Implementations of an unsupervised neural network model on an experimental multiprocessor system

A dissertation submitted to
the Faculty of Electrical Engineering and Computer Science
of The Norwegian Institute of Technology
in candidacy for the degree of Doktor Ingeniør

By

Gaute Myklebust

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

February 2, 1996
Abstract

This thesis represents a practical work on parallel implementations of an Artificial Neural Network Model, the Self-organizing Map. The Self-organizing Map model is investigated and analyzed in a parallel context. A description of parallel implementations of this model, made by other research groups, is also given.

Several parallel implementations of the SOM model have been constructed. The implementations have been executed on RENNS, a REconfigurable Neural Network Server, which is a multiprocessor system designed and built at the Norwegian Institute of Technology. The reconfigurability of the communication system enables experimentation with different communication topologies and, if required, different communication protocols. The different parallel implementations of the SOM model cover several aspects of parallelism.

Initially, attention was payed towards large networks i.e. networks with a large number of neurons and/or high dimensional training data. Motivated by an application of the SOM model – visualization of multispectral medical images, a short survey of the applications of the SOM model was undertaken. This showed that typical applications of the model use rather small networks, whereas implementations made by other researchers have focused on large networks. This fact justified a shift of focus from large networks to smaller networks.

It is showed how the size of the neural network impacts on which kind of algorithm to use in order to obtain the best performance. For large networks, the communication overhead in a neuron parallel algorithm is very low, and near linear speedup can easily be achieved. For smaller networks, the communication overhead becomes larger in neuron parallel algorithms and therefore, a shift towards training example parallel algorithms gives better results.
# Nomenclature

## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>Dimensionality</td>
</tr>
<tr>
<td>$E_p$</td>
<td>Efficiency of a parallel algorithm on $p$ processors</td>
</tr>
<tr>
<td>$m_i$</td>
<td>Weight vector, neuron $i$</td>
</tr>
<tr>
<td>$\Delta m_i^j$</td>
<td>Weight change vector for neuron $i$ on processor $j$</td>
</tr>
<tr>
<td>$l$</td>
<td>Word length of vector</td>
</tr>
<tr>
<td>$m_{ij}$</td>
<td>Weight component $j$, neuron $i$</td>
</tr>
<tr>
<td>$N_c$</td>
<td>The neighborhood around a neuron</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of neurons</td>
</tr>
<tr>
<td>$n_{\text{upd}}$</td>
<td>Number of updated neurons</td>
</tr>
<tr>
<td>$out_i$</td>
<td>Output of neuron $i$</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of processors</td>
</tr>
<tr>
<td>$p_i$</td>
<td>Number of intermediate processors</td>
</tr>
<tr>
<td>$S_p$</td>
<td>Speedup of a parallel algorithm on $p$ processors</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Time needed by an algorithm executed on $p$ processors</td>
</tr>
<tr>
<td>$t$</td>
<td>Time</td>
</tr>
<tr>
<td>$t'$</td>
<td>Epoch number</td>
</tr>
<tr>
<td>$t_{\text{byp}}$</td>
<td>Time needed to bypass a module</td>
</tr>
<tr>
<td>$t_{cc}$</td>
<td>Cycle time of the communication subsystem</td>
</tr>
</tbody>
</table>

## Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{div}$</td>
<td>Integer division</td>
</tr>
<tr>
<td>$\text{mod}$</td>
<td>Rest, integer division</td>
</tr>
<tr>
<td>$f()$</td>
<td>Activation function</td>
</tr>
<tr>
<td>$\text{net}()$</td>
<td>Basis function</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\lfloor \cdot \rfloor & \quad \text{Lowest integer not less than} \\
\lceil \cdot \rceil & \quad \text{Largest integer not greater than} \\
\| \cdot \| & \quad \text{Norm} \\
\langle \cdot, \cdot \rangle & \quad \text{Inner product}
\end{align*}
\]

## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>ART</td>
<td>Adaptive Resonance Theory</td>
</tr>
<tr>
<td>BMU</td>
<td>Best Matching Unit</td>
</tr>
<tr>
<td>BP</td>
<td>Backpropagation</td>
</tr>
<tr>
<td>CPS</td>
<td>Connections Processed per Second</td>
</tr>
<tr>
<td>CUPS</td>
<td>Connection Updates Per Second</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>DS</td>
<td>Data Stream</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>LBF</td>
<td>Linear Basis Function</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>Millions of Floating Point Instructions Per Second</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instructions Multiple Data</td>
</tr>
<tr>
<td>MIPS</td>
<td>Millions of Instructions Per Second</td>
</tr>
<tr>
<td>MR</td>
<td>Magnetic Resonance</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PM</td>
<td>Processing Module</td>
</tr>
<tr>
<td>PS</td>
<td>Processing Subsystem</td>
</tr>
<tr>
<td>RBF</td>
<td>Radial Basis Function</td>
</tr>
<tr>
<td>RENNS</td>
<td>REconfigurable Neural Network Server</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SOFM</td>
<td>Self Organizing Feature Maps (same as SOM)</td>
</tr>
<tr>
<td>SOM</td>
<td>Self Organizing Maps</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>WTA</td>
<td>Winner Take All</td>
</tr>
</tbody>
</table>
## Greek letters

<table>
<thead>
<tr>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Learning rate, Self-organizing Map</td>
</tr>
<tr>
<td>$\alpha'$</td>
<td>Learning rate, modified Self-organizing Map</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Iterations done on each processor between updating</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>Fraction of neurons updated</td>
</tr>
<tr>
<td>$\theta_i$</td>
<td>Neuron bias</td>
</tr>
</tbody>
</table>
Contents

List of Figures xi
List of Tables xv
Preface xvii

1 Introduction 1
1.1 The objective ............................................. 1
1.2 The RENNS project ........................................... 2
1.3 Limitations .................................................. 2
1.4 Contributions ................................................ 4
1.5 Outline of this thesis ....................................... 5

2 Artificial Neural Networks 7
2.1 History ....................................................... 7
2.2 Structure of Artificial Neural Networks ......................... 8
2.2.1 The neuron ................................................ 8
2.2.2 Interconnection of neurons ............................... 11
2.2.3 Learning algorithms ..................................... 11
2.2.4 Retrieval phase .......................................... 12
2.3 Various ANN models ......................................... 12
2.3.1 Hopfield networks ....................................... 13
2.3.2 Self-organizing Maps ................................... 14
2.3.3 Backpropagation networks .............................. 15

3 Parallelism in Neural Networks 17
3.1 Parallel processing ............................................ 17
3.1.1 Evaluating parallel algorithms .......................... 18
3.1.2 Limitations on parallel executions ....................... 19
3.2 Parallelism in Neural Networks ............................... 20
3.2.1 Training session parallelism ............................. 20
3.2.2 Training example parallelism ............................ 21
3.2.3 Layer level parallelism .................................. 22
3.2.4 Neuron level parallelism ........................................ 23
3.2.5 Weight level parallelism ........................................ 23
3.2.6 Evaluation of ANN algorithms .................................. 24
3.2.7 Benchmarks .................................................... 26

3.3 Simulation of Artificial Neural Networks ......................... 26
3.3.1 Describing computer architectures ............................ 26
3.3.2 General purpose single processor computers ................ 27
3.3.3 General purpose multiprocessor computers .................. 28
3.3.4 General purpose neurocomputers ............................... 30
3.3.5 Special purpose neurocomputers ............................... 31
3.3.6 Proposed architectures ....................................... 32

4 Parallel Self-organizing Maps .................................... 35
4.1 The SOM model revisited ........................................ 35
4.1.1 Background .................................................. 35
4.1.2 A detailed description ....................................... 36
4.1.3 Practical considerations ...................................... 37
4.1.4 Learning Vector Quantization ................................ 39
4.2 Parallelization strategy ......................................... 41
4.2.1 Training example parallelism .................................. 41
4.2.2 Layer level parallelism ....................................... 41
4.2.3 Neuron parallelism ............................................ 42
4.2.4 Load balancing ............................................... 45
4.3 Performance .................................................... 48
4.3.1 Parameters influencing measurements ....................... 48
4.3.2 Previous measurements ...................................... 54
4.4 Applications of the SOM model .................................. 54

5 The RENNS Computer System ..................................... 59
5.1 History .......................................................... 59
5.2 RENNS Architecture ............................................. 60
5.3 RENNS implementation ......................................... 61
5.3.1 The Processing Subsystem .................................... 61
5.3.2 The Communication Subsystem ............................... 62
5.3.3 Reconfiguration of the Communication Network .......... 64
5.4 Communication Subsystem Configurations ....................... 66
5.4.1 Communication configurations ............................... 66
5.4.2 Utility configurations ....................................... 68
5.4.3 The configuration used in the implementations .......... 68
5.4.4 Modeling the Vector configuration ........................... 69
5.5 The RENNS environment ........................................ 72
5.5.1 Program development ........................................ 72
5.5.2 Interfacing RENNS .......................................... 72
CONTENTS

6 Parallel SOM implementations 75
   6.1 The NP-Ring implementation ........................................ 76
   6.2 The NP-Tree implementation ........................................ 82
   6.3 The NP-Wave implementation ........................................ 87
   6.4 The TP-Ring implementation ........................................ 93
   6.5 The NP/TP implementation ......................................... 97

7 Results 105
   7.1 The NP-Ring implementation ........................................ 105
       7.1.1 Performance .................................................. 106
       7.1.2 The frequency of the Communication Subsystem .......... 109
   7.2 The NP-Tree implementation ........................................ 109
       7.2.1 Performance .................................................. 109
       7.2.2 Comparison ................................................... 109
       7.2.3 Communication times ........................................ 110
   7.3 The NP-Wave implementation ........................................ 112
       7.3.1 Performance .................................................. 112
       7.3.2 Comparison ................................................... 113
   7.4 The TP-Ring implementation ........................................ 115
       7.4.1 Performance .................................................. 116
   7.5 The NP/TP implementation .......................................... 117
       7.5.1 Performance .................................................. 117
   7.6 The MR-visualization application ................................. 120
       7.6.1 SOM in visualization ....................................... 121
       7.6.2 Weight exchange ............................................. 123
       7.6.3 Performance comparisons for the MR-image visualization application ........................................ 125
   7.7 Performance on a larger application ............................ 127

8 Conclusions and recommendations 129
   8.1 Summary and Conclusions .......................................... 129
   8.2 Recommendations for future work ................................. 130

A The RENNS reconfigurability 133
   A.1 Reconfiguration on the logic level ................................ 133
   A.2 Reconfigurable interconnection network ........................ 136

B Source code 137
   B.1 Common SOM routines .............................................. 137
   B.2 The NP-Ring implementation ....................................... 140
   B.3 The NP-Tree implementation ....................................... 149
   B.4 The NP-Wave implementation ...................................... 159
   B.5 The TP-Ring implementation ...................................... 173
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.6 The NP/TP implementation</td>
<td>180</td>
</tr>
<tr>
<td>C Tables</td>
<td>193</td>
</tr>
<tr>
<td>Bibliography</td>
<td>203</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Artificial Neuron ................................................. 9  
2.2 Activation functions ............................................ 10  
2.3 Hopfield network ............................................... 13  
2.4 Example of a Self Organizing Map with a 2 dimensional ordering of the  
neurons .............................................................. 14  
2.5 The Backpropagation network ................................... 16  
2.6 Gradient search .................................................. 16  

3.1 Training session parallelism, Backpropagation network ............. 21  
3.2 Training example parallelism, Backpropagation network .......... 22  
3.3 Layer level parallelism, Backpropagation network ................ 23  
3.4 Neuron level parallelism, Backpropagation network ............... 24  
3.5 Weight level parallelism ......................................... 25  

4.1 The development of the neighborhood function in a 2 dimensional SOM  
network .............................................................. 38  
4.2 Mapping of a 1 dimensional SOM chain onto a set of processors .... 47  
4.3 Obermayer’s mapping of a 2 dimensional SOM network onto a set of  
processors ........................................................... 48  
4.4 Hodges’ mapping of a 2 dimensional SOM network onto a set of processors 49  
4.5 Demian’s mapping of a 2 dimensional SOM network onto a set of processors 50  
4.6 Fraction of neurons updated in the 1 and 2 dimensional case ....... 53  

5.1 The building blocks in the RENNS architecture - the Processing Module  
with associated Data Streams ....................................... 60  
5.2 Operational modes of the Data Streams ................................ 61  
5.3 Block diagram of a RENNS module ................................ 62  
5.4 Block diagram of the RENNS Processing Subsystem ................. 63  
5.5 Block Diagram of the RENNS Communication Subsystem ............ 64  
5.6 Overview of the data paths in the Communication Subsystem ....... 65  
5.7 Vector protocol with dummy Stream Controller inserted ............ 67  
5.8 A packet in the Vector protocol .................................. 67  
5.9 Communication in the NP-Wave implementation ..................... 69  
5.10 The RENNS environment, intermodule communication not indicated . 73
LIST OF FIGURES

6.1 Communication based on one ring ........................................ 79
6.2 Estimated performance of the NP-Ring implementation .................. 82
6.3 Estimated performance of the NP-Ring implementation for a large number of processors ........................................ 83
6.4 Communication based on one global ring and a tree structure of sub rings 84
6.5 Estimated performance of the NP-Tree implementation .................... 88
6.6 Estimated performance of the NP-Tree implementation for a larger number of processors ........................................ 89
6.7 Example on the communication in the NP-Wave implementation with 4 modules operating in the normal mode. ................................. 91
6.8 Estimated performance of the NP-Wave implementation .................... 93
6.9 Estimated performance of the NP-Wave implementation for a larger number of processors ........................................ 94
6.10 Estimated performance of the TP-Ring implementation .................... 98
6.11 2D-torus interconnection network ........................................ 100
6.12 Estimated performance of the NP/TP implementation ..................... 102
6.13 Comparison of the estimated performances on the MR-image visualization problem ........................................ 103

7.1 Performance of the NP-Ring implementation ............................... 106
7.2 Effect of altering the frequency of the communication subsystem, NP-Ring implementation ........................................ 107
7.3 Performance of the NP-Tree implementation ................................ 108
7.4 Comparison of the performance of the NP-Ring and the NP-Tree implementations ........................................ 110
7.5 Communication times, NP-Ring and NP-Tree implementations ............. 111
7.6 Performance of the NP-Wave algorithm ..................................... 113
7.7 Performance comparisons, NP-Ring and NP-Wave .......................... 114
7.8 Performance of TP-Ring implementation with 64 neurons .................. 115
7.9 Performance of TP-Ring implementation with 256 neurons ................ 116
7.10 Performance of the NP/TP implementation, 64 neurons, $\epsilon = 10$ ........ 118
7.11 Performance of the NP/TP implementation, 64 neurons, $\epsilon = 1000$ .... 119
7.12 Performance of the NP/TP implementation, 256 neurons, $\epsilon = 10$ ........ 120
7.13 Performance of the NP/TP implementation, 256 neurons, $\epsilon = 1000$ .... 121
7.14 Visualization of an MR-image by using the PCA transform ............... 122
7.15 Visualization of an MR-image by using the SOM model ................... 123
7.16 Convergence for different weight exchange values, and 20000 iterations .... 124
7.17 Online convergence when varying the initial learning rate ................ 125
7.18 Comparison of the different implementations on the MR-image visualization application ................................. 126
7.19 Performance in the first break arrival application ........................ 127
A.1 The format of the packet in the Winner configuration (arrow indicates direction) ................................................................. 134
A.2 Block diagram of the stream controller, WINNER protocol. ....... 135
List of Tables

1.1 An overview over the persons who have contributed in the RENNS project 3

3.1 Nordström’s Classification ........................................ 27

4.1 Performance of SOM implementations .......................... 56

4.2 Problem parameters for some reported applications of Self-organizing Maps 57

6.1 Outline of the NP-Ring implementation, masters part ........ 77

6.2 Outline of the NP-Ring implementation, slaves’ part .......... 78

6.3 Outline of the NP-Tree implementation, master’s part ........ 85

6.4 Outline of the NP-Tree implementation, slaves’ part .......... 86

6.5 Outline of the NP-Wave implementation ....................... 90

6.6 Outline of the TP-Ring implementation ....................... 95

6.7 Weight exchange when the weight change matrix fits in FIFO 95

6.8 Weight exchange when the weight change matrix does not fit in FIFO 96

6.9 Outline of the NP/TP implementation .......................... 99

C.1 NP-Ring performance, MCUPS ................................... 193

C.2 NP-Ring performance on 8 MHz and 10 MHz, MCUPS .......... 194

C.3 NP-Tree performance, MCUPS ................................... 194

C.4 Communication times, Clocks .................................... 195

C.5 NP-Wave performance, MCUPS ................................... 195

C.6 Performance of NP-Ring and NP-Wave, MCUPS (d = 4) .... 196

C.7 TP-Ring performance, MCUPS (n = 64, d = 4) ................ 196

C.8 TP-Ring performance, MCUPS (n = 256, d = 4) ............... 197

C.9 NP-Ring performance, MCUPS (n = 64, d = 4, ε = 10) .... 197

C.10 NP-Ring performance, MCUPS (n = 64, d = 4, ε = 1000) ... 198

C.11 NP-Ring performance, MCUPS (n = 256, d = 4, ε = 10) ... 198

C.12 NP-Ring performance, MCUPS (n = 256, d = 4, ε = 1000) ... 199

C.13 Error as a function of how often weight update is done ... 200

C.14 Convergence as a function of learning rate .................. 201

C.15 Performance of NP-Ring, NP/TP and TP-Ring, MCUPS (n = 2048, d = 32, ε = 10) ........................................... 202
Preface

The work presented in this thesis has been conducted by Professor Olav Landsverk at the Group of Computer Architecture and Design, Department of Computer Systems and Telematics, The Norwegian Institute of Technology. The work was started in January 1992 and was completed by the end of 1995.

I would like to express my thanks to the Department of Computer Systems and Telematics for the granting of the scholarship which made this work possible. I would also like to thank both the CS&T Department, the NTH Fund and the Computer Architecture and Design group for granting travel funds, making it possible to participate on various conferences throughout the world. During my work I have attended scientific conferences in Norway, Cyprus, USA (twice), India and Australia.

Professor Olav Landsverk has been my supervisor in the work with the thesis, and I am very thankful for all advices he has given me during the past four years. The working conditions in the Computer Architecture and Design Group have been excellent under his leadership.

Most of the work has been done in the Computer Architecture and Design Group, and I am thankful to the members of the group. In the laboratory, I have had the pleasure to work with Jon Solheim, who has given many helpful hints regarding programming of the Communication Subsystem of RENNS, and has helped in troubleshooting when things went wrong (things actually tend to go wrong when you attempt to write parallel programs for an experimental computer). I have been working with and sharing an office with Jim Tørresen, and I am thankful to him for all the valuable feedback he has given on my work. I would also like to thank Jarle Greipsland, who has helped me on several occasions, especially when I was stuck with programming errors.

I would also like to thank Erik Steen for letting me work with him in the field of applying neural networks for visualization of multispectral images. The work on that specific application has been the motivation for several of the elements which this thesis is constructed of.
Thanks to Pauline Haddow, who has taken the time to proof read the thesis. Her skills, both in the professional field of the thesis, and as a native speaker of English, is a valuable combination.

My family has always encouraged me to do well at school and to get an education. This support has been of great value, and I would like to thank my family for this support and for giving me a fine adolescence.

I would also like to thank my social surroundings for making the years at the Norwegian Institute of Technology (R.I.P.) more than just school and education. This especially applies to my dear friends Mr. Blaker, Mr. Husstad and Dr. Sagatun (alphabetic order).

Finally, I would like to thank Anne Marie for her support, eventhough she is of the opinion that saying thanks to people who have not participated directly in the work should be avoided.

Trondheim February 2, 1996,
Gaute Myklebust
Chapter 1

Introduction

1.1 The objective

Neural Networks have gained increasing importance during the last 10 years. One of the main problems that often arises when simulating Artificial Neural Networks (ANNs) on conventional single processor computers, is the long time needed to train the networks. This problem can be overcome in several ways. Many researchers have been working on modifications of the already known ANN models in order to make the network train faster, or even development of new ANN models. Another way to reduce the training time is to take advantage of the parallel nature of the neural network models. By exploiting parallel execution of the ANN models, on conventional parallel computers or on computers specially designed for execution of neural networks, the time needed to train networks can be reduced substantially.

The main goal of this thesis has been to examine one specific ANN model – the Self-organizing Maps, with a view to efficiently parallelizing the model for the RENNS computer system. Within the thesis, a number of parallel implementations of the Self-organizing Map model are described. The implementations are all executed on the RENNS computer system. RENNS is a general purpose neurocomputer designed at the Norwegian Institute of Technology. The RENNS computer system was designed to experiment with various kinds of ANN models. Although the parallel implementations were designed for this computer, the principles are general and implementations can be moved to other parallel computer systems.

In cooperation with the Algorithmic Construction Group at the Norwegian Institute of Technology, a study of the application of the SOM model for visualization of multi-spectral
medical images was undertaken. Using a four channel MR-image (Magnetic Resonance) and gray scale visualization, this turned out to be a relatively small application. The parallel SOM implementations we had developed by that time turned out to give very poor performance on problems of the actual size. Therefore, an investigation into the size for typical applications of the SOM was carried out. It turned out, that even though parallel algorithms for simulating the SOM network had been known for several years, the averaged sized actual application of the SOM network was still relatively small. The parallel algorithms previously implemented by other researchers mainly report results on large networks. Many of these previously developed algorithms showed poor results on small networks. As a result of this, much attention in this work has been payed towards speeding up the small sized applications of the SOM model.

1.2 The RENNS project

The work within this thesis is a part of the RENNS project. The RENNS project was started in late 1989 by Professor Olav Landsverk. The aim of the RENNS project was to design and build a general purpose neurocomputer in order to experiment with various types of ANN models. As the work presented in this thesis was started early in 1992, the design of the computer system was finished. During the years from 1992-1995, the work done within the group has concentrated on making the RENNS computer system fully operational, to develop various communication configurations, and to develop parallel implementations of ANN models.

During these years, a large number of people have been involved in the RENNS project. Table 1.1 gives a brief overview over the persons which have contributed to the project, and in which area the contribution has been made.

1.3 Limitations

Writing a Dr.ing. thesis is a process where you start off trying to solve all problems related to a field, and end up with dealing with a very small subset of those initial problems.

The SOM model is central in this thesis. It is however outside the scope of the thesis to deal with theoretical issues regarding the model. We have chosen to use the model in its initial form. The main focus has been to make efficient parallel implementations of the model in this initial form.
### 1.3. Limitations

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Years</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olav Landsverk</td>
<td>Professor</td>
<td>1989-1995</td>
<td>Project initiator and leader Supervision</td>
</tr>
<tr>
<td>Håkon Dahle</td>
<td>PhD Student</td>
<td>1989-1992</td>
<td>Design, building and testing of the Processing Subsystem</td>
</tr>
<tr>
<td>Jarle Greipsland</td>
<td>PhD Student</td>
<td>1989-1995</td>
<td>Design, building and testing of the Processor Subsystem VME interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software Environment</td>
</tr>
<tr>
<td>Gaute Myklebust</td>
<td>PhD Student</td>
<td>1992-1995</td>
<td>Building and testing of the Processor Subsystem and the Communication Subsystem Implementation of the Self-organizing Maps</td>
</tr>
<tr>
<td>Jim Tørresen</td>
<td>PhD Student</td>
<td>1992-1995</td>
<td>Configuration Loader Implementation of the Backpropagation algorithm</td>
</tr>
<tr>
<td>Lisbet Utne</td>
<td>PhD Student</td>
<td>1989-1995</td>
<td>Design, building and testing of the Communication Subsystem Implementation of Hopfield Networks</td>
</tr>
<tr>
<td>Ingvald Baugstø</td>
<td>MSc Student</td>
<td>1992-1993</td>
<td>Software Environment</td>
</tr>
<tr>
<td>Ingvar Fagerland</td>
<td>MSc Student</td>
<td>1992-1993</td>
<td>Software Environment</td>
</tr>
<tr>
<td>Per K. Knutsen</td>
<td>MSc Student</td>
<td>1994</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>Kjetil Laugsand</td>
<td>MSc Student</td>
<td>1993</td>
<td>Design of a mathematical interpreter for RENNS</td>
</tr>
<tr>
<td>Ole-Jørgen Liium</td>
<td>MSc Student</td>
<td>1993-1994</td>
<td>Design of a loader</td>
</tr>
<tr>
<td>Staal Vinterbo</td>
<td>MSc Student</td>
<td>1993</td>
<td>Design of a mathematical interpreter for RENNS</td>
</tr>
<tr>
<td>Uwe Zachman</td>
<td>MSc student</td>
<td>1992</td>
<td>A study of a reconfigurable interconnection switch for RENNS</td>
</tr>
</tbody>
</table>

Table 1.1: An overview over the persons who have contributed in the RENNS project
A number of experiments in this work are based on real medical images. However, using the SOM algorithm for visualization of medical images does not imply advocating that this algorithm gives better or worse images than other visualization techniques. The reason for using the MR-image application is that it represents a real application of the SOM model.

1.4 Contributions

This thesis represents a practical work with developing parallel implementations of Kohonen's Self-organizing Maps on an experimental computer system. The contributions can be summarized as follows:

- Several parallel implementations of the Self-organizing Map model have been constructed for RENNS. The implementations are published at the Joint Conference on Information Sciences [81] (also published at the Norwegian Neural Network Symposium [85]), the IEEE Symposium on Computer-Based Medical Systems [83], and the International Conference on Digital Signal Processing [84].

- Special attention has been payed to speeding up small sized networks. A short review of applications of the SOM model has shown that typical applications of the model are within this range. This work was presented at the IEEE International Conference on Neural Networks [82].

- It has been demonstrated that there exists no sole parallel algorithm which is best suited to the SOM model, but that the size of the problem decides which algorithm should be used. This work was published at the International Conference on Signal Processing Applications and Technology [80].

- The RENNS Computer System is now operational. Although the design of the computer system was finished by the time the work on this thesis was started, much time has been used in building, testing and debugging the RENNS computer system. This work has to a large degree been a collaboration with the other members of the RENNS project. The work done on the RENNS computer system is published at the Nordic Symposium on Artificial Neural Networks [62], the International Workshop on Parallel Processing [107], and at the International Conference on Algorithms and Architectures of Parallel Processing [108].

- Work undertaken in collaboration with the Algorithm Construction Group regarding the application of neural networks in visualization of multi-spectral medical images has been published at the Joint Conference on Information Sciences [86], at the SPIE Conference on Medical Imaging [113], and will also appear in the Information Sciences
1.5 Outline of this thesis

Journal [87]. This work is not in the mainstream of the thesis, and the results from this work figures only in the thesis as examples.

1.5 Outline of this thesis

Chapter 2 gives an introduction to the field of Artificial Neural Networks. A description of the main components in an ANN model are described. A brief outline of some of the most frequently used ANN models, representing a large variety in network components, is also given.

Chapter 3 describes parallel processing in general and how parallel algorithms are evaluated. In addition, the parallel aspects of ANN models are described. We also give a description of different computer architectures which have been used for simulations of ANNs.

Chapter 4 describes the Self-organizing Map model in detail. The different parallel aspects of the model are described and performance issues are discussed. It also contains an overview of implementations made by other researchers and a short review of problem sizes for applications of the SOM model.

Chapter 5 gives a description of the RENNS architecture and the implemented RENNS computer system. Special attention is given to the communication subsystem and the configuration used in the implementations of the SOM model. A description of the environment of the RENNS computer system is also given.

Chapter 6 describes five different implementations of the Self-organizing Map model which have been constructed for the RENNS Computer System. A theoretical model of each of the implementations is also given, making it possible to give qualitative information on how the implementations would behave on systems with a larger number of processors than are present in the RENNS Computer System.

Chapter 7 is the results chapter. In this chapter, the performance measurements of the implementations are presented. Other related results are also included in this chapter.

Finally, in chapter 8 the concluding remarks are given, including recommendations for future work.
Chapter 2

Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models which are inspired by the operation of the human brain. ANNs share some of the characteristics of its biological inspirator: they learn, they generalize, they have the ability to extract essentials from corrupted or incomplete data, and they are relatively fault tolerant. Among the areas where ANNs have successfully been used, we find the areas where the traditional von Neuman style of computers have been less successful, such as speech and image recognition. These are the same areas in which the ANN’s inspirator, the human brain, is highly successful.

2.1 History

The first artificial neuron was introduced by McCulloch and Pitts in 1943 [74]. This first neuron was a simple logic device. The first physiological learning rule was stated by Hebb in 1949 [27]. Hebb’s learning rule states that when a neuron A repeatedly or persistently takes part in firing another neuron B, a change takes place in the connection between the neurons such that A’s efficiency, as one of the neurons firing B, is increased. The Perceptron presented by Rosenblatt [99] in 1958 and the ADALINE presented by Widrow and Hoff [128] in 1960 are complete systems including computational elements, interconnection and learning schemes. Great optimism followed the development of these models, which seemingly could compute everything. However, in 1969, Minsky and Papert [78] pointed out severe restrictions in what the Perceptron was able to compute. Furthermore it was predicted that extending the Perceptron to multilayer networks would probably share the same limitations as the single layer networks. As a result of the work of Minsky and Papert, research in ANNs almost came to a standstill. In the 1970’s research activity in ANN was low. There were however small groups of researchers still working in the ANN
area or closely related areas, among these Teuvo Kohonen, Stephen Grossberg, and James Anderson. The modern era in ANN research started with Hopfield's presentation [32] of the network, later to be known as the Hopfield Network. In 1986 the Backpropagation algorithm was proposed by the Parallel Distributed Processing Group at University of California – San Diego led by McClelland and Rumelhart [73]. Following the development of the Backpropagation algorithm, there has been an enormous amount of research into ANNs.

Today, the field of ANNs is more established. During the last 10 years, several textbooks including Wasserman [126], Freeman et al [17] and Kung [57] and a number of reviewing papers [33, 64] describing ANNs have been published. A short presentation of the field of ANNs, based on these references is given in the following section.

2.2 Structure of Artificial Neural Networks

A large variety of ANN models have been proposed. However, there are three common characteristics. The first is the processing element, deciding what the neurons are computing. The second is the interconnection of the processing elements. The third component is the learning algorithm, describing how the parameters in a neural network are adapted in a learning situation.

2.2.1 The neuron

In the literature, there exists several names for the processing element of an ANN. The most common names are neuron, node, and unit. In the rest of this thesis, the term neuron will be used.

The artificial neuron exists in multiple variants which however share some characteristics. Each neuron has an unspecified number of input signals. An input signal can be an output signal from another neuron or it may be a signal fed into the neural network from an external source. The input signals to the neuron are weighted. The neuron performs some computations on its input signals and produces a single output signal. The output signal may be real valued or binary, depending on the model. This output signal of the neuron may be used as an input signal to other neurons and/or it can be an output signal from the network. A schematic illustration of an artificial neuron is shown in figure 2.1.
2.2. Structure of Artificial Neural Networks

![Artificial Neuron Diagram]

**Figure 2.1: Artificial Neuron**

### 2.2.1.1 Basis function

The neurons first compute a basis function $\text{net}(w, x)$, where $w = \{w_1, w_2, \cdots, w_d\}$ is the connection strengths (weights) and $x = \{x_1, x_2, \cdots, x_d\}$ is the input vector to the neuron (see figure 2.1). The most common forms for this basis function are described by Kung in [57] as:

1. A linear basis function (LBF). An LBF is a linear combination of the inputs.

   $\text{net}_i = \sum_{j=1}^{d} x_j w_{ji}$

   \hspace{1cm} (2.1)

2. A radial basis function (RBF). The net value represents the distance to a reference pattern given by the weights.

   $\text{net}_i = \sqrt{\sum_{j=1}^{d} (x_j - w_{ji})^2}$

   \hspace{1cm} (2.2)

Nesvik [88] gives examples of other forms of basis functions. There is no formal restriction as to which form the basis function must take.

### 2.2.1.2 Activation function

The result of the net function is passed through an activation function. Among the most common we find [57]:

$\text{out} = f(\text{net}(w, x))$
1. Sigmoid function
\[ f(\text{net}) = \frac{1}{1 + e^{-\text{net}}} \] (2.3)

2. Gaussian function
\[ f(\text{net}) = ce^{-\frac{\text{net}^2}{\sigma^2}} \] (2.4)

3. Step function (hard limiter)
\[ f(\text{net}) = \begin{cases} 0 & \text{if net < 0} \\ 1 & \text{otherwise} \end{cases} \] (2.5)

4. Ramp function
\[ f(\text{net}) = \begin{cases} y_{\min} & \text{if net < } y_{\min} \\ \text{net} & \text{if } y_{\min} \leq \text{net} \leq y_{\max} \\ y_{\max} & \text{otherwise} \end{cases} \] (2.6)

The four described kinds of activation functions are illustrated in figure 2.2. Not all models use an activation function, which is for instance the case in the SOM model. It can be argued that even these models have an activation function which is the linear activation function \( f(\text{net}) = \text{net}. \)

![Graphs of Sigmoid, Gaussian, Step, and Ramp functions.](image)

**Figure 2.2: Activation functions**

The activation functions in the figure are all unipolar, but can just as well be bipolar, ranging from, for example, \(-1\) to \(1\).
2.2. Structure of Artificial Neural Networks

2.2.2 Interconnection of neurons

The neurons of a neural network may be connected in various ways. The networks are often *layered* into an input layer, possible hidden layers and an output layer. The different connection types are:

1. Feedforward connections: the outputs from one layer are propagated to the inputs of a succeeding layer.
2. Feedback connections: the outputs from one layer are fed back to the inputs of a former layer.
3. Lateral connections: the outputs from one layer is connected to neurons within the same layer.

2.2.3 Learning algorithms

There are two phases involved in the application of ANNs to a problem: the learning phase and the retrieval phase. The objective of the learning phase is to set the parameters of the network (for instance the weights and eventual thresholds in the activation functions). These parameters are adapted by the use of a learning algorithm. The learning algorithms can broadly be divided into three classes: the fixed weight learning algorithms, the supervised learning algorithms and the unsupervised learning algorithms.

