DESIGN OF A RECONFIGURABLE NEUROCOMPUTER
PERFORMANCE ANALYSIS BY IMPLEMENTATION OF
RECURRENT ASSOCIATIVE MEMORIES

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
Lisbet Utne
May 1995

THE NORWEGIAN INSTITUTE OF TECHNOLOGY
THE UNIVERSITY OF TRONDHEIM
DEPARTMENT OF COMPUTER SYSTEMS AND TELEMATICS
COMPUTER SYSTEMS GROUP
Abstract

The main objective has been to design a neurocomputer suitable for experimenting with different system architectures for simulation of artificial neural networks. RENNS, REconfigurable Neural Network Server, is a moderately parallel programmable neurocomputer designed at the Norwegian Institute of Technology. The RENNS architecture is scalable, with processing modules and their attached data streams as building blocks. Being dynamically reconfigurable, RENNS can form a range of communication architectures, in particular those suggested for neural networks. This allows the user to evaluate different configurations without building new hardware, hence reducing development time and cost.

Each module consists of one TMS320C30 digital signal processor with a capacity of 33 MFLOPS, 128–256 K 32-bit words of SRAM, 1–4 M 32-bit words of DRAM, and a separate communication subsystem. The SRAM can be accessed at full processor speed, while the DRAM requires 1 wait-state for burst mode access and 2 wait-states otherwise. The modules can be connected through four bidirectional communication links, each with a bandwidth of 5 Mbyte/s. Besides flexibility and high computational capacity, other design goals are minimum communication overhead and close to linear speedup when new modules are added.

The interprocessor communication can be changed dynamically at several levels during simulations. The most fundamental way of altering the communication network is to physically redirect the data streams between the processors. For another level of flexibility, most of the logic in the communication subsystem is implemented in field-programmable gate arrays. Depending on this logic, command registers in the communication controllers can be programmed. Reconfiguration at this level has been used extensively, to set up various addressing schemes for the communication, and for routing data internally in the communication subsystem. Both the processing part and the communication are optimised for operation on vectors. Although there is a large number of possible configurations, design decisions are taken to the advantage of topologies based on rings. A two-dimensional toroidal mesh is easily formed by rings, while several rings can be used to mimic the layered structure of many neural networks. Topologies based on one-dimensional rings with token ring protocol have shown to be a general and efficient way of intermodule communication for many applications.

For performance analysis, recurrent associative memories, or more specific, variants of Hop-
field nets are implemented and applied on a simple pattern recognition problem. One of the methods overcomes many of the serious limitations of the original Hopfield net but requires extensive resources of memory and computational capacity. Various parallelisation and mapping strategies have been tried out on a one-dimensional ring and a two-dimensional toroidal mesh. An eight-processor configuration with 1 M words of DRAM on each module can take a maximum pattern size of around 50×50. Highest performance is achieved with close to maximum pattern size on the eight modules. 40.8 MCPS (Million Connections Processed per Second) was measured, which means a utilisation of the numerical capacity of 5.9 FLOPS/CPS. On eight processors, a speedup of 7.7 compared to the one-processor case is achieved for large patterns. For small patterns, communication costs and the sequential part of the program become more significant, reducing the speedup to 5.7. The time spent on waiting for data is 2–10% of the total execution time for small problems, while it is negligible compared to the total execution time for large patterns. The most computational intensive part is multiply-accumulate, which is proportional to square the pattern size and is 40–60% of the total execution time, depending on the pattern size. The large pattern sizes common in Hopfield nets require the weight matrix to be stored in DRAM, giving a maximum speed of 2 processor clock cycles per multiply-accumulate.

A straightforward parallelisation strategy showed to be the most efficient. This simple method is non-overlapping communication and computation on a one-dimensional ring, using token ring protocol. Attempts to pipeline computations and communication, and to increase communication bandwidth by using a two-dimensional toroidal mesh, do not lead to improved performance. Although the waiting time for data is reduced, more complex code a and larger non-parallelisable part of the program instead reduce the performance, compared to the simpler approach. Experiments with one-dimensional ring and two-dimensional mesh architectures for a scalable application have shown that the significance of several performance limiting factors differ from assumptions based on theoretical considerations.
Contents

Preface ...
Nomenclature ...

