Myklebust, Jon Gunnar Solheim, and Jim Tørresen. Parallel implementations of self organizing maps on RENNS. In *Proceedings of The Norwegian Neural Network Seminar*. SINTEF Instrumentation, December 1994. Report no. STF31 S94026, ISBN 82-595-8893-5.

- [99] Reza Nekovei and Ying Sun. Back-propagation network and its configuration for blood vessel detection in angiograms. *IEEE Trans. on Neural Networks*, 6(1):64–72, January 1995.
- [100] Geir Ove Nesvik. *An empirical study of selected learning algorithms for feed-forward neural networks*. PhD thesis, Norwegian Institute of Technology, 1993. ISBN 82-7119-548-4.
- [101] 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.
- [102] National Institute of Standards and Technology. *fl3* subset of the NIST Special Database 1, obtainable from sequoyah.ncsl.nist.gov in pub/databases/data/.
- [103] S. Olikar et al. Design architectures and training of neural networks with a distributed genetic algorithm. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 199–202, 1993.
- [104] Hiroyuki Onda. Neural network application to welding defect identification. *FUJITSU Sci. Tech. Journal*, 29(3):271–277, September 1993.
- [105] 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.
- [106] Alain Petrowski, Gérard Dreyfus, and Claude Girault. Performance analysis of a pipelined backpropagation algorithm. *IEEE Trans. on Neural Networks*, 4(6):970–981, November 1993.
- [107] D. A. Pommerleau et al. Neural network simulation at Warp speed: How we got 17 million connections per second. In *Proc. of IEEE Int. Conference on Neural Networks*, 1988.
- [108] K. Wojtek Przytula, Viktor K. Prasanna, and Wei-Ming Lin. Parallel implementations of neural networks. *Journal of VLSI Signal Processing*, 4(2-3):111–123, May 1992.
- [109] U. Ramacher et al. Multiprocessor and memory architecture of the neurocomputer SYNAPSE-1. In *Proc. of World Congress on Neural Networks*, volume 4, pages 775–778, 1993.
- [110] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, and M. Weßeling. SYNAPSE-1 – a general-purpose neurocomputer. Technical report, SIEMENS AG, February 1993.

- [111] C.R. Rosenberg and G. Brelloch. An implementation of network learning on the Connection Machine. In D. Walz and J. Feldman, editors, *Connectionist Models and their Implications.*, pages 329–340. Ablex, Norwood, NJ, 1988.
- [112] 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.
- [113] E. Sánchez, S. Barro, and C.V. Regueiro. Artificial neural networks implementation on vectorial supercomputers. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 3938–3943, Orlando, FL, June 28 - July 2, 1994.
- [114] Akira Sato. An analytical study of the momentum term in a backpropagation algorithm. In Proc. of ICANN–91. In T. Kohonen et al., editors, *Artificial Neural Networks*, volume 1, pages 617–622. Elsevier Science Publishers B.V. (North-Holland), 1991.
- [115] Yuji Sato et al. Development of a high-performance, general purpose neuro-computer composed of 512 digital neurons. In *Proc. of Int. Joint Conference on Neural Networks*, pages 1967–1970, 1993.
- [116] Terrence J. Sejnowski. NETtalk corpus, obtainable from ftp.idiap.ch in pub/benchmarks/neural/nettalk.tar.z.
- [117] Terrence J. Sejnowski. Sonar, mines vs. rocks, obtainable from ftp.idiap.ch in pub/benchmarks/neural/sonar.data.z.
- [118] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168, 1987.
- [119] Soheil Shams and Jean-Luc Gaudiot. Implementing regularly structured neural networks on the DREAM machine. *IEEE Trans. on Neural Networks*, 6(2):407–421, March 1995.
- [120] Shigetoshi Shiotani et al. Fast recognition of overlapping targets using neural networks. In *Proc. of Int. Conference On Neural Network Processing*, pages 614–619, Seoul, 1994.
- [121] A. Singer. Exploiting the inherent parallelism of artificial neural networks to achieve 1300 million interconnects per second. In *Proc. of International Neural Networks Conference*, pages 656–660, Paris, France, July 9-13 1990.
- [122] Alexander Singer. Implementation of artificial neural networks on the Connection Machine. *Parallel Computing*, 14:305–315, Summer 1990.
- [123] David J. Slate. Letter image recognition data, obtainable from ics.uci.edu in pub/machine-learning-database/letter-recognition, 1991.