2.2.3.1 Fixed-weight learning algorithms

For some neural networks models the weights can be found in deterministic time by some pre-defined computations on the training vectors. This is for instance the case in Hopfield Networks, see section 2.3.1, and bidirectional associative memories. The ANNs which incorporate such deterministic learning algorithms are often trained very fast compared to those using supervised or unsupervised learning algorithms.

2.2.3.2 Supervised learning algorithms

In a supervised learning algorithm, the training set consists not only of the training (input) vector but also the desired output vector. The training vector and the corresponding desired output vector is referred to as a training pair. The desired output vectors are used
as a correction mechanism in the sense that the weights are changed in order to make the actual output produced by the network more similar to the desired output. Supervised learning algorithms are iterative and stop when the results produced by the network are reduced to an acceptable error level. The most popular learning algorithm in this class is the Backpropagation algorithm, see section 2.3.3.

2.2.3.3 Unsupervised learning algorithms

Unsupervised algorithms are algorithms where the training set consists only of training (input) vectors. The algorithms are iterative and training is normally stopped when the network has become organized in some sense, for instance when similar inputs produce similar outputs. The Self-organizing Maps (see section 2.3.2) is an example of a network using an unsupervised learning algorithm.

Unsupervised learning algorithms are claimed to be more biologically plausible than supervised learning algorithms. This comes from the fact that it is difficult to imagine a training mechanism in the brain which compares actual with desired outputs [126].

2.2.4 Retrieval phase

After a network is trained, it is only used to produce an output vector for each input vector presented to the network. This phase is called the retrieval phase. When a network is in the retrieval phase all the network parameters, such as the weights, are fixed and no learning takes place.

2.3 Various ANN models

A number of ANN models have been proposed. According to Hecht-Nielsen [28] there were, in 1988, over fifty ANN models developed, of which at least thirteen were in common use. In this work, we have chosen to describe the Hopfield network, the Self-organizing Maps and the Backpropagation network. The reasons are twofold: firstly, they represent three of the most commonly used ANN models and secondly, they represent a large variety with respect to the three ANN components described earlier.
2.3. Various ANN models

2.3.1 Hopfield networks

The Hopfield networks were introduced in 1982 by John Hopfield [32]. A Hopfield network is a network with only one layer. The output from each neuron is fed back to all the other neurons in the network, see figure 2.3. The neurons use an LBF and pass the value from this LBF to a step function which produces the output of the network. The Hopfield network has a fixed weight learning algorithm. The weights are formed by taking the outer products of the \( d \) training vectors \( \mathbf{x}_i \) as shown:

\[
w_{ij} = \sum_{m=1}^{d} x_{mi} x_{mj}, \quad i \neq j
\]  

(2.7)

The Diagonal of the weight matrix is 0 i.e. \( w_{ii} = 0 \) for all \( i \). The neurons compute an LBF and use a hard limiter as the activation function. The neurons have an offset, \( \theta \), which can be set to 0, eventually \( \theta_i = \sum_{j=1}^{d} w_{ij} \) [57]. In the first iteration, the external inputs are used as input to the system. In the following iterations, the external inputs are replaced by the feedback connections. More formally, the neurons compute the following basis function:

\[
net_i(t+1) = \sum_{j=1}^{d} w_{ij}out_j(t) + \theta_i
\]  

(2.8)

The output of neuron \( i \), \( out_i(t+1) = f(net_i(t+1)) \) takes the value 0, 1, or \( out_i(t) \) depending on whether the value of \( net_i \) is less than, more than, or equal to 0 respectively.

![Figure 2.3: Hopfield network](image)

Typical applications of the Hopfield networks are optimization problems. Hopfield networks have been used to give semi-optimal solutions to problems like the Traveling Salesman's Problem (TSP).
2.3.2 Self-organizing Maps

Kohonen's Self-organizing Map is another ANN model which has gained a lot of attention. A short description of the model is given here and a more detailed description is found in chapter 4.

SOM networks consist of an ordered set of neurons. The ordering is usually one or two dimensional, see figure 2.4 for an example of a two dimensionally ordered network. All neurons have the same inputs. The most used basis function is the following RBF:

\[ out_i = \sqrt{\sum_{j=1}^{d} (m_{ij} - x_j)^2 } \]  

(2.9)

where \( out_i \) is the output of neuron \( i \), \( m_{ij} \) is the \( j \)th weight to neuron \( i \), \( x_j \) is the \( j \)th component of the input vector and \( d \) is the dimensionality of the input vector.

![Figure 2.4: Example of a Self Organizing Map with a 2 dimensional ordering of the neurons](image)

In the training phase, a training vector is selected and presented to all the neurons in the network. The output values of all the neurons are computed. The neuron which closest resembles the input vector i.e. the neuron with the lowest output value, is selected as the winner. The winning neuron and its neighbors are then updated in order to be even more similar to the input vector:

\[ m_{ij}(t + 1) = m_{ij}(t) + \alpha(x_j - m_{ij}(t)) \]  

(2.10)
The neighborhood size and learning parameter ($\alpha$) are reduced with time.

In the retrieval phase, the operation of the network is reduced to select the winning neuron of the applied input vector.

### 2.3.3 Backpropagation networks

The Backpropagation algorithm was first described by Werbos in 1974 [127]. It was later reinvented by Parker in 1985 [94] and finally by Rumelhart and McClelland [73] in 1986. The Backpropagation network is the most popular neural network model, and a number of commercial applications using this model have been developed [129]. It is a layered network using an LBF and a sigmoid function as the activation function. The name Backpropagation network comes from the fact that the error is propagated from the output layer back towards former layers. The Backpropagation algorithm operates on multilayer feed-forward networks. We use the term Backpropagation networks to denote this combination of algorithm and network.

The number of layers in a Backpropagation network may vary, but typically it is structured as one input layer, one hidden layer and one output layer. The number of neurons in the input layer matches the number of dimensions in the input vectors and the number of neurons in the output layer matches the number of dimensions in the output vectors. The number of neurons in the hidden layer are more difficult to determine. If the number of neurons in the hidden layer is too low, the network may not be able to learn the desired mapping from input vectors to output vectors. If there are too many neurons in the hidden layer, the network may, on the other hand, be overfitting i.e. it learns the training set well, but will not behave well on vectors not present in the training set. As such, the number of neurons in the hidden layer cannot be determined by the number of inputs and outputs alone, but is very dependent on the training set of the application at hand. A figure of a Backpropagation network can be seen in figure 2.5.

The learning algorithm is a supervised iterative algorithm. The algorithm used is a gradient search method. The error is derivated with respect to each of the weights. Each weight is then moved slightly in the opposite direction of the gradient in order to reduce the error. Figure 2.6 shows the error of the network as a function of one specific weight, when all the other weights are fixed. $w(t)$ indicates the weight at time $t$. By moving the weight in the opposite direction of the gradient, the error is likely to decrease (but only likely because all the weights are changed, thereby changing the influence of each of them).

After training is completed, the Backpropagation network is used only in feed-forward mode, producing an output vector when an input vector is applied.
Figure 2.5: The Backpropagation network

Figure 2.6: Gradient search
Chapter 3

Parallelism in Neural Networks

Training an artificial neural network may be very time consuming. When an application is to be adapted to a neural network, several network parameters are required. The values of these parameters are often found by trial and error. Even after the size of a network is fixed, a trained network may not give acceptable performance and new training will have to take place. Each of these training sessions may take a very long time.

One way of reducing the time needed is to make better training algorithms. Several variations of the standard ANN models have been proposed in the literature. The variations claim to improve the original models in either it’s accuracy or in the training time. One method for reducing the training time, which is the direction we are following in this thesis, is by the use of parallel processing.

The human brain itself also uses massively parallel processing, having as much as $10^{12}$ processing elements, and $10^{15}$ connections between neurons (approximate figures).

3.1 Parallel processing

In parallel processing several processors cooperate to solve a problem, in order to provide reduced execution time compared to what which can be achieved on one processor [39]. The impact of parallel processing is that larger problems can be solved within the same time frame, or a fixed sized problem can be solved faster.
3.1.1 Evaluating parallel algorithms

It is important to measure what is achieved by using a parallel algorithm to solve a problem compared to using a sequential algorithm which is solving the same problem. There may exist a number of sequential algorithms solving the problems at hand, but the algorithm solving the problem fastest may not be suitable for parallelization. Comparison should in all cases be based on the sequential algorithm solving the problem fastest in order to get a real idea of what has been achieved by using parallel processing.

3.1.1.1 Speedup

Let the execution time for the best known sequential algorithm on a sequential computer be denoted $T_1$, and let the execution time of the parallel algorithm when executed on $p$ processors be denoted $T_p$. Then the speedup, denoted $S_p$, becomes:

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

If $S_p = p$ then the speedup of the parallel algorithm is said to be linear on $p$ processors. If $S_p < p$, the algorithm is said to be sublinear, and if $S_p > p$, the algorithm is said to be superlinear. The processing element used for the sequential algorithm is the same as the processing element used for the parallel algorithm.

**Superlinear speedup** If an algorithm is to be superlinear, some special effects have to occur. This is because every parallel algorithm can be sequentialized and executed on a single processor computer. A superlinear speedup can for instance be observed if the processors in the computer have access to a restricted amount of resources like for instance high speed on-chip memory. When a problem is executed on a sequential computer, the amount of data may not fit into the memory residing on-chip on the processor, and must then be placed in slower external memory. In a parallelization of the algorithm, it may be possible to divide the data among the processors, and the reduced amount of data on each processor may now fit in faster memory. The result can be a parallel implementation with superlinear speedup. Examples of such superlinear speedup can be observed in chapter 7.
3.1. Parallel processing

3.1.1.2 Efficiency

Efficiency is a measure of how well the processors are utilized when used in a parallel algorithm. The efficiency of an algorithm on \( p \) processors, denoted \( E_p \), is given by:

\[
E_p = \frac{T_1}{T_{pp}} = \frac{S_p}{p}
\]  

(3.2)

In an ideal system, with linear speedup, the efficiency will be exactly 1. In most parallel implementations of an algorithm, only sublinear speedup is achieved, with a resulting efficiency between 0 and 1.

3.1.1.3 Scalability

The scalability of a system is a measure of its capacity to increase speedup in proportion to the number of processors [56]. As the number of processors is increased, the efficiency tends to drop if the problem size is fixed. If the efficiency can be maintained by increasing the size of the problem, then the parallel system is said to be scalable. When the problems are scaled when executed on more processors, artificial effects such as superlinear speedup caused by different memory speeds, can be avoided.

Load balancing is another problem which arises when the number of processors is increased for a given problem size. If the problem size is held constant, then less work is placed on each processor. By scaling the problem, this effect can be avoided.

3.1.2 Limitations on parallel executions

There are several reasons for why the efficiency drops when more processors are added. Amdahl’s law places an upper bound the speedup which can be achieved by parallel execution of a problem. Amdahl states that practical computations have a sequential part (i.e. a part which can not be parallelized) and a parallel part. Denote the time needed to execute the serial part \( t_{seq} \) and the time needed to execute the parallel part \textit{when executed on a single processor} \( t_{par} \). As the number of processors rises, the time spent doing the parallel part drops, whereas the sequential part is constant. More precisely, the maximum speedup which can be achieved when executing on \( p \) processors is given by:
\[ S_{Amdahl} = \frac{T_1}{T_p} = \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{p}} \] (3.3)

Since \( t_{seq} \) is not scaled with \( p \), linear speedup can only be achieved if the problem does not have any sequential part. Since most problems do have a sequential part, the maximum speedup which can be achieved is limited by:

\[ S_{\text{max}} = \lim_{p \to \infty} \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{p}} = \frac{t_{seq} + t_{par}}{t_{seq}} \] (3.4)

### 3.2 Parallelism in Neural Networks

Neural network models contain several inherently parallel structures which can be utilized for high performance implementations on parallel architectures [102].

The different levels of parallelism in neural networks have been described by several researchers including Nordström et al [90] who identified six levels. The levels of parallelism coexist in a neural network, and many of the levels may be exploited simultaneously. Which level to use is dependent on a number of elements, including the complexity of the processing element, the number of processors and other aspects of the computer architecture.

#### 3.2.1 Training session parallelism

With training session we understand a complete training of a neural network. When a problem is to be applied to neural networks, a lot of experiments usually takes place in order to set parameters such as the learning rate and the number of neurons in the different layers. By using training session parallelism, several network configurations may be explored simultaneously. An example of training session parallelism is given in figure 3.1. The only difference between the two training sessions is the difference in learning rate.

Furthermore, the results of training a neural network is often sensitive to the initial weight values. This is the case in the Backpropagation network – where a gradient search technique is used and in Hopfield networks – when these (i.e. the Hopfield networks) are used in optimization problems. Using training session parallelism, networks with different initial settings may be explored simultaneously.
3.2. Parallelism in Neural Networks

![Diagram of parallel processors with training vectors](image)

**Figure 3.1: Training session parallelism, Backpropagation network**

When using training session parallelism, linear speedup is easily achieved since there is no need for interprocessor communication. As it is trivial, we will not investigate this level of parallelism any further.

### 3.2.2 Training example parallelism

There are often a large number of training vectors associated with a neural network application. On a sequential computer, these vectors will be presented to the network one at a time. In a parallel system, these training vectors can be divided among the processors. Each processor then needs a complete copy of the neural network. The processors can then train on different training vectors simultaneously. Figure 3.2 shows an example of training example parallelism.

After a number of iterations, the local weight changes computed on one processor must be distributed to the other processors. Using training example parallelism raises the question of how often the local weight changes computed locally on one processor should be communicated to the other processors in order for the neural network to train properly, i.e. converge in a similar way as if weight update was done after each training vector.
3.2.3 Layer level parallelism

Neural network models such as the Backpropagation network and the Neocognitron are layered as shown in figure 2.5. In such layered models, the training vectors can be pipelined through the network. Using layer level parallelism, several training vectors are present in the network simultaneously. An example of layer parallelism is given in figure 3.3.

For some networks, the pipeline can be folded backwards. In the Backpropagation algorithm, the error is propagated backwards layer by layer. This operation can be taken care of by other processors than those used in the forward pass.

Models which are not layered, such as the SOM model, can use a similar scheme where the training vectors are pipelined through the processors. After the training vector has passed through all the processors, a winner is selected. This winner is sent through the pipeline once more so that all the processors can update their weights according to this winner.
3.2.4 Neuron level parallelism

Neuron level parallelism is the most obvious level of parallelism present in a neural network model, since the neuron is also the processing element of the neural network. Neuron level parallelism is to divide the neurons (within a layer if a layered model is used) among the processors and then compute the neurons in parallel. One or more neurons are mapped to each processor. Neuron level parallelism is present in all the neural network models, and is common in most parallel implementation, regardless of which ANN model is used. Neuron level parallelism is illustrated in figure 3.4.

3.2.5 Weight level parallelism

The computation within a neuron can also be divided among several processors. This is a very fine grained parallelism, and is primarily present in direct hardware implementations. An example of weight level parallelism is given in figure 3.5.
3.2.6 Evaluation of ANN algorithms

Performance metrics for parallel algorithms include speedup, efficiency and scalability. These are relative metrics of how well a parallelization of an algorithm is. In the field of ANNs, two additional metrics exist. These metrics are absolute in the sense that they are related to wall clock time. These two measurements are Connections Processed per Second (CPS) and Connection UPdatet per Second (CUPS).

CPS measures the speed of the computer system in the retrieval phase. A connection is said to be processed when an input signal is combined with a weight (connection) in, for example, a multiplication (in an LBF) or a subtraction (in an RBF). By counting how many such connections can be processed per time unit, the measurement is established.

CUPS is a measurement of how fast the training is done. This is often a more interesting figure, as it is in the training where most time is consumed. When the training algorithm results in the modification of a weight, a connection is said to be updated. The measurement is established by counting how many of these updates a computer can manage per time unit.

Although they are absolute, the CPS and CUPS figures are not objective. There are a number of factors which influence the measurements of these figures and great care must
be taken when analyzing reported CPS or CUPS measurements. The measurements are, of course, dependent on the ANN model. This is due to the different learning algorithms. Modifying one weight in a SOM network does not require the same amount of computation as in a Backpropagation network. For some models, like the SOM model, the CUPS figure is not constant over the training period since the neighborhood size decreases. Another factor influencing the performance is the resolution of the numbers being used. The CNAPS computer, for instance, can choose between eight bit and sixteen bit precision, resulting in different CUPS measurements, see Hammarstrom et al [25]. The influence of the parameters on measured performance of a parallel SOM implementation is further investigated in section 4.3.1.
3.2.7 Benchmarks

The figures of CPS and CUPS are also dependent on the parameters of the network, such as how many neurons are present in the network. In order to make the measurements comparable, the settings of the network parameters should be equal in all experiments.

There exists some applications of ANNs which have become de facto benchmarks. Nettalk [101] is an application which transforms written text into phonemes. It consists of a 203 dimensional input vector, 60-120 neurons in the hidden layer, and it has 26 outputs. The input vector is a window of 7 characters, where we want to classify the sound of the character in the center of the window.

Nettalk was one of the first applications of the Backpropagation algorithm, and has become a frequently used benchmark for ANN models with supervised learning rules.

For SOM, there are no such de facto benchmarks. This can be observed in the reported results of parallel implementations of this model. There is often no reason given for the chosen parameters. This difference in parameter settings makes comparisons of performance between the implementations very difficult.

3.3 Simulation of Artificial Neural Networks

An ANN model is basically a mathematical description. This description can be executed on a broad range of computers, ranging from simple single processor computers such as a PC, to massively parallel computers such as the MasPar MP-1.

Even though neurocomputing attempts to mimic the operation of the brain, the implementation techniques are often conventional [121], which means simulating the processing elements of the network on a traditional von-Neuman style computer.

3.3.1 Describing computer architectures

A number of different computer architectures have been used for simulation of ANNs, and a description of some of these is given in the following section. In order to help out in describing computer architectures, a number of well known architecture classification schemes can be used. A commonly used classification scheme is known as Flynn's classification [34].

Flynn divides the range of computer architectures into four classes, depending on the
multiplicity of instruction streams and data streams. These four classes are:

**SISD:** Single Instruction Single Data. Computers in this category sequentially execute the instructions from a single instruction source on a single data stream.

**SIMD:** Single Instruction Multiple Data. Computers in this category have multiple processors, executing the same instruction from a single stream on data from multiple streams.

**MISD:** Multiple Instruction Single Data. Computers in this category would execute several instructions on the same data. According to Hwang and Briggs [34] this is not a realistic class.

**MIMD:** Multiple Instruction Multiple Data. Computers in this category have multiple processors, each with its own instruction stream and its own data stream.

There are large variations in the number of processing elements present in the different architectures used in simulating ANNs and a number of informal terms are used to describe the degree of parallelism. Nordström and Svenson [90] made a quantification of some of these commonly used terms:

<table>
<thead>
<tr>
<th>Number of PEs</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0 - 2^4$</td>
<td>Barely Parallel</td>
</tr>
<tr>
<td>$2^4 - 2^8$</td>
<td>Moderately Parallel</td>
</tr>
<tr>
<td>$2^8 - 2^{12}$</td>
<td>Highly Parallel</td>
</tr>
<tr>
<td>$2^{12}-     $</td>
<td>Massively Parallel</td>
</tr>
<tr>
<td>$\infty$</td>
<td>Continuously Parallel</td>
</tr>
</tbody>
</table>

**Table 3.1: Nordström’s Classification**

A factor of $2^4$ was chosen to separate the classes. Nordström’s classification is assumed in this thesis.

### 3.3.2 General purpose single processor computers

Due to availability, the general purpose single processor computers, such as personal computers or UNIX workstations, is probably the most common platform for execution of ANNs.
From the mathematical description of an ANN model, an arbitrary high level language can be used to implement a simulator of the model. Textbooks describing software implementation issues in neural network programming are available [17]. There are no restrictions on which kinds of networks can be simulated on such computers, but the extensive training times often impose practical restrictions.

A number of commercially available tools for simulating neural networks on general purpose single processor computers are available. An overview of different commercially available neural network tools is given by Nesvik [88].

3.3.3 General purpose multiprocessor computers

In order to reduce training times in neural networks researchers have been forced to move to other kinds of computer systems. The following is a brief description of some general purpose multiprocessor computers, with some references to implemented ANNs on these computers.

Thinking Machines Corporation has constructed a series of parallel computers, known as Connection Machines. There are three distinct computers. The Connection Machine CM-1 [123] is a massively parallel SIMD computer and consists of up to 64K one bit processors, each with 4Kbit local memory. The processors have neighborhood communication facilities, in addition to a general routing facility. Singer [105] has described 5 different implementations of the Backpropagation algorithm on the CM-1. The Connection Machine CM-2 [123] is the successor of the CM-1. A floating point accelerator per 32 processing element, additional memory and I/O features are included. Obermayer et al [91, 92] have used the CM-2 for implementing the Self-organizing Map algorithm. In contrast to the CM-1 and CM-2, the CM-5 [63] is a MIMD computer. It is a massively parallel computer with up to 16K processors. Each processing unit has a SPARC processor, 32 MByte memory and a vector processing unit. A fat-tree is used for interprocessor communication. Adamo et al [1] and Liu et al [65] have used the CM-5 for implementing the Backpropagation algorithm.

The Transputer is another common processor platform for implementation of neural network models. Transputers are microprocessors designed for multiprocessing with four bidirectional communication channels for communication with other Transputers. Transputers can be connected in various topologies. The SOM model has been implemented on a transputer array by Obermayer et al [91], Ceccarelli et al [7], Siemon et al [104], and Wu et al [30, 130].

Warp is a parallel computer designed and built at Carnegie Mellon University with support from DARPA. It was later manufactured and sold commercially by The General
Electric Company. Warp is a linear array of cells (ten cells) working in a systolic fashion. Each cell consists of a 5 MFLOPS adder chip, a 5 MFLOPS multiplication chip, and a 10 MIPS integer ALU. The Backpropagation algorithm was implemented by Pomeerleau et al [97] and the SOM model was implemented by Mann et al [69] on the Warp computer system.

The MasPar MP-1 is a massively parallel SIMD computer with up to 16K PEs. The computer consists of a processor element array (PE array), an array control unit (ACU) and a UNIX based front end (FE). The ACU controls the PE array by broadcasting data and instructions simultaneously to the PEs. Interprocessor communication is available by point-to-point nearest neighbor communication (each PE has 8 neighbors) or between two arbitrary PEs through a slower general purpose Global Router. Each PE consists of a 4 bit processor, 40 x 32 bit registers, 16 - 32 KByte memory and communication circuitry. The MasPar MP-1 has been used for implementation of the Backpropagation algorithm by Chinn et al [8, 22]. The SOM model was implemented by Demian et al [12] and Grajski et al [23], and a competitive learning algorithm by Syu et al [114].

Sequent System's Balance 8000 is a symmetric multiprocessor system. The computer has multiple identical processors, and shared memory. Communication between the processors occurs over a shared bus. Buhusi [5] used the Balance 8000 computer for implementing the SOM model.

Fujitsu AP1000 [38] is the third generation of a highly parallel computer constructed by Fujitsu. It consists of from 64 to 1024 processors (cells). The computer system has three independent communication networks; a barrier synchronization network for synchronization, a broadcast network for communication between the host computer and the cells and a 2D-torus point-to-point network for communication between the cells. Each cell consists of a SPARC integer unit, a SPARC floating point unit, 16MB DRAM, and communication controllers. Tørrersen et al [116, 117, 118, 119] have used the AP1000 for implementing a number of different Backpropagation algorithms.

NCube/x is a parallel computer with a x-dimensional hypercube interconnection network. Each NCube node consists of a 32 bit processor and additional circuitry for memory management and inter processor communication. The system is hosted by a personal computer. Kerckhoffs et al [44] have used an NCube/4+ for implementing the Backpropagation algorithm. Tang et al [115] have also used an NCube system for implementing Backpropagation, with the number of nodes ranging from 1 to 128.
3.3.4 General purpose neurocomputers

General purpose neurocomputers are architectures that are designed to execute neural networks. This implies strong communication facilities and a large numeric computing capacity. The general purpose neurocomputers have the generality to support a wide range of ANN models. The General purpose neurocomputers can broadly be classified into one of two classes, depending on the processing elements. The first category is based on off-the-shelf components and nearly always ends up with Digital Signal Processors (DSPs) as the processing elements due to the mathematical operations involved in neural network simulations. The second class uses processing elements constructed especially for simulating neural networks.

RENN5 [59, 60, 61] is a MIMD reconfigurable general purpose neurocomputer based on the Texas Instruments TMS320C30 Digital Signal Processor. It was built to simulate a wide range of ANN models. RENN5 is described in more detail in chapter 5. The ANN models implemented to date include the Backpropagation network [106, 107], the Hopfield networks [124] and the SOM model [83, 84, 85].

RAP (Ring Array Processor) [79] is a MIMD computer system built at the University of California, Berkeley. The computer system is based on the Texas Instruments TMS320C30 Digital Signal Processor, and a system of 40 DSPs have been constructed. Each processor is equipped with 4-16 MByte DRAM and 256 KByte-1 MByte SRAM. The processors are connected in a ring realized in Field Programmable Gate Arrays. The RAP has been used for implementing the Backpropagation algorithm. The RAP computer simulating the Backpropagation algorithm has been used in speech recognition.

Music [24] (MUlti Signal processor system with Intelligent Communication) is a MIMD parallel distributed computer architecture. The MUSIC implementation consists of 30 PEs. A PE consists of a Motorola DSP 96002, 2MB Video-DRAMs, 256KB SRAM and an intelligent communication controller. The communication can be handled independently from the computation. The PEs are connected in a ring. The MUSIC computer system has been used for implementing the Backpropagation algorithm.

Synapse-1 [98] is a general purpose neurocomputer designed by Siemens. Although it was primarily designed for aiding neural network application research within Siemens, it has also been made commercially available. A Synapse neurocomputer consist of a hosting Sparcstation, a 68040 based Control Unit, memory for holding weights, a data unit and a MA16 processor array. The processing element in Synapse is the MA16 Neural Signal Processor (NSP). Each MA16 NSP holds 4 identical processor modules. The operations supported by the processor modules are operations typical for neural processing. The operations not supported by the processor module must be handled by the data unit.
Adaptive Solutions’ CNAPS (Connected Network of Adaptive Processors) system [26] is a general-purpose SIMD neural network architecture, allowing simulation of various ANN models. The CNAPS implementation consists of 4 PN (Processor Node) array chips (N64000) and a sequencer hosting the system. The N64000 chip, produced by Inova Microelectronics Corporation, has 80 fabricated PNs, of which 64 are in use. Each PN has a number of registers, a multiplier, an adder, 4K local memory and some conditional logic. The PN can do one multiply and add per cycle. The CNAPS computer system has been used for implementing the Backpropagation algorithm [72] and the Self-organizing Maps [25].

HNC has developed a commercially available SIMD general-purpose neurocomputer called SNAP (SIMD Numerical Array Processor) [75, 76]. The computer system consists of 16, 32 or 64 processors connected in a ring. There are four HNC100 processors in each VLSI circuit. Each HNC100 has 512 KByte of external memory. In addition, all the processors have access to a global memory. The system can be hosted by a Sparc station. The Backpropagation algorithm has been implemented, and performance measurements are presented. An implementation of the SOM algorithm is briefly described.

MANTRA I [36] is an accelerator dedicated to ANN simulations. It is based on a systolic array of up to 40 \( \times \) 40 processing elements. The processing elements are implemented in a circuit named GENES IV. Each GENES IV circuit holds 2 \( \times \) 2 processing elements. The accelerator is controlled by a TMS320C40 DSP from Texas Instruments. Descriptions on how to implement the Backpropagation algorithm, the Hopfield networks and the Self-organizing Maps has been presented [35, 36], but no actual implementation has to our knowledge been reported as yet.

3.3.5 Special purpose neurocomputers

Special purpose neurocomputers are computers specialized for executing a specific ANN model. Special purpose neurocomputers are often based on special purpose digital or analog VLSI circuits.

Speckman et al [109, 110, 111, 112] have developed two coprocessor boards for simulation of the SOM model (KOKOS: a coprocessor for Kohonen’s selforganizing map) and for the Backpropagation network (KOBOLD: a coprocessor for Backpropagation with online learning). The KOKOS coprocessor consist of 8 MABs (Memory and Arithmetic Board) utilizing weight level parallelism. The KOKOS implementation of the SOM network is described in section 4.2.3.1. The KOBOLD coprocessors consist of up to 128 subprocessors (SP). The subprocessors compute one neuron from each layer in parallel. A total of 8 layers can be used, with 128 neurons in each layer. Alternatively, 256 neurons in 4 layers can be used. The coprocessor boards have a PC-EISA interface. Software for controlling
the board and visualizing results has been constructed.

Melton [77] constructed a VLSI chip which simulates one neuron using the TInMANN (The Integer Markovian artificial neural network) variation of the original SOM model. The modifications simplify the hardware implementation. Each chip, or neuron, consists of a number of registers, two adders and some conditional logic. Several restrictions are made in the prototype implemented. There are only 6 registers available to hold the weights and the position of the neuron within the SOM network. Furthermore, the word length is restricted to 10 bit. The TInMANN implementation is further described in section 4.2.3. Several TInMANN circuits can be cascaded in order to construct larger networks.

Hirai [29] gives a review of hardware implementations of neural networks in Japan. Examples are given of both analog, digital and optical implementations. He summarizes the advantages and disadvantages of the different approaches in the following way:

**Analog:** Is compact and fast, but is sensitive to process parameter variation and noise, lacks scalability, has limited precision on weights and it is difficult to make modifiable synapses.

**Optical:** Can have large fan-in and fan-out and modifiable synapses. The disadvantages are the need for electro-optic and opto-electronic transformation and that it is sensitive to noise and parameter variation.

**Digital:** High precision and scalability and has modifiable weights. The disadvantage is the large circuit size.

### 3.3.6 Proposed architectures

There exists a number of articles where computer architectures are proposed. For the following architectures, we are not aware of any articles describing that an implementation of the architecture has actually been constructed.

Duranton and Sirat have proposed a VLSI circuit architecture for simulation of multiple ANN models [13]. The implementation of SOM, Backpropagation and Hopfield network on this chip is described. Each circuit contains 32 neurons with a maximum input vector dimensionality of 32. Several circuits can be cascaded in order to simulate larger networks.

Kotilainen et al [53, 54] have presented a multiprocessor architecture for a general purpose neurocomputer. The processing elements in a realization of the architecture is proposed to be Digital Signal Processors connected in a tree structure. It is described how
different ANN models can be mapped onto this architecture -- Backpropagation network, SOM and Sparse Distributed Memory (SDM).

Cavalieri et al [6] have proposed a DSP-based multiprocessor system for simulating ANNs. The system is based on the Texas Instruments TMS320C30, where all the processors are connected to a common bus. The processors are equipped with local memory and a bus interface which is realized in Field Programmable Gate Arrays. A description of how the Backpropagation algorithm can be mapped to the architecture is described.

Mann [68] proposed an analog implementation of the SOM model with modifiable weights. The implementation consists of two circuits. The first circuit is the neuron implementation. Four neurons reside on each circuit and the input vector dimensionality is 4 giving a total of 16 weights per circuit. The second circuit is for weight update and refresh. Some modifications on the original model have been made in order to get an easier implementation. The proposal is not complete, some of the functionality needed was still not implemented at the time the article referred to was written.
Chapter 4

Parallel Self-organizing Maps

4.1 The SOM model revisited

In this section, we present Kohonen's SOM model in detail, and discuss the model in a parallel context.

4.1.1 Background

In many areas the brain is organized in such a way that aspects of the sensory environment is represented in the form of two-dimensional maps. In for instance the visual system, there are several topographic mappings of visual space onto the surface of the visual cortex. Kohonen is attempting to construct an artificial system that exhibits the same behavior [3].

Kohonen made the discovery that topologically correct maps of structured distributions of signals can be formed in a one- or two-dimensional array of processing units which did not have this structure initially [47]. This discovery led to the artificial neural network model which we today know as the Self-organizing Maps.

Kohonen [49, 50, 51] has published a number of books and articles about several aspects of the Self-organizing Maps model. The presentation of the SOM model in this section is based on these references.
4.1.2 A detailed description

The SOM model exists in several variants. A common feature of these variants is an ordering of the neurons in the form of a neighborhood. However, they differ in the choice of basis function and update rule.

Let the weight vector related to neuron \( i \) be denoted \( \mathbf{m}_i = \{m_{i1}, m_{i2}, \ldots, m_{id}\} \) where \( d \) is the dimensionality of the input vector. A random training vector is denoted \( \mathbf{x} = \{x_1, x_2, \ldots, x_d\} \). Let \( c \) denote the index of the neuron which most resembles the training vector, referred to as the winning neuron or the best matching unit (BMU).

In the common variant of the SOM model, the neurons compute the following basis function:

\[
\text{out}_i = \| \mathbf{m}_i - \mathbf{x} \| \tag{4.1}
\]

where the norm \( (\| \cdot \|) \) is the Euclidean norm:

\[
\| \mathbf{a} \| = \sqrt{\sum_{j=1}^{d} a_j^2} \tag{4.2}
\]

where \( \mathbf{a} \) is an arbitrary \( d \)-dimensional vector. Combining the basis function given in equation 4.1 with the norm given in equation 4.2 we obtain:

\[
\text{out}_i = \| \mathbf{m}_i - \mathbf{x} \| = \sqrt{\sum_{j=1}^{d} (m_{ij} - x_j)^2} = \sqrt{(m_{i1} - x_1)^2 + \cdots + (m_{id} - x_d)^2} \tag{4.3}
\]

The winning neuron is then defined as the neuron with index \( c \) where:

\[
\| \mathbf{m}_c - \mathbf{x} \| = \min_{i} \| \mathbf{m}_i - \mathbf{x} \| \tag{4.4}
\]

The update rule reads:

\[
m_{ij}(t + 1) = m_{ij}(t) + \alpha(t) h(c, j, t)(x_j - m_{ij}(t)) \tag{4.5}
\]
where \( h(c, j, t) \) is the neighborhood function:

\[
h(c, j, t) = \begin{cases} 
1 & \text{if } j \in N_c(t) \\
0 & \text{if } j \notin N_c(t) 
\end{cases}
\] (4.6)

\( N_c(t) \) is the neighborhood set of \( c \) at time \( t \) and includes all the neurons which are to be updated. See section 4.1.3.3 for more details.