1 Introduction
1.1 Computing with Artificial Neural Networks .... 1
1.2 Simulation of Large ANNs Requires Parallel Hardware ... 4
1.3 Hardware for ANN Simulations - Different Approaches ... 5
1.4 RENNS, a REconfigurable Neural Network Server ... 5
1.4.1 Why a Reconfigurable Neurocomputer ... 6
1.4.2 Performance Analysis of RENNS ... 6
1.5 Contributions of this Work ... 7
1.6 Outline of this Thesis ... 8

2 Parallel Processing in ANN Simulations ...
2.1 ANN Paradigms, A Short Survey ... 11
2.2 A Unifying Mathematical Framework for Various ANN Models ... 15
2.3 Levels of Parallelism in ANN Algorithms ... 17
2.4 Mapping Neural Networks to Parallel Architectures ... 20
2.4.1 Ring Architectures ... 21
2.4.2 Mesh Architectures ... 21
2.5 Parallelisation of Different ANN Models ... 23
2.6 The Efficiency of the Parallelisation ... 28
2.6.1 Benchmarks ... 28
2.6.2 Comparison of Communication Topologies ... 29
2.6.3 Maximum Speedup and Optimal Size of a Processor Network ... 30
2.6.4 Prototyping ... 31

3 Neurocomputers ...
3.1 Special Purpose Neurocomputers ... 33
3.2 Programmable Parallel Computers Used in ANN Simulations ... 34
3.2.1 Massively Parallel General-purpose Computers ... 36
3.2.2 Highly Parallel and Moderately Parallel Neurocomputers ... 37

iii
3.3 Other Architectures ........................................... 39

4 Hardware Design Considerations 41
  4.1 The Processors ............................................. 41
    4.1.1 Numerical Precision .................................. 42
  4.2 Storage Capacity .......................................... 43
  4.3 Communication Bandwidth .................................. 43
  4.4 Processor Topologies ....................................... 45
    4.4.1 Broadcast Communication .......................... 46
    4.4.2 Ring and Linear Array Architectures ............. 47
    4.4.3 Two-Dimensional Mesh Structures ................. 47
    4.4.4 Higher Dimensional Architectures ................. 48
    4.4.5 Communication by General Routing ............... 48
    4.4.6 Other Communication Schemes ..................... 48
    4.4.7 Speeding Up the Simulations Through a Pipelined Architecture 49
    4.4.8 Comparison of Processor Topologies ............. 49
  4.5 Implementation Technologies ............................. 49
  4.6 Performance Criteria ..................................... 50
    4.6.1 Measurements and Metrics ........................ 50
    4.6.2 Speedup and Efficiency ............................ 52
    4.6.3 Benchmarks .......................................... 52

5 The RENNS Architecture and Implementation 53
  5.1 Design Goals .............................................. 53
  5.2 Building Blocks .......................................... 54
  5.3 The Modules ............................................... 55
  5.4 The Processors ............................................ 56
    5.4.1 Choosing a Processor ................................ 56
    5.4.2 The TMS320C30 ........................................ 56
  5.5 Memory Configuration ..................................... 59
  5.6 Clock Distribution ........................................ 60
  5.7 Interprocessor Communication ............................ 61
  5.8 The Processing Subsystem ................................ 61