- [124] SNAP – SIMD Numerical Array Processor. High-performance parallel computing, 1994. Product information, Technology Development Division, HNC.
- [125] J. G. Solheim. *The RENNS approach to neural computing*. PhD thesis, Norwegian Institute of Technology, In preparation.
- [126] Jon Gunnar Solheim and Gaute Myklebust. RENNS - an experimental computer system with a reconfigurable interconnection network. In *Proceedings of the First International Workshop on Parallel Processing (IWPP '94)*, pages 205–210, Bangalore, India, December 1994. Tata McGraw Hill.
- [127] Jon Gunnar Solheim and Gaute Myklebust. RENNS - a Reconfigurable Computer System for Artificial Neural Networks. In *Proceedings of first IEEE International Conference on Algorithms and Architectures for Parallel Processing, Brisbane, Australia*, pages 197–206, April 1995.
- [128] N. Sonehara et al. Image data compression using a neural network model. In *Proc. of Int. Joint Conference on Neural Networks*, volume 2, pages 35–40, Washington, D.C., 1989.
- [129] Harold Sue. Adaptive wavelet transforms by wavenets. In *Proc. of Int. Conference On Neural Network Processing*, pages 3–11, October 1994.
- [130] Y. Suzuki and L. Atlas. A comparison of processor topologies for a fast trainable neural network for speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech and Signal Processing, Glasgow*, May 1989.
- [131] Fumiaki Takeda and Sigeru Omatu. High speed paper currency recognition by neural networks. *IEEE Trans. on Neural Networks*, 6(1):73–77, January 1995.
- [132] Shumping Tang, Marino J. Niccolai, and Michael W. Bringmann. A model for the execution of a neural network upon a multiprocessor. In *Proc. of the Fifth Workshop on Neural Networks: Academic/Industrial/NASA/Defence*, pages 259–262, 1993.
- [133] A. Thepaut and Y. Autret. Handwritten digit recognition using combined neural networks. In *Proc. of Int. Joint Conference on Neural Networks*, pages 1365–1368, 1993.
- [134] Jim Tørresen. Jakten på en kunstig hjerne. *Teknisk Ukeblad*, (19):30–31, 1995.
- [135] 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, Seoul, Korea*, volume 1, pages 140–145, October 1994.

- [136] 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 *Proc. of The third Parallel Computing Workshop*, pages P2–E–1 – 9, Kawasaki Japan, November 1994.
- [137] 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 *Proc. of Summer Workshop On Parallel and distributed Processing, Okinawa, Japan, CPSY94–30*, pages 25–32, July 1994.
- [138] Jim Torresen, Shin-ichiro Mori, Hiroshi Nakashima, Shinji Tomita, and Olav Landsverk. The ap1000 computer running back propagation. In *Proc. of the Norwegian Neural Network Seminar. SINTEF Instrumentation*, November 1994.
- [139] 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*, pages 483–488, Cambridge, UK, June 1995. IEE.
- [140] 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*, Perth, Western Australia, 27 November – 1 December 1995. IEEE.
- [141] Jim Torresen and Jon Solheim. Implementing backpropagation training on a reconfigurable computer using pipelining of the training patterns. In *Proceedings of International Conference On Neural Information Processing, Hong Kong*, volume 1, September 1996. Submitted.
- [142] 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*, volume 1, pages 679–682, Washington, D.C., July 1995. INNS Press.
- [143] P. Treleaven and R.V. Rocha. Towards a general-purpose neurocomputing system. In *Silicon Architectures for Neural Nets. Proceedings for the IFIP WG 10.5 Workshop*, pages 167–177. North Holland, November 1991. ISBN: 0-444-89113-7.
- [144] Philip Treleven. *Neurocomputers*. Research Note 89/8, Department of computer science, University College London, January 1989.
- [145] Lampros Tsinas and Volker Graefe. Coupled neural networks for real-time road and obstacle recognition by intelligent road vehicles. In *Proc. of Int. Joint Conference on Neural Networks*, Nagoya, Japan, 1993.
- [146] Lewis W. Tucker and George G. Robertson. Architecture and Applications of the Connection Machine. *Computer*, 21(8):26–38, August 1988.