The other common variant of the SOM model has an other basis function and update rule. In this variant the training vectors and weights are normalized. The reason for normalizing the vectors is that the reference vectors then tend to have the same dynamic range and therefore the numerical accuracy may be improved. Instead of using the Euclidean norm, an inner product \( \langle \cdot, \cdot \rangle \) is used. In this case, the neurons compute the following function:

\[
\langle x^T, m_c \rangle = \max_i \langle x^T, m_i \rangle
\] (4.7)

and the update rule becomes

\[
m_{ij}(t + 1) = \begin{cases} 
\frac{m_{ij}(t) + \sigma'(t)x_j}{\|m_i + \sigma'(t)x\|} & \text{if } i \in N_c(t) \\
m_{ij}(t) & \text{if } i \notin N_c(t)
\end{cases}
\] (4.8)

As can be seen from the learning rule, the weights are normalized in each training iteration.

Other variants of the model exist, but these two are the variants most often encountered in the literature. Throughout the thesis, we will however be using the first presented model with the basis function given in equation 4.3 and the learning rule given in equation 4.5.

4.1.3 Practical considerations

4.1.3.1 The number of iterations

There does not exist any definite way of determining how many iterations a training session of a SOM network should have. According to Kohonen [51] a rule of thumb is that the number of iterations should at least be 500 times the number of neurons in the network.
Kohonen also states that the dimensionality of the training vectors does not effect the number of iterations required.

### 4.1.3.2 Initialization of the weights

The weights vectors are initialized to arbitrary values. The values may be randomly chosen but the weight vectors should initially be distinct.

### 4.1.3.3 Neighborhood

The neighborhood $N_c = N_c(t)$ decides how many of the neighboring neurons of the winning neuron should be updated. The size of the neighborhood should be large in the beginning of the training, covering most of the neurons, in order to assure a global ordering. The size of the neighborhood should also shrink with time. It is suggested that some time should be used to fine-tune the map by only updating the winning neuron. Figure 4.1 shows the development of a rectangular neighborhood set. In the figure, $t_1 < t_2 < t_3$. We will use such a rectangular neighborhood later in our parallel implementations.

![Diagram](image.png)

**Figure 4.1**: The development of the neighborhood function in a 2 dimensional SOM network
The neighborhood function can take several forms. Besides the rectangular form, there are two other much used functions. The first is a circular neighborhood where the circle is drawn around the winning neuron and all neurons within the circle are updated. The second is the Gaussian neighborhood, where a Gaussian function is centered on the best matching neuron. The update is then proportional to the value of the Gaussian function at each neuron. Using the Gaussian neighborhood, the neighborhood function becomes:

$$h(c, j, t) = \exp \left( -\frac{d^2}{\sigma^2} \right)$$ \hspace{1cm} (4.9)

When the Gaussian neighborhood function is used we have to update all the weights in each training iteration. Erwin et al [15] and later Siemon [103] have investigated the learning speed as a function of the parameters of the Gaussian neighborhood function.

### 4.1.3.4 Learning rate

The learning rate $\alpha = \alpha(t)$ is time dependent. The range of values it can take is $0 < \alpha < 1$. The value of $\alpha$ should be decreased with time. There is no definite statements as to how this decrease must be taken. We will throughout the thesis use a linearly decreasing learning rate.

### 4.1.4 Learning Vector Quantization

If the SOM network is to be used for pattern classification, supervised learning algorithms may be applied to the network. Learning Vector Quantization (LVQ) is the name of a set of such algorithms.

Kohonen in [51] describes 3 LVQ algorithms, denoted LVQ1, LVQ2 and LVQ3. In addition, Poirier [96] has suggested an alternative algorithm known as Dynamic Vector Quantization (DVQ). DVQ is, however, not based on SOM and does not have the topology preserving abilities which the SOM has and the LVQ maps to some extent inherit. A brief description of DVQ is included as it is often described together with the LVQ algorithms.

LVQ1 is the simplest algorithm. LVQ2 was made in order to comply better with Bayes’ decision making philosophy and LVQ3 is an improvement on this feature of LVQ2 [51].

In all the cases LVQ1, LVQ2 and LVQ3, the network is initially organized with the standard unsupervised SOM algorithm. When the initial training of the SOM network is
finished, the training vectors are marked with their class. The whole training set is then applied to the network and each neuron is marked with the class of the majority of the training vectors which had this neuron as the winner. After all neurons have been marked with a class, a new training procedure is started.

In LVQ1, the following training step is repeated: A training vector \( x \) is randomly selected. A winning neuron \( i \) is selected and is updated according to the following rule:

\[
m_i(t + 1) = m_i(t) + \alpha(x - m_i(t)) \tag{4.10}
\]

if \( x \) and neuron \( i \) belongs to the same class, otherwise

\[
m_i(t + 1) = m_i(t) - \alpha(x - m_i(t)) \tag{4.11}
\]

Note that equation 4.10 is identical to the normal SOM update rule (equation 4.5).

In LVQ2 an update can only occur if the nearest neuron \( (i) \) does not belong to the same class as the training vector \( x \). In this case, the nearest neuron from the same class as \( x \), neuron \( j \), must be identified. If the training vector falls within a window centered around the midplane between \( m_i \) and \( m_j \) then the neurons are updated according to the following rule:

\[
m_i(t + 1) = m_i(t) - \alpha(x - m_i(t)) \tag{4.12}
\]

\[
m_j(t + 1) = m_j(t) + \alpha(x - m_j(t)) \tag{4.13}
\]

otherwise,

\[
m_k(t + 1) = m_k(t), \forall k \tag{4.14}
\]

In LVQ3, the same update rule is used as in LVQ2 (equations 4.12 and 4.13). In addition, a modification is done if both the training vector and the two nearest neurons belong to the same class. Both these two vectors are then moved towards the training vector:

\[
m_k(t + 1) = m_k(t) + \epsilon \alpha(x - m_i(t)), k \in \{i, j\} \tag{4.15}
\]
\( \epsilon \) is a small constant, typically between 0.1 and 0.5.

DVQ differs from the LVQ algorithms by the fact that the number of neurons changes. Initially, there is one neuron for each class. If the best matching unit and the training example does not belong to the same class, and the nearest neuron which belongs to the same class as the training vector is too far away, then a new neuron is constructed, where the weights equals the training vector components.

\section*{4.2 Parallelization strategy}

As pointed out in chapter 3, there are several inherently parallel structures in neural network models. This is, of course, also the case for the SOM model. A number of different levels of parallelism can be exploited to parallelize the SOM algorithm.

\subsection*{4.2.1 Training example parallelism}

In training example parallelism, several instances of the SOM network exist on different processors, training several vectors simultaneously and exchanging the results obtained at each network instance.

Training example parallelism is not a common method used to parallelize the SOM model. Mann and Haykin [69] used this approach in their implementation on a 10 PE Warp system. The Warp modules were connected in a linear array operating in a systolic fashion. The systolic operation of the computer is reported as the main reason for using a training example parallel algorithm.

A training example parallel algorithm is implemented for RENNS [83], and this implementation is described further in section 6.4. A combination of training example and neuron parallelism has also been constructed for RENNS, see section 6.5.

\subsection*{4.2.2 Layer level parallelism}

Unlike the Backpropagation network, the SOM network is not a layered network model, unless the distribution of the training vectors is regarded as a layer. Since the input layer does not do any computing, nothing can be gained by allocating computational resources to the input layer.
A pipelined implementation of the SOM model has, however, been constructed by Ceccarelli et al [7]. In this implementation the neurons are divided among the processors. A linear chain of Transputers is used. A training vector is applied to the first processors, which computes its neurons and passes it forward to the next processor. After a training vector has passed through the pipeline, a winning neuron has been selected and the winner is passed through the pipeline once more in order to update the weights according to this winner. An implementation following the same approach has been constructed for RENNS [84].

4.2.3 Neuron parallelism

The by far most common way of implementing parallel SOM algorithms is by using neuron level parallelism. As will be seen from the following overview of published implementations, there are a number of differences in the implementations.

Obermayer et al [91, 92] implemented node parallel SOM algorithms on a Connection Machines CM-2 and on a network of Transputers connected in a linear array. In both implementations, the neurons compute an inner product as given in equation 4.7 and uses the corresponding update rule (equation 4.8). A Gaussian neighborhood function is used. The experiments are conducted on relatively large networks, the number of neurons ranging from a few hundred to tens of thousands, with an input vector dimensionality ranging from around 32 to 900. Performance results are given as wall clock time.

Hammarstrom and Nguyen [25] implemented a neuron parallel SOM algorithm on a CNAPS system consisting of 4 CNAPS circuits. The computer had a total of 256 PEs and used the standard SOM model. Finding the global maximum was achieved through a maximum function implemented on the CNAPS circuits. Performance measurements were made on a 2-dimensional network consisting of 512 neurons. The dimensionality of the input vector was 256. The performance is reported in CUPS.

Buhusi's [5] node parallel implementation uses a Balance 8000 computer system. The description of the implementation is somewhat unclear, but seemingly it is based on a dynamic load balancing scheme. When several neurons are to be computed or updated, a number of processes are started and assigned to available processors. The performance measurements are reported by the efficiency of the processors and as wall clock time. A 2-dimensional ordering was used, the number of neurons ranged from 100 to 900, and the dimensionality of the input vector ranged from 10 to 30.

Ceccarelli et al [7] have also made a neuron parallel implementation of the SOM model on their network of Transputers in addition to the pipelined algorithm. The processors are connected in a binary or ternary tree. The training vectors are passed down through
the tree, and each processor computes the local winner and passes it back up the tree. The global winner is finally passed down through the tree, and all modules update their weights accordingly. The neuron parallel algorithm is compared with the pipelined algorithm and the performance is reported in terms of speedup, efficiency and wall clock time. In the experiments they used 5000-15000 neurons and a 100 dimensional input vector. The dimensionality of the neuron ordering is not reported.

Demian and Mignot have made two implementations of the SOM algorithm [11, 12], both using neuron parallelism. The first implementation is constructed for a iPSC/860 MIMD system with an unspecified number of processors (probably eight since the speedup approaches eight as the size of the network increases) and a block update scheme. Local winners for several training vectors are computed on each processor. Then, the global winners are selected for these local winners. If a neuron is within the neighborhood of several winners, it is only updated towards the the training vector it was closest to. The second implementation uses a MasPar MP-1 computer of 1024 PEs. The MasPar implementation uses the same block updating scheme. The performance of the iPSC/860 implementation is reported in terms of speedup.

Marguerat [71] has constructed a number of neuron level parallel implementations for a system based on Texas Instruments TMS320C40 Digital Signal Processor. The difference between the various implementations is in the interconnection topology chosen – i.e. linear array, b-ary tree, broadcast bus, 2D Mesh and 2D Torus. Simplified theoretical models of algorithms using the different interconnection schemes are presented. The implemented algorithms are executed on a system consisting of four processors. The performance metric reported is speedup for networks ranging from 100 to 400 neurons and input dimensionality ranging from two to ten. A performance comparison is also given, where the measurements are reported in CPS.

Grajski [23] has implemented a neuron parallel SOM algorithm on the MasPar MP-1 computer. The implementation uses a computer with 4096 processors and one neuron per processing element. The inner product version of the basis function is used. Measurements are taken from training vectors with a dimensionality ranging from 2 to 256. The neighborhood is held constant in the measurements. Either all the neurons are updated for each training vector or just the nearest neighbors to the winning neuron.

Siemon and Ultsch implemented a neuron parallel SOM algorithm on a computer system consisting of 16 Transputers connected in a ring [104]. The implementation is straightforward and follows the standard SOM model. The performance measurements are reported as CUPS, and were conducted on a network consisting of 16K neurons, a 17 dimensional input vector and a 2 dimensional ordering of the neurons.

Hodges et al [30, 130] constructed a neuron parallel SOM implementation for a network of Transputers. The implementation uses a circular neighborhood function. Two different
implementations are constructed. In the first implementation, a linear array of Transputers is used, and in the second the Transputers are connected as a 2D torus. The two implementations use different mappings of the neurons to the processors. The performance metric reported is speedup. In the linear chain implementation, the number of processors ranges from 1 to 12, the number of neurons is 840 and the input vector dimensionality is 2. In the 2D torus implementation, measurements are taken from 1, 4 and 9 processors, the number of neurons is 1296 and the input vector dimensionality is 2.

Melton et al [77] uses neuron level parallelism in their hardware implementation of the TINMANN modified SOM model. The basis function of the neurons is the Manhattan distance – also known as the taxicab metric [55]. The use of this metric does not imply a change in the update rule. The update rule is the same as when a Euclidian distance measure is used. The selection of the neurons which are within the neighborhood is done according to the integer Markovian learning algorithm. The reason for doing these changes in the original algorithm is that the hardware realization becomes simpler. The implementation puts some restrictions on the problem parameters. There are 6 registers in each node which hold the neuron’s weights and the index in the SOM network. Using a 2 dimensional ordering, the dimensionality of the input vector cannot exceed four. Performance results report the number of training vectors which can be trained per second.

König et al have proposed two VLSI implementations of the SOM model [52]. The first proposal is to be a low cost implementation, featuring a pipelined Subtract-Multiply-Accumulate structure. In this first proposal, the Euclidean distance is used as a similarity measure. One such processor slice realizes one neuron, and takes care of the operations needed both for computing the neurons’ output values and for training. Several processor slices are combined in order to construct larger networks. The second implementation is based on simpler neurons, using the Manhattan distance as basis function. Using this simpler neuron model, it is possible to increase the number of neurons on each circuit.

Kotalainen et al [53, 54] have described a parallel SOM implementation for their proposed multiprocessor architecture. The implementation uses neuron level parallelism, and a tree structure is described in order to determine global winners.

Ienne et al [35] are implementing a neuron parallel SOM algorithm on the MANTRA I computer system. The development of the algorithm is undergoing a final phase. The proposed algorithm uses a batch updating scheme. A modification of the original SOM model is present in this implementation. This modification is that if there are several winners, then all of these are selected for updating. Since the algorithm has not been completed, there exists no performance results.

Nordström [89] has done an investigation of the SOM model in order to design a bit serial processing element for a SIMD computer which is especially tuned towards the SOM model. Nordström describes both a neuron parallel and a weight parallel processing ele-
ment. Predicted performance results are reported in CUPS.

Saarinen discusses a node parallel implementation of the SOM algorithm on programmable circuits [100] which are to be used as a coprocessor in a PC. The proposed architecture can process 16 neurons concurrently.

4.2.3.1 Weight level parallelism

Weight level parallelism is present in the KOKOS coprocessor constructed by Speckman et al [109, 110, 111, 112]. The coprocessor consists of Memory and Arithmetic Boards (MABs). The weights of a neuron are distributed to the different MABs. The MABs concurrently compute the distance between a weight and the corresponding component of the input vector. The MABs are connected in an adder tree, which adds the distances computed in each MAB. If the input vector dimensionality exceeds the number of MABs, the computation of the neuron is split. Training vectors are selected by the hosting PC and presented to the coprocessor, which selects the best matching unit. The identification of the winning neuron is passed back to the host, which computes the excitation matrix. In the presented results, a Gaussian neighborhood function was used. The performance results are reported in CUPS.

4.2.4 Load balancing

One of the necessary conditions which must be satisfied in order to achieve an efficiency close to 1, is to have the computational load evenly spread over the processing elements. This is the problem of load balancing.

4.2.4.1 Training example level

When using training example parallelism, load balancing is implicit in the model. All processing modules compute the same number of neurons, such that a balanced load is assured, assuming identical processing elements. During the update phase, some minor differences in computation time may be present. This is due to the fact that the neighborhood in some cases exceed the borders of the network. Since the different instances of the network computes different training vectors, the different instances may end up with different numbers of neurons to update.
4.2.4.2 Neuron level

At the neuron level, the problem of load balancing is the problem of how to map the neurons onto the processing elements. There are two computational phases in the SOM model. The first is the computation of the neurons’ outputs and the second is the updating of the weights.

Load balancing the first phase only requires that the processing elements compute the same number of neurons. This can be perfectly achieved if the number of neurons is a multiple of the number of processors. In this case, exactly \( \frac{n}{p} \) neurons are mapped to each processor. If the number of neurons is not a multiple of the number of processors, then we have to map \( n \text{ div } p \) neurons to each processor. In addition, the first \( n \mod p \) processors would be responsible for computing yet another neuron. This means that some processors need to compute one neuron more than the others (each processor computes either \( \lceil \frac{n}{p} \rceil \) or \( \lfloor \frac{n}{p} \rfloor \) neurons). If the number of neurons per processor is high, the load balancing is kept at a fairly high level, since no processor is responsible for more than one neuron more than any other processor.

In the updating phase, a load balanced system should assure that no processor is responsible for updating more than \( \lceil \frac{n_{\text{upd}}}{p} \rceil \) and less than \( \lfloor \frac{n_{\text{upd}}}{p} \rfloor \), where \( n_{\text{upd}} \) is the number of neurons in the neighborhood of the winning neuron. The size of the neighborhood changes, so that even if the number of neurons is a multiple of the number of processors, there will be situations where the number of neurons within the neighborhood does not the divide number of processors, and a perfect load balancing can not be achieved.

The 1-dimensional case A mapping of a 1-dimensional SOM network is given in figure 4.2. Processor number \( i \) is responsible for each neuron \( j \) where \( j \mod p = i \). The conversion from a local index to a global index is very straightforward:

\[
\text{global index} = \text{local index} \times p + i
\]  
(4.16)

The 2-dimensional case Load balancing is more difficult in the 2-dimensional case. The problem has been addressed by Hodges et al [30, 130], Demian et al [11, 12], Petrowski et al [95] and Obermayer et al [91]. The easiest mapping is one of the mappings suggested by Obermayer. The mapping is shown in figure 4.3. The map is vertically divided into \( p \) slices, giving a mapping of one slice per processor. The mapping ensures that, approximately, the same number of neurons are mapped to each processor depending on if the number of neurons in the horizontal direction divides the number of processors, but in the update
4.2. Parallelization strategy

![Diagram](image)

Figure 4.2: Mapping of a 1 dimensional SOM chain onto a set of processors

phase, there are large differences in the workload on the processors. In the worst case, all updating is done by a single processor, where the number of neurons which are to be updated is the square of the slicewidth – if a rectangular neighborhood is used.

The second mapping Obermayer et al used was the mapping which was first introduced by Hodges et al. The mapping is shown in figure 4.4. In this mapping, there is one slice for each column of neurons and the slices are interleaved when mapped onto the processors. The load balancing of the first phase of the computation is the same as in the previous mapping. In the update phase however, a larger degree of load balancing is obtained compared to the previous mapping.

A third mapping has been proposed by both Hodges et al and Demian et al in order to obtain a more even number of neurons on each processor in the update phase. One weakness in the mapping is that it assumes that the number of processors is a square number. The mapping is outlined in figure 4.5. The patterns of the neurons indicate which processor they are mapped on.

The mapping is made by placing the neuron with position \((i, j)\) in the grid, on processor \((i \mod \sqrt{p}, j \mod \sqrt{p})\).

Demian et al makes a change in this mapping because the algorithm he uses is sensitive to the fact that neurons on the border of the map have a lower probability to be selected. In the mapping in figure 4.5, all border neurons are mapped to the processors on the edges of the processor mesh. This is however no problem when the update is calculated for each training vector – it merely means that it is always the same processors which have slightly less work during the update phase.
4.3 Performance

Measuring performance of an SOM algorithm is a science in its own. As is the case with other ANN models, the performance is very dependent on the network parameters such as dimensionality of the input vector and the number of neurons in the network. However, there exists no de facto benchmark for SOM networks, resulting in incomparable performance measurements between different research groups. This is very clear from table 4.1, where a presentation of results is given for the different implementations.

4.3.1 Parameters influencing measurements

The network parameters have crucial influence on the performance measurements. In the following, we identify these parameters and describe how they influence on the performance.
Figure 4.4: Hodges’ mapping of a 2 dimensional SOM network onto a set of processors

4.3.1.1 Neighborhood

The neighborhood influences the performance measurements in two ways. The first is by the dimensionality of the neighborhood. The second is by what the size of the neighborhood is when the performance measurements are done, since this size is time dependent, and the performance varies with this size.

The dimensionality of the neighborhood influences how many neurons are updated for each training vector. Assume that the size of the neighborhood decreases linearly, starting with updating the whole map and continuing until only the winner is updated. It should be noted that we do not take into account the effect where the neighborhood exceeds the map. If the number of neurons is \( n \), where \( n \) is an odd number, then using a 1-dimensional ordering we first update all \( n \) neurons, then \( n - 2 \) neurons, \( \cdots \), 5 neurons, 3 neurons, and finally one neuron. In average, the number of neurons updated is:
Figure 4.5: Demian’s mapping of a 2 dimensional SOM network onto a set of processors
4.3. Performance

\[ n_{\text{upd1D}} = \frac{\sum_{i=1}^{n+1} (2i - 1)}{n+1} \]
\[ = 2^{\frac{n+1}{2}(\frac{n+1}{2}+1)} \frac{n+1}{2} - \frac{n+1}{2} \]
\[ = \frac{(n+1)^2}{2} + \frac{n+1}{2} - \frac{n+1}{2} \]
\[ = \frac{n + 1}{2} \]

(4.17)

In average, \( \frac{n+1}{2} \) neurons will be updated for each iteration or in other words, the fraction of updated neurons per iteration is:

\[ \kappa_{1D} = \frac{\frac{n+1}{2}}{n} = \frac{n + 1}{2n} \]

(4.18)

\[ \lim_{n \to \infty} \kappa_{1D} = \frac{1}{2} \]

(4.19)

For large networks, this means that we in average update one half of the neurons in the network.

For a two dimensional network, we make the assumption that the number of neurons \( n \) is a square number and that the neurons are arranged in a \( n_i \times n_i \) grid. Training this network we would first have to update \( n_i^2 \) neurons, then \( (n_i - 1)^2 \) neurons, \( \cdots \), then 9 and finally only the winning neuron. In average, the number of neurons updated is – substituting \( x \) for \( \frac{n+1}{2} \):

\[ n_{\text{upd2D}} = \frac{\sum_{j=1}^{n_i+1} (2j - 1)^2}{n_i+1} \]
\[ = \frac{\sum_{j=1}^{n_i+1} (4j^2 - 4j + 1)}{n_i+1} \]
\[ = \frac{x}{4} \left( \frac{x(x+1)(2x+1)}{6} - \frac{4x(x+1)}{2} + x \right) \]
\[ = \frac{2}{3} (2x^2 + 3x + 1) - 2(x + 1) + 1 \]
\[ \begin{align*}
\frac{2}{3}(2x^2 + 1) &+ 2x - 2x - 1 \\
= \frac{2}{3}(2x^2 + 1) - 1 \\
= \frac{2}{3}(2\left(\frac{n_i + 1}{2}\right)^2 + 1) - 1 \\
= \frac{1}{3}(n_i^2 + 2n_i + 1) + 1 - 1 \\
= \frac{1}{3}(n_i^2 + 2n_i + 3) - 1 \\
= \frac{n_i^2 + 2n_i}{3} \\
= \frac{n + 2\sqrt{n}}{3} 
\end{align*} \] 

(4.20)

The fraction of updated neurons in a training iteration hence becomes:

\[ \kappa_{2D} = \frac{n + 2\sqrt{n}}{n} = \frac{n + 2\sqrt{n}}{3n} \] 

(4.21)

\[ \lim_{n \to \infty} \kappa_{2D} = \frac{1}{3} \] 

(4.22)

In other words, the fraction of neurons updated per iteration is less in the two dimensional case than in the one dimensional case. Since the amount of computation in the forward pass is equivalent in the two cases, this means that the one dimensional case uses a larger fraction of its time for updating weights, resulting in a larger CUPS measurement. Equations 4.18, 4.19, 4.21 and 4.22 are graphically displayed in figure 4.6.

Performance of a parallel SOM implementation is often reported in CUPS i.e. how many weights are updated per second. In the SOM model, the CUPS measurement will vary with the size of the neighborhood, since the output values of all the neurons must be computed independent of the size of the neighborhood. This computation of the winning neuron does, however, not contribute to the performance measurement. If the neighborhood is large, then a larger percentage of the total time is due to updating weights than is the case if if the neighborhood is small. There are even examples where the reported results are calculated as the product of weights and number of training iterations per time unit, even though many of the weights are not updated. This is for instance the case in Mann and Haykin’s performance measurements [69].
4.3. Performance

Figure 4.6: Fraction of neurons updated in the 1 and 2 dimensional case

4.3.1.2 The number of neurons and dimensionality of training vector

The numbers of neurons and the dimensionality of the training vector greatly influence performance. The communications in a simple neuron parallel algorithm are not dependent on the number of neurons in the network, and are only dependent on the size of the training vector in the case where all the training vectors are distributed to the processors before the training starts. By increasing the number of neurons and the dimensionality of the training vector, the percentage of the time used for communication decreases and a larger percentage of time is used for computation.

The amount of computation which has to be done, both in the computing phase and in the update phase, is dependent on the product of the number of neurons and the dimensionality of the training vector.
4.3.1.3 Update frequency

If the update of the weights is accumulated instead of being added to the weights before the next training vector is applied, then the number of iterations between each update influences the performance. For the backpropagation algorithm, experiments have shown that the number of iterations required for a network to learn a mapping increases when the interval between weight updates is increased [120]. Hence, for the backpropagation algorithm, the CUPS measurement for online and offline algorithms are not necessarily directly comparable.

For the SOM network the effect of using offline algorithms is also applicable. If the weights are updated too seldom then the network may not manage to learn [83]. The decision as to how often the weights should be updated is dependent on the data and therefore an investigation of the data at hand should be carried out.

4.3.2 Previous measurements

In table 4.1 the performance of parallel SOM implementations reported by various researchers is presented. As can be seen from the table, large differences in the parameters which influence the measurements are present, making the measurements difficult to compare. In many cases, it is not possible to read from the article exactly what is measured.

4.4 Applications of the SOM model

The main reason for designing parallel SOM algorithms is to speed up the execution of applications of the model and to make the SOM model applicable to problems which have been considered too time consuming to be practically implemented. However, the researchers who develop applications of the SOM model are seldom the same researchers who develop the parallel implementations of the SOM model.

Table 4.2 presents the problem sizes for applications of the SOM model reported by various researchers. A broad range of applications are present. It can be observed that most of the applications uses networks of a limited size. This holds both for the number of neurons and the dimensionality of the input vector. The number of operations in one training cycle in the SOM model (see equations 4.3 and 4.5) is dependent on the product of these two parameters.

The applications of the SOM model are typically based on relatively small networks
4.4. Applications of the SOM model

despite the fact that parallel algorithms for the model have existed for several years. The reasons for this could be the unavailability of parallel computers and/or the parallel implementations of the SOM model for application developers. For larger problems, these two fields need to be coupled. Otherwise, the large amount of time needed to train the networks will make it impractical to develop applications requiring large networks.

For smaller problems, however, many applications are already developed. Introducing parallel algorithms for these problems will not necessarily generate any new applications, but it may make the applications more competitive to traditional models for solving the problems.
<table>
<thead>
<tr>
<th>Author</th>
<th>Computer</th>
<th>Problem size</th>
<th>Performance</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buhusi</td>
<td>Balance 8000</td>
<td>100-900 neurons</td>
<td>$E_{10} = 0.68-0.95$</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>10 PE</td>
<td>10-30 inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ceccarelli</td>
<td>Transputer</td>
<td>5000 neurons</td>
<td>$S_{60} = 47$</td>
<td>Layer parallel</td>
</tr>
<tr>
<td></td>
<td>60 PE</td>
<td>100 inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ceccarelli</td>
<td>Transputer</td>
<td>5000 neurons</td>
<td>$S_{60} = 38-43$</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>60 PE</td>
<td>100 inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demian</td>
<td>MasPar MP-1</td>
<td>1024 neurons</td>
<td>Not given</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>1024 PE</td>
<td>2 inputs</td>
<td></td>
<td>Block version</td>
</tr>
<tr>
<td>Demian</td>
<td>iPSC/860</td>
<td>64-25600</td>
<td>$S_8 = 0.5-7.5$</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>8 PE</td>
<td>unknown input</td>
<td></td>
<td>Block version</td>
</tr>
<tr>
<td>Grajski</td>
<td>MasPar MP-1</td>
<td>4096 neurons</td>
<td>7.4-18.0 MCUPS</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>4096 PE</td>
<td>2-256 inputs</td>
<td></td>
<td>IP version</td>
</tr>
<tr>
<td>Hammarstrom</td>
<td>CNAPS</td>
<td>512 neurons</td>
<td>54-65 MCUPS</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>256 PE</td>
<td>256 input</td>
<td></td>
<td>8 bit-16 bit</td>
</tr>
<tr>
<td>Hodges</td>
<td>Transputer</td>
<td>840-1296 neurons</td>
<td>$S_{12} = 11.5$</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>12 PE</td>
<td>2 inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mann</td>
<td>Warp</td>
<td>1024 neurons</td>
<td>8-12.5 MCUPS</td>
<td>Training example</td>
</tr>
<tr>
<td></td>
<td>10 PE</td>
<td>16-128 inputs</td>
<td>(see 4.3.1.1, last</td>
<td>parallel. Epoch size 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>paragraph)</td>
<td></td>
</tr>
<tr>
<td>Marguerat</td>
<td>TMS320C40</td>
<td>100-400 neurons</td>
<td>$S_4 = 2.5-3.4$</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>4 PE</td>
<td>2-10 inputs</td>
<td>8.4 MCPS</td>
<td></td>
</tr>
<tr>
<td>Melton</td>
<td>TinMANN</td>
<td>1 neuron</td>
<td>Wall clock time</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>1 PE</td>
<td>3 inputs</td>
<td></td>
<td>Tinmann version</td>
</tr>
<tr>
<td>Nordström</td>
<td>REMAP</td>
<td>1024-2048 neurons</td>
<td>17.5-280 MCUPS</td>
<td>8 bit-16 bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10-128 inputs</td>
<td>(estimated)</td>
<td></td>
</tr>
<tr>
<td>Obermayer</td>
<td>CM-2</td>
<td>16K-128K neurons</td>
<td>Wall clock time</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>16K PE</td>
<td>30-800 inputs</td>
<td></td>
<td>IP version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Gaussian</td>
</tr>
<tr>
<td>Obermayer</td>
<td>Transputer</td>
<td>2K-22K neurons</td>
<td>Wall clock time</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>30 PE</td>
<td>30-250 inputs</td>
<td></td>
<td>IP version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Gaussian</td>
</tr>
<tr>
<td>Siemon</td>
<td>Transputer</td>
<td>16K neurons</td>
<td>2.7 MCUPS</td>
<td>Neuron parallel</td>
</tr>
<tr>
<td></td>
<td>16 PE</td>
<td>17 inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speckman</td>
<td>COCOS</td>
<td>3600 neurons</td>
<td>16MCUPS</td>
<td>Weight parallel</td>
</tr>
<tr>
<td></td>
<td>8 PE</td>
<td>8 inputs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Performance of SOM implementations
<table>
<thead>
<tr>
<th>Author</th>
<th>Description</th>
<th>Neurons</th>
<th>Input dim.</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohen [9]</td>
<td>Synthetic speech</td>
<td>96</td>
<td>9</td>
<td>2D</td>
</tr>
<tr>
<td>Elo [14]</td>
<td>EEG Classification</td>
<td>500</td>
<td>12</td>
<td>1D</td>
</tr>
<tr>
<td>De Haan [10]</td>
<td>Digit Recognition</td>
<td>400</td>
<td>17</td>
<td>2D</td>
</tr>
<tr>
<td>Garavaglia [18]</td>
<td>Data analysis</td>
<td>64</td>
<td>12</td>
<td>2D</td>
</tr>
<tr>
<td>Gemello [19]</td>
<td>Contour Detection</td>
<td>256</td>
<td>2</td>
<td>1D</td>
</tr>
<tr>
<td>Ghosal [20]</td>
<td>Range Image Segmentation</td>
<td>3</td>
<td>9</td>
<td>1D</td>
</tr>
<tr>
<td>Göppert [21]</td>
<td>Gas concentration</td>
<td>144</td>
<td>32</td>
<td>2D</td>
</tr>
<tr>
<td>Göppert [21]</td>
<td>Corrosion</td>
<td>625</td>
<td>26-512</td>
<td>2D</td>
</tr>
<tr>
<td>Hogden [31]</td>
<td>Tracking objects</td>
<td>Unknown</td>
<td>256</td>
<td>2D</td>
</tr>
<tr>
<td>Kallio [40]</td>
<td>Lung Sound Classification</td>
<td>120</td>
<td>16</td>
<td>2D</td>
</tr>
<tr>
<td>Kang [41]</td>
<td>Color Classification</td>
<td>256</td>
<td>3</td>
<td>2D</td>
</tr>
<tr>
<td>Kangas [42]</td>
<td>Image Transmission</td>
<td>512</td>
<td>16</td>
<td>2D</td>
</tr>
<tr>
<td>Kaski [43]</td>
<td>EEG Monitoring</td>
<td>154</td>
<td>80</td>
<td>2D</td>
</tr>
<tr>
<td>Kita [46]</td>
<td>Tonotic Map</td>
<td>400</td>
<td>144</td>
<td>2D</td>
</tr>
<tr>
<td>Kohonen [48]</td>
<td>Speech recognition</td>
<td>96</td>
<td>15</td>
<td>2D</td>
</tr>
<tr>
<td>Lu [66]</td>
<td>Pattern Classification</td>
<td>12/400</td>
<td>36</td>
<td>1D/2D</td>
</tr>
<tr>
<td>Manduca [67]</td>
<td>Visualization</td>
<td>256</td>
<td>4</td>
<td>1D</td>
</tr>
<tr>
<td>Mann [70]</td>
<td>Radar Clutter Classification</td>
<td>100-400</td>
<td>11</td>
<td>2D</td>
</tr>
<tr>
<td>Tryba [122]</td>
<td>Process Control</td>
<td>4761</td>
<td>11</td>
<td>2D</td>
</tr>
<tr>
<td>Walter [125]</td>
<td>Non-linear Prediction</td>
<td>144</td>
<td>Unknown</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Table 4.2:** Problem parameters for some reported applications of Self-organizing Maps
Chapter 5

The RENNS Computer System

5.1 History

The RENNS project was started up in the end of 1989 by Professor Olav Landsverk. The main goal of the project was to construct application specific hardware. Artificial Neural Networks was chosen as the application area.