6 The Communication Subsystem 63
  6.1 Design Goals .............................................. 63
  6.2 First Approach, with Serial Intermodule Communication 64
  6.3 Testing the Bandwidth of Flat Ribbon Cable .......... 65
  6.4 Current Implementation ................................... 65
  6.5 Use of Reconfigurable Logic ............................ 67
  6.6 Processor Interface ....................................... 68
  6.7 Data Streams .............................................. 68
  6.8 FIFO Memory ............................................... 69
6.9 Control Logic .......................................................... 70
   6.9.1 Partitioning of the Control Logic ......................... 71
   6.9.2 Data Paths through the Communication Subsystem .... 73
   6.9.3 FIFO Control ................................................. 74
   6.9.4 Data Stream Control ..................................... 75
6.10 Configuring the LCAs ............................................. 78
6.11 Synchronisation .................................................. 79
6.12 Communication Protocols ....................................... 79
   6.12.1 Signalling on the Data Stream ......................... 79
   6.12.2 Token Ring Protocol .................................. 80
   6.12.3 Addressing on the Ring Bus ............................ 81
6.13 Balancing the Processor Speed ................................. 81

7 Configuring the Communication Network .......................... 83
   7.1 Reconfiguration on Three Levels ............................ 83
   7.1.1 Data-path Level .......................................... 84
   7.1.2 Logic Level ............................................... 84
   7.1.3 Command level .......................................... 85
   7.1.4 Possible Configurations ................................. 85
   7.2 One-Dimensional Ring Architectures ....................... 85
   7.2.1 Token Ring ............................................... 85
   7.2.2 Direct Communication .................................. 87
   7.3 Architectures Based on Several Rings ...................... 87
   7.4 Mesh Topologies ............................................. 88

8 Parallelisation of Recurrent Associative Memories .............. 93
   8.1 Motivation for the Choice of Test Problem ................ 93
   8.2 A Simple Pattern Recognition Problem ..................... 96
   8.3 Processor Architectures for Recurrent Associative Memories 96
   8.4 Determining the Weights and Thresholds .................... 97
   8.4.1 Methods for Specification of Weights and Thresholds .... 98
   8.4.2 Synthesis Procedure .................................. 99
   8.4.3 Algorithms ............................................. 102
   8.4.4 Parallelisation ....................................... 104
   8.5 Recall Phase .............................................. 107
   8.5.1 Calculations .......................................... 107
   8.5.2 Algorithm .............................................. 109
   8.5.3 Parallelisation ....................................... 109
   8.6 Scaling the Problem ....................................... 110
   8.7 Storage Capacity and Recall Results ...................... 111
   8.8 Memory Requirements for Scaled-up Problems .............. 112

9 Implementation of Recurrent Associative Memories .............. 115
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.4 Parallel Recall, Different Parallelisation and Mapping Strategies</td>
<td>172</td>
</tr>
<tr>
<td>C.5 Calculation of Weights and Biases</td>
<td>192</td>
</tr>
<tr>
<td>C.6 Configuring the Communication</td>
<td>197</td>
</tr>
</tbody>
</table>

D Result Tables                                | 203  |
List of Figures

1.1 A simple neuron, which forms a weighted sum of N inputs and passes the result through a nonlinearity. ........................................ 2
1.2 Three representative nonlinearities. ........................................ 2
1.3 General multilayer feed-forward neural network. ..................... 3

2.1 A three layer perceptron. .................................................. 13
2.2 Pipelined backpropagation learning (from [50]). ..................... 19
2.3 Data dependency graph for multilayer ANNs. .......................... 20

3.1 Spectrum of computer architectures used in ANN simulations (from [73]). 34
3.2 A general digital neurocomputer system (from [45]). ......... 35

4.1 Storage capacity versus computing speed in neural networks (from DARPA study [11]). .................................................. 44
4.2 Some of the most common processor topologies used in ANN simulations. 45

5.1 A processing module with its attached data streams (from [50]). .... 54
5.2 Different configurations for data stream operation (from [50]). .... 54
5.3 Block diagram of a RENNS module (from [48]). .................. 55
5.4 Memory organisation in a RENNS module ............................ 60
5.5 The processing subsystem. .............................................. 61
5.6 RENNS modules in a rack. ............................................. 62

6.1 Block diagram of the RENNS communication subsystem (from [50]). ... 66
6.2 The FIFO buffers. ....................................................... 70
6.3 Components and data paths in the communication subsystem (from [79]). 72
6.4 The communication subsystem board. ................................ 73
6.5 The data paths between the communication links and the FIFOs. ... 74
6.6 Outgoing data to the data streams. .................................. 76
6.7 Incoming data from the data streams. ................................ 77
6.8 Data paths through the channel controller. ........................ 78