- [147] Lisbet Utne. *Design of a reconfigurable neurocomputer Performance analysis by implementation of recurrent associative memories*. PhD thesis, Norwegian Institute of Technology, 1995. ISBN 82-7119-793-2.
- [148] T.P. Vogl et al. Accelerating the convergence of the back propagation method. *Biological Cybernetics*, 59:257–263, 1988.
- [149] Philip D. Wasserman. *Neural Computing – Theory and Practice*. Van Nostrand Reinhold, 1989.
- [150] D.M. Weber. Parallel implementation of time delay neural networks for phoneme recognition. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 1583–1587, 1993.
- [151] John N. Weinstein. Neural networks in the biomedical sciences: A survey of 386 publications since the beginning of 1991. In *Proc. of World Congress on Neural Networks*, volume 1, pages 121–126, 1994.
- [152] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, 1985.
- [153] Bernard Widrow et al. Neural networks: Applications in industry, business and science. *Communication of ACM*, 37(3):93–105, March 1994.
- [154] Michael Witbrock and Marco Zagha. An implementation of backpropagation learning on GF11, a large SIMD parallel computer. *Parallel computing*, 14:329–346, 1990.
- [155] James Wolfer et al. Robust multispectral road classification in Landsat thematic mapper imagery. In *Proc. of World Congress on Neural Networks*, volume 1, pages 260–268, 1994.
- [156] Hyunsoo Yoon and Jong H. Nang. Multilayer neural networks on distributed-memory multiprocessors. In *INNC*, pages 669–672, 1990.
- [157] Hyunsoo Yoon, Jong H. Nang, and S.R Maeng. Parallel simulation of multilayered neural networks on distributed-memory multiprocessors. *Microprocessing and Microprogramming*, 29:185–195, 1990.
- [158] Hideki Yoshizawa and Kazuo Asakawa. Highly parallel architecture for back-propagation using a ring register data path. *Fujitsu Sci. Tech. J.*, pages 227–233, September 1993.
- [159] Takashi Yukawa and Tsutomu Ishikawa. Optimal parallel back-propagation schemes for mesh-connected and bus-connected multiprocessors. In *Proc. of IEEE Int. Conference on Neural Networks*, pages 1748–1753, 1993.

-
- [160] 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.
- [161] Xiru Zhang. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14:317–327, Summer 1990.
- [162] S. Zickenheimer, M. Wendt, B. Klauer, and K. Waldschmidt. Pipelining and parallel training of neural networks on distributed-memory multiprocessors. In *Proc. of IEEE Int. Conference on Neural Networks*, volume 4, pages 2052–2057, 1994.

Index

- 2APC, 70
- 3APC, 74
- algorithmic mapping, 34, 57
- AP1000
 - description, 59
 - programming, 60
- application adaptable mapping, 98
- artificial neural networks, 1
- backpropagation, 9
- backpropagation (BP), 2
- bias, 9
- cell, 60
- coarse grain parallelism, 34
- communication, 33
- complexity, 34
- contributions, 3
- convergence, 12
- CPS, 14
- CUPS, 14
- degree of parallelism, 13
 - node, 28, 68
 - pipelining, 28
 - training set, 27, 67
- delta error, 11
- error measure, 9, 12
- estimation
 - convergence, 110
 - execution time, 108
- feed-forward network, 7
- fine grain parallelism, 34
- generalization, 17
- global reduction functions, 60
- heuristic mapping, 34, 57
- iteration, 13
- learning by
 - block, 13
 - epoch, 13
 - pattern, 13
- learning phase, 2
- learning rate, 11
- memory modeling, 110
- MIMD, 33
- momentum, 14
- multi-layer perceptron, 2
- neural network
 - applications, 20
 - image compression, 23
 - NETtalk, 20
 - sonar classification, 24
 - speech recognition, 20
 - size, 13
- neurocomputer, 31
- notation, 13
- outline, 5
- overtraining, 17
- parallel computer
 - classification, 32
 - communication, 33
 - grain of parallelism, 33
 - load balancing, 33
 - topology, 32
- parallel impl. of neural networks

- commercial neurocomputers, 50
- DSP systems, 49
- FPGA, 44
- general purpose machines, 34
- machines models, 39
- other technologies, 53
- transputers, 45
- parallelism
 - node, 28, 68
 - neuron, 29
 - synapse, 30
 - pipelining, 28
 - training set, 27, 67
- perceptron, 7
- preprocessing, 18
- processing elements, 31
- processor topology, 32

- recall phase, 2
- reinforcement learning, 2
- RENNS
 - description, 61
 - programming, 63
- ring bus, 32

- SIMD, 33
- supervised learning, 2

- torus network, 32
- training
 - iteration, 13
 - set, 13

- unsupervised learning, 2

- weight update
 - block, 13
 - delayed, 19
 - epoch, 13
 - interval, 13
 - pattern, 13