The design of the RENNS computer was undertaken in 1990 and 1991. The circuit boards were produced late in 1991. Design of the Processor Subsystem was undertaken by Jarle Greipsland and Haakon Dahle, whereas the Communication Subsystem board was designed by Jon Solheim, Lisbet Utne and Jan Anders Mathisen. The first version of the Communication Subsystem was erroneous and required a minor revision. This final revision was finished in the spring of 1992.

From mid 1992 to the spring of 1994, effort was concentrated on getting RENNS fully operational. The main part of this work was to construct the interior logic of the Field Programmable Gate Arrays on the Communication Subsystem which was completed by Jon Solheim and Lisbet Utne.

By the end of 1993, 10 RENNS modules were operational, and has since that time been used for simulation of neural networks. The final 6 RENNS modules, giving a total of 16 modules, were built in mid 1994. Simulations executed on the RENNS computer system include the Backpropagation network, the Hopfield network, and the Self-organizing Maps.
5.2 RENNS Architecture

Both the RENNS architecture and the implementation of this architecture have been described in several papers and theses [59, 60, 61, 62, 106, 107, 108, 124]. This description of the architecture and its implementation is based on these references.

The RENNS architecture is based on two main building blocks. These are the Processing Module (PM) and the Data Stream (DS). Both the Processing Modules and the Data Streams are optimized for operations on vectors.

The Processing Module's task is to do vector operations. It accepts vectors as inputs, and produces new vectors as outputs. The task of each of the Data Streams is to interconnect two or more RENNS modules so that vectors can be passed from one module to another. The Data Streams are reconfigurable, and support several modes of operation.

![Diagram of RENNS Architecture](image)

Figure 5.1: The building blocks in the RENNS architecture - the Processing Module with associated Data Streams

The RENNS architecture is illustrated in figure 5.1. The figure shows a Processing Module and its associated Data Streams. The number of Data Streams associated to each Processing Module is an implementation issue. As will be seen in the next section, the current implementation of the RENNS module contains 8 such Data Streams.

The architecture specifies a series of modes which must be supported by the Data Streams. These are Send, Receive, Bypass, and Broadcast. The different operational modes are illustrated in figure 5.2.
5.3 RENNS implementation

The RENNS Computer System is an implementation of the RENNS Architecture. The main building block in the RENNS Computer System is the RENNS Module. The RENNS Module contains a Processing Subsystem (PS) and a Communication Subsystem (CS). The RENNS Module implements one Processing Module and eight Data Streams.

A block diagram of the RENNS module is given in Figure 5.2.

Figure 5.2: Operational modes of the Data Streams

5.3.1 The Processing Subsystem

A block diagram of the Processing Subsystem is given in Figure 5.4. The processor on the PS is Texas Instruments' TMS320C30 Digital Signal Processor (DSP). A DSP was chosen due to its performance on executing floating point operations. The DSP on the PS is clocked at 32MHz and can achieve a peak performance of 32MFLOPS when calculating inner products.

Each PS is equipped with 4-16 MByte of DRAM and 256 KByte-1 MByte of SRAM. In addition, the DSP is equipped with an 8KByte internal SRAM which can be used for programs or data. The code is usually placed in external SRAM. The rest of the external SRAM, the DRAM, and the internal SRAM can be used for user data.

A VME interface was placed on the PS to give access to the PS from a host computer situated in the same VME rack, see section 5.5.2 for more information.
5.3.2 The Communication Subsystem

A block diagram of the communication subsystem is given in figure 5.5, and an overview of the data paths within the communication subsystem is given in figure 5.6.

The main components of the communication subsystem are the FIFO-memory, the data path control, the FIFO control, and the data stream control. The latter three of these components are realized in FPGAs which are all in-system reprogrammable Logic Cell Arrays [131]. The communication subsystem features eight data streams, which can be connected to communication subsystems on other RENNS modules. The data streams are independent, unidirectional (half-duplex) byte-wide communication channels. In addition to the eight bits of each channel, a 9th control bit is added when values are sent from the processor to the FIFO.

The processor accesses the communication subsystem through memory-mapped FIFO-banks. There are four FIFO-banks present, two for incoming data and two for outgoing data (all directional references are seen from the processor side). The FIFO-banks have several flags which can be read by the processor. The FIFO-flags must be read prior to accessing the FIFO-banks in order to assure correct operation, and avoid situations like...
writing to a full FIFO-bank or reading from an empty FIFO-bank.

The routing of data between the FIFO-banks and the Stream Controllers is done through the DEMUX and MUX circuits. The DEMUX circuit can connect any of the outgoing FIFO-banks to any Stream Controller, and the MUX circuit can connect any Stream Controller to any incoming FIFO-bank.

The FIFO controller controls the transmission of data between the FIFO-banks and the Stream Controllers. This includes converting data between word and byte format. The FIFO controller is also responsible for setting up the paths within the MUX and DEMUX circuits.

The four FPGA-based Stream Controllers are responsible for transmitting data to and receiving data from the interprocessor network. Each of these Stream Controllers is responsible for two Data Streams. The communication protocol is determined by the interior logic programmed into the Stream Controllers. Several configurations for these controllers have been constructed, and a brief description of the protocols is given in section 5.4.

As there are only two incoming and two outgoing FIFO-banks, but four Stream Controllers, there is not a dedicated pair of FIFO-banks to each Stream Controller. If all the eight data streams are to be used at the same time, then some kind of sharing the FIFO-banks has to take place. Such sharing is not used in the algorithms presented in this thesis.

The communication on a data stream follows a simple asynchronous protocol. Current peak performance of one data stream is 5Mbyte per second i.e. the peak performance of a
Communication Subsystem is 40 MByte per second.

5.3.3 Reconfiguration of the Communication Network

The communication network was implemented using reconfigurable components in order to meet the varying communication demands of the different ANN algorithms. The network was made reconfigurable on three levels: the data path level, the logic level, and the command level.

Data-path level Reconfigurability at the data path level involves changing the interconnection structure of the RENNS modules. Reconfigurability at this level was planned to be achieved by a programmable switch. However, the communication switch has not, at yet, been constructed, and reconfiguration at this level is done by manual recabling.
Figure 5.6: Overview of the data paths in the Communication Subsystem
Logic level The operation of the communication subsystem is defined by the internal logic programmed into the FPGAs which can be reconfigured by the processor. It is mainly the Stream Controller which is subject to reconfiguration at this level. Reconfiguration can be achieved under simulation, but has up to now only been done between simulations.

Command level The different configurations in the FPGAs of the communication subsystem will normally include several command registers which can be programmed by the processor. The number of registers and their functions are defined by the current state of the logic level. By changing the contents of such registers one can, for instance, change the interior data path of the communication subsystem.

5.4 Communication Subsystem Configurations

A number of configurations for the Communication Subsystem have been developed by Solheim and Utne. There are two configurations designed for intermodule data communication, and two configurations for utility purposes.

Most of the Communication Subsystem is realized in reconfigurable FPGAs. The configurations of the MUX, DEMUX, and the FIFO-controller are, however, the same in all the configurations constructed so far. It is in the Stream Controllers that the reconfigurability exploits.

5.4.1 Communication configurations

The two implemented communication configurations for the Stream Controllers are the Vector configuration and the Word configuration. In the parallel implementations of the SOM model – presented in the next chapter, we have used the Vector configuration, and as such is described in greater detail than the Word configuration

5.4.1.1 The Vector configuration

The Vector configuration was the first configuration constructed. Besides this, its general nature has made it the most used configuration in the RENNS Computer System. The Vector configuration implements a token based protocol, but it can also be set up for point-to-point communication.
When the Vector configuration is used as intended i.e. as a token ring, the modules are connected in a ring. A minimum of 4 modules must be present in each ring. If less than 4 modules are to communicate using the Vector configuration, then several Stream Controllers from one or more of the modules must be inserted into the ring such that the total number of Stream Controllers in the ring is at least 4. An example on this situation is given in figure 5.7 where initially only 3 modules wish to communicate. A fourth Stream Controller is inserted - a dummy Stream Controller, by using Stream Controller #2 from the first module. The address of this extra inserted Stream Controller must be set such that it does not attempt to receive any data.

![Figure 5.7: Vector protocol with dummy Stream Controller inserted](image)

Initially, one of the modules sets a token on the ring. A module which wants to send data takes the passing token from the ring, and then inserts a packet in the ring. The format of the packet is given in figure 5.8.

![Figure 5.8: A packet in the Vector protocol](image)

The address field consists of 4 bytes, but only 4 bits are used for addressing, allowing a total of $2^4 = 16$ unique addresses. The address of the Stream Controller is set from the processor by writing to a register in the Stream Controller. There are no restrictions on the length of a packet, but in practical use, it is restricted by the size of the FIFO-banks.
5.4.1.2 The Word configuration

The Word configuration is a specialized configuration designed for the scenario where all modules wish to communicate the same amount of data to all other modules. Using this configuration, each module sends one word at a time to all the other modules in a Round Robin fashion.

The Word configuration has not been used in any of the SOM implementations, but has been used in the implementations of the Backpropagation network and the Hopfield network.

5.4.2 Utility configurations

Two utility configurations have been constructed. The first is named the Check configuration. The Check configuration is typically used on power-up and for diagnostics. The Check configuration reports current connections to other RENNS modules and cable damages, if any.

The other utility configuration is called the Monitor configuration. A Stream Controller using this configuration is inserted between two communicating modules, and can report statistics on the traffic.

5.4.3 The configuration used in the implementations

All but one of the implementations to be described in chapter 6 use the Vector configuration in the ordinary way. The exception is the NP-Wave implementation (see section 6.3).

In the NP-Wave implementation, the modules only communicate with their succeeding module, requiring only a unidirectional channel between each module and its succeeding module. Using a token based protocol for this purpose is more complex than necessary as there is only one transmitter and one receiver on each ring. In this case, the sender would have to wait for the token each time it wanted to transmit a message. It would also result in an extra overhead, since the address and the token would need to be retransmitted. An alternative use of the vector configuration is to connect the modules as indicated in figure 5.9.

Each module is set up with a connection from its Stream Controller #2 to the succeeding module’s Stream Controller #1. Stream Controllers #1 and #2 on each module are set
up with addresses 1 and 2 respectively. Each module initializes one of the connections by sending through Stream Controller 2 (on the sender module) the address of the Stream Controller on the succeeding module (address=1). The token is never inserted on the ring. Since there are no limitation on the size of a packet, we have an open connection from Stream Controller #2 in one module to Stream Controller #1 in the succeeding module.

5.4.4 Modeling the Vector configuration

In order to develop simplified models of the parallel implementations as are described in chapter 6, we require expressions for the communication time for each of these implementations. Solheim [106] in his PhD work, has developed expressions for the time taken for various operations using the Vector configuration. Some of the expressions presented in this section are taken directly from Solheim’s thesis, the others are combinations of his expressions.

5.4.4.1 Normal sending of a vector without competition

If one module wishes to send a vector of length \( l \) words to another module, with \( p_i \) intermediate modules between the sender and the receiver, and no other modules are attempting to communicate, then this can be done in the following time \( (\ell_{vec}) \) [106]:

Figure 5.9: Communication in the NP-Wave implementation
\[ t_{vec}(p_i, p, l) = \left( \frac{5}{2} t_{cc} + t_{sdel} \right)p_i + \left( \frac{5}{4} t_{cc} + \frac{1}{2} t_{sdel} \right)p + 8t_{cc}l + \frac{94}{4} t_{cc} + t_{sdel} \] (5.1)

Where \( t_{cc} \) is the cycle time of the communication subsystem, and \( t_{sdel} \) is the propagation time between two Stream Controllers. The expression is constructed by adding the time needed to:

- move the data from FIFO the the Stream Controller \( t_{wout} = \frac{7}{2} t_{cc} \)
- wait for the token \( t_{tok} = \frac{5}{2} t_{byp} = \frac{5}{2} \left( \frac{5}{2} t_{cc} + t_{sdel} \right) \)
- pass the \( p_i \) intermediate modules \( p_i t_{byp} = p_i \left( \frac{5}{2} t_{cc} + t_{sdel} \right) \)
- incremental cost per word in the package \( t_{word} = 8t_{cc} \)
- synchronize at the last Stream Controller \( t_{ins} = 2t_{cc} + t_{sdel} \)
- remove the address at the receiver \( t_{remaddr} = 7t_{cc} \)
- move the incoming data from the Stream Controller to the FIFO \( t_{win} = 9t_{cc} \)

In the original equations [106] there are three additional constant terms, covering possible delays. These constants are omitted in this thesis for simplicity reasons.

The present maximum value of the cycle time is \( t_{cc} = 100 ns \). A value of 60 ns has been measured for \( t_{sdel} \). Using these values, we obtain the following equation:

\[ t_{vec}(p_i, p, l) = p_i 310ns + p 155ns + l 800ns + 2410ns \] (5.2)

For the rest of the equations, we will assume \( t_{cc} = 100 ns \) and \( t_{sdel} = 60 ns \).

5.4.4.2 All to all communication

In the case were all modules wish to broadcast block of length \( l \) we do not need to take into account acquisition of the token. The time needed before the first module can start sending is given by \( t_{wout} \). After this, the vector is passed around the ring. The time needed to do this is \( p_i t_{byp} + l t_{word} \). In addition, the time needed from when the last Stream Controller
has received the last vector until this is available to the processor through the FIFO-bank is \( t_{ins} + t_{remaddr} + t_{win} \).

The total time needed for all the modules to send a vector of length \( l \) to all the other modules hence becomes (values have been inserted for \( t_{cc} \) and \( t_{adel} \)):

\[
 t_{allall}(p, l) = 350ns + p(p310ns + l800ns) + 1860ns = p(p310ns + l800ns) + 2210ns
\]

(5.3)

### 5.4.4.3 All to one communication

Another typical situation is when one module is to receive data from all the other modules. In the previous case (section 5.4.4.2), all the intermediate modules receive the data on the ring. In the current Vector implementation, a module is not able to receive the data in a packet and also pick up the token at the end of the vector. In the all to one case, a module can start sending once it has taken off the token on the end of the current vector on the ring.

The time needed from when the communication starts to the receiving module receives data in its incoming FIFO-bank is (in average) \( t_{wout} + \frac{p}{2} t_{byp} + t_{ins} + t_{remaddr} + t_{win} \). Thereafter, it will receive \( p - 1 \) transmissions, each taking \( lt_{word} \). There are, however, \( p - 2 \) intervals between the \( p - 1 \) data transmissions, each taking approximately \( t_{byp} \). In summary this gives rise to:

\[
 t_{alone}(p, l) = 350ns + p155 + 1860ns + (p - 1)l800ns + (p - 2)310ns \\
= p(155ns + l800ns + 310ns) - l800ns + 350ns + 1860 - 6200ns \\
= p(465ns + l800ns) - l800ns + 1590ns
\]

(5.4)

### 5.4.4.4 Point-to-point communication

When the Vector configuration is used for point-to-point communication (as described in section 5.4.3) then the time needed to send a message of \( l \) words from one module to another with \( p_i \) intermediate modules, is composed of the time needed for the data to pass through the Communication Subsystem \( t_{wout} \), the time needed to pass the intermediate modules \( (p_it_{byp}) \), the time needed to send the rest of the message \( (8t_{word}) \), and the time
needed for the packet to pass through the communication subsystem in the receiving Stream Controller \((t_{ins} + t_{win})\). As such, \(t_{direct}\) can be given as:

\[
t_{direct}(p_i, l) = 350ns + p_i310ns + 1800ns + 1160ns
= p_i(310ns) + 1800ns + 1510ns
\]  

(5.5)

These four functions, \(t_{vec}(p_i, p, l)\), \(t_{allali}(p, l)\), \(t_{alilone}(p, l)\), and \(t_{direct}(p_i, l)\), are all used in chapter 6 in the models of the parallel algorithms.

5.5 The RENNS environment

5.5.1 Program development

All the programs developed for RENNS have been written in the high-level programming language C [45]. The standard optimizing compiler used, from Texas Instruments [37], contains no parallel extensions of the programming language. Development was undertaken on UNIX workstations. Due to the lack of debugging tools, the debugging is done upon execution of the programs on RENNS.

The RENNS environment is outlined in figure 5.10.

5.5.2 Interfacing RENNS

The RENNS modules are situated in two VME racks. Since one module occupies two slots, only 10 RENNS modules can be placed in each rack. There is also room for an additional card occupying the last slot available (a total of 21 slots in each rack).

The intention was to interface RENNS through a host computer placed in one of the VME racks together with the RENNS modules (see figure 5.10). However, since we have faced problems making this solution work, we have been forced to use the RS-232 serial line interface as a temporary arrangement for accessing the system.
5.5. The RENNS environment

![Diagram of the RENNS environment]

**Figure 5.10: The RENNS environment, intermodule communication not indicated**

### 5.5.2.1 The Sparc host

The Sparc host is a Force CPU-3CE [16]. The host uses Solaris 1.1F Unix operating system, and has access to the same file system as is used when developing the programs for RENNS. The programs developed on the UNIX workstations can as such be fetched by the Sparc host, and downloaded on the RENNS modules residing in the same VME rack as the host computer. The communication between the Sparc host and the RENNS modules is done over the VME Bus.

There has been several problems with this solution. The initial problem was an error in the VME interface to the RENNS modules. This problem seems now to be overcome, and a VME driver is currently under construction by Jarle Greipsland.
5.5.2.2 The RS-232 interface

The RENNS modules were equipped with an RS-232 serial line interface. This interface has been our only way to communicate with the RENNS modules. The RS-232 interface to the RENNS modules are connected to one or several PCs, also connected to the LAN. A cable arrangement was made such that programs and data can be downloaded to all modules simultaneously. The interface to the RENNS system is however a severe bottleneck, since the communication speed achievable is a maximum of 38400 bit per second. One result of this poor interface is that training of the developed parallel algorithms has often been done on synthetic data generated on the RENNS modules.
Chapter 6

Parallel SOM implementations

This chapter describes the implementations of the SOM algorithm which has been constructed for the RENNS Computer System. The implementations are presented in the order in which they were constructed. The implementations are described in high-level pseudo code supplied by a verbal description.

For each of the implementations a theoretical model of the average time needed to execute one training iteration of the SOM algorithm is presented. The reason for constructing models is to explicitly describe the implementations’ dependency on the parameters of the SOM-network and to estimate the effect on performance of a system with a larger number of processors.

We have made some assumptions in the models which make them not directly comparable to the measured performances. First, we have assumed a perfect loadbalance, which is a fair assumption when the number of neurons per processor is large but is not correct when only a few neurons are computed on each module. Second, estimating the time needed for computation involves counting only the operations involved in the computation of the neurons and the modification of the weights. The rest of the overhead is not taken into account. Third, we have assumed that there are no sequential part in the algorithms. The reason for making these assumptions are that they simplify the models. As a result of this, the constants in the models are not accurate enough, and the models cannot be used directly to determine execution times of the different implementations. They can however still give a qualitative description of the implementations.
6.1 The NP-Ring implementation

The first approach to implementing a parallel SOM algorithm used neuron level parallelism exclusively. The implementation is denoted the NP-Ring implementation (Neuron Parallel implementation on one Ring), NPR for short, due to the interconnection structure of the modules.

The NP-Ring implementation is a straightforward approach. The RENNS modules are all connected in one ring (see figure 6.1) using a token based protocol (the Vector configuration).

One of the modules has a special role as a master module. This master module has the following responsibilities in each iteration:

- Broadcasting of the training vector.
- Selection of the winning neuron.
- Broadcasting of the winning neuron's identification.

The remaining modules, named the slaves, are responsible for computing their share of neurons and for updating them after a winner is selected.

The computation of the learning rate and the neighborhood size is done locally on each module. The transformation to and from a global neuron index is also done locally.

The implementation ensures a high degree of loadbalancing when the number of neurons per processor is high. For a 1-dimensional SOM network, the mapping of the neurons to the modules are done according to figure 4.2, and for a 2 dimensional SOM network, the neurons are mapped according to figure 4.4.

The parts of the master and the slaves in the implementation are outlined in tables 6.1 and 6.2 respectively. The source code can be found in appendix B.2.

Communication is over one token based ring. All slave modules share the same address, whereas the master module has a unique address. When the master sends a message it is thus received by all the slave modules in the ring. The slave modules report information about their locally winning neuron by using the address of the master. The identification of locally winning neurons is translated to a global index before it is sent to the master. Hence, the master does not need any information regarding which module the neuron resides on.
1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values

2. Select training vector

3. Broadcast training vector

4. Compute our own neurons, and select the local winning neuron.

5. Compute the local winning neurons global index.

6. For each slave in the system:
   (a) Receive the index and value of the slaves’ winner
   (b) Compare the value of the slaves’ winner to the current winning neurons’. If this new winner is better, then replace the current global winner.

7. Broadcast the global winner’s identification.

8. Compute the new learning rate and neighborhood size.

9. Compute which of the local neurons are within the neighborhood of the global winner.

10. Update the local neurons within the neighborhood.

11. If more training steps, goto step 2.

Table 6.1: Outline of the NP-Ring implementation, masters part

When the master is communicating the training vector or the global winning neuron’s identification, the slave modules do not attempt to send. Hence, there is no contention on the ring in this part of the communication. On the other hand, when the slave modules have finished the computation of their neurons, they will finish at approximately the same time, and will attempt to use the ring simultaneously.

As can be seen from the implementation outlined, and more directly from equation 6.5, the amount of communication is independent of the number of neurons in the network. It has a dependency on the dimensionality of the training vector because the training vector is broadcast to the slave modules. This dependency can be eliminated by issuing a copy of the training vectors to each slave module. Assuming they have enough memory to hold the entire training set, the master would only have to broadcast the index of the training vector.
1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values

2. Receive new training vector

3. Compute our own neurons, and select the local winning neuron.

4. Compute the local winning neurons global index.

5. Send the local winner’s global index and value to the master.

6. Receive the global winner’s index.

7. Compute the new learning rate and neighborhood size.

8. Compute which of the local neurons are within the neighborhood of the global winner.

9. Update the local neurons within the neighborhood.

10. If more training steps, goto step 3.

Table 6.2: Outline of the NP-Ring implementation, slaves’ part

The computational part on the other hand is very dependent on the number of neurons and the dimensionality of the training vector. By increasing the size of these parameters, more computations must be done in each iteration.

The computation of the neighborhood size and the learning factor $\alpha$ are computed locally on each module. This is redundant computation, but since it is too simple to be computed distributed on a computer system such as RENNNS, it is a faster solution than computing it locally on one module and then broadcasting it to the other modules.

We have used a linearly decreasing function for the learning factor $\alpha = \alpha(t)$:

$$\alpha(t) = \alpha(0) \left(1 - \frac{t}{t_T}\right)$$  \hspace{1cm} (6.1)

where $\alpha(0)$ is the starting value of $\alpha$, $t$ is the iteration number and $t_T$ is the total number of iterations in the training session.

A linearly decreasing function for the neighborhood size is used. We have chosen to use a neighborhood which initially can cover the whole network. We have not used wrapping of
the neighborhood. In the case of a 2-dimensional ordering of neurons in a \( n_x \times n_y \) grid, and \( n_{\text{max}} = \max(n_x, n_y) \) then the width and height of the neighborhood at iteration \( t \) becomes:

\[
N_{\text{height}}(t) = N_{\text{width}}(t) = n_{\text{max}}(1 - \frac{t}{t_T}) \tag{6.2}
\]

A width and height of 0 means that only the winning neuron is updated. In a 1-dimensional ordering, \( n_y = 1 \).

The time needed to do one iteration of the NP-Ring implementation, \( T_{\text{NP-Ring}} \), is the sum of the time needed for computation \( (T_{\text{NP-Rcomp}}) \) and for communication \( (T_{\text{NP-Rcomm}}) \).

The TMS320C30 can do one addition, subtraction, or multiplication in one clock period. The time needed to do one of these floating point operations is \( t_{\text{float}} \).

The NP-Ring implementation uses the standard equations for the basis function and the learning rule (equations 4.3 and 4.5).

Let \( n \) be the number of neurons, \( d \) the input vector dimensionality, and \( p \) the number of processors. In order to compute one neuron, we would need to do \( d \) subtractions, \( d \) multiplications, and \( d - 1 \) additions. The square root part of the computation can be omitted since the magnitude of the output is of no concern, and the square root does not affect which neuron becomes the winner. This gives a total of \( 3d - 1 \) floating point operations for each neuron, giving a total of \( n(3d - 1) \) floating point operations.

In the learning phase, \( d \) subtractions, \( d \) multiplications, and \( d \) additions are done for each neuron \textit{within the neighborhood of the winner}, giving a total of \( 3(kn)d \) floating point operations per iteration, where \( k \) is the average fraction of neurons within the neighbor-
hood.

This gives a total of \( n(3d + 3d\kappa - 1) \) floating point operations in one training iteration. Each of these floating point operations takes \( t_{\text{float}} = 62.5\,\text{ns} \) in the current system which has a 32MHz clock. Hence, the total time needed for computation of the neuron outputs and weight adjustments in one training iteration becomes:

\[
T_{\text{NPRecompfloat}}(n, d, \kappa) = n(3d + 3d\kappa - 1)62.5\,\text{ns}
\] (6.3)

The \( T_{\text{NPRecompfloat}} \) is too optimistic since we only take into account the time needed to do the floating point operations. In order to make it somewhat closer to the actual time required, we must look at the assembly code generated by the compiler.

In the part of the code where the neuron outputs are computed, the inner loop of computing each neuron requires 3 floating point operations per dimension. However, there are an additional 14 instructions surrounding the inner loop. These instructions must be executed for each neuron. There are also 34 instructions encapsulating the computations, giving a total of \( n(3d + 14) + 34 \) instructions each time we want to compute the output of \( n \) neurons with dimensionality \( d \).

In the weight update part of the code, the inner loop is now handled by 5 instructions for each dimension, and 10 additional instructions per neuron. The update code is, in addition, encapsulated by 27 instructions which must be executed each time weight update is to be done, independent of \( n \) and \( d \). This gives a total of \( n\kappa(5d + 10) + 27 \) instructions which are executed when \( n\kappa \) neurons of dimensionality \( d \) are to be updated. Using a 2-dimensional SOM-network, the weight update code would even have to be executed several times.

If we assume that each instruction is executed in one clock period then the total time needed to compute the neurons and change the weights in one training iteration becomes:

\[
T_{\text{NPRecomp}}(n, d, \kappa) = (n(3d + 14) + 34)62.5\,\text{ns} + (n\kappa(5d + 10) + 27)62.5\,\text{ns}
= (3nd + 14n + 34 + 5kn\kappa + 10kn + 27)62.5\,\text{ns}
= (nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5\,\text{ns}
\] (6.4)

The time needed for communication in one iteration is the sum of the time needed to broadcast a training vector, the time needed to send the local winners from all the slaves to the master, and the time needed to broadcast the winner.

The time needed to broadcast the training vector, including acquiring the token, is
6.1. The NP-Ring implementation

given by \( t_{vec}(p-2, p, d) \) (equation 5.2). The \( p-2 \) parameter comes from the fact that this is the number of intermediate modules between the master and the last module.

In the phase where the \( p-1 \) slave modules send their 2 word messages regarding local winners, the time needed is given by \( t_{alone}(p, 2) \) (equation 5.4).

In the final phase, one word is broadcast, and the time needed is given by \( t_{vec}(p-2, p, 1) \) (equation 5.2).

The total communication time in one iteration hence becomes:

\[
T_{NP\text{R}comm}(p, d) = t_{vec}(p-2, p, d) + t_{alone}(p, 2) + t_{vec}(p-2, p, 1)
\]

\[
= (p-2)310\text{ns} + p155\text{ns} + d800\text{ns} + 2410\text{ns}
\]

\[
+ p(310\text{ns} + 2*800\text{ns} + 310\text{ns}) - 2*800\text{ns} + 1590\text{ns}
\]

\[
+ (p-2)310\text{ns} + p155\text{ns} + 800\text{ns} + 2410\text{ns}
\]

\[
= p465\text{ns} + d800\text{ns} + 1790\text{ns}
\]

\[
+ p2220\text{ns} - 10\text{ns}
\]

\[
+ p465\text{ns} + 2590\text{ns}
\]

\[
= p3150\text{ns} + d800\text{ns} + 4370\text{ns}
\]  \hspace{1cm} (6.5)

If we assume a perfect loadbalance, then the time needed for executing one iteration of the NP-Ring implementation becomes:

\[
T_{NP-Ring}(p, n, d, \kappa) = \frac{T_{NP\text{R}comm}(n, d, \kappa)}{p} + T_{NP\text{R}comm}(p, d)
\]

\[
= \frac{nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5\text{ns}}{p}
\]

\[
+ p3150\text{ns} + d800\text{ns} + 4370\text{ns}
\]  \hspace{1cm} (6.6)

Figure 6.2 shows the estimated speedup of this implementation as a function of the number of processors for varying numbers of neurons. The input vector dimensionality is fixed at 26 (randomly chosen), and it is assumed that in average, one fourth of the neurons are updated (\( \kappa = 0.25 \)).

For large networks, the estimated speedup nearly follows the linear speedup. For smaller networks however, the speedup is not that good. In the network with 100 neurons, the speedup is less than \( \frac{p}{2} \) when \( p = 16 \).
We also include the estimated speedups when the number of processors is increased beyond 16. This is shown in figure 6.3. This figure shows that for a network consisting of 1000 neurons and a 26 dimensional input vector, it is limited how many processors the implementation can take advantage of.

### 6.2 The NP-Tree implementation

One of the disadvantages of the NP-Ring implementation is the contention for communication resources when the global winner is selected. All the modules wish to send information about their local winner at approximately the same time over a common ring.

In order to reduce the time needed to select the global winner, an implementation using a different interprocessor topology was constructed. In this implementation, called the NP-Tree implementation, we still have a global ring used by the master module to broadcast training vectors and the global winner’s identification. However, in addition, a number of local rings connect the modules in a tree structure (see figure 6.4). The root node of the tree is the master module, with the same responsibilities as in the NP-Ring
implementation. The tree structure is only used in the global winner selection procedure.

The leaf nodes of the tree does only select the local winner, and pass this local winner's value and identification up the tree. Each intermediate node i.e. each node which is not a leaf node or the root node, receives the information about local winners from nodes at a lower level of the tree structure, compares these local winners (1 or 2, depending on the balancing of the tree) with its own local winner, and passes the new winner further up the tree. The root node (master) is therefore only required to compare it's local winner with the winners passed from its children in order to select the global winner.

The parts of the master and the slaves in the implementation are outlined in table 6.3 and 6.4 respectively. The source code can be found in appendix B.3.

The communication protocol used, both in the global ring and in the local rings, is the same token based protocol as was used in the NP-Ring implementation. The reason for using a token based protocol in the subrings is that each module has only two incoming FIFO-banks, and each module with two children has three sources of information (it’s two children and the global ring).
Figure 6.4: Communication based on one global ring and a tree structure of sub rings
6.2. The NP-Tree implementation

1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values

2. Select training vector

3. Broadcast training vector using the global ring

4. Compute the local neurons, and select a local winning neuron

5. Compute the local winner's global index

6. Await the winners from the eventual children, and select a global winner

7. Broadcast the global winner's index on the global ring

8. Compute the new learning rate and neighborhood size

9. Compute which of the local neurons are within the neighborhood of the global winner

10. Update the local neurons within the neighborhood

11. If more iterations, goto step 2.

Table 6.3: Outline of the NP-Tree implementation, master's part

The time needed to perform one iteration using the NP-Tree implementation is given by the sum of the time needed for computation and the time needed for communication
\( T_{NP-Tree} = T_{NP\text{comp}} + T_{NP\text{comm}} \).

The time needed for computing the values of the neurons and the weight changes is the same as in the NP-Ring implementation:

\[
T_{NP\text{comp}}(n, d, \kappa) = (nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5ns
\]

The time needed for communication is divided into three parts: broadcasting of the training vector, the winner selection, and the broadcasting of the winner's index. The first part and the third part are the same as in the NP-Ring implementation.

The second part is dependent on the number of levels in the tree. For each level in addition to the root-level, the children need to send a 2 word message to the parent node.
1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values
2. Receive new training vector from the global ring
3. Compute the local neurons, and select the local winner
4. Compute the local winner’s global index
5. Await the winners from eventual children. Compare these winners to the local winners, and send the winner of this comparison to the parent in the tree.
6. Await the global winner’s index from the global ring
7. Compute the new learning rate and neighborhood size
8. Compute which local neurons are within the neighborhood of the global winner
9. Update the local neurons within the neighborhood
10. If more iterations, goto step 2.

Table 6.4: Outline of the NP-Tree implementation, slaves’ part

If we assume that we have a balanced tree, the number of levels which must be passed is \( \log_2(p+1) - 1 \). The time needed to pass one level \( (t_{level}) \) is the sum of the time needed by the children to send their messages.

\[
t_{\text{level}} = t_{\text{vec}}(0,4,2) + t_{\text{vec}}(1,4,2) \\
= 4(155\text{ns}) + 2(800\text{ns}) + 2410\text{ns} + 1(310\text{ns}) + 4(155\text{ns}) + 2(800\text{ns}) + 2410\text{ns} \\
= 9570\text{ns}
\] (6.8)

For one of the children, there are no intermediate modules between itself and the parent. For the other child, there is one intermediate node which is the first child.

The total time needed for communication hence becomes:

\[
T_{NPT_{\text{comm}}} = t_{\text{vec}}(p-2,p,d) + (\log_2(p+1) - 1)t_{\text{level}} + t_{\text{vec}}(p-2,p,1)
\]
6.3. The NP-Wave implementation

\[
= p465ns + d800ns + 1790ns \\
+ (\log_2(p + 1) - 1)9570ns \\
+ p465ns + 2590ns \\
= p930ns + \log_2(p + 1)9570ns + d800ns - 5190ns
\] (6.9)

The time needed in order to perform one iteration in the NP-Tree implementation on \( p \) modules hence becomes:

\[
T_{NP-Tree}(p, n, d, \kappa) = \frac{T_{NPT_{comp}}(n, d, \kappa)}{p} + T_{NPT_{comm}}(p, d) \\
= \frac{(nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5ns}{p} \\
+ p930ns + \log_2(p + 1)9570ns + d800ns - 5190ns
\] (6.10)

The estimated performance of the NP-Tree implementation is shown in figure 6.5. Compared to the estimated performance of the NP-Ring implementation, an increase in performance can be observed for the smallest network. For the larger networks, the differences are very small.