7.1 Three reconfiguration levels. ......................................... 84
7.2 Internal data paths in a module connected to two rings. .......... 86
7.3 RENNS modules connected in two one-dimensional rings. ........ 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>RENNS modules set up with direct connections.</td>
<td>87</td>
</tr>
<tr>
<td>7.5</td>
<td>Layered neural networks implemented with rings (from [50]).</td>
<td>88</td>
</tr>
<tr>
<td>7.6</td>
<td>Implementation of pipelined backpropagation learning using several rings.</td>
<td>89</td>
</tr>
<tr>
<td>7.7</td>
<td>Internal data paths in a module connected in a TLA.</td>
<td>89</td>
</tr>
<tr>
<td>7.8</td>
<td>RENNS modules connected in TLA.</td>
<td>91</td>
</tr>
<tr>
<td>8.1</td>
<td>Recurrent neural network.</td>
<td>94</td>
</tr>
<tr>
<td>8.2</td>
<td>Example patterns, (a) original pattern (b) pattern with 25% statistical noise.</td>
<td>97</td>
</tr>
<tr>
<td>8.3</td>
<td>Sequential calculation of weights and thresholds in a Hopfield net, outer product method.</td>
<td>102</td>
</tr>
<tr>
<td>8.4</td>
<td>Sequential calculation of weights and thresholds in a Hopfield net, orthogonalisation method.</td>
<td>103</td>
</tr>
<tr>
<td>8.5</td>
<td>Algorithm for calculating matrix exponential functions for the Hopfield net.</td>
<td>104</td>
</tr>
<tr>
<td>8.6</td>
<td>Parallel calculation of weights and thresholds in a Hopfield net, orthogonalisation method.</td>
<td>104</td>
</tr>
<tr>
<td>8.7</td>
<td>Parallel routine for calculating matrix exponential functions for the Hopfield net.</td>
<td>106</td>
</tr>
<tr>
<td>8.8</td>
<td>Sequential algorithm for the Hopfield net recall phase.</td>
<td>108</td>
</tr>
<tr>
<td>8.9</td>
<td>Parallelisation of the Hopfield net recall phase.</td>
<td>109</td>
</tr>
<tr>
<td>9.1</td>
<td>Ideal workload distribution.</td>
<td>110</td>
</tr>
<tr>
<td>10.1</td>
<td>Parallelisation with non-overlapping communication and computation.</td>
<td>116</td>
</tr>
<tr>
<td>10.2</td>
<td>Speed measurements for non-overlapping communication and computation recall phase.</td>
<td>127</td>
</tr>
<tr>
<td>10.3</td>
<td>Execution time per iteration for small pattern sizes.</td>
<td>128</td>
</tr>
<tr>
<td>10.4</td>
<td>Execution time per iteration for larger pattern sizes.</td>
<td>129</td>
</tr>
<tr>
<td>10.5</td>
<td>Parallelisation with pipelined communication and computation.</td>
<td>129</td>
</tr>
<tr>
<td>10.6</td>
<td>Speed measurements for pipelined communication and computation recall phase.</td>
<td>130</td>
</tr>
<tr>
<td>10.7</td>
<td>Parallelisation by computing and communicating partial sums.</td>
<td>131</td>
</tr>
<tr>
<td>10.8</td>
<td>Speed measurements for recall phase based on computing and communicating partial sums.</td>
<td>131</td>
</tr>
<tr>
<td>10.9</td>
<td>Comparison of parallelising strategies for token ring protocol.</td>
<td>132</td>
</tr>
<tr>
<td>10.10</td>
<td>Broadcast stages in ring of direct connections.</td>
<td>133</td>
</tr>
<tr>
<td>10.11</td>
<td>Comparison of direct connections versus token ring protocol.</td>
<td>133</td>
</tr>
<tr>
<td>10.12</td>
<td>Parallelisation with a computation order based on handling single words.</td>
<td>134</td>
</tr>
<tr>
<td>10.13</td>
<td>Speed measurements for recall phase with direct connections, single word based algorithm.</td>
<td>135</td>
</tr>
<tr>
<td>10.14</td>
<td>Distribution of weight matrix for toroidal lattice architecture.</td>