The estimated performance for an increasing number of processors is shown in figure 6.6. Differences can be observed even for larger networks, due to the advantages of using the tree structure for selection of the global winner.

As the number of processors is increased above 16 modules, it becomes advantageous to use the tree structure also for distributing the training vector and the global winner’s index instead of using the global ring.

6.3 The NP-Wave implementation

The NP-Wave implementation was constructed due to the poor performance observed in the NP-Ring and the NP-Tree implementations on small networks. The NP-Wave implementation was the first implementation which focussed on relatively small networks.

The problem with small networks in the NP-Ring and the NP-Tree implementations is the high percentage of the time used for communication. The NP-Wave implementation is based on an implementation that was made by Ceccarelli et al [7], which denotes it a
systolic algorithm. Since the RENNS modules does not operate under the control of a
global clock, we have chosen to call this implementation for the TP-Wave implementation
according to Kung's description of wavefront algorithms [58].

In the NP-Wave implementation, the modules are connected in one ring. In contrast to
the NP-Ring and NP-Tree implementations, communication is only between neighboring
modules. The communication is unidirectional, see figure 6.7.

By using the Vector configuration in a point to point operation mode, all modules can
communicate with their neighbor simultaneously.

A training vector needs $p$ communication steps to pass through all the modules, result-
ing in a delayed update scheme. Training of a vector is done by first passing the vector
through all the modules in order to determine the global winner. The global winner is
then passed through all the modules in order to update the weights. In the time between
a local winner is selected on a module, and the updating of the weights is done, $p - 1$ other
training vectors have passed through the module.

The operation of the NP-Wave implementation can be divided into steps. In each step,
a module receives a tuple $<i_{d_{i-p}}, v_i, b_i, i_d>$, where $i_{d_{i-p}}$ is the identification of an earlier
6.3. The NP-Wave implementation

![Graph showing speedup vs. number of processors]

**Figure 6.6: Estimated performance of the NP-Tree implementation for a larger number of processors**

A winning neuron (the winning neuron according to the training vector which was at the actual module $p$ steps ago), $v_i$ is the new training vector, and $b_i$ and $id_i$ is the output and identification of the winning neuron from the preceding modules respectively. $p$ is the number of modules in the ring.

An outline of the NP-Wave implementation is given in table 6.5. The source code can be found in appendix B.4.

The sequence of the tuple is selected to minimize communication overhead. First, the identification of the earlier winning neuron is received and forwarded to the next module. The neurons within the neighborhood of this winner are then updated. The training vector can then simultaneously be received by the Communication Subsystem to the Processing Module updating the neurons, provided that the previous module starts sending it. After this, the module reads the new training vector from the CS and immediately starts retransmitting it to the next module. The CS can then transmit the training vector to the next module while the PS is busy computing the local winner.

The first module, the master, holds the complete set of training vectors. If sufficient memory resources exist, a complete copy of the training set could be placed on each module,
1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values

2. Repeat p times (fill pipe)
   (a) If not master module, then receive new training vector \( v_i \), else select new training vector \( v_i \), and set \( b_i = \infty \).
   (b) If not last module, then send training vector \( v_i \) to next module.
   (c) Compute local neurons and select local winner.
   (d) Receive \( b_i \) and \( v_i \), compare with local winner, and replace with local winner if this is more similar to the training vector.
   (e) If not last module, then send \( b_i \) and \( id_i \) to next module, else send \( id_i \) to the next module.

3. Do while more than \( p \) iterations left (Normal operation)
   (a) Receive \( id_{i-p} \) from the previous module.
   (b) If not the last module then send \( id_{i-p} \) to the next module.
   (c) Update the weights of the neurons within the neighborhood of \( id_{i-p} \).
   (d) If not master module, then receive new training vector \( v_i \), else select new training vector \( v_i \), and set \( b_i = \infty \).
   (e) If not last module, then send training vector \( v_i \) to next module.
   (f) Compute local neurons and select local winner.
   (g) Receive \( b_i \) and \( v_i \), compare with local winner, and replace with local winner if this is more similar to the training vector.
   (h) If not last module, then send \( b_i \) and \( id_i \) to next module, else send \( id_i \) to the next module.

4. Repeat p times (flush pipe)
   (a) Receive \( id_{i-p} \) from the previous module.
   (b) If not the last module then send \( id_{i-p} \) to the next module.
   (c) Update the weights of the neurons within the neighborhood of \( id_{i-p} \).

Table 6.5: Outline of the NP-Wave implementation
6.3. The NP-Wave implementation

![Diagram of NP-Wave implementation](image)

**Figure 6.7:** Example on the communication in the NP-Wave implementation with 4 modules operating in the normal mode.

requiring only the sending of the index of the training vector. However, this is not the case in the present version. Instead the module’s local buffer holds $p$ training vectors, allowing for updating of the neuron weights without the need for retransmission of the training vector. After updating, the training vector is no longer of any use and is overwritten by the next incoming training vector.

The last module in the ring has a special role in the NP-Wave implementation. Since all modules have updated their weights according to the winner $i_d_{i-p}$, the winner is not required any more, as is the case for the training vector $v_i$ when it has passed through all modules. The value of the winning neuron, $b_i$ can also be discarded. Hence, the only information which is passed back to the master from the last node is the identification of the winner according to the current training vector, $i_d_i$.

Since the NP-Wave implementation is a delayed algorithm, attention should be payed towards the convergence of the algorithm. For the RENNS system, having only a total of 16 modules, updating is done relatively soon after the training vector is applied.

The time needed to compute the neurons, and to do the updating is the same in the NP-Wave implementation as it is in the two previous implementations:
\[ T_{NPW\text{comp}} = (nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5\text{ns} \]  

(6.11)

The time needed for communication in one training iteration is the time needed to send the \( d \) dimensional training vector, and the three word message regarding current and previous winners. The time needed to communication in one iteration is hence given by:

\[ T_{NPW\text{comm}}(d) = t_{direct}(0,1) + t_{direct}(0,d) + t_{direct}(0,2) \]
\[ = 800\text{ns} + 1510\text{ns} + d800\text{ns} + 1510\text{ns} + 2(800\text{ns}) + 1510\text{ns} \]
\[ = d800\text{ns} + 6930\text{ns} \]  

(6.12)

The time needed for each iteration hence becomes:

\[ T_{NP-\text{Wave}}(p, n, d, \kappa) = \frac{T_{NPW\text{comp}}(n, d, \kappa)}{p} + T_{NPW\text{comm}}(d) \]
\[ = \frac{(nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5\text{ns}}{p} \]
\[ + d800\text{ns} + 6930\text{ns} \]  

(6.13)

In figure 6.8, the estimated performance of the NP-Wave implementation is given as a function of the number of processors. In the figure, the dimensionality of the input vector is 26, and the fraction of the updated neurons, \( \kappa \), is 0.25. As can be seen from the figure, the estimated speedup is improved in comparison to those given for the NP-Ring and NP-Tree implementations.

We have not calculated the time needed to fill the pipeline at the start or flush the pipeline at the end of the training. The model only covers the normal operation part of the implementation i.e. when the operation between filling and flushing of the pipeline.

In figure 6.9 the estimated performance of the NP-Wave implementation is given for a larger number of processors. The estimated speedup is, as expected, an improvement over the previous implementations, since the communication in the TP-Wave implementation is independent of the number of processors.
6.4 The TP-Ring implementation

The next implementation which was constructed is a training example parallel algorithm. This implementation is named the TP-Ring implementation (Training example parallel, communicating on one Ring).

The programming of a training example parallel algorithm is essentially simpler than programming a neuron parallel algorithm, since the training example parallel algorithm is very similar to the sequential SOM algorithm. There is, however, one major difference – the weight updates are not immediately added to the weights but are accumulated in a weight change matrix. The update of the weights occurs after the weight changes from all the modules have been accumulated.

The communication structure of the TP-Ring implementation is the same as in the NP-Ring implementation. All the modules are connected in one ring using a token based protocol (see figure 6.1). In contrast to all the previously presented implementations, this implementation does not have any master. All the modules have the same address, such that all data sent on the ring is received by all other modules.
Figure 6.9: Estimated performance of the NP-Wave implementation for a larger number of processors

To repeat the main idea of training example parallel algorithms: All the modules have a complete copy of the SOM-network, whereas the training vectors are divided among the modules. After each module has trained a specified number of training vectors, the local weight change matrixes are exchanged among the modules and the weight update becomes:

\[ m_i(t' + 1) = m_i(t') + \sum_{j=1}^{p} \Delta m_i^j(t'), \forall i \]  

(6.14)

where \( t' \) denotes the batch number and \( \Delta m_i^j(t') \) is the local weight change vector of module \( j \) for neuron \( i \).

The implementation is outlined in table 6.6. The source code can be found in appendix B.5.

Special care has to be taken in the communication of the weight change matrix when the product of the training vector dimensionality \( d \) and the number of neurons \( n \) exceed the depth of the FIFO-banks on the Communication Subsystem. If the product of the
1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values
   (c) Divide training set among the modules

2. Each module selects a local training vector, and computes the winner.

3. The weight change according to this winner is computed. The weight change is added to a weight change matrix.

4. If more iterations within this batch, go to 2.

5. All modules broadcast their weight change matrix.

6. All weight changes received from other modules is added to the local weight change matrix.

7. Add the accumulated weight change matrix to the weight matrix. Reset the weight change matrix.

8. If more batches, go to 2.

Table 6.6: Outline of the TP-Ring implementation

training vector dimensionality and the number of neurons is less than the depth of the FIFO-banks, then the approach outlined in table 6.7 will be sufficient.

1. Send the contents in the local weight change matrix to the outgoing FIFO.

2. Receive the $p - 1$ local weight change matrixes from the other modules from the incoming FIFO and combine them with the local weight change matrix.

Table 6.7: Weight exchange when the weight change matrix fits in FIFO

If the size of the weight change matrix is small enough to fit in the FIFO-bank, all modules will send the local weight change matrixes to the FIFO-bank, and then start receiving the weight changes from the other modules. If, however, the size of the weight change matrix is too large, the ring will be blocked by the succeeding module of the module which was sent first. When the incoming FIFO-bank of this succeeding module is filled up, then the ring will be blocked until it starts reading from the FIFO. This will never happen, as it is awaiting available space in it’s own outgoing FIFO.
This problem can be solved by partitioning the weight change matrix into smaller pieces, and using the approach outlined in table 6.7 several times. When using the Vector configuration in this way, the maximum number of elements in each partition is 1022 since one word is used for the address of the receiving modules and one word is used for the token (the FIFO depth is 1024 words). The partitioned weight exchange scheme is outlined in table 6.8.

1. Send one part of the weight change matrix to the outgoing FIFO.
2. Receive the \( p - 1 \) parts from the other modules.
3. If there are more parts, goto 1.

**Table 6.8: Weight exchange when the weight change matrix does not fit in FIFO**

The time needed to execute one iteration in the TP-Ring implementation depends on how often the weight exchange is done. We therefore introduce a new parameter, \( \epsilon \), which describes how many iterations are carried out on each processor between the weight exchange.

Even though each training vector is trained locally on one processor, we approximate the time needed for computation in each iteration with the expressions used in the previously presented implementations:

\[
T_{TPR\text{comp}} = (nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5ns
\]  
(6.15)

The time needed to exchange the weights between all the modules is approximated by:

\[
t_{TPR\text{exchange}}(p, n, d) = t_{all}(p, nd)  
= p(p310ns + nd800ns + 2210ns)
\]  
(6.16)

Equation 6.16 does not take into account the extra time needed if the weight exchange is partitioned.

If we have \( p \) modules, and exchange weights after \( \epsilon \) training vectors on each module, then the weight exchange is done after \( p\epsilon \) training vectors have been applied. The time needed for communication per iteration hence becomes:
\[ T_{TPR_{\text{comm}}}(p, n, d, \epsilon) = \frac{t_{TPR_{\text{exchange}}}(p, n, d)}{\epsilon p} = \frac{p(p310ns + nd800ns + 2210ns)}{\epsilon p} = \frac{p310ns + nd800ns + 2210ns}{\epsilon} \] (6.17)

The average time needed to train one training vector hence becomes:

\[ T_{TP-Ring}(n, d, p, \epsilon, \kappa) = \frac{T_{TP_{\text{comp}}}(n, d, \kappa)}{p} + T_{TPR_{\text{comm}}}(p, n, d, \epsilon) = \frac{(nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5ns}{p} + \frac{p310ns + nd800ns + 2210ns}{\epsilon} \] (6.18)

In figure 6.10, the estimated performance of the TP-Ring implementation is given as a function of the number of processors. In the figure, the number of neurons is 100, the dimensionality of the input vector is 26, and the fraction of updated neurons, \( \kappa \), is 0.25. The weights are exchanged after each module has trained \( \epsilon = 10 \), \( \epsilon = 20 \), and \( \epsilon = 30 \) vectors. Increasing the number of neurons above 100 will not give any substantial change in speedup since the time needed to execute the algorithm, both a singular or several processors, is dominated by a term proportional to \( n \).

The figure clearly shows how important it is for the performance to have a high value of \( \epsilon \). The performance of the TP-Ring implementation is however relative poor even for networks of this size.

The TP-Ring implementation cannot compete with the previous implementations on large networks, and the figure describing performance on systems with a larger number of processors is not included. The performance on smaller networks is included in the next section.

### 6.5 The NP/TP implementation

The final SOM implementation uses both neuron parallelism and training example parallelism. The \( p \) modules are separated into \( c \) clusters, each having \( p_c = \frac{p}{c} \) RENNS modules,
Figure 6.10: Estimated performance of the TP-Ring implementation

i.e. each cluster has the same number of RENNS modules.

Within each cluster, a neuron parallel algorithm is used, whereas between the clusters, a training example parallel algorithm is used. This means that the training vectors are divided among the clusters. After each cluster has trained a certain number of iterations, the weight change matrixes in each cluster are exchanged.

Within a cluster, a neuron parallel algorithm is used. The algorithm used within a cluster is close, but not identical, to the NP-Ring implementation. The difference being that the weights are not updated at once, but are held in weight change matrixes in the same way as in the TP-Ring implementation.

There are several reasons to base the neuron parallel part on the NP-Ring implementation. First, there are not sufficient communication resources to base it on the NP-Tree implementation. The alternative would be to use the NP-Wave implementation which is estimated to be faster than the NP-Ring implementations when it operates in its normal mode. However, since we would have to fill and flush the pipeline each time a weight change is done, the NP-Ring implementation was selected to serve as a basis for the neuron parallel part of the NP/TP implementation.
6.5. The NP/TP implementation

The modules are connected in a 2D-torus topology, see figure 6.11. The communication protocol used in all the rings is the token based protocol of the Vector configuration. The vertically aligned modules complete one cluster. The horizontally aligned modules are responsible for the same set of neurons within their cluster. Between the weight exchanges, each cluster uses the vertical communication channel for broadcasting training data and reporting local winners etc. The weight changes need only be distributed on the horizontal communication channels.

The NP/TP implementation is outlined in table 6.9. The complete source code for the program can be found in appendix B.6.

1. Initialize:
   (a) Set up communication subsystem
   (b) Set weights to random values
   (c) Divide training set among the masters of the clusters

2. The master module in each cluster selects and broadcasts a training vector to the other modules within the cluster.

3. All modules compute their locally winning neuron

4. All slaves sends the identification and value of their locally winning neuron to the master in their cluster.

5. All masters select a global winner, and broadcast the global winners identification to the other modules within the cluster.

6. All modules compute the weight changes, and adds this to the local weight change matrix.

7. If more iterations within this batch, goto 2.

8. All modules broadcast their local weight change matrix to the modules in the other clusters which are responsible for the same set of neurons. All weight changes received from the other modules are added to the local weight change matrix.

9. All modules add the accumulated weight change matrix to the weight matrix. Reset the weight change matrix.

10. if more batches, goto 2.

Table 6.9: Outline of the NP/TP implementation
For the NP/TP implementation, it is necessary to determine how often the weight change matrixes need be exchanged between clusters. This factor can be found experimentally. Once this factor has been obtained, the values of $p_c$ and $c$ are selected in such a way that the performance is maximized.

In order to estimate the performance of the NP/TP implementation, we approximate the time needed for computation with the same expression as was used as estimate for the previously described implementations.

\[ T_{NP/TP_{comp}} = (nd(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5ns \]  \hspace{1cm} (6.19)

The time needed to exchange the weights between clusters is approximated by (as before, we do not take into account the fact that the weight exchange may be partitioned):

\[ t_{NP/TP_{exchange}}(c, p_c, n, d) = t_{all}(c, \frac{n}{p_c}d) \]
\[ = c(c310ns + \frac{nd}{p_c}800ns + 2210ns) \]  \hspace{1cm} (6.20)
6.5. The NP/TP implementation

The number of modules on each ring exchanging weight change matrixes is \( c \) (the number of clusters). The number of neurons on each module is \( \frac{n}{p_c} \), where \( p_c \) is the number of processors in each cluster.

In addition, there is communication within each cluster for each training iteration, which is the same as the communication in the TP-Ring implementation (equation 6.5):

\[
\begin{align*}
t_{\text{cluster}}(p_c, d) &= t_{\text{vec}}(p_c - 2, p_c, d) + t_{\text{alone}}(p_c, 2) + t_{\text{vec}}(p_c - 2, p_c, 1) \\
&= p_c3150\text{ns} + d800\text{ns} + 4370\text{ns} \\
&= (c(c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns})) + p_c3150\text{ns} + d800\text{ns} + 4370\text{ns} \\
&= c(c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns}) + p_c3150\text{ns} + d800\text{ns} + 4370\text{ns} \\
&= \frac{c(c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns})}{c} + \frac{p_c3150\text{ns} + d800\text{ns} + 4370\text{ns}}{c} \\
&= \frac{c(c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns})}{c} + \frac{p_c3150\text{ns} + d800\text{ns} + 4370\text{ns}}{c}
\end{align*}
\]

(6.21)

If the weights are exchanged after each cluster has trained \( \epsilon \) training vectors, then the average time needed for communication for each iteration becomes:

\[
T_{NP/TPcomm} = \frac{t_{NP/TPexchange}}{cc} + \frac{t_{\text{cluster}}}{c}
\]

\[
= \frac{c(c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns})}{c} + \frac{p_c3150\text{ns} + d800\text{ns} + 4370\text{ns}}{c}
\]

(6.22)

The \( t_{\text{cluster}} \) must be divided by \( c \), since one training vector is trained on all the \( c \) clusters simultaneously. The total time needed to do one iteration in the implementation becomes:

\[
T_{NP/TP} = \frac{T_{NP/TPcomp}}{p} + T_{NP/TPcomm}
\]

\[
= (n d(3 + 5\kappa) + n(14 + 10\kappa) + 61)62.5\text{ns}
\]

\[
+ \frac{c310\text{ns} + \frac{n d}{p_c}800\text{ns} + 2210\text{ns}}{\epsilon} + \frac{p_c3150\text{ns} + d800\text{ns} + 4370\text{ns}}{c}
\]

(6.23)

A problem with describing the performance of the NP/TP implementation is that it depends on a large number of variables. The dependency on the variables can be seen from equation 6.23.

As was the case with the TP-Ring implementation, the NP/TP implementation was designed for simulation of smaller networks. Therefore, we choose a relatively small network for visualizing the performance of the implementation. In figure 6.12, the estimated
performance of the NP/TP implementation is given as a function of the number of processors. In the figure, the number of neurons is 100, the dimensionality of the input vector is 26, and the fraction of updated neurons, \( \kappa \), is 0.25. The number of neurons in each cluster \( p_c \) is 4. Although the value of \( p_c \) restricts the possible values of \( p \) to 4, 8, 12, and 16, we have chosen to plot the continuous function, as in all the previous cases (the performance functions in the previous cases do strictly only take values for an integer number of processors). The performance has been estimated for \( \epsilon = 10, \epsilon = 20 \) and \( \epsilon = 30 \).

![Figure 6.12: Estimated performance of the NP/TP implementation](image)

The TP-Ring and the NP/TP implementations were constructed in order to achieve increased performance on small networks. This was motivated by one specific application – visualization of multispectral medical images. We include estimated performance figures of the implementations for this application.

In the visualization application, the training set consists of a 4 channel MR-image, giving 4 dimensional input vectors. A typical number of neurons is 64. Figure 6.13 shows the estimated performances of the implementations for this application. The parameter setting is: \( n = 64, d = 4, \kappa = 0.25, \epsilon = 20, p_c = 4 \).

As can be seen from the figure, the NP-Wave, the TP-Ring, and the NP/TP implementation are the three implementations which give rise to the best estimated performance for networks of this size. However, at problems of this size, the constants in the model
Figure 6.13: Comparison of the estimated performances on the MR-image visualization problem

becomes very important and the estimates do not give a very accurate picture of the actual speedup obtained. We will come back to the measured performance of this problem in the results chapter (chapter 7).
Chapter 7

Results

This chapter contains the performance measurements of the implementations described in chapter 6. Results regarding convergence of the SOM algorithm for the case of accumulated weight update are also included.

In the current RENNS implementation, 16 modules are present. Due to stability problems with some of the modules, simulations have been carried out on 13, 14, 15 and 16 modules.

In all the simulations conducted on RENNS, the on-chip timers on the TMS320C30 have been used for measuring the time used by the implementations. Using such timers influence performance measurements to a limited extent. However, this is assumed to be negligible with respect to the results sought.

7.1 The NP-Ring implementation

The NP-Ring implementation was the first implementation to be constructed. It can be said to be a very simple implementation and was designed without any special application size in mind.

The performance of the NP-Ring implementation was first reported at the Joint Conference on Information Sciences [81], and it is these results which are presented here.
7.1.1 Performance

The simulations were run on a 1-dimensional SOM network consisting of between 100 and 10000 neurons and using a 26 dimensional training vector. The dimensionality of the training vector was randomly selected, as standard values for the parameters of the network do not appear to exist. At the time the measurements were taken, there were 15 operational RENNS modules. We measured the number of weight updates per second (CUPS). A total of 1000 iterations were done on each network size. This is not a sufficient number of iterations for the network to train properly, but gives us measurements over the whole range of neighborhoodsizes. The main reason for using a limited number of iterations is of course the reduced time needed to obtain the required results. The training vectors were generated at the master module. Each component of the training vector was a random number between 0 and 1.

In figure 7.1 the performance of the NP-Ring implementation is given as a function of the number of RENNS modules. The exact values in the figure can be found in table C.1.

![Graph showing performance of NP-Ring implementation](image_url)

Figure 7.1: Performance of the NP-Ring implementation
As described in section 5, the Processing Subsystem is equipped with three kinds of memory: Internal SRAM, external SRAM and DRAM. When memory for the weights is allocated, then an area of the requested size is reserved in the fastest available of these three memory types. Each module needs to allocate $\lceil \frac{n}{d} \rceil$ words of memory for the weight matrix. When this block becomes too large to fit in the internal SRAM, it is put in the external SRAM instead. If it is too large to fit in the external SRAM, then it is placed in the DRAM which is the largest and slowest memory type. The effect of this memory allocation scheme can be observed in the figure as sudden leaps in performance. For instance, the curve representing the performance of 1000 neurons shows a sharp rise when the number of processors is increased from 12 to 13. This is due to the fact that the weight matrix fits in internal SRAM when divided amongst 13 modules, but not when divided amongst 12. In the case where we have 10000 neurons, and only 1 or 2 modules, then the weight matrix must be placed in the DRAM. In the 10000 neuron case, the speedup is superlinear because of this. The superlinear speedup is, however, a special case which would not occur if the modules were equipped with more memory resources.

Figure 7.2: Effect of altering the frequency of the communication subsystem, NP-Ring implementation
As expected (and estimated), larger networks gain better speedup and performance (measured in CUPS) than the smaller networks. The reason for this is that when the network size is increased, more time is used for computing the neurons, whereas the time needed for communication is independent of the number of neurons.

The difference in performance when changing the number of neurons from 5000 to 10000 is not large. This is because even with 5000 neurons, most of the time is used for computing neurons and changing weights. Doubling the number of neurons from 5000 to 10000 mainly results in using twice the time for computation, and updating twice as many weights.

For the small network (100 neurons), the performance flattens fast, and almost nothing is gained by increasing the number of modules beyond ten.

Figure 7.3: Performance of the NP-Tree implementation
7.1.2 The frequency of the Communication Subsystem

An experiment was conducted to measure the effect of reduced frequency on the performance of the communication subsystem. This was done in order to investigate how large an increase in performance could be achieved if the frequency could be raised. We used two different frequencies – 10MHz, which is the fastest which can be used on the present configuration, and 8MHz. As can be seen from figure 7.2 (see also table C.2), the effect of the change in frequency is very small, and redesigning the interiors of the FPGAs on the communication subsystem was not considered the way to go in order to achieve increased performance.

7.2 The NP-Tree implementation

The estimates for the performance of the NP-Tree implementation are almost identical to those of the NP-Ring implementation. This is confirmed by the performance measurements taken, the results of which were presented at the Joint Conference on Information Sciences [81].

7.2.1 Performance

The measured performance of the NP-Tree implementation is shown in figure 7.3 (see also table C.3). We used the same parameters when measuring performance on this implementation as we did for the NP-Ring implementation, making them directly comparable.

Memory usage effects in the NP-Tree simulation are similar to those of the NP-Ring simulation. This was as expected, since the only difference between the implementations is how they select the globally winning neuron.

7.2.2 Comparison

In figure 7.4 the performance of the NP-Ring implementation and the NP-Tree implementation are compared directly in order to see if there are any differences in the performance. The results confirm estimates that for large networks, the two implementations achieve similar performance levels and for small networks, a slight variation in performance is seen.
Figure 7.4: Comparison of the performance of the NP-Ring and the NP-Tree implementations

7.2.3 Communication times

Since the number of modules in the present RENNS system is as low as it is, not much can be gained by using advanced interprocessor topologies. When the number of processor is raised further, using more sophisticated topologies than a ring will have an effect as we saw from the estimates of the performance on a larger number of processors. Other topologies than the tree structure, such as a 2D-torus, should also be considered, but the comparison of the NP-Ring against the NP-Tree implementations indicate that the number of processors should be larger in order to obtain any increased performance.

The time needed for communication was also measured. The results of these measurements are shown in figure 7.5 (see also table C.4). We have measured the number of timer counts taken at the master, before all the messages regarding local winners are received. The measurements are started from when the master is finished computing it's
7.2. The NP-Tree implementation

own neurons.

![Graph](image)

**Figure 7.5: Communication times, NP-Ring and NP-Tree implementations**

Experiments were conducted for communication subsystem clock frequencies of both 8MHz and 10MHz. The reason for doing these experiments was that it was impossible to see from the performance figures the effect of using the different interconnection structures.

As can be seen from the figure, different frequencies give rise to different communication times. This difference is small when the number of processors are small, but increases when the number of modules reaches about 10. For large networks, most of the total time is taken by computation, and therefore the relatively small difference in the communication times, between the two implementations, does not give rise to a noticeable difference in the total performance. For smaller networks, however, a small difference in the total performance was observable. This is because less time is used for computation, whereas the time needed for communication is independent of the number of neurons.

As stated, this experiment was limited to 15 modules. However, it is expected that the difference in performance will continue to increase as the number of modules increases. This
is due to the communication time dependance on the number of modules (or processors $p$) in the case of NP-Ring and $\log p$ in the case of NP-Tree.

The number of levels in the tree structure is also observable in figure 7.5. A new level is reached when 2, 4, and 8 modules are involved in the computations.

### 7.3 The NP-Wave implementation

As stated earlier, the NP-Wave implementation was the first implementation constructed with small networks in mind. We therefore started to measure the performance on even smaller networks than we had been measuring on before. The performance measurements of the NP-Wave implementation was first presented at the International Conference on Digital Signal Processing [84].

#### 7.3.1 Performance

The performance of the NP-Wave implementation is shown in figure 7.6 (see also table C.5). In order to make the measurements comparable to those obtained for the NP-Ring and NP-Tree implementations, we included measurements of networks consisting of 1000 and 10000 neurons and a 26 dimensional training vector. A 1-dimensional ordering of the neurons was used in the measurements.

Since the focus was shifted to smaller networks, experiments for networks containing 64 and 256 neurons were also conducted.

As expected from the estimates, the performance of the NP-Wave implementation is very good on large networks. The performance for the network consisting of 10000 neurons is approximately equal to the other implementations due to the high percentage of time used for computation.

As the network size is decreased to 1000, a difference in performance is observed. The NP-Wave performs slightly better than the NP-Ring and NP-Tree implementations.

When the number of neurons is only 256, the performance of the NP-Wave implementation is clearly lower than is the case for 1000 neurons, but the performance is still increasing as the number of modules is increased.

For the smallest network, with only 64 neurons, the development of the performance
Figure 7.6: Performance of the NP-Wave algorithm

figure as the number of modules is increased does not look too promising. Increasing the number of modules above 8 does not give any significant improvement in performance.

7.3.2 Comparison

The motivation for designing the NP-Wave implementation, was that the NP-Ring and NP-Tree implementations did not give satisfactory performance for networks of a rather limited size. The results in figure 7.6 demonstrate that the NP-Wave implementation does not give rise to the performance as expected in the estimates. However, the estimates are not accurate for small networks, as the increased computational overhead is not reflected in the model.

We also wanted to investigate how the NP-Wave implementation performed on even smaller networks. A series of experiments were conducted on the NP-Ring and NP-Wave
implementations. The NP-Tree was not used in these experiments due to its similar performance to the NP-Ring implementation. The input vector dimensionality was set to 4, and the number of neurons ranged from 64 to 256. The reason for choosing these parameters is that they represent a real world application of the SOM model which will be described later in this chapter.

The performance of the NP-Ring and the NP-Wave implementation on these small sized networks is shown in figure 7.7 (see also table C.6).

![Graph showing performance comparisons, NP-Ring and NP-Wave](image)

**Figure 7.7: Performance comparisons, NP-Ring and NP-Wave**

The performance of the NP-Wave implementation is shown to be greater than that of the NP-Ring implementation for both the network sizes. For the network consisting of 64 neurons, the performance of the NP-Wave implementation is still carefully rising as the number of processors increases, whereas the NP-Ring implementation has a maximum performance when using 9 modules. Even though the increase in performance of the NP-Wave implementation is about 1.4 times the performance of the NP-Ring implementation on 14 modules, the performance of the NP-Wave implementation is not as high as expected due to the reasons mentioned earlier.
The performance of both implementations improve when the number of neurons is increased to 256. However, as the number of modules reaches 8, the performance curve of the NP-Ring implementation starts to flatten.

The performance of the NP-Wave implementation is higher than that of the NP-Ring and NP-Tree implementations, both on small and large networks. As the size of the networks increases, the difference in performance approaches zero, since the computational part of the implementations are very similar.

![Graph showing performance of TP-Ring implementation with 64 neurons](image)

**Figure 7.8: Performance of TP-Ring implementation with 64 neurons**

### 7.4 The TP-Ring implementation

The TP-Ring implementation was also constructed after the focus was set to smaller networks. As can be seen from the equation describing the communication needs of this implementation (equation 6.17), one of the terms is proportional to the product of the
number of neurons and the training vector dimensionality. This makes the TP-Ring implementation unsuitable for large networks, and therefore performance measurements are taken for small networks exclusively. The performance measurements presented here were first presented at the Symposium on Computer-Based Medical Systems [83].

7.4.1 Performance

The TP-Ring implementation includes $\epsilon$, termed weight update interval. This parameter, describes how many training vectors are to be applied locally on each module before an exchange of weight change matrixes is to take place. In order to demonstrate the influence of this parameter, we have chosen to construct one figure for each network size, with several curves representing different values of the $\epsilon$ parameter.

![Graph showing performance of TP-Ring implementation with 256 neurons](image)

Figure 7.9: Performance of TP-Ring implementation with 256 neurons

The performance of the TP-Ring implementation with 64 neurons and a 4-dimensional input vector is shown in figure 7.8 (see also table C.7). Different curves represent the
following values of the weight update interval - $\epsilon = 10$, $\epsilon = 100$, $\epsilon = 1000$. As can be seen from the estimates, the value of $\epsilon$ has a tremendous effect on performance, and should be set as high as possible, without changing the convergence properties of the training. A value of $\epsilon = 1000$ is assumed to be too high for most applications of the SOM model. Using a value of $\epsilon = 10$ means that 150 training vectors are trained (simultaneously on all modules in a 15 module system) before a weight exchange. Which value of $\epsilon$ to use can be gained from convergence experiments on the training set of the actual application.

In figure 7.9 (see also table C.8), the performance of the TP-Ring implementation with 256 neurons is shown. The shape of the curves in this figure are similar to the curves in figure 7.8, but the performance is somewhat higher for the larger network.

A comparison between the TP-Ring implementation and the other implementations will be presented later.

7.5 The NP/TP implementation

The NP/TP implementation was also constructed with small networks in mind, and performance measurements are hence only done on small networks for this implementation. The performance of the NP/TP implementation was presented together with the results of the TP-Ring implementation at the Symposium of Computer-Based Medical Systems [83].

7.5.1 Performance

In the NP/TP implementation, yet another parameter is introduced, making it even more difficult to present the performance, as yet another parameter can be varied. This additional parameter is the number of clusters.

The performance of the NP/TP implementation on a network consisting of 64 neurons and a 4-dimensional input vector is shown in figure 7.10 (see also table C.9). In this figure, the value of $\epsilon$ is 10.

The performance obtained on 1 module is included as a reference. We have not measured the performance when there are only one module in each cluster, since this is the same as the TP-Ring implementation. In the case where the number of clusters is 6 and 7, there are no curve, but merely one point, representing 12 and 14 modules respectively.

For a network of the given size, the best performance is obtained when the number of
clusters is 5. When using 4 clusters a similar performance is achieved. High performance is obtained in all the configurations where the number of clusters is high. The poorest performance is obtained when the number of clusters is only 2. This is because we are now approaching the NP-Ring implementation.

When the value of the weight update interval, $\epsilon$, is increased, it is expected that the best configuration will be a configuration with a higher number of clusters, since the communication in such configurations are very dependent on the size of the local weight matrix. When $\epsilon$ is increased, the weight matrixes are exchanged more seldom, whereas the communication within a cluster is unchanged.

In figure 7.11 (see also table C.10), the network size is the same as in figure 7.10, but the value of $\epsilon$ is raised to 1000. Performance increases as the number of clusters increase due to the reduction in the time needed to exchange weight matrixes, and the constant time taken for communication within one cluster.
Figure 7.11: Performance of the NP/TP implementation, 64 neurons, $\epsilon = 1000$

We also include the measurements of a larger network. In figure 7.12 (see also table C.11), the performance of the NP/TP implementation is given for a larger network consisting of 256 neurons. The value of the parameter $\epsilon$ in this figure is 10.

As the number of neurons is increases, the configurations with a few clusters becomes more competitive. There are many reasons for this. If there are many clusters, then the amount of data which must be exchanged between the modules increases as the number of neurons is increased. The communication within a cluster, on the other hand, is not dependent on the number of neurons in the network. The performance of the different configurations are fairly similar. The highest performance is achieved when the number of clusters is 5, as is the case for a 64 neuron network.

We also include results of the simulations where the value of $\epsilon$ is increased to 1000. As was the case in the performance of the network consisting of 64 neurons, the configuration with a large number of clusters takes over as the configuration giving the highest performance. The results of this experiment is shown in figure 7.13 (see also table C.12).
Figure 7.12: Performance of the NP/TP implementation, 256 neurons, $\epsilon = 10$

For both the networks with 64 and with 256 neurons, the choice of number of modules to use in each cluster varies with the choice of the parameter $\epsilon$. The larger this value is, the better the performance gained. In the next section, we will look closer into an application of the SOM model which we have been investigating. This application is visualization of multispectral images. By doing experiments on real data, it is demonstrated how this value of $\epsilon$ can be selected.

### 7.6 The MR-visualization application

The use of the SOM model for visualization of MR-images (Magnetic Resonance Images) was first introduced by Manduca [67]. We start by giving a brief description of how the SOM model is used in this application, and give an example of the results achieved by using the SOM model on a 4 channel MR-image.
Figure 7.13: Performance of the NP/TP implementation, 256 neurons, $\epsilon = 1000$

### 7.6.1 SOM in visualization

The SOM model takes as input training data from a high dimensional input space. The training process ensures that neurons that are close in the lower dimensional (1- or 2-dimensional) neuron ordering are also close in the original input space.

This ordering can be used as a basis for visualization. If a gray scale visualization is to take place, then a 1-dimensional ordering of the neurons is sufficient. The network is trained until convergence on the multidimensional training vectors. When the training is completed, the neurons in the 1-dimensional chain are labeled with a gray scale value, starting with a value of 0 at one end of the chain, and a value of 1 at the other end of the chain. The training vectors are then applied once more, and is given the color of the best matching neuron.
In a 4 channel MR-image, there are 4 measurements for each spacial position. In order to visualize such an image, many different approaches can be used. A straightforward algorithm would be to take the average of the 4 measured components. A more sophisticated approach is to use the PCA (Principal Component Analysis) transform in order to maximize the image variance [93]. An example of the result of using the PCA-transform for visualizing an MR-image is shown in figure 7.14.

Figure 7.14: Visualization of an MR-image by using the PCA transform

PCA is a linear projection of an image onto a space with lower dimensionality. Often, it may be difficult or impossible to find a linear projection which differentiates all the important clusters in feature space. In such cases, nonlinear projection methods may be more appropriate.

The result of using the SOM model on the MR-visualization problem is shown in figure 7.15. In the experiment, we used a network consisting of 64 neurons.
7.6.2 Weight exchange

As we have seen from the performance measurements, we get increased performance (measured in MCUPS) by increasing the number of vectors trained on each module between weight exchange. We will now investigate how the value of this parameter $\epsilon$ impacts on the convergence rate of a SOM training session. In order to achieve the best performance, we should choose an as high as possible value whilst preserving the convergence rate in the training.

In figure 7.16 (see also table C.13) we have looked on the convergence of the training when the value of $\epsilon$ is varied. The experiments were conducted on a $218 \times 245$ 4 channel MR-image.

In these experiments, we used a network consisting of 64 neurons, and the training vector dimensionality is, as would be expected, four. It turned out that the learning rate, $\alpha$, also had an influence on how often the weight exchange had to be done. The initial learning rate in the figure is $\alpha(0) = 0.01$. The measurements were done on a single module.
Figure 7.16: Convergence for different weight exchange values, and 20000 iterations

The process followed is that the weight changes are accumulated in a weight change matrix and then added to the weight matrix after a number of training vectors has been applied as done in the TP-Ring and the NP/TP implementations.

We first measured how the error, produced by the network, developed when the update took place after each training vector. Then, we measured the effect of increasing the number of vectors trained between each weight update. As can be seen from the figure, there are no large differences in development of the error, even if the weights are not updated until after 250 training vectors are applied. As the number is increased to 300, a variation in the curve is seen with respect to updates after each training vector. However, both curves yield approximately the same error in the end. When the number of iterations between weight update is increased to 500, a large difference between the methods is seen, and when increased to 750, the average error does not decrease at all.

This experiment showed that the weights should be exchanged after at least 250 training
vectors are applied.

![Graph showing online convergence with varying learning rates](image)

**Figure 7.17: Online convergence when varying the initial learning rate**

As the learning rate affects how often the weights should be updated, we investigated how the network learned as the learning rate was varied. Using an initial learning rate of $\alpha(0) = 0.001$ would have allowed us to do a less frequent weight exchange, but as can be seen from figure 7.17 (see also table C.14), this would mean a higher average error. Using $\alpha(0) = 0.01$ gives comparable results to using larger values.

### 7.6.3 Performance comparisons for the MR-image visualization application

Having established approximately how often the weight exchange should be done, we can now start to choose which implementation should be used for the MR-image visualization application. In order to compare the performance of the different implementations, we use
the previous measurements where the value of $\epsilon$ is 10. Using this value, the number of training iterations between weight update is at most 10$p$, which is in all cases below 250 in the present RENNS system where the number of modules is 16.

In figure 7.18, the performance of the NP-Ring, NP-Wave, TP-Ring and NP/TP implementations are shown. The performance of the NP-Tree implementation is omitted due to its similarity with the NP-Ring implementation.

![Graph showing performance comparison](image)

**Figure 7.18:** Comparison of the different implementations on the MR-image visualization application

As can be seen from this figure, the neuron parallel implementations cannot compare with the TP-Ring and the NP/TP implementations. The best results are achieved when using the NP/TP implementation. For the NP/TP implementation with 5 clusters, only 50 training vectors are applied between each weight update. The results are, as such, not directly comparable since the number of trained vectors in the TP-Ring implementation and the various NP/TP implementations are not identical. In any case, it would be the NP/TP implementation which gives the best performance, since weight update is done more often in the NP/TP implementation than in the TP-Ring implementation.
7.7 Performance on a larger application

In order to demonstrate that the size of the application affects which implementation to choose, we consider the performance of a larger application.

We have conducted some experiments with using the SOM model to pick first break arrivals in seismic traces. The use of the SOM model with this application is still immature, but it represents a problem which requires a larger network than was the case in the MR-image visualization case.

![Graph showing performance comparison]

Figure 7.19: Performance in the first break arrival application

When we experimented with this application, we varied the dimensionality of the training vector from 16 to 128 samples. The number of neurons in the network was varied from 256 to 4096 neurons.

In figure 7.19 (see also table C.15), the performances of the NP-Ring, the TP-Ring, and the NP/TP implementations are presented. In these measurements, the size of the
network was 2048 neurons and a training vector dimensionality of 32. The weights were exchanged after each module had trained 10 vectors.

As can be seen from the figure, the best performance is obtained by the NP-Ring implementation. When using networks of this size, the amount of computation on each module is so large that the communication part does not have as significant effect as in the case of a network with only 64 neurons. For the TP-Ring implementation, however, each module must send $2048 \times 32$ words to all other modules. This communication takes long time, and therefore the performance of the TP-Ring implementation is not especially good on such large networks. Since the NP-Ring implementation performed this good on this problem, we chose to use the configuration of the NP/TP implementation with only 2 clusters, since this is the configuration which is most like the NP-Ring implementation. We measured the performance for a $2 \times 2$ and a $2 \times 4$ module configuration, but as it was clear that the NP/TP implementation could not compare with the NP-Ring implementation even for a small number of modules, we did not do any measurements for a larger number of modules for the NP/TP implementation. Using the NP-Wave implementation would give a slightly increased performance compared to the NP-Ring implementation.
Chapter 8

Conclusions and recommendations

8.1 Summary and Conclusions

The work presented in this thesis concentrates on parallel implementations of the Self-organizing Map neural network model for the RENNNS computer system. The SOM model has been described, and analyzed in a parallel context.

The results presented in this thesis include the following:

- The Self-organizing Map model has several parallel aspects which can be utilized in a parallel implementation. Several different implementations have been undertaken, exploiting many of these aspects. Theoretical models of the implementations have been developed, and the dependencies on problem parameters can be extracted from the models.

- A short review of the actual applications of the SOM model has shown that most of these are of a limited size. The implementations carried out by other researchers have focused on larger networks. Parallelizing large networks is a relatively easy task with respect to getting near linear speedup, since the amount of communication in a neuron parallel SOM implementation is independent on the number of neurons. In this thesis, we have set focus on speeding up the smaller sized networks, thus investigating how to construct improved algorithms for the actual applications of the SOM model.

- It is demonstrated that the size of the network has a large impact on which parallelization scheme should be used. For large networks, neuron parallel implementations should be used, whereas for small networks, implementations using training example
parallelism are more advantageous. Thus, in order to obtain the highest performance for an arbitrarily sized problem, a number of different parallel programs is required.

8.2 Recommendations for future work

Working in a research field in 4 years leaves a lot of loose ends which can be followed in order to improve the results obtained in this work. Among the things which would deserve further investigation are:

- The theoretical models of the implementations should be expanded. At the time being, the models only cover the part of the computation which involves computation of the neurons and update of the weights. The rest of the operations involved in one iteration of the implementations should be included. If we had a more detailed model, then the model could be used for predicting which implementation to use.

- There is room for improvement of the implementations. We emphasize the following points

  - In most cases, and especially for the small networks, we have the ability to distribute the training vectors to all the modules, and the time needed for communication in the neuron parallel implementations could be reduced.

  - The distribution of the training vector (or training vector index, if the training vectors are distributed) could be done together with the distribution of the previously winning neuron. This would reduce the time needed for communication.

  - Although an optimizing compiler is used, the code generated is not optimal. The code could be improved by manually altering the code generated by the compiler.

  - The NP-Tree implementation could use the direct mode of the Vector configuration for distributing the training vector and the winner's identification, since the master is the only module sending on the global ring.

  - When the neighborhood size and learning rate are decreased linearly, a constant term could be computed prior to the training determining how much these values should decrease for each iteration.

- The effect of updating the weight matrixes after a number of training vectors are applied should be investigated further, both with respect to convergence and the ordering.

- The reconfigurability of the communication subsystem should be investigated further in order to adapt it more towards the needs of the SOM implementation. In appendix
8.2. Recommendations for future work

A, we have described a possible configuration where this is done. The configuration has not been implemented.

- The modifications of the original SOM model should be investigated to see if any of these could lead to more efficient parallel implementations.
The RENNS reconfigurability

RENNS is a reconfigurable computer, and the reconfiguration can be exploited on several levels. In this appendix, we discuss how this reconfigurability can be exploited by parallel SOM algorithms on the two main levels, the logic-level and the data path level.

A.1 Reconfiguration on the logic level

The communication subsystem, described in section 5.3.2, contains a number of reconfigurable FPGAs. The implementations described in the previous chapter did not take advantage of this reconfigurability, they all used the same general Vector configuration.

An investigation was done on how the reconfigurability of the communication subsystem could be exploited better by tailoring the configuration directly towards the SOM model.

A neuron parallel SOM implementation, such as the NP-ring implementation, has three communication steps:

1. Master broadcasts a training vector
2. Slaves send local winner to Master
3. Master broadcasts global winner

By utilizing the reconfigurability, the two last steps can be moved from the processing subsystem to the communication subsystem. The proposed configuration, the Winner
configuration, does also take care of some of the computation which is presently done on
the processing subsystem. More specific, we wish the communication subsystem to take
care of the comparison of both the local winners and the global winners.

In the Winner configuration, the RENNS modules are connected in one global ring.
Other communication topologies can be constructed by using local rings, as done in the
NP-Tree implementation (see section 6.2). On this ring, a packet is circulating. The packet
format is shown in figure A.1.

| Tagfield (16b) | Neuron (16b) | Minimal value (32b) |

Figure A.1: The format of the packet in the Winner configuration (arrow indicates direction)

The packet contains information about the current global winner, and consists of three
fields. The minimal field holds the current lowest neuron value reported from one of the
RENNS modules, the neuron ID is the identification the corresponding neuron which pro-
duced this lowest value, and the tagfield indicates which RENNS modules have finished
computing all their neurons for the current training vector.

When a RENNS module has finished computing it’s neurons, it writes a command to
the communication subsystem, making it clear its bit in the tagfield the next time the
packet passes the communication subsystem. When the tagbits for all the modules are
cleared, the communication subsystem on each module sends the global winning neuron’s
identification to the incoming FIFO bank, where it is read by the processor.

The interior of a stream controller using the Winner configuration is outlined in figure
A.2. The shift register holds the current best matching unit received either from the
processor or from the ring. The stream controller has two sources of inputs: The outgoing
FIFO and the previous RENNS module in the ring. When data are present in the FIFO,
a busy signal to the previous RENNS module is set. The first 4 bytes in the FIFO is
the value of the neuron computed. This value is compared, byte by byte, with the value
in the shift register, and the larger of the two is shifted back into the register. Once a
difference in the values is detected, the multiplexer selectors are fixed until the tagfield is
encountered. The tagfield is always shifted back from the shift register when the FIFO is
the input source.

When a packet is received from the ring, the operation is the same with a few exceptions.
First, the values shifted back into shift register are also forwarded to the next module, secondly the tag-field is always taken from the ring, combined with the current value of the local tag-bit.

![Block diagram of the stream controller, WINNER protocol.](image)

The Winner configuration would move certain tasks away from the processor. In for instance the NP-Ring implementation, the processor needs to compare the output of each computed neuron with the value of the so far best matching unit. If the newly computed neuron is less, its id and value is kept as the current best matching unit. Both this comparison and the conditional setting of new best matching unit will be taken care of in the stream controller operating in parallel with the processor.
A.2 Reconfigurable interconnection network

When the RENNS computer system was designed, it was decided to construct a reconfigurable interconnection switch. An analysis of such a switch was done by Zachman [132]. No switch has been built as yet, and reconfiguration at this level is done by manual recabling.

The different SOM implementations does to some extent use different interprocessor topologies. As will be pointed out in the results chapter, the size of the network, and the properties of the training data impacts which implementation gives the highest performance. In order to determine which implementation to use, it would be a great advantage to have the ability to execute some iterations with each of the different implementations, and then decide which of them to use. With a reconfigurable programmable interconnection network, this task could be done automatically, by making the implementation configure the interconnection network to its desired topology.

In the NP/TP implementation, a 2D-torus interconnection network is used. It must however be decided how many processors should be allocated to each cluster. Changing the number of processors within a cluster means changing the interprocessor topology, and it would be very convenient if the program itself could handle the change in interconnection of the modules.
B

Source code

This appendix contains the source code.

B.1 Common SOM routines

#include <math.h>

int
computebest(double *somnet,
    double *vector,
    double *mindist,
    int numneur,
    int dimension)

{
    int index,minindex,i,j;
    double dist,tempdist;

    *mindist=10000000.00;
    minindex=-1;
    index=0;
    for(i=0;i<numneur;i++)
    {
        dist=0.0;
        for(j=0;j<dimension;j++)
        {
        }
tempdist=(somnet[index]-vector[j]);
dist+=tempdist*tempdist;
index++;}

if(dist<*mindist)
{
    *mindist=dist;
    minindex=i;
}

*mindist = sqrt(*mindist);
return(minindex);
}

void updatenetwork(double *somnet,

    double *vector,
    int startupd,
    int stopupd,
    int dimension,
    double alpha)
{
    int index;
int i,j;

index=startupd*dimension;
for(i=startupd;i<stopupd;i++)
{
    for(j=0;j<dimension;j++)
    {
        somnet[index]+=alpha*(vector[j]-somnet[index]);
        index++;
    }
}
}

void computechange(double *somnet,

    double *vector,
    double *delta,
    int startupd,
    int stopupd,
    int dimension,
double alpha)
{
    int index;
    int i,j;

    index=startupd*dimension;
    for(i=startupd;j<stopupd;i++)
    {
        for(j=0;j<dimension;j++)
        {
            delta[index]+=alpha*(vector[j]-somnet[index]);
            index++;
        }
    }
}

void updateclrdelta(double *somnet,
                      double *delta,
                      int dimension,
                      int numneur)
{
    int i;
    for(i=0;i<numneur*dimension;i++)
    {
        somnet[i]+=delta[i];
        delta[i]=0;
    }
}
B.2 The NP-Ring implementation

/* This program is a parallel version of the Self Organizing Feature Map
   The program uses the vector configuration of the communication
   subsystem. Only one ring is used. The modules must be connected in a ring
   on stream controller 1. Fifo 1 is used for sending, and fifo 3 is used
   for receiving. */

#define NUMITER 1000 /* These could just as well been */
define ALPHA 0.05 /* read from console */
define MINIM(x,y)((x) < (y)) ? (x) : (y))
define MAXIM(x,y)((x) > (y)) ? (x) : (y))

#include <stdio.h>
#include <math.h>
#include "com_addr.h"

struct sc
{
    int cmd;
    int adr;
};

void init_memory_pointers(); /* Memory allocation */
double *mem_alloc(int,unsigned *); /* routines for RENNS */

int getint();

int computebest(double *,double *,double *,int,int); /* SOM routines */
void updatenetwork(double *,double *,int,int,double);

/* Configuration values for comsys */
int fregs[4] = {1,3,1,3}; /* F1->S1, F2->S3, S1->F3, S3->F */
int sregs[12] = {1,0,0,0,
    1,0,0,0,
    1,0,2,0}; /* enable stream ctrl 1 only */
int aregs[4]; /* stream 1 uses F1, stream 3 uses F2 */

main()
{
    /* RENNS specific memory pointers */
    volatile unsigned *id = (unsigned *)0x804400;
}
volatile unsigned *oflags = (unsigned *)0x805608;
volatile unsigned *iflags = (unsigned *)0x805609;
volatile unsigned *sfp = (unsigned *)0x801100;
volatile unsigned *fpi = (unsigned *)0x801000;
volatile float *fpf = (float *)0x801000;
volatile unsigned long *tctrl = (unsigned long *)0x808020;
volatile unsigned long *tcount = (unsigned long *)0x808024;
volatile unsigned long *tperiod = (unsigned long *)0x808028;
volatile struct sc *str1 = STREAM1;

/*local variables*/
int node,numnodes,error,numneur,startpoint,ournumeur;
int dimension,minindex,i,j,k,m,index,/startx,stopx,starty,stopy;
int startupd,stopupd,neighsize,numneurx,numneury,row,col,maxxy;
double alpha,startalpha;
double *testvector;
double *somnet,*vector;
double mindist,tempdist,dist;
unsigned long time,numitter;

/* Timing variables */
unsigned c0,c1,c2,c3,c4,c5;
unsigned memtype;
double secs,cup,kcups;
*tperiod = 0xffffffff;

numnodes = 1; /* one node is always participating */
time = 0;
cup = 0.0;

node = *id & 0xff;
xlat7seg(node); /* Read our nodes identification */

/* Write this identification to the 7seg display */

printf("RENNNS Self Organizing Feature Maps, NP-Ring version\n\n");
printf("How many modules in this run? ");

numnodes = getint();
printf("How many neurons in the x-dimension? ");
umneurx = getint();
printf("How many neurons in the y-dimension? ");
numneury = getint();
printf("How many dimensions in input? ");
dimension = getint();
umneurx=numneury*numneury;
maxxy=MAXIM(numneurx,numneury);

printf("Starting with %d modules, %d x %d neurons\n", numnodes,numneurx,numneury);

for(i=0;i<4;i++)
    aregs[i]=(dimension / 2); /* Set FIFO-flags */
asm(" OR 800h,ST"); /* Enable cache */

if(node==0) /* The master is responsible for inserting first token */
    {
        sregs[4]=9;
        sregs[0]=2; /* The master has address = 2 */
    }

error=init_comsubsyst(fregs,sregs,aregs); /* Initialize CS */
if(error)
    {
        printf("Error in init_comsubsyst: %d\n",error);
        exit(0);
    }
printf("Communication initialized. Press any key to start\n");
getchar();
printf("\n\n");

init_memory_pointers(); /* Initialize memory allocation */
time=0;
cup=0.0;

if(node==0) /* master */
    {
        for(i=0;i<100000;i++); /* Need to delay the master */
        startalpha=ALPHA;
        startpoint = (node % numnodes); /* Our first neuron in x direction*/

        /* Determine our number of neurons per row */
        ournumnumneur = (numneurx/numnodes);
if((numneurons*ournumneur*numnodes)>startpoint) ournumneur++;

/* Allocate space for training vector */
testvector = (double *) mem alloc(dimension,&memtype);

/* Allocate space for weights */
somnet = (double *) mem alloc(ournumneur*dimension*numneury,&memtype);
if(somnet==0)
{
    printf("Not enough memory for this network\n");
    printf("Tried to allocate %d * %d * %d * %d\n",  
            ounumneur,dimension,numneury,sizeof(float));
    exit(0);
}

srand(node); /* Take a fixed seed */

/* Initialize weights to values between -0.5 and 0.5 */
for(i=0;i<ournumneur*dimension*numneury;i++)
    somnet[i]=(((rand() % 1000000) + 1) / 1000000.0)-0.5;

*tctrl = 0x2c0;
ct0 = *tcount; /* Start counter */

for(numiter=0;numiter<NUMITER;numiter++)
{
    for(i=0;i<dimension;i++) /* generate new trainingvector */
        testvector[i]=(((rand() % 1000000) + 1) / 1000000.0);

    /* Wait until space in outgoing FIFO */
    while(!(*oflags & 0x4))
        ;

    *sfp=0x61000000; /* Address of slaves = 1*/
    for(i=0;i<dimension;i++) /* Broadcast the */
        {
            *fip=testvector[i]; /* training vector */
        }
    *sfp=TOKEN; /* Reinsert token */
/* compute local winner */
minindex=computebest(somnet,testvector,&mindist,ournumneur*numneury,
dimension);

/* Determine global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex=row*numneurx+col*numnodes+startpoint;

/* await best match from the other modules */
for(i=0;i<numnodes-1;i++)
{
    while((*iflags & 0xf)==0x4) /* Await data in FIFO */
    ;
    dist=*fpf; /* Read best value */
    while((*iflags & 0xf)==0x4) /* Await data in FIFO */
    ;
    index=*fpi; /* Read index */
    if(dist<mindist) /* Check if better than current winner */
    {
        mindist=dist; /* If so, replace */
        minindex=index;
    }
}

/* Time to broadcast the index of the winner */
while(!(*oflags & 0x4)) /* Await space in outgoing FIFO */
    ;
sfp=0x61000000; /* Set address of slaves */
*fpi=minindex; /* Send index */
sfp=TOKEN; /* Reinsert token */

/* Compute size of neighborhood */
neighsize=(int)((float)maxxy/(float)NUMITER)
    *(float)time+maxxy);

/* Adapt learning rate */
alpha=startalpha*(1.0-(float)(numiter/((float)NUMITER)));
/* Determine row and column of winner */
row=minindex/numneurx;
col=minindex%numneurx;

/* Check if neighborhood exceeds border of network */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);
starty=MAXIM(row-neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/* Number of neurons which will be changed */
cup+==(stopx-startx+1)*(stopy-starty+1);

/* Determine which of our neurons in each row are within the neighborhood */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes<startpoint))
    stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(somnet,testvector,m+startupd,
                  m+stopupd,dimension,alpha);

time++;
}

c5 = *tcount;                         /* Read the timer */
cup* =dimension;                      /* Find the number of weights updated */
secs = (double)(c5-c0)/8000000.0;     /* Find the CUPS figure */
kcups = (cup/sec) / 1000.0;
printf("\nPerformance : %.3f KCups/Sec",kcups);

} else                                        /* slave */
{

numiter=NUMITER;
startalpha=ALPHA;

startpoint = (node % numnodes);  /\ Same as for master */
ournumneur = (numneurx/numnodes);
if((numneurx-ournumneur*numnodes)>startpoint) ournumneur++;

vector = (double *) mem_alloc(dimension,&memtype);
if(vector==0)
{
    printf("Could\'nt allocate vector\n");
    exit(0);
}

somnet = (double *) mem_alloc(ournumneur*dimension*numneury,&memtype);
if(somnet==0)
{
    printf("Not enough memory for this network\n");
    printf("Tried to allocate %d * %d * %d * %d\n",  
            ournumneur,dimension,numneury,sizeof(float));
    exit(0);
}

rand(node);
for(i=0;i<ournumneur*dimension*numneury;i++)
    somnet[i]=((rand() % 1000000) + 1) / 1000000.0-0.5;

for(i=0;i<numiter;i++)
{
    while(!(*iflags & 0x2))  /\ Await training vector */
        ;
    for(k=0;k<dimension;k++)  /\ Read training vector */
        vector[k]=*fpf;

    /* Compute local winner */
    minindex=computebest(somnet,vector,&mindist,ournumneur*numneury, 
            dimension);

    /* Determine global index */
    row=minindex/ournumneur;
    col=minindex%ournumneur;
    minindex=row*numneurx+col*numnodes+startpoint;

/* Send local winner to master */
while(!(*oflags & 0x4)) /* Check if space in outgoing FIFO */
    ;
*sfp=0x62000000; /* Master address is 2 */
*fpf=mindist; /* Send winners value */
*fpi=minindex; /* and index */
*sfp=TOKEN; /* Reinsert token */

/* compute size of neighborhood and learning rate */
    while awaiting global winner from master */
neighsize=(int)-((float)maxxy/(float)numiter)
    *(float)time+maxxy);
alpa=startalpha*(1.0-(float)((float)/(float)numiter)));

while(*iflags & 0xf) == 0x4) /* Global winner present ? */
    ;
minindex=*fpi; /* Got it */

/* Determine row and column of winner */
row=minindex/numneurx;
col=minindex%numneurx;

/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy =MINIM(row-neighsize,numneury-1);

/* Determine which of our neurons in each row are within neighborhood */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes<startpoint))
    stopupd--;

/* update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(somnet,vector,m+startupd,m+stopupd,dimension,alpha);
time++;
B. Source code

}

B.3 The NP-Tree implementation

/*
   This program is a parallel version of the Self Organizing Feature Map.
   The program uses the vector configuration of the communication subsystem.
   One ring is used for broadcasting training data and winner information.
   In addition, the modules are connected in a tree by making a ring of
   the parent and its two children. This tree structure is made for determining
   the winner.
*/

#define NUMITER 1000
#define ALPHA 0.05
#define MINIM(x,y) (((x) < (y)) ? (x) : (y))
#define MAXIM(x,y) (((x) > (y)) ? (x) : (y))

#include <stdio.h>
#include <math.h>
#include "com_addr.h"

struct sc
{
    int cmd;
    int adr;
};

void init_memory_pointers(); /* Memory allocation */
double *mem_alloc(int,unsigned *); /* routines for RENNS */

int getint();

int computebest(double *,double *,double *,int,int); /* SOM routines */
void updatenetwork(double *,double *,int,int,int,double);

/* Configuration values for comsys */
int fregs[4] = {1,4,1,2}; /* F1->S1, F2->S4, S1->F3, S2->F4 */
int sregs[12] = {1,2,3,1,
                 1,0,0,0,
                 1,0,0,2}; /* adr:S1=1,S2=2,S3=3,S4=1 */
int aregs[4]; /* enable stream ctrl 1 only */
/* stream 1 uses F1, stream 4 uses F2 */

main()
{    /* RENNNS specific memory pointers */    volatile unsigned *id = (unsigned *)0x804400;    volatile unsigned *oflags = (unsigned *)0x805608;    volatile unsigned *iflags = (unsigned *)0x805609;    volatile unsigned *sfp = (unsigned *)0x801100;    volatile int *fpi = (int *)0x801000;    volatile double *fpi0 = (double *)0x801000;    volatile unsigned *sfpi0 = (unsigned *)0x801300;    volatile int *fpilocal = (int *)0x801200;    volatile double *fpi0local = (double *)0x801200;    volatile int *seed = (int *) 0x808024;    volatile unsigned long *tctrl = (unsigned long *)0x808020;    volatile unsigned long *tcount = (unsigned long *)0x808024;    volatile unsigned long *tperiod = (unsigned long *)0x808028;    volatile struct sc *strl = STREAMl;

    /*local variables*/    int node, numnodes, error, numneur, startpoint, ournumneur;
    int dimension, minindex, ci, i, j, k, m, index, neigh, startx, stopx, starty, stopy;
    int startupd, stopupd, neighsize, children, row, col, maxxy, numneurx, numneury;
    double alpha, startalpha, *testvector, *somnet, *vector;
    double mindist, tempdist, dist, temp;
    unsigned long time, numiter;
    unsigned memtype;

    /* Timing variables */
    unsigned c0, c1, c2, c3, c4, c5;
    double secs, cup, kcups;

    *tperiod = 0xffffff;

    printf("RENNNS Self Orgranizing Feature Maps, NP-Tree version\n\n");
    printf("How many modules in this run? ");
    numnodes = getint();
    printf("How many neurons in the x-dimension ");
    numneurx = getint();
    printf("How many neurons in the y-dimension ");
    numneury = getint();
    printf("How many dimensions in input ? ");
    dimension = getint();
    numneur = numneurx*numneury;
    maxxy = MAXIM(numneurx, numneury);
printf("Starting with %d nodes, %d x %d neurons\n", 
numnodes,numneurx,numneury);

for(i=0;i<4;i++)
    aregs[i] = (dimension / 2);                       /* Set FIFO-flags */

node = *id & 0xf;
xlat7seg(node);                                       /* Read our nodes identification */
                                                        /* Write identification on 7seg display */
asm(" OR 800h,ST");                                   /* Enable cache */

/* Determine how many children we have */
temp = numnodes / 2.0;
if(temp<node) children=0;
if(temp==node) children=1;
if(temp>node) children=2;

if(node==1)                                              /* Master module */
    { sregs[4]=9;                                          /* Master is responsible for settin token on ring */
      sregs[0]=2;                                          /* Master is given another address than the slaves */
    }
else
    { sregs[7]=1;                                         /* Stream 4 activated */
    }

if(children)                                             /* Also father */
    { sregs[5]=1;                                         /* Father is given another adress */
      sregs[1]=2;
      sregs[6]=1;
      sregs[2]=3;
    }
if(!(node==1) && !(node%2))                               /* Insert token on subring */
    { sregs[7]=9;
    }

if((numnodes==node) && !(node % 2))                       /* Only one child => 2 dummy stream controllers */
    { sregs[2]=3;
    }
sregs[6]=1;
}

/* Initialize communication subsystem */
error=init_comsubsys(fregs,sregs,aregs);
if(error)
{
    printf("Error in init_comsubsys: %d\n",error);
    exit(0);
}
printf("\n\nInitialized... Press any key to start\n");
getchar();

time = 0;
cup = 0.0;

init_memory_pointers();          /* Initialize memory allocation */

if(node==1)
    /* master */
{
    for(i=0;i<1000000;i++);
    startalpha=ALPHA;
    /* Delay master module */

    /* Find our first neuron in the x-direction */
    startpoint = (node % numnodes);

    /* Find our number of neurons per row */
    ournumneur = (numneurx/numnodes);
    if(((numneurx-ournumneur*numnodes)>startpoint) ournumneur++;

    /* Allocate space for training vector */
    testvector = (double *) mem_alloc(dimension,&memtype);

    /* Allocate space for weights */
    somnet = (double *) mem_alloc(ournumneur*dimension*numneury,&memtype);
    if(somnet==0)
    {
        printf("Not enough memory for this network\n");
        printf("Tried to allocate %d * %d * %d\n",
            ournumneur,dimension,sizeof(float));
        exit(0);
    }
srand(node); /* Fixed seed */

/* Initialize weights */
for(i=0;i<ournumneur*dimension;i++)
    somnet[i]=((rand() % 1000000) + 1) / 1000000.0;

tctrl = 0x2c0; /* Start counter */
c0 = *tcount;

for(numiter=0;numiter<NUMITER;numiter++)
{
    /* Generate new trainingvector (random) */
    for(i=0;i<dimension;i++)
        testvector[i]=(((rand() % 1000000) + 1) / 1000000.0);

while(!(oflags & 0x4)) /* Wait until space in FIFO */
    ;

*sfp=0x61000000; /* Set address of slaves (=1) */
for(i=0;i<dimension;i++)
{
    *fpf=testvector[i]; /* Broadcast training vector */
}
*sfp=TOKEN; /* Reinsert token */

/* Compute local winner */
minindex=computebest(somnet,testvector,&mindist,ournumneur*numneury,
    dimension);

/* Determine global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex=row*numneurx+col*numnodes+startpoint;

/* Await best match from our children */
for(i=0;i<children;i++)
{
    while((oflags & 0xf0)==0x40) /* Data in FIFO ? */
        ;
    dist=*fpflocal; /* Local winners value */
while(*(iflags & 0xf0)==0x40) /* Data in FIFO */
    ;
    index=*fpilocal; /* Local winners id */
if(dist<mindist) /* Compare with current winner */
    {
        mindist=dist;
        minindex=index;
    }
    /* Replace */

/* Broadcast the winners identification */
while(!(iflags & 0x4)) /* Space in outgoing FIFO */
    ;
    *sfp=0x61000000; /* Slaves address */
    *fpi=minindex; /* Winners index */
    *sfp=TOKEN; /* Reinsert index */