<td>136</td>
</tr>
<tr>
<td>10.15</td>
<td>The stages in the computations for toroidal lattice architecture.</td>
<td>137</td>
</tr>
<tr>
<td>10.16</td>
<td>Speed measurements for recall phase with toroidal lattice architecture.</td>
<td>138</td>
</tr>
<tr>
<td>10.17</td>
<td>Speed-up from keeping data in fastest possible memory.</td>
<td>139</td>
</tr>
</tbody>
</table>
10.18 Speedup from utilising the programmable FIFO flags........... 141
10.19 Time to determine weights and thresholds, outer product method.... 142
10.20 Time to determine weights and thresholds, orthogonalisation method, small patterns................................................... 143
10.21 Time to determine weights and thresholds, orthogonalisation method, larger patterns................................................... 143
A.1 RENNS as a neural network server (from [50]).......................... 158
A.2 Typical programming environment (based on Treleaven).............. 160
B.1 Component placement on the communication subsystem printed circuit board......................................................... 166
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A mathematical framework for various ANN models.</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Speed measurements for ANN simulations on different computer systems.</td>
<td>51</td>
</tr>
<tr>
<td>8.1</td>
<td>Storage capacity in Hopfield neural networks.</td>
<td>95</td>
</tr>
<tr>
<td>8.2</td>
<td>Scaling up the pattern recognition problem.</td>
<td>111</td>
</tr>
<tr>
<td>8.3</td>
<td>Problem size range for a varying number of processors.</td>
<td>112</td>
</tr>
<tr>
<td>8.4</td>
<td>Memory used by matrices and vectors for the Hopfield net, orthogonalisation method.</td>
<td>113</td>
</tr>
<tr>
<td>10.1</td>
<td>System scaling for different problem sizes.</td>
<td>144</td>
</tr>
<tr>
<td>D.1</td>
<td>Speed measurements for non-overlapping communication and computation recall phase.</td>
<td>203</td>
</tr>
<tr>
<td>D.2</td>
<td>Execution time per iteration for small pattern sizes (128).</td>
<td>204</td>
</tr>
<tr>
<td>D.3</td>
<td>Execution time per iteration for larger pattern sizes (1024).</td>
<td>204</td>
</tr>
<tr>
<td>D.4</td>
<td>Speed measurements for pipelined communication and computation recall phase.</td>
<td>204</td>
</tr>
<tr>
<td>D.5</td>
<td>Speed measurements for recall phase based on computing and communicating partial sums.</td>
<td>205</td>
</tr>
<tr>
<td>D.6</td>
<td>Comparision of parallelisation strategies for token ring (pattern size of 512 elements).</td>
<td>205</td>
</tr>
<tr>
<td>D.7</td>
<td>Comparision of direct connections versus token ring protocol.</td>
<td>205</td>
</tr>
<tr>
<td>D.8</td>
<td>Speed measurements for the recall phase with direct connections, single word based algorithm.</td>
<td>206</td>
</tr>
<tr>
<td>D.9</td>
<td>Speed measurements for toroidal lattice architecture (TLA), recall phase.</td>
<td>206</td>
</tr>
<tr>
<td>D.10</td>
<td>Speed-up from keeping data in fastest possible memory.</td>
<td>206</td>
</tr>
<tr>
<td>D.11</td>
<td>Time to determine weights and thresholds, outer product method.</td>
<td>207</td>
</tr>
<tr>
<td>D.12</td>
<td>Time to determine weights and thresholds, orthogonalization method, small pattern (128).</td>
<td>207</td>
</tr>
<tr>
<td>D.13</td>
<td>Time to determine weights and thresholds, orthogonalization method, larger patterns (1024).</td>
<td>207</td>
</tr>
</tbody>
</table>
Preface