/* Compute size of neighborhood */
neighsize=(int)(-(float)maxxy/(float)NUMITER)
        *(float)time+maxxy);

/* Adapt learning rate */
alpha=startalpha*(1.0-(float)((float)numiter/(float)NUMITER)));

/* Determine the row and column of winner */
row=minindex/numneurx;
    col=minindex%numneurx;

/* Determine if neighborhood exceeds border of map */
startx=MAXIM(col-neighsize,0);
    stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
    stopy =MINIM(row+neighsize,numneurx-1);

/* Number of neurons which will be changed */
cup+==(stopx-startx+1)*(stopy-starty+1);

/* Determine which of our neurons within each row
    are within the neighborhood */
    startupd = startx / numnodes;
    stopupd = stopx / numnodes;

if((startx*numnodes)>startpoint)
    startupd++;
if((stopx*numnodes<startpoint))
    stopupd--;

    /* Update weights */
    for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    
        updatenetwork(somnet,testvector,m+startupd,m+stopupd,dimension,
             alpha);

    time++;
}

c5 = *tcount;                          /* Read timer */
cup* =dimension;                     /* Compute the number of weight updates */
secs = (double)(c5-c0)/8000000.0;    /* Compute how long time it has taken */

kcups = (cup/secs) / 1000.0;
printf("\nPerformance : %0.3f KCups/Sec",kcups);
}
else                                           /* slave */
{
    numiter=NUMITER;
    startalpha=ALPHA;

    startpoint = (node % numnodes);            /* Same as for master */
    ournumneur = (numneurx/numnodes);
    if((numneurx-ournumneur*numnodes)>startpoint) ournumneur++;

    /* Allocate space for training vector */
    vector = (double *) mem_alloc(dimension,&memtype);
    if(vector==0)
    {
        printf("Couldnt allocate vector\n");
        exit(0);
    }

    /* Allocate space for weights */
    somnet = (double *) memAlloc(ournumneur*dimension*numneury,&memtype);
    if(somnet==0)
    {

printf("Not enough memory for this network\n");
printf("Tried to allocate \%d * \%d * \%d\n",
    ournumneur, dimension, sizeof(float));
exit(0);
}
srand(node);
/* fixed seed */

/* Initialize weights to random numbers between -0.5 and 0.5 */
for(i=0;i<ournumneur*dimension*numneury;i++)
    somnet[i]=((rand() % 1000000) + 1) / 1000000.0-0.05;

for(i=0;i<numiter;i++)
{
    while(!(*iflags & 0x2))        /* Await training vector */
    ;
    for(k=0;k<dimension;k++)
        vector[k]=*fpf;

    /* Compute local winner */
    minindex=computebest(somnet, vector, &mindist, ournumneur, dimension);

    /* Compute local winners global index */
    row=minindex/numneurt;
    col=minindex%numneurt;
    minindex=row*numneurt+col*numnodes+startpoint;

    /* Await local winners from eventual children */
    for(ci=0;ci<children;ci++)
    {
        while(!(*iflags & 0xf0)==0x40)
            ;
        dist=*fpflocal;        /* Local winners value */

        while(!(*iflags & 0xf0)==0x40)
            ;
        index=*fpilocal;        /* Local winners index */

        if(dist<mindist)        /* Compare to current winner */
            {
                mindist=dist;
                minindex=index;        /* Replace */
            }
} }

/** Time to send local winner of this subtree to parent */
while(!(*oflags & 0x40)) /**< Check if space in FIFO */
;
*sfplocal=0x62000000; /**< Send winners value */
*fplocal=minindex; /**< and index */
*sfplocal=TOKEN; /**< Reinsert token */

/** compute size of neighborhood while waiting for answer */
neighsize=(int)-((float)maxxy/(float)numiter)
 *(float)time+maxxy);
alpha=startalpha*(1.0-(float)((float)i/((float)numiter)));

while((*iflags & 0xf) == 0x4) /**< Check for global winner */
;
minindex=*fpi /**< Got it */

/** Determine row and column of global winner */
row=minindex/numneurx;
col=minindex%numneury;

/** Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/** Determine which neurons in each row are within neighborhood */
startupd = startx / numnodes;
stopudp = stopx / numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes<startpoint))
    stopudp--;

for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(somnet, vector, startupd, stopudp, dimension, alpha);
time++;


```java
}
} }
```
B.4 The NP-Wave implementation

/*
 This program is a parallel version of the Self Organizing Feature Map
 with 1 dimensional linear ordering of the nodes. The program uses the
 Vector configuration in a point to point mode. Only one ring is used. The
 nodes must be connected in a ring from stream 1 on the sender to stream 2
 on the receiver.
 */

#define ALPHA 0.4
#define MINIM(x,y) (((x) < (y)) ? (x) : (y))
#define MAXIM(x,y) (((x) > (y)) ? (x) : (y))

#include <stdio.h>
#include <math.h>
#include "com_addr.h"

int getint();

void init_memory_pointers(); /* Memory allocation */
double *mem_alloc(int,unsigned *); /* routines for RENNS */

int computebest(double *,double *,double *,int,int);
/* SOM routines */
void updatenetwork(double *,double *,int,int,int,double);

int fregs[4] = {1,3,2,4}; /* F1->S1, dummy, S2->F3, dummy */
int aregs[4];

main()
{
    /* RENNS specific memory pointers */
    volatile unsigned *id = (unsigned *)0x804400;
    volatile unsigned *oflags = (unsigned *)0x805608;
    volatile unsigned *iflags = (unsigned *)0x805609;
    volatile unsigned *sfp = (unsigned *)0x801100;
    volatile unsigned *sendint = (unsigned *)0x801000;
    volatile double *senddbl = (double *)0x801000;
    volatile unsigned *recint = (unsigned *)0x801000;
    volatile double *recdbl = (double *)0x801000;
    volatile unsigned long *tcntrl = (unsigned long *)0x808020;
volatile unsigned long *tcount = (unsigned long *)0x808024;
volatile unsigned long *tperiod = (unsigned long *)0x808028;

/* Local variables */
int i,j,k,m,node,curridx,error,lowid,minindex,lastwinner;
int startupd,stopupd,row,col,maxxy;
int numnodes,numneurx,numneury,numneury, dimension, ournumeur;
int startpoint, startx, stopx, starty, stopy, neighsize;
unsigned memtype;
double *vector,*weight,alpha,mindist,lowest;

/* Timing variables */
unsigned c0,c1,c2,c3,c4,c5, time;
double cup,mcup,secs;

node = *id & 0xff; /* Read this nodes identification */
xlat7seg(node); /* Write it to the 7seg display */
asm(" OR 800h,ST"); /* Enable cache */
*tperiod= 0x00000000;

printf("Welcome to RENNS, a REconfigurable Neural Network Server\n\n");
printf("This is the NP-Wave version of the Self Organizing Maps,\n");
printf("Please report problem parameters\n");
printf("Number of RENNS modules : ");
umnodes = getint();
printf("How many neurons in the x-dimension? ");
umneurx = getint();
printf("How many neurons in the y-dimension? ");
umneury = getint();
printf("Number of iterations : ");
umiter = getint();
printf("Dimensionality : ");
dimension = getint();
umneury=numneurx*numneury;
maxxy=MAXIM(numneurx,numneury);

for(i=0;i<4;i++) /* Set FIFO flags */
    aregs[i]=dimension / 2;

printf("Starting with %d modules, %dx%d neurons, %d iterations, dim = %d\n",numnodes,numneurx,numneury,numiter,dimension);
/* Initialize communication subsystem */
error = init_comsubsys(fregs,aregs);
if(error)
{
    printf("Error in init_comsubsys: %d\n",error);
    exit(0);
}

init_pointstream(1,2,1,0x3,2); /* Make the receivers ready */
for(i=0;i<1000000;i++); /* Delay */
init_pointstream(0,1,2,0xB,1); /* Initialize the channels */

printf("Communication initialized. Press any key to start\n");

getchar();
printf("\n\n");

time=0; /* Start timers */
cup=0.0;
stctrl=0x2c0;

init_memory_pointers(); /* Initialize memory allocation */

alpha = ALPHA;
startpoint = (node % numnodes ); /* Our first neuron in x direction */
ournumneur = (numneurx/numnodes); /* Number of neurons per row */
if((numneurx-ournumneur*numnodes)>startpoint) ournumneur++;

/* Allocate space for training vectors */
vector = (double *) mem_alloc(dimension*numnodes,&memtype);

/* Allocate space for weights */
weight = (double *) mem_alloc(ournumneur*dimension*numneury,&memtype);
if(weight==0)
{
    printf("Not enough memory for this network\n");
    printf("Tried to allocate %d * %d * %d * %d\n", ournumneur,dimension,numneury,sizeof(double));
    exit(0);
}

srand(node); /* Fixed seed */
/* Initialize weights to random values between -0.5 and 0.5 */

for(i=0;i<ournumneur*dimension;i++)
    weight[i]=((rand() % 1000000) + 1) / 1000000.0)-0.5;

c0 = *tcount;

if(node==1) /* Master */
{
    for(i=0;i<numnodes;i++) /* Fill pipe */
    {
        curridx=i*dimension; /* Index in ring buffer of training vectors */

        /* Generate and send new training vector */
        for(j=0;j<dimension;j++)
        {
            vector[curridx+j]=((rand() % 1000000) +1) / 1000000.0);
            *senddbl = vector[curridx+j];
        }

        /* Compute local winner */
        minindex=computebest(weight,&vector[curridx],&mindist,
                               ournumneur*numneury,dimension);

        /* Compute local winners global index */
        row=minindex/ournumneur;
        col=minindex%ournumneur;
        minindex=row*numneurx+col*numnodes+startpoint;

        /* Send local winners index and value */
        *senddbl = mindist;
        *sendint = minindex;
    } /* Pipe filled */

    c1=*tcount;
    for(i=0;i<numiter-numnodes;i++) /* Normal operation */
    {
        curridx = (i % numnodes)*dimension;

        while((*iflags & 0xf)==0x4) /* Await winner from last module */
        {
            lastwinner = *recint;
            *sendint = lastwinner; /* Receive and */
            /* retransmit */
        }
    }
}
/* Compute size of neighborhood */
neighsize=(int)(-(double)maxxy/(double)numiter)
  *(double)time+maxxy);

/* Adapt learning rate */
alpha=ALPHA*(1.0-(double)((double)i/((double)numiter)));

/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;

/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/* Compute number of neurons updated */
cup+==(stopx-startx+1)*(stopy-starty+1);

/* Determine which neurons in each row which will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
  startupd++;
if((stopx%numnodes<startpoint))
  stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
  updatenetwork(weight,&vector[curridx],m+startupd,
  m+stopupd,dimension,alpha);

time++;

/* Generate and send new training vector */
for(j=0;j<dimension;j++)
  {
    vector[curridx+j]=(((rand() % 1000000) +1) / 1000000.0);
    *senddbl = vector[curridx+j];
  }
/* Compute local winner */
minindex=computebest(weight,&vector[curridx],&mindist,
ournumeur*numneury,dimension);

/* Compute local winners global index */
row=minindex/ournumeur;
col=minindex%ournumeur;
minindex=row*numneurx+col*numnodes+startpoint;

/* Send local winners index and value */
*senddbl=mindist;
*sendint=minindex;
}

for(i=0;i<numnodes;i++) /* Empty pipe (only weight update) */
{
    while((*iflags & 0xf)==0x4) /* Await winner */
    {
        lastwinner=*recint;
        *sendint=lastwinner;
    }
    /* Receive and */
    /* Retransmit */

    /* Compute size of neighborhood */
    neighsize=(int)(((double)maxxy/(double)numiter)*((double)time+maxxy);

    /* Adapt learning rate */
    alpha=ALPHA*(1.0-((double)maxxy/(double)numiter));

    /* Compute global winners row and column */
    row=lastwinner/numneurx;
col=lastwinner%numneurx;

    /* Check if neighborhood exceeds map */
    startx=MAXIM(col-neighsize,0);
    stopx =MINIM(col+neighsize,numneurx-1);

    starty=MAXIM(row-neighsize,0);
    stopy =MINIM(row+neighsize,numneury-1);

    /* Compute number of neurons updated */
cup+=(stopx-startx+1)*(stopy-starty+1);
/* Determine which neurons in each row which
will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
startupd++;
if((stopx%numnodes<startpoint))
stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(weight,&vector[curridx],m+startupd,
                   m+stopupd,dimension, alpha);

  time++;
}

  c5= tcount;
  cup* = dimension;  /* Compute number of weights updated */
  secs = (double)(c5-c0)/8000000.0;  /* Number of seconds used */
  mcup=(cup/secs)/1000000.0;
  printf("Node 1 finished, \%f Mcup/s (inc fill)\n",mcup);
}

if((node<numnodes)&(node>1))  /* common nodes */
{
    for(i=0;i<numnodes;i++)  /* Fill pipe (only computation) */
        {
            curridx=i*dimension;

            while(!(*flags & 0x2))  /* Await new training vector */
;
    for(j=0;j<dimension;j++)
    {
        vector[curridx+j]=*recdbl;  /* Receive and */
        *senddbl=vector[curridx+j];  /* retransmit */
    }

    /* Compute local winner */
    minindex=competebest(weight,&vector[curridx],&mindist,
                  ounumneur*numneury,dimension);
/* Compute local winners global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex=row*numneurx+col*numnodes+startpoint;

/* Receive current winner */
while(!flags & 0xf)="0x4")
;
lowest = *recdbl;            /* Winners value */

while(!flags & 0xf)="0x4")
;
lowid = *recint;           /* Winners index */

if(mindist<lowest)         /* Compare local winner to previous winner */
{
    *senddbl = mindist;       /* Send our local winner */
    *sendint = minindex;
}
else
{
    *senddbl = lowest;        /* Send winner from previous module */
    *sendint = lowid;
}

for(i=0;i<numiter-numnodes;i++)       /* Normal operation */
{
    curridx = (i % numnodes)dimension;

    while(!flags & 0xf)="0x4")         /* Await earlier winner */
;
    lastwinner = *recint;
    *sendint = lastwinner;

    /* Compute neighborhood */
    neighsize=(int)((double)maxxy/(double)numiter)
        *(double)time+maxxy);

    /* Adapt learning rate */
alpha=ALPHA*(1.0-(double)i/(double)numiter));
/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;

/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/* Determine which neurons in each row which will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes<startpoint))
    stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(weight,&vector[curridx],m+startupd,
                   m+stopupd,dimension,alpha);

time++; 

while(!(flags & 0x2))   /* Await new training vector */
    
for(j=0;j<dimension;j++)
    
    vector[curridx+j]=*recdbl;
    *senddbl = vector[curridx+j];  /* Receive and */
    
    }  /* retransmit */

/* Compute local winner */
minindex=computebest(weight,&vector[curridx],&mindist,
                    ournumneur*numneury,dimension);

/* Compute local winners global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex= row*numneurx+col*numnodes+startpoint;

while((*iflags & 0xf)==0x4)                      /* Await previous winner */
{
    lowest = *recdbl;                             /* Value */
}
while((*iflags & 0xf)==0x4)                      /* and identification */
{
    lowid = *recint;
}
if( mindist<lowest)                              /* Compare to local winner */
{
    *senddbl = mindist;                           /* Replace if less */
    *sendint = minindex;
}
else                                              /* else retransmit */
{
    *senddbl = lowest;
    *sendint = lowid;
}
}
for(i=0;i<numnodes;i++)                         /* Empty pipe (only weight update) */
{
    curridx = (i % numnodes)*dimension;

    while((*iflags & 0xf)==0x4)                  /* Await previous winner */
    {
        lastwinner = *recint;                    /* Read and */
        *sendint = lastwinner;                   /* retransmit */
    }
/* Compute size of neighborhood */
neighsize=(int)(-(double)maxxy/(double)numiter)
*(double)time+maxxy);
/* Adapt learning rate */
alpha=ALPHA*(1.0-(double)((double)time/((double)numiter)));
/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;
/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx = MINIM(col + neighsize, numneurx - 1);
starty = MAXIM(row - neighsize, 0);
stopy = MINIM(row + neighsize, numneury - 1);

/* Determine which neurons in each row which
 will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx % numnodes) > startpoint)
    startupd++;
if((stopx % numnodes < startpoint))
    stopupd--;

/* Update weights */
for(m = starty * ournumneur; m < stopy * ournumneur; m += ournumneur)
    updatenetwork(weight, &vector[curridx], m + startupd,
                   m + stopupd, dimension, alpha);
time++;
}
printf("Node %d finished\n", node);
}

if(node == numnodes) /* Last node */
{
    for(i = 0; i < numnodes; i++) /* Fill pipe (only computation) */
    {
        curridx = i * dimension;
        while(!(*(iflags & 0x2))) /* Await training vector */
            ;

        for(j = 0; j < dimension; j++) /* We do not need to retransmit */
            vector[curridx + j] = *recdbl;

    /* Compute local winner */
    minindex = computebest(weight, &vector[curridx], &mindist,
                            ournumneur * numneury, dimension);

    /* Compute local winners global index */
    row = minindex / ournumneur;
    col = minindex % ournumneur;
minindex=row*numneurx+col*numnodes+startpoint;

while((*iflags & 0xf)==0x4) /* Await current winner */
;
lowest = *recdbl;
while((*iflags & 0xf)==0x4) /* Compare to local winner */
;
lowid = *recint;

if(mindist<lowest) /* Send our own or */
    *sendint = minindex;
else /* winner from previous module */
    *sendint = lowid;

for(i=0; i<numiter-numnodes;i++) /* Normal operation */
{
    curridx = (i % numnodes)*dimension;

    while((*iflags & 0xf)==0x4) /* Await previous winner */
    ;
    lastwinner = *recint; /* Do not need to retransmit */

    /* Compute size of neighborhood */
    neighsize=(int)(-((double)maxxy/(double)numiter)
    *(double)time+maxxy);

    /* Adapt learning rate */
    alpha=ALPHA*(1.0-(double)((double)time/((double)numiter)));

    /* Compute global winners row and column */
    row=lastwinner/numneurx;
    col=lastwinner%numneurx;

    /* Check if neighborhood exceeds map */
    startx=MAXIM(col-neighsize,0);
    stopx =MINIM(col+neighsize,numneurx-1);

    starty=MAXIM(row-neighsize,0);
    stopy =MINIM(row+neighsize,numneury-1);

    /* Determine which neurons in each row which */
    /* will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes<startpoint))
    stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(weight,&vector[curridx],m+startupd,
    m+stopupd,dimension,alpha);
time++;

while(!(*iflags & 0x2)) /* Await new training vector */
    ;

for(j=0;j<dimension;j++)
    vector[curridx+j]=*recdbl; /* Do not need to retransmit */

/* Compute local winner */
minindex=computebest(weight,&vector[curridx],&mindist,
ournumneur*numneury,dimension);

/* Compute local winners global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex= row*numneurx+col*numnodes+startpoint;

while((*iflags & 0xf)==0x4) /* Await previous winner */
    ;
lowest = *recdbl; /* Value and */
while((*iflags & 0xf)==0x4)
    ;
lowid = *recint; /* identification */

if(mindist<lowest) /* Compare */
    *sendint = minindex; /* Send index of the smallest */
else
    *sendint = lowid;
}
for(i=0;i<numnodes;i++) /* Empty pipe (only weight update) */
{
    lastwinner = *recint; /* Do not need to retransmit */
/* Compute size of neighborhood */
neighsize=(int)((double)maxxy/(double)numiter)*((double)time+maxxy);

/* Adapt learning rate */
alpha=ALPHA*(1.0-((double)((double)time/((double)numiter))));

/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;

/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx=MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy=MINIM(row+neighsize,numneury-1);

/* Determine which neurons in each row which will be updated */
startupd=startx/numnodes;
stopupd=stopx/numnodes;
if((startx%numnodes)>startpoint)
    startupd++;
if((stopx%numnodes)<startpoint)
    stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
    updatenetwork(weight,&vector[curridx],m+startupd,
                   m+stopupd,dimension,alpha);
    time++;

printf("Last node (node %d) finished\n",node);
}
B.5  The TP-Ring implementation

/*
   This program is a training set parallel version of the
   Self-organizing Maps. The program uses the vector
   configuration. All the modules are connected in one ring.
*/

#define ALPHA 0.05
#define MINIM(x,y) (((x) < (y)) ? (x) : (y))
#define MAXIM(x,y) (((x) > (y)) ? (x) : (y))
#define EFFPACKETSIZE 1022
/* Effective packet size is size of fifo - 2 because of address and token */

#include <stdio.h>
#include <math.h>
#include "com_addr.h"

struct sc
{
   int cmd;
   int adr;
};

void init_memory_pointers();   /* Memory allocation routines */
double *mem_alloc(int, unsigned *); /* for RENNS */

int getint();

int computebest(double *, double *, double *, int, int);
/* SOM routines */
void updatenetwork(double *, double *, int, int, int, double);
void updateclrdelta(double *, double *, int, int);
void computechange(double *, double *, double *, int, int, int, double);

/* Configuration values for comsys */
int fregs[4] = {1, 3, 1, 3};  /* F1->S1, F2->S3, S1->F3, S3->F4 */
int aregs[4] = {1, 1, 1, 1};

main()
{

/* RENNS specific memory pointers */
volatile unsigned *id = (unsigned *)0x804400;
volatile unsigned *oflags = (unsigned *)0x805608;
volatile unsigned *iflags = (unsigned *)0x805609;
volatile unsigned *sfpdpa = (unsigned *)0x801100;
volatile unsigned *fpdpa = (unsigned *)0x801000;
volatile double *fpdpa = (double *)0x801000;
volatile unsigned long *tctrl = (unsigned long *)0x808020;
volatile unsigned long *tcount = (unsigned long *)0x808024;
volatile unsigned long *tperiod = (unsigned long *)0x808028;
volatile struct sc *strl = STREAM1;

/* local variables*/
int node,numnodes,error,numneurx,numneury,numneur,startpoint;
int dimension,minindex,i,j,k,l,m,index,neigh,startx,stopx,starty,stopy;
int neighsize,iterupd,adr,cmd,numbatch,bytestosend,packindex,row,col,maxxy;
double alpha, startalpha;

double *somnet,*vector,*delta;
double mindist,tempdist,dist;
unsigned long masternode,time,numiter;
unsigned memtype;

/* Timing variables */
unsigned c0,c1,c2,c3,c4,c5;
double secs,kcups;
int cup;

tperiod = 0xffffffff;

node = *id & 0xff; /* Read this modules identification */
xlat7seg(node);

/* Write it to the 7seg display */
printf("Welcome to RENNS, a REconfigurable Neural Network Server\n\n");
printf("This is the TP-Ring version of the Self Organizing Maps,\n");
printf("Please report problem parameters\n");
printf("Number of modules : ");
umnodes = getint();
printf("Number of neurons in the x-direction : ");
umneurx = getint();
printf("Number of neurons in the y-direction : ");
umneury = getint();
printf("Total number of iterations : ");
numiter = getint();
printf("Number of iterations per update: ");
iterupd = getint();
printf("Dimensionality: ");
dimension = getint();
umnneurx = numnneury;
maxxy = MAXIM(numnneurx,numnneury);

numbatch = numiter / iterupd;
aregs[0] = aregs[2] = dimension / 2;
aregs[1] = aregs[3] = 2; /* Set FIFO flags */
asm(" OR 800h,ST"); /* Enable cache */

/* Initialize communication subsystem */
error = init_comsubsys(fregs,aregs);
if(error)
{
    printf("Error in init_comsubsys: \%d\n",error);
    exit(0);
}

if(node==1) /* Master is responsible for inserting token */
{
    if(numnodes<4) /* If 3 nodes, then a dummy controller on master is used */
        init_vector_stream(1,7,0x3,1);
    cmd = 0xb;
    adr = 1;
    for(i=0;i<100000;i++);
}
else /* If only 2 nodes, then a dummy controller on slave is used as well */
{
    if(numnodes<3)
        init_vector_stream(1,7,0x3,1);
    cmd = 0x3;
    adr = 1;
}

init_vector_stream(0,adr,cmd,1); /* Start communication subsystem */
printf("Communication initialized. Press any key to start\n");

cgetchar();
B. Source code

printf("\n\n");

init_memory_pointers();            /* Initialize memory allocation */

/* Allocate space for training vector */
vector = (double *) mem_alloc(dimension,&memtype);
if(vector==0)
    {
        printf("Could'nt allocate vector
");
        exit(0);
    }

/* Allocate space for weights */
somnet = (double *) mem_alloc(numneur*dimension,&memtype);
if(somnet==0)
    {
        printf("Not enough memory for this network
");
        exit(0);
    }

/* Allocate space for weight change matrix */
delta = (double *) mem_alloc(numneur*dimension,&memtype);
if(delta==0)
    {
        printf("Not enough memory to allocate delta
");
    }

srand(0);                        /* Fixed seed */

/* Initialize weights and weight change matrix */
for(i=0;i<numneur*dimension;i++)
    {
        somnet[i]=((rand() % 1000000) + 1) / 1000000.0)-0.5;
        delta[i]=0;
    }

*tcntcr = 0x2c0;                  /* Start timer */

time=0;
cup=0;
c0 = *tcntcr;
for(i=0;i<numbatch;i++)
{
    for(j=0;j<iterupd;j++) /* Loop between weight exchange */
    {
        /* generate new local trainingvector */
        for(k=0;k<dimension;k++)
            vector[k]=(((rand()%1000000)+1)/1000000.0);
        /* Compute winner */
        minindex=computebest(somnet,vector,&mindist,numneur,dimension);
        row=minindex/numneurx;
        col=minindex%numneury;
        /* Compute neighborhood */
        neighsize=(int)(((float)maxxy/(float)numbatch)
                        *(float)i+maxxy);
        /* Adapt learning rate */
        alpha=ALPHA*(1.0-((float)(int)/(float)numbatch));
        /* Check if neighborhood exceeds map */
        startx=MAXIM(col-neighsize,0);
        stopx =MINIM(col+neighsize,numneurx-1);
        starty=MAXIM(row-neighsize,0);
        stopy =MINIM(row+neighsize,numneury-1);
        /* Compute the number of weights updated */
        cup+=(stopx-startx+1)*(stopy-starty+1);
        /* Compute the weight exchange and place it in weight change matrix*/
        for(m=starty*numneurx;m<stopy*numneurx;m+=numneurx)
            computechange(somnet,vector,delta,startx,stopx,dimension,alpha);
        time++;
    }
    /* Exchange weights */
    packindex=0; /* Index into weight change matrix */
    for(j=0;j<=((numneur*dimension)/EFFPACKETSIZE);j++) /* Full packets */
    {
        ...
while(!(oflags & 0xf)==0x4)) /* Check if space in FIFO */
;
*sfpdpa=0x61000000;
for(k=0;k<EFFPACKETSIZE;k++)
    *fpfdpa=delta[packindex+k]; /* Send our part of the matrix */
    *sfpdpa=TOKEN;

for(k=1;k<numnodes;k++) /* Receive parts from others */
for(l=0;l<EFFPACKETSIZE;l++)
{
    while((iflags & 0xf)==0x4)
;
    delta[packindex+l]+=fpfdpa; /* And accumulate all changes */
}

packindex+=EFFPACKETSIZE; /* Update index */

/* Send the rest of the matrix */
while(!(oflags & 0xf)==0x4) /* Check for space in FIFO */
;
*sfpdpa=0x61000000;

for(k=0;k<((numneur*dimension)%EFFPACKETSIZE);k++) /* Send our part */
    *fpfdpa=delta[packindex+k];
*sfpdpa=TOKEN;

for(j=1;j<numnodes;j++) /* Receive from the other modules */
{
    for(k=0;k<((numneur*dimension)%EFFPACKETSIZE);k++)
    {
        while((iflags & 0xf)==0x4)
    ;
        delta[packindex+k]+=fpfdpa; /* Accumulate changes */
    }
/* Add changes to weight matrix and clear weight change matrix */
updateclrdelta(somnet,delta,dimension,numneur);
}
c5 = *tcount;
cup* = dimension;
*sfpdpa=0x61000000;  /* Need to know how much work done by other modules */
*fpidpa=cup;
*sfpdpa=TOKEN;
for(i=1;i<numnodes;i++)
{
    while((*iflags & 0xf)==0x4)  /* Receive amount of updates from others */
    {
        j=*fpidpa;
        cup+=j;
    }
}
secs = (double)(c5-c0)/8000000.0;
kcups = (cup/secs) / 1000.0;  /* Compute performance */
printf("\nPerformance : %0.3f KCups/Sec",kcups);
B.6 The NP/TP implementation

/*
* This program is a combined training example parallel and neuron parallel version of the Self-organizing Map model. The modules are connected in a 2D-torus, and uses the vector configuration for communication.
*/

#define PACKETSIZE 1022
#define ALPHA 0.05
#define MINIM(x,y)(((x) < (y)) ? (x) : (y))
#define MAXIM(x,y)(((x) > (y)) ? (x) : (y))

#include <stdio.h>
#include <math.h>
#include "com_addr.h"

void init_memory_pointers(); /* RENNS memory routines */
double *mem_alloc(int,unsigned *);

int getint();

int computebest(double *,double *,double *,int,int);
/* SOM routines */
void updatenetwork(double *,double *,int,int,double);
void updatedeltadelta(double *,double *,int,int);
void computechange(double *,double *,double *,int,int,int,double);

/* Configuration values for comsys */
int fregs[4] = {1,3,1,3}; /* F1->S1, F2->S3, S1->F3, S3->F4 */
int aregs[4];

main()
{
/* RENNS specific memory pointers */
volatile unsigned *id = (unsigned *)0x804400;
volatile unsigned *oflags = (unsigned *)0x805608;
volatile unsigned *iflags = (unsigned *)0x805609;
volatile unsigned *sfpnpa = (unsigned *)0x801100;
volatile unsigned *fpinpa = (unsigned *)0x801000;
volatile double *fpmpa = (double *)0x801000;
volatile unsigned *sfpdpa = (unsigned *)0x801300;
volatile unsigned *fpidpa = (unsigned *)0x801200;
volatile double *fpfdpa = (double *)0x801200;
volatile unsigned long *tctrl = (unsigned long *)0x808020;
volatile unsigned long *tcount = (unsigned long *)0x808024;
volatile unsigned long *tperiod = (unsigned long *)0x808028;

/*local variables*/
int node,numnets,numnodes,error,numneur,startpoint,ournumneur;
int dimension,minindex,i,j,k,index,neigh,startx,stopx,starty,stopy;
int numneurx,numneury,row,col,maxxy,startudp,stoppudp;
int neighsize,iterudp,adr,cmd,numbatch,sendindex,numsendsend;
double alpha,startalpha;
unsigned memtype;

double *somnet,*vector,*delta;
double mindist,tempdist,dist;
unsigned long masternode,time,numiter;

/* Timing variables */
unsigned c0,c1,c2,c3,c4,c5;
double secs,kcups;
int cup;

*tperiod = 0xffffffff;

node = *id & 0xff;
xsat7seg(node & 0xf);

/* Read node identification */
printf("Welcome to RENN, a REconfigurable Neural Network Server\n\n");
printf("This is the NP/TP version of the Self Organizing Maps,\n");
printf("Please report problem parameters\n");
printf("Number of copies of the network : ");
umnets = getint();
printf("Number of RENN nodes per copy : ");
umnodes = getint();
printf("Number of neurons in the x direction : ");
umneurx = getint();
printf("Number of neurons in the y direction : ");
umneury = getint();
printf("Total number of iterations : ");
numiter = getint();
printf("Number of iterations per update : ");
iterupd = getint();
printf("Dimensionality : ");
dimension = getint();
numneur=numneurx*numneury;
maxxy=MAXIM(numneurx,numneury);
printf("Starting with %d x %d modules, %d x %d neurons, %d iterations, 
     \text{dim} = \%d\n",numnets,numnodes,numneurx,numneury,numiter,dimension);

numbatch = numiter / iterupd; /* Determine the number of batches */

/* Determine our neurons in the x direction */
startpoint = ((node&0xf) % numnodes);
ournumneur = (numneurx/numnodes);
if((numneurx-ournumneur*numnodes)>startpoint) ournumneur++;

aregs[0]=aregs[2]= dimension / 2;  /* Set FIFO flags */

asm(" OR 800h,ST");  /* Enable cache */

/* Initialize communication subsystem */
error=init_comsubsys(fregs,aregs);
if(error)
{
    printf("Error in init_comsubsys: \%d\n",error);
    exit(0);
}

if((node & 0xf)==1) /* Master within a cluster inserts token on local ring */
{
    if(numnodes<4) /* If few modules, we need dummy controllers */
        init_vector_stream(1,7,0x3,1);
    cmd = 0xb;
    adr = 2;
    for(i=0;i<100000;i++);
}
else
{
    if(numnodes<3) /* If only 2 modules per cluster we need 2 dummies */
        init_vector_stream(1,7,0x3,1);
    cmd = 0x3;
adr = 1;
}

/* Start intracluster communication */
init_vector_stream(0,adr,cmd,1);

if(node & 0xf0) /* Need to initialize intercluster communication as well */
{
    if(numnets<4) /* Insert dummy controller */
        init_vector_stream(3,7,0x3,1);
    cmd = 0xb;
    for(i=0;i<1000000;i++);
}

else
{
    if(numnets<3) /* Insert another dummy controller */
        init_vector_stream(3,7,0x3,1);
    cmd = 0x3;
}

/* Start intercluster communication */
init_vector_stream(2,1,cmd,2);

printf("Communication initialized. Press any key to start\n");
getchar();
printf("\n\n");

init_memory_pointers(); /* Initialize memory allocation */
time=0;
cup=0;

/* Allocate space for training vector */
vector = (double *) mem_alloc(dimension,&memtype);
if(vector==0)
{
    printf("Couldn't allocate vector\n");
    exit(0);
}

/* Allocate space for weight matrix */
somnet = (double *) mem_alloc(ournumneur*dimension*numneury,&memtype);
if(somnet==0)

{
    printf("Not enough memory for this network\n");
    printf("Tried to allocate %d * %d * %d * %d\n",
            ournumneur,dimension,numneury,sizeof(float));
    exit(0);
}