The REconfigurable Neural Network Server (RENNS) project was started in the autumn 1989 at the Department of Computer Systems and Telematics, the Norwegian Institute of Technology (NTH). At the Computer Systems Group, there was a growing interest in application specific hardware, especially hardware for neural network simulations. A project on building a general purpose neurocomputer was started late in 1989, supervised by Professor Olav Landsverk. To build a parallel processor requires extensive resources with respect to working hours and financial support. Financial support was given from NTH for a three year period, from 1990 to 1993. This covered most of the hardware costs, like the components and the production of professional printed circuit boards.

The RENNS project was divided into two main parts, the processing part, including the host interface, and the interprocessor communication. I have been involved in the design and implementation of the communication subsystem, with Jon Gunnar Solheim and Jan Anders Mathisen. Jon Gunnar Solheim is currently finishing his Ph.D. thesis on specialised hardware for neural network simulations. The processor board was designed by Jarle Greipsland, and Håkon Dahle was also involved in this part. The work with the host interface has been done mainly by Jarle Greipsland. He has also administered the work on the software part, by supervising students doing their diploma thesis on the software environment. Ph.D. students Gaute Myklebust and Jim Tørresen will continue developing and evaluating RENNS for neural network applications.

In this thesis, I will give a motivation for and a description of the RENNS architecture. I will cover, but not in-depth, the processing part, for understanding of the platform for neural network simulations. I will go into more detail on the communication subsystem, describing the hardware, functionality, and operation, and discussing the performance in relation to the design objectives.

Performance analyses are accomplished as individual projects. Different neural network models and applications are selected for test purposes. Among these I have been concentrating on recurrent associative memories, or more specific, variants of Hopfield type neural networks.
This thesis contains first a general part, which is a study of ANN models, parallelisation strategies, neurocomputers, and neurocomputer design considerations. Based on this study, the architecture of the REconfigurable Neural Network Server is proposed, followed by a description of the implementation. The third part comprises performance analysis of RENNS, including parallel implementation of test applications, speed measurements, and conclusions.

Various PC based design tools are used for implementation of the RENNS hardware. The schematics for the communication subsystem printed circuit board (PCB) is drawn in SCHEMA III (Ovation Inc.) and imported into ULTIBoard (ULTIMATE Technology), which is used as layout tool. ULTICap is used to generate files for PCB production. As design entry and simulation environment for LCA design, Viewdraw and Viewsim (VIEWlogic Systems Inc.) are used. The state machines are written in ABEL (Data I/O Corporation), and merged with the schematics from Viewdraw, before translation into Xilinx netlist files. Mapping and routing of the LCAs are done in the Xilinx Design Manager environment (Xilinx Inc.), which includes the XACT Design Editor, APR (partitioning and routing program for the Xilinx 3000-family LCAs), Xdelay (for timing information), etc. Program development for testing of RENNS has been carried out on a SUN SPARCstation 10, and cross-compiled for the TMS320C30 processor.

**Acknowledgements**

The work described in this dissertation has been carried out mainly at the Department of Computer Systems and Telematics (IDT), NTH, from the start of the RENNS project in October 1989. I am most grateful for the financial support from NTH, the first year as a Research Assistant, then as a full time Ph.D. student, until I moved to Stavanger in April 1992. As an Assistant Professor at Rogaland University Centre (now Stavanger College) almost a thousand kilometres distance from Trondheim and the hardware lab. at IDT, some practical problems with finishing this work occurred. I will therefore like to thank Department of Electrical and Computer Engineering, Rogaland University Centre, for allowing me to stay at NTH in Trondheim for longer periods, working with the RENNS hardware and neural network implementations.

My supervisor at NTH, professor Olav Landsverk, has deep knowledge and long experiences in computer architecture and implementation. I would like to thank him for being always available for questions and discussions during this work, and for giving me valuable comments on a preliminary version of this thesis.

I gratefully acknowledge the inspiring and supportive environment provided by my fellow students as well as the staff of the Computer Systems Group at IDT, NTH. In particular,
I will thank Jon Gunnar Solheim, who has always been most helpful when I had problems with unstable hardware. I will also express my gratefulness to Jarle Greipsland and Jan Anders Mathisen for supporting the accomplishment of this work. Gaute Myklebust has taken the time to review early versions of this thesis, which I greatly appreciate.

Finally, special thanks to Dag for constant support and encouragement.

Stavanger, May 1995

Lisbet Utne
# Nomenclature

## Words and Phrases

- **Activation function**: Transfer function in a neuron
- **Activation value**: Output from a neuron
- **Epoch**: A complete cycle of learning vector presentations
- **Fan-in**: The number of incoming connections to a neuron
- **n layer network**: A neural network with n layers of neurons
- **Neural network**: Artificial neural network
- **Neuron**: An artificial neuron
- **Module**: RENNS module
- **Target**: The desired output from a neural network
- **Weight**: The connection weight value

## Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</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>DMUX</td>
<td>2-4 Demultiplexer for 9-bit wide busses</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EPROM</td>
<td>Erasable Programmable Read-Only Memory</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out memory</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating Point Operations Per Second</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LCA</td>
<td>Logic Cell Array (LCA™ is a registered trademark of Xilinx Inc.)</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-Accumulate</td>
</tr>
</tbody>
</table>
MIMD  Multiple Instruction Multiple Data
MUX   4–2 Multiplexer for 9-bit data busses
PE    Processing Element
PLA   Planar Lattice Architecture
PLANNS Planar Lattice Architecture for Neural Network Simulations
RENNS Reconfigurable Neural Network Server
RISC  Reduced Instruction Set Computer
SIMD  Single Instruction Multiple Data
SOFM  Self-Organising Feature Maps
SPARC Scalable Processor ARCHitecture
SPMD  Single Program Multiple Data
SRAM  Static Random Access Memory
SVD   Singular Value Decomposition
TLA   Toroidal Lattice Architecture

Symbols

e_i   Output error of neuron_i
E     Error vector
f()   Activation function, nonlinear or piecewise linear
m     Number of patterns to be stored in a neural network
n, N  Number of input variables
Net_i The value achieved by accumulating weighted inputs
to a neuron, before it is passed through the non-linearity
o_i, Out_i Output value from neuron_i
p     Number of processors
t_i   Target output of neuron_i
w_ij  Weight from neuron_i to neuron_j, or from input_i to neuron_j
W     Weight matrix
x_i   State from neuron_i, or input_i
X     State vector or input vector

δ     Rumerhart’s delta value
Δw    Weight change
η     Learning rate, constant less than 1
θ_i   Threshold in neuron_i
Chapter 1

Introduction

1.1 Computing with Artificial Neural Networks

Which Problems Can Be Solved by Artificial Neural Networks?

In the human brain there are about $10^{12}$ neurons or nerve cells, each connected to up to tens of thousands of other neurons. Neurons are often grouped into larger structures formed by hundreds of thousands neurons. Artificial Neural Networks, abbreviated ANNs, are simplified and to some extent also modified models of these complex biological neural networks.

ANNs have lead to improvements in solving problems in areas where conventional computing has not succeeded, but where the human brain is superior. In practice, ANNs are especially useful for problems which are tolerant of a high error rate, have lots of example data available, but to which rules can not easily be applied. In this category of problems we find processing of speech and vision, trying to achieve human-like performance in speech and image recognition, and all kinds of pattern recognition tasks. Other industrial applications of ANNs are in the field of process modelling, identification, and control. ANNs are also successfully applied as elements in knowledge based systems.

The Neuron Model

ANNs are simplified models of neurons and their interconnections in biological systems. In the simplest neuron model, the neuron sums $N$ weighted inputs and passes the result
through a nonlinear or piecewise linear function (in the following denoted nonlinearity or nonlinear function). Figure 1.1 shows a simple neuron model. The three most common nonlinearities are illustrated in Figure 1.2. The nonlinearities can easily be modified to give an output in the range \([-1, 1]\).