/* Allocate space for weight change matrix */
delta = (double *) mem_alloc(ournumneur*dimension*numneury,&memtype);
if(delta==0)
{
    printf("Not enough memory to allocate delta\n");
}

srand(node&0xf); /* Fixed seed */

/* Initialize weights and weight change matrix */
for(i=0;i<ournumneur*dimension;i++)
{
    somnet[i]=((rand() % 1000000) + 1) / 1000000.0)-0.5;
    delta[i]=0;
}

if((node&0xf)==1) /* masters within each cluster */
{
    *tctrl = 0x2c0;
    /* Start timer */
    c0 = *tcount;

    for(i=0;i<numbatch;i++)
    {
        for(j=0;j<iterupd;j++) /* Loop between weight exchanges */
        {
            /* generate new training vector */
            for(k=0;k<dimension;k++)
                vector[k]=((rand() % 1000000) + 1) / 1000000.0;

            while(!(*oflags & 0x4)) /* Check if space in FIFO */
            {

            }

            *sfnpna=0x61000000;
            /* Address of slaves */
            for(k=0;k<dimension;k++)
{  
    *fpfnpa=vector[k];
}

*spfnpa=TOKEN;                                      /* Reinsert token */

/* compute our own neurons */
minindex=computebest(somnet,vector,&mindist,ournumneur*numneury,
            dimension);

/* compute local winners global index */
row=minindex/ournumneur;
col=minindex%ournumneur;
minindex=row*numneurx+col*numnodes+startpoint;

/* Await winners from slaves */
for(k=0;k<numnodes-1;k++)
{
    while(*isflags & 0xf)==0x4)    /* Check for contents in FIFO */
        ;
    dist=*fpfnpa;
    while(*isflags & 0xf)==0x4)
    {
        index=*fpinpa;
        /* and index */

        if(dist<mindist)           /* Compare to current winner */
        {
            mindist=dist;
            minindex=index;
            /* Replace */
        }
    }

/* Broadcast index of winner */
*spfnpa=0x61000000;                                 /* Address of slaves */
*fpinpa=minindex;                                     /* Send index */
*spfnpa=TOKEN;                                       /* Reinsert token */

/* Compute neighborhood */
neighsize=(int)-((float)maxxy/(float)numbatch)
           *(float)i+maxxy);

/* Adapt learning rate */
alpha=ALPHA*(1.0-(float)((float)i/((float)numbatch)));
/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;

/*/ Check if neighborhood exceeds map */
startx=MAXIM(col+neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row+neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/*/ Compute number of neurons updated */
cup+=(stopx-startx+1)*(stopy-starty+1);

/*/ Determine which neurons in each row which
will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
  startupd++;
if((stopx%numnodes)<startpoint)
  stopupd--;

/*/ Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
  computechange(somnet, vector, delta, m+startupd,
               m+stopupd,dimension,alpha);

  time++;
}

/*/ Exchange weight matrix */

sendindex=0;                        /* Index into weight matrix */
/*/ First send full packets (partitioned exchange */
for(numsend=0;numsend<((ournumneur*numneury*dimension)/PACKETSIZE);numset
    {
      while(!((oflags & 0xf0)==0x40))     /* Check if space in FIFO */
        ;
      *spdpa=0x61000000;                      /* Common address for all */
      for(j=0;j<PACKETSIZE;j++)
        {

*fpfdpa=delta[sendindex+j];
}
*sfpdpa=TOKEN; /* Reinsert token */

for(j=1;j<numnets;j++) /* Receive from other clusters */
{
    for(k=0;k<PACKETSIZE;k++)
    {
        while( (*iflags & 0xf0)==0x40) /* Check if data in FIFO */
        {
            delta[sendindex+k]=*fpfdpa; /* Accumulate changes */
        }
    }
    sendindex+=PACKETSIZE; /* Update index */
}

while( !( (oflags & 0xf0)==0x40) ) /* Check if space in FIFO */
{
    *sfpdpa=0x61000000;
    for(j=0;j<((ournumneur*dimension)%PACKETSIZE);j++) /* Rest of matrix*/
    {
        *fpfdpa=delta[sendindex+j];
    }
    *sfpdpa=TOKEN; /* Reinsert token */
}

for(j=1;j<numnets;j++) /* Receive rest from other clusters */
{
    for(k=0;k<((ournumneur*dimension)%PACKETSIZE);k++)
    {
        while( (*iflags & 0xf0)==0x40)
        {
            delta[sendindex+k]=*fpfdpa; /* Accumulate weight changes */
        }
    }
}

/* Update weights and reset weight change matrix */
updatedcrdelta(somnet,delta,dimension,numneur);
}
c5 = *tcount;
cup* = dimension; /* Compute number of weights updated */
*sfpdpa=0x61000000; /* Exchange with other clusters */
*fpidpa=cup;
*sfpdpa=TOKEN;
for(i=1;i<numnets;i++)
{
    while((*iflags & 0xf0)==0x40)
    {
        j=*fpidpa;
        printf("Receiving %d",j);
        cup+=j;
    }
    secs = (double)(c5-c0)/8000000.0;
    k cups = (cup/sec) / 1000.0; /* Compute performance */
    printf("\nPerformance : %0.3f KCups/Sec",kcups);
}
else /* slave */
{
    *tcrl = 0x2c0;
    c0 = *tcnt;
    /* Start timer */
    for(i=0;i<numbatch;i++)
    {
        for(j=0;j<iterupd;j++) /* Loop between weight exchanges */
        {
            while(!(*iflags & 0x2)) /* wait for training vector */
            {
                for(k=0;k<dimension;k++)
                {vector[k]=*fphna;
                 /* compute our own neurons */
                 minindex=computebest(somnet,vector,&mindist,
                                       ournumneur*numneur, dimension);

                 /* compute local winners global index */
                 row=minindex/ournumneur;
                 col=minindex%ournumneur;
                 minindex=row*numneurx+col*numnodes+startpoint;

                while(!(*oflags & 0x4)) /* Check if space in FIFO */
                {
                    *fphna=0x62000000;
                    *fphna=mindist; /* Send value and index of */
                    *fphna=mindindex; /* local winner */
                    *fphna=0x62000000; /* Masters address */
                }
            }
        }
    }
}
/* compute size of neighborhood while waiting for answer */
neighsize=(int)((float)maxxy/(float)numbatch)
 *(float)i+maxxy);

/* Adapt learning rate */
alpha=ALPHA*(1.0-(float)((float)i/((float)numbatch)));

while((*iflags & 0xf) == 0x4) /* Is global winner ready ?*/
  ;
minindex=*fpinpa; /* Got it ! */

/* Compute global winners row and column */
row=lastwinner/numneurx;
col=lastwinner%numneurx;

/* Check if neighborhood exceeds map */
startx=MAXIM(col-neighsize,0);
stopx =MINIM(col+neighsize,numneurx-1);

starty=MAXIM(row-neighsize,0);
stopy =MINIM(row+neighsize,numneury-1);

/* Determine which neurons in each row which will be updated */
startupd = startx / numnodes;
stopupd = stopx / numnodes;
if((startx%numnodes)>startpoint)
  startupd++;  
if((stopx%numnodes<startpoint))
  stopupd--;

/* Update weights */
for(m=starty*ournumneur;m<stopy*ournumneur;m+=ournumneur)
  computechange(somnet,vector,delta,m+startupd,
    m+stopupd,dimension,alpha);

  time++;  
}

/* Weight exchange */
sendindex=0; /* Index into weight exchange matrix */
/* First send the full packets */
for(numsend=0;numsend<((ournumneur*dimension)/PACKETSIZE);numsend++)
{
    while(!(*(oflags & 0xf0)==0x40)) /* Check if space in FIFO */
    {
        *sfpdpa=0x61000000;
        /* Common address */
        for(j=0;j<PACKETSIZE;j++)
        {
            *fpfdpa=delta[sendindex+j];
        }
        *sfpdpa=TOKEN; /* Reinsert token */
    }
    for(j=1;j<numnets;j++) /* Receive parts from other clusters */
    {
        for(k=0;k<PACKETSIZE;k++)
        {
            while(!(*(oflags & 0xf0)==0x40)) /* Data in FIFO ? */
            {
                delta[sendindex+k]+=*fpfdpa; /* Accumulate */
            }
            sendindex+=PACKETSIZE; /* Update index */
        }
    }

    /* Send and receive last part */
    while(!(*(oflags & 0xf0)==0x40)) /* Space in FIFO ? */
    {
        *sfpdpa=0x61000000;
        for(j=0;j<((ournumneur*dimension)%PACKETSIZE);j++)
        {
            *fpfdpa=delta[sendindex+j];
        }
        *sfpdpa=TOKEN;
    }
    for(j=1;j<numnets;j++) /* Receive from others */
    {
        for(k=0;k<((ournumneur*dimension)%PACKETSIZE);k++)
        {
            while(!(*(oflags & 0xf0)==0x40)) /* Data in FIFO */
            {
                delta[sendindex+k]+=*fpfdpa; /* Accumulate */
            }
        }
    }
} /* Update and clear weight change matrix */
updatelnrdelta(somnet,delta,dimension,numneur);
}
}

This appendix contains tables of the measured values presented in chapter 7, as it can be difficult to read these values directly from the figures.

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>100 neurons</th>
<th>500 neurons</th>
<th>1000 neurons</th>
<th>5000 neurons</th>
<th>10000 neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
<td>1.0</td>
<td>1.0</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>1.8</td>
<td>1.9</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>1.9</td>
<td>2.6</td>
<td>2.8</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>2.3</td>
<td>3.4</td>
<td>3.7</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>4.1</td>
<td>4.5</td>
<td>4.9</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>2.7</td>
<td>4.8</td>
<td>5.3</td>
<td>5.9</td>
<td>6.0</td>
</tr>
<tr>
<td>7</td>
<td>2.9</td>
<td>5.8</td>
<td>6.1</td>
<td>6.8</td>
<td>6.9</td>
</tr>
<tr>
<td>8</td>
<td>3.0</td>
<td>6.4</td>
<td>6.8</td>
<td>7.8</td>
<td>7.9</td>
</tr>
<tr>
<td>9</td>
<td>3.2</td>
<td>6.9</td>
<td>7.5</td>
<td>8.7</td>
<td>8.9</td>
</tr>
<tr>
<td>10</td>
<td>3.2</td>
<td>7.4</td>
<td>8.2</td>
<td>9.6</td>
<td>9.9</td>
</tr>
<tr>
<td>11</td>
<td>3.3</td>
<td>7.8</td>
<td>8.8</td>
<td>10.6</td>
<td>10.8</td>
</tr>
<tr>
<td>12</td>
<td>3.3</td>
<td>8.3</td>
<td>9.4</td>
<td>11.4</td>
<td>11.8</td>
</tr>
<tr>
<td>13</td>
<td>3.4</td>
<td>8.7</td>
<td>10.9</td>
<td>12.3</td>
<td>12.7</td>
</tr>
<tr>
<td>14</td>
<td>3.5</td>
<td>9.0</td>
<td>11.4</td>
<td>13.2</td>
<td>13.7</td>
</tr>
<tr>
<td>15</td>
<td>3.5</td>
<td>9.4</td>
<td>12.0</td>
<td>14.1</td>
<td>14.6</td>
</tr>
</tbody>
</table>

Table C.1: NP-Ring performance, MCUPS
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>100 neurons 8 MHz</th>
<th>100 neurons 10 MHz</th>
<th>10000 neurons 8 MHz</th>
<th>10000 neurons 10 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
<td>0.8</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>1.5</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>1.9</td>
<td>1.9</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>2.2</td>
<td>2.3</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>2.5</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>2.7</td>
<td>2.7</td>
<td>6.0</td>
<td>5.9</td>
</tr>
<tr>
<td>7</td>
<td>2.8</td>
<td>2.9</td>
<td>6.9</td>
<td>6.9</td>
</tr>
<tr>
<td>8</td>
<td>3.0</td>
<td>3.0</td>
<td>7.9</td>
<td>7.9</td>
</tr>
<tr>
<td>9</td>
<td>3.1</td>
<td>3.1</td>
<td>8.9</td>
<td>8.9</td>
</tr>
<tr>
<td>10</td>
<td>3.2</td>
<td>3.2</td>
<td>9.9</td>
<td>9.9</td>
</tr>
<tr>
<td>11</td>
<td>3.3</td>
<td>3.3</td>
<td>10.8</td>
<td>10.8</td>
</tr>
<tr>
<td>12</td>
<td>3.3</td>
<td>3.4</td>
<td>11.8</td>
<td>11.8</td>
</tr>
<tr>
<td>13</td>
<td>3.4</td>
<td>3.5</td>
<td>12.7</td>
<td>12.7</td>
</tr>
<tr>
<td>14</td>
<td>3.4</td>
<td>3.5</td>
<td>13.7</td>
<td>13.7</td>
</tr>
<tr>
<td>15</td>
<td>3.5</td>
<td>3.6</td>
<td>14.6</td>
<td>14.6</td>
</tr>
</tbody>
</table>

Table C.2: NP-Ring performance on 8 MHz and 10 MHz, MCUPS

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>100 neurons</th>
<th>500 neurons</th>
<th>1000 neurons</th>
<th>5000 neurons</th>
<th>10000 neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
<td>1.0</td>
<td>1.0</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>1.9</td>
<td>2.7</td>
<td>2.8</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>2.3</td>
<td>3.4</td>
<td>3.7</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>4.1</td>
<td>4.5</td>
<td>4.9</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>2.7</td>
<td>4.7</td>
<td>5.3</td>
<td>5.9</td>
<td>5.9</td>
</tr>
<tr>
<td>7</td>
<td>2.9</td>
<td>5.8</td>
<td>6.1</td>
<td>6.9</td>
<td>6.9</td>
</tr>
<tr>
<td>8</td>
<td>3.0</td>
<td>6.3</td>
<td>6.8</td>
<td>7.8</td>
<td>7.9</td>
</tr>
<tr>
<td>9</td>
<td>3.1</td>
<td>6.9</td>
<td>7.5</td>
<td>8.7</td>
<td>8.9</td>
</tr>
<tr>
<td>10</td>
<td>3.2</td>
<td>7.4</td>
<td>8.2</td>
<td>9.6</td>
<td>9.9</td>
</tr>
<tr>
<td>11</td>
<td>3.3</td>
<td>7.8</td>
<td>8.8</td>
<td>10.5</td>
<td>10.8</td>
</tr>
<tr>
<td>12</td>
<td>3.4</td>
<td>8.3</td>
<td>9.4</td>
<td>11.5</td>
<td>11.8</td>
</tr>
<tr>
<td>13</td>
<td>3.5</td>
<td>8.7</td>
<td>10.8</td>
<td>12.3</td>
<td>12.7</td>
</tr>
<tr>
<td>14</td>
<td>3.5</td>
<td>9.2</td>
<td>11.5</td>
<td>13.2</td>
<td>13.7</td>
</tr>
<tr>
<td>15</td>
<td>3.6</td>
<td>9.5</td>
<td>12.2</td>
<td>14.1</td>
<td>14.6</td>
</tr>
</tbody>
</table>

Table C.3: NP-Tree performance, MCUPS
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>NP-Ring 8 MHz</th>
<th>NP-Tree 8 MHz</th>
<th>NP-Ring 10 MHz</th>
<th>NP-Tree 10 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>363</td>
<td>359</td>
<td>287</td>
<td>285</td>
</tr>
<tr>
<td>3</td>
<td>409</td>
<td>360</td>
<td>324</td>
<td>296</td>
</tr>
<tr>
<td>4</td>
<td>412</td>
<td>393</td>
<td>325</td>
<td>341</td>
</tr>
<tr>
<td>5</td>
<td>438</td>
<td>418</td>
<td>346</td>
<td>341</td>
</tr>
<tr>
<td>6</td>
<td>430</td>
<td>411</td>
<td>334</td>
<td>331</td>
</tr>
<tr>
<td>7</td>
<td>444</td>
<td>421</td>
<td>350</td>
<td>336</td>
</tr>
<tr>
<td>8</td>
<td>509</td>
<td>472</td>
<td>399</td>
<td>380</td>
</tr>
<tr>
<td>9</td>
<td>531</td>
<td>502</td>
<td>418</td>
<td>413</td>
</tr>
<tr>
<td>10</td>
<td>563</td>
<td>499</td>
<td>446</td>
<td>403</td>
</tr>
<tr>
<td>11</td>
<td>589</td>
<td>508</td>
<td>451</td>
<td>428</td>
</tr>
<tr>
<td>12</td>
<td>592</td>
<td>471</td>
<td>458</td>
<td>399</td>
</tr>
<tr>
<td>13</td>
<td>622</td>
<td>483</td>
<td>474</td>
<td>407</td>
</tr>
<tr>
<td>14</td>
<td>641</td>
<td>480</td>
<td>486</td>
<td>397</td>
</tr>
<tr>
<td>15</td>
<td>671</td>
<td>498</td>
<td>522</td>
<td>407</td>
</tr>
</tbody>
</table>

**Table C.4: Communication times, Clocks**

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>64 neurons</th>
<th>256 neurons</th>
<th>1000 neurons</th>
<th>10000 neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.84</td>
<td>0.94</td>
<td>1.00</td>
<td>0.71</td>
</tr>
<tr>
<td>2</td>
<td>1.33</td>
<td>1.75</td>
<td>1.95</td>
<td>1.42</td>
</tr>
<tr>
<td>3</td>
<td>1.68</td>
<td>2.47</td>
<td>2.88</td>
<td>3.04</td>
</tr>
<tr>
<td>4</td>
<td>1.91</td>
<td>3.36</td>
<td>3.77</td>
<td>4.04</td>
</tr>
<tr>
<td>5</td>
<td>2.07</td>
<td>3.95</td>
<td>4.62</td>
<td>5.03</td>
</tr>
<tr>
<td>6</td>
<td>2.21</td>
<td>4.43</td>
<td>5.45</td>
<td>6.04</td>
</tr>
<tr>
<td>7</td>
<td>2.34</td>
<td>4.90</td>
<td>6.25</td>
<td>7.01</td>
</tr>
<tr>
<td>8</td>
<td>2.43</td>
<td>5.35</td>
<td>7.02</td>
<td>8.00</td>
</tr>
<tr>
<td>9</td>
<td>2.51</td>
<td>5.70</td>
<td>7.75</td>
<td>8.98</td>
</tr>
<tr>
<td>10</td>
<td>2.53</td>
<td>6.03</td>
<td>8.47</td>
<td>9.98</td>
</tr>
<tr>
<td>11</td>
<td>2.60</td>
<td>6.33</td>
<td>9.17</td>
<td>10.94</td>
</tr>
<tr>
<td>12</td>
<td>2.62</td>
<td>6.61</td>
<td>9.78</td>
<td>11.89</td>
</tr>
<tr>
<td>13</td>
<td>2.68</td>
<td>6.91</td>
<td>10.46</td>
<td>12.84</td>
</tr>
<tr>
<td>14</td>
<td>2.70</td>
<td>7.08</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table C.5: NP-Wave performance, MCUPS**
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>64 neurons NP-Ring</th>
<th>64 neurons NP-Wave</th>
<th>256 neurons NP-Ring</th>
<th>256 neurons NP-Wave</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.44</td>
<td>0.44</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>2</td>
<td>0.67</td>
<td>0.72</td>
<td>0.97</td>
<td>1.01</td>
</tr>
<tr>
<td>3</td>
<td>0.82</td>
<td>0.91</td>
<td>1.36</td>
<td>1.42</td>
</tr>
<tr>
<td>4</td>
<td>0.91</td>
<td>1.02</td>
<td>1.66</td>
<td>1.77</td>
</tr>
<tr>
<td>5</td>
<td>0.97</td>
<td>1.12</td>
<td>1.92</td>
<td>2.08</td>
</tr>
<tr>
<td>6</td>
<td>1.02</td>
<td>1.21</td>
<td>2.13</td>
<td>2.35</td>
</tr>
<tr>
<td>7</td>
<td>1.06</td>
<td>1.28</td>
<td>2.30</td>
<td>2.57</td>
</tr>
<tr>
<td>8</td>
<td>1.08</td>
<td>1.33</td>
<td>2.51</td>
<td>2.85</td>
</tr>
<tr>
<td>9</td>
<td>1.09</td>
<td>1.38</td>
<td>2.60</td>
<td>3.02</td>
</tr>
<tr>
<td>10</td>
<td>1.08</td>
<td>1.38</td>
<td>2.69</td>
<td>3.19</td>
</tr>
<tr>
<td>11</td>
<td>1.09</td>
<td>1.42</td>
<td>2.81</td>
<td>3.37</td>
</tr>
<tr>
<td>12</td>
<td>1.08</td>
<td>1.44</td>
<td>2.87</td>
<td>3.51</td>
</tr>
<tr>
<td>13</td>
<td>1.07</td>
<td>1.47</td>
<td>2.92</td>
<td>3.70</td>
</tr>
<tr>
<td>14</td>
<td>1.07</td>
<td>1.50</td>
<td>2.95</td>
<td>3.80</td>
</tr>
</tbody>
</table>

Table C.6: Performance of NP-Ring and NP-Wave, MCUPS ($d = 4$)

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$\epsilon = 10$</th>
<th>$\epsilon = 100$</th>
<th>$\epsilon = 1000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.43</td>
<td>0.45</td>
<td>0.48</td>
</tr>
<tr>
<td>2</td>
<td>0.75</td>
<td>0.88</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>1.05</td>
<td>1.31</td>
<td>1.42</td>
</tr>
<tr>
<td>4</td>
<td>1.28</td>
<td>1.73</td>
<td>1.90</td>
</tr>
<tr>
<td>5</td>
<td>1.53</td>
<td>2.13</td>
<td>2.37</td>
</tr>
<tr>
<td>6</td>
<td>1.74</td>
<td>2.53</td>
<td>2.84</td>
</tr>
<tr>
<td>7</td>
<td>1.88</td>
<td>2.93</td>
<td>3.31</td>
</tr>
<tr>
<td>8</td>
<td>2.09</td>
<td>3.31</td>
<td>3.77</td>
</tr>
<tr>
<td>9</td>
<td>2.17</td>
<td>3.68</td>
<td>4.24</td>
</tr>
<tr>
<td>10</td>
<td>2.25</td>
<td>4.05</td>
<td>4.71</td>
</tr>
<tr>
<td>11</td>
<td>2.32</td>
<td>4.41</td>
<td>5.17</td>
</tr>
<tr>
<td>12</td>
<td>2.39</td>
<td>4.77</td>
<td>5.63</td>
</tr>
<tr>
<td>13</td>
<td>2.45</td>
<td>5.11</td>
<td>6.10</td>
</tr>
<tr>
<td>14</td>
<td>2.51</td>
<td>5.45</td>
<td>6.56</td>
</tr>
<tr>
<td>15</td>
<td>2.56</td>
<td>5.78</td>
<td>7.02</td>
</tr>
</tbody>
</table>

Table C.7: TP-Ring performance, MCUPS ($n = 64, d = 4$)
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$\epsilon = 10$</th>
<th>$\epsilon = 100$</th>
<th>$\epsilon = 1000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.49</td>
<td>0.52</td>
<td>0.55</td>
</tr>
<tr>
<td>2</td>
<td>0.85</td>
<td>1.02</td>
<td>1.10</td>
</tr>
<tr>
<td>3</td>
<td>1.19</td>
<td>1.52</td>
<td>1.65</td>
</tr>
<tr>
<td>4</td>
<td>1.49</td>
<td>2.00</td>
<td>2.20</td>
</tr>
<tr>
<td>5</td>
<td>1.79</td>
<td>2.48</td>
<td>2.74</td>
</tr>
<tr>
<td>6</td>
<td>2.07</td>
<td>2.95</td>
<td>3.29</td>
</tr>
<tr>
<td>7</td>
<td>2.27</td>
<td>3.42</td>
<td>3.83</td>
</tr>
<tr>
<td>8</td>
<td>2.55</td>
<td>3.87</td>
<td>4.38</td>
</tr>
<tr>
<td>9</td>
<td>2.68</td>
<td>4.38</td>
<td>4.92</td>
</tr>
<tr>
<td>10</td>
<td>2.80</td>
<td>4.76</td>
<td>5.46</td>
</tr>
<tr>
<td>11</td>
<td>2.92</td>
<td>5.20</td>
<td>6.00</td>
</tr>
<tr>
<td>12</td>
<td>2.94</td>
<td>5.62</td>
<td>6.54</td>
</tr>
<tr>
<td>13</td>
<td>3.13</td>
<td>6.04</td>
<td>7.08</td>
</tr>
<tr>
<td>14</td>
<td>3.16</td>
<td>6.45</td>
<td>7.62</td>
</tr>
<tr>
<td>15</td>
<td>3.31</td>
<td>6.86</td>
<td>8.16</td>
</tr>
</tbody>
</table>

Table C.8: TP-Ring performance, MCUPS ($n = 256, d = 4$)

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$p_c = 2$</th>
<th>$p_c = 3$</th>
<th>$p_c = 4$</th>
<th>$p_c = 5$</th>
<th>$p_c = 6$</th>
<th>$p_c = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.57</td>
<td>1.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.79</td>
<td>2.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>2.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.90</td>
<td></td>
<td>2.51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.98</td>
<td>2.52</td>
<td>2.82</td>
<td>2.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2.01</td>
<td></td>
<td></td>
<td></td>
<td>2.88</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>2.73</td>
<td>3.24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.9: NP-Ring performance, MCUPS ($n = 64, d = 4, \epsilon = 10$)
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$p_c = 2$</th>
<th>$p_c = 3$</th>
<th>$p_c = 4$</th>
<th>$p_c = 5$</th>
<th>$p_c = 6$</th>
<th>$p_c = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.10: NP-Ring performance, McUPS ($n = 64, d = 4, \epsilon = 1000$)

<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$p_c = 2$</th>
<th>$p_c = 3$</th>
<th>$p_c = 4$</th>
<th>$p_c = 5$</th>
<th>$p_c = 6$</th>
<th>$p_c = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.11: NP-Ring performance, McUPS ($n = 256, d = 4, \epsilon = 10$)
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>$p_c = 2$</th>
<th>$p_c = 3$</th>
<th>$p_c = 4$</th>
<th>$p_c = 5$</th>
<th>$p_c = 6$</th>
<th>$p_c = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.70</td>
<td>2.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.31</td>
<td>3.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>4.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.85</td>
<td></td>
<td>4.85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4.26</td>
<td>4.96</td>
<td>5.38</td>
<td>5.81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.77</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>5.79</td>
<td></td>
<td></td>
<td>6.72</td>
<td></td>
</tr>
</tbody>
</table>

Table C.12: NP-Ring performance, MCUPS ($n = 256, d = 4, \epsilon = 1000$)
<table>
<thead>
<tr>
<th>Iterations</th>
<th>Update after each</th>
<th>Update after 250</th>
<th>Update after 300</th>
<th>Update after 500</th>
<th>Update after 750</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.382</td>
<td>0.414</td>
<td>0.482</td>
<td>0.492</td>
<td>0.463</td>
</tr>
<tr>
<td>2000</td>
<td>0.241</td>
<td>0.328</td>
<td>0.411</td>
<td>0.504</td>
<td>0.563</td>
</tr>
<tr>
<td>3000</td>
<td>0.234</td>
<td>0.248</td>
<td>0.355</td>
<td>0.371</td>
<td>0.578</td>
</tr>
<tr>
<td>4000</td>
<td>0.228</td>
<td>0.224</td>
<td>0.280</td>
<td>0.359</td>
<td>0.566</td>
</tr>
<tr>
<td>5000</td>
<td>0.226</td>
<td>0.238</td>
<td>0.239</td>
<td>0.371</td>
<td>0.569</td>
</tr>
<tr>
<td>6000</td>
<td>0.221</td>
<td>0.198</td>
<td>0.229</td>
<td>0.356</td>
<td>0.563</td>
</tr>
<tr>
<td>7000</td>
<td>0.212</td>
<td>0.189</td>
<td>0.214</td>
<td>0.355</td>
<td>0.559</td>
</tr>
<tr>
<td>8000</td>
<td>0.219</td>
<td>0.219</td>
<td>0.222</td>
<td>0.352</td>
<td>0.564</td>
</tr>
<tr>
<td>9000</td>
<td>0.227</td>
<td>0.240</td>
<td>0.235</td>
<td>0.366</td>
<td>0.579</td>
</tr>
<tr>
<td>10000</td>
<td>0.233</td>
<td>0.241</td>
<td>0.224</td>
<td>0.355</td>
<td>0.569</td>
</tr>
<tr>
<td>11000</td>
<td>0.228</td>
<td>0.235</td>
<td>0.216</td>
<td>0.352</td>
<td>0.572</td>
</tr>
<tr>
<td>12000</td>
<td>0.198</td>
<td>0.201</td>
<td>0.234</td>
<td>0.354</td>
<td>0.565</td>
</tr>
<tr>
<td>13000</td>
<td>0.159</td>
<td>0.164</td>
<td>0.218</td>
<td>0.359</td>
<td>0.577</td>
</tr>
<tr>
<td>14000</td>
<td>0.145</td>
<td>0.145</td>
<td>0.164</td>
<td>0.209</td>
<td>0.568</td>
</tr>
<tr>
<td>15000</td>
<td>0.134</td>
<td>0.135</td>
<td>0.139</td>
<td>0.189</td>
<td>0.550</td>
</tr>
<tr>
<td>16000</td>
<td>0.127</td>
<td>0.127</td>
<td>0.129</td>
<td>0.183</td>
<td>0.566</td>
</tr>
<tr>
<td>17000</td>
<td>0.114</td>
<td>0.115</td>
<td>0.115</td>
<td>0.143</td>
<td>0.545</td>
</tr>
<tr>
<td>18000</td>
<td>0.117</td>
<td>0.117</td>
<td>0.117</td>
<td>0.146</td>
<td>0.561</td>
</tr>
<tr>
<td>19000</td>
<td>0.123</td>
<td>0.123</td>
<td>0.123</td>
<td>0.152</td>
<td>0.582</td>
</tr>
<tr>
<td>20000</td>
<td>0.108</td>
<td>0.108</td>
<td>0.132</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.13: Error as a function of how often weight update is done
<table>
<thead>
<tr>
<th>Iterations</th>
<th>$\alpha(0) = 0.001$</th>
<th>$\alpha(0) = 0.01$</th>
<th>$\alpha(0) = 0.1$</th>
<th>$\alpha(0) = 0.25$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.415</td>
<td>0.382</td>
<td>0.382</td>
<td>0.380</td>
</tr>
<tr>
<td>2000</td>
<td>0.389</td>
<td>0.241</td>
<td>0.249</td>
<td>0.261</td>
</tr>
<tr>
<td>3000</td>
<td>0.387</td>
<td>0.234</td>
<td>0.252</td>
<td>0.267</td>
</tr>
<tr>
<td>4000</td>
<td>0.366</td>
<td>0.228</td>
<td>0.244</td>
<td>0.254</td>
</tr>
<tr>
<td>5000</td>
<td>0.362</td>
<td>0.226</td>
<td>0.238</td>
<td>0.251</td>
</tr>
<tr>
<td>6000</td>
<td>0.354</td>
<td>0.221</td>
<td>0.233</td>
<td>0.249</td>
</tr>
<tr>
<td>7000</td>
<td>0.352</td>
<td>0.212</td>
<td>0.224</td>
<td>0.235</td>
</tr>
<tr>
<td>8000</td>
<td>0.357</td>
<td>0.219</td>
<td>0.232</td>
<td>0.240</td>
</tr>
<tr>
<td>9000</td>
<td>0.370</td>
<td>0.227</td>
<td>0.247</td>
<td>0.258</td>
</tr>
<tr>
<td>10000</td>
<td>0.368</td>
<td>0.233</td>
<td>0.253</td>
<td>0.264</td>
</tr>
<tr>
<td>11000</td>
<td>0.366</td>
<td>0.228</td>
<td>0.211</td>
<td>0.217</td>
</tr>
<tr>
<td>12000</td>
<td>0.359</td>
<td>0.198</td>
<td>0.166</td>
<td>0.166</td>
</tr>
<tr>
<td>13000</td>
<td>0.366</td>
<td>0.159</td>
<td>0.170</td>
<td>0.184</td>
</tr>
<tr>
<td>14000</td>
<td>0.360</td>
<td>0.145</td>
<td>0.158</td>
<td>0.172</td>
</tr>
<tr>
<td>15000</td>
<td>0.347</td>
<td>0.134</td>
<td>0.130</td>
<td>0.148</td>
</tr>
<tr>
<td>16000</td>
<td>0.356</td>
<td>0.127</td>
<td>0.109</td>
<td>0.118</td>
</tr>
<tr>
<td>17000</td>
<td>0.344</td>
<td>0.114</td>
<td>0.093</td>
<td>0.093</td>
</tr>
<tr>
<td>18000</td>
<td>0.351</td>
<td>0.117</td>
<td>0.092</td>
<td>0.090</td>
</tr>
<tr>
<td>19000</td>
<td>0.368</td>
<td>0.123</td>
<td>0.089</td>
<td>0.083</td>
</tr>
<tr>
<td>20000</td>
<td>0.341</td>
<td>0.108</td>
<td>0.076</td>
<td>0.068</td>
</tr>
</tbody>
</table>

Table C.14: Convergence as a function of learning rate
<table>
<thead>
<tr>
<th>Number of modules</th>
<th>NP-Ring</th>
<th>NP/TP (c = 2)</th>
<th>TP-ring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.92</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.89</td>
<td>1.17</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.86</td>
<td>1.79</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.88</td>
<td>2.73</td>
<td>2.24</td>
</tr>
<tr>
<td>5</td>
<td>4.82</td>
<td>2.39</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5.74</td>
<td>2.61</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6.61</td>
<td>2.85</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.65</td>
<td>4.97</td>
<td>2.93</td>
</tr>
<tr>
<td>9</td>
<td>8.61</td>
<td>3.09</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9.37</td>
<td>3.28</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10.22</td>
<td>3.36</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11.13</td>
<td>3.47</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>11.92</td>
<td>3.57</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>13.03</td>
<td>3.66</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14.16</td>
<td>3.71</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14.87</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.15: Performance of NP-Ring, NP/TP and TP-Ring, MCUPS (\(n = 2048, d = 32, \epsilon = 10\))
Bibliography