Figure 1.1: A simple neuron, which forms a weighted sum of N inputs and passes the result through a nonlinearity.

Figure 1.2: Three representative nonlinearities.

The output function is

\[
o_i = f(\sum_{i=0}^{N-1} w_{ij} x_j - \theta_i)
\]  

(1.1)

where \(f\) is a nonlinear function, \(x_j\) are the inputs with their associated weights \(w_{ij}\), and \(\theta_i\) is the internal offset (threshold) in the neuron. The neurons can be thought of as very simple processing elements, or units, each possibly having a small amount of local memory. The neurons are connected by unidirectional communication channels, connections, which carry numerical data. The neurons operate only on their local data and on the inputs they receive via the connections.
Neural Network Models

In an artificial neural network, the neurons are connected in various structures. The simplest are feed-forward single layer networks, which have full connectivity between the inputs and the neurons. Another category of single layer networks are recurrent networks, where the outputs are fed back to the inputs. Research has shown that the "computing" in the human brain is organised in layers, and many of the ANN models reflect this by connecting the artificial neurons in a layered structure.

The best-known ANN model is the multilayer perceptron, which fits into the more general model shown in Figure 1.3. A multilayer perceptron consists of one input and one output layer, and one to several hidden layers. In general, the number of neurons in each layer is varying. The output of a neuron is broadcasted to all the neurons in the following layer.

![Figure 1.3: General multilayer feed-forward neural network.](image)

The weights and bias values are determined through a learning process, which can be either supervised or unsupervised. In unsupervised learning, the weights and bias values are organised based on the input patterns presented to the network. If the learning is supervised, the weights and biases are adjusted according to the correct output values for each input pattern.

The most frequently used learning algorithm for the multilayer feed-forward networks is backpropagation [71, pages 318–364]. A sequence of input and target pattern pairs is presented, and the weights are changed until the network has converged. Many repetitions may be necessary. It may also be required to adjust some of the updating parameters and repeat the whole learning process.
1.2 Simulation of Large ANNs Requires Parallel Hardware

The computations involved in ANN simulations can often exceed millions of multiplications and additions. ANN simulations are computationally intensive for several reasons:

**Massive parallelism:** Real-world applications require large ANNs of several thousand neurons where all can be connected to a large set of other neurons. Each connection usually requires one multiplication and one addition.

**Large training data sets:** The training of large ANNs require processing and updating of millions of weights iterating through the training set a number of times. Many of the problems tried solved with ANNs involve large training data sets of up to several thousands training patterns.

**Convergence:** The convergence properties of ANNs are varying depending on the network architecture, size of the network, learning algorithm, learning parameters and the available training data or choice of training data. Taking small steps in updating the weights may be necessary to obtain convergence but increases learning time.

**Trial and error:** Many learning algorithms do not guarantee convergence at a global minimum. Repeated training runs with different initial weights or training parameters may be necessary.

**Application development and improvements:** Algorithms and architectures for artificial neural systems are continuously evolving. Researchers and users may wish to improve the system, which means going through new training sessions and experimentations.

The data retrieving time in ANNs depends on the size of the network, which is expressed by the numbers of neurons, layers and interconnections. The learning time is in addition influenced by the number of training patterns and of the convergence properties of the network. Some applications are characterised by a great number of neurons, other by a large training set. Speed in ANN simulations is usually measured in connections processed per second (CPS) during the retrieving phase, and connections updated per second (CUPS) during learning.

The research in the field of computing with neural networks is now progressing to the point where PCs, workstations, mainframe computers, and in some cases, even supercomputers are inadequate. Conventional computers consist of one single CPU, which means that the
processing are done sequentially. For a wide range of applications conventional computers do not meet the speed requirements, which suggests that parallel hardware should be used.

The perhaps most important issue why parallel hardware is necessary for ANN simulation, lies in the nature of the problem itself. The strength of biological neural networks is not at all the computational capabilities of each neuron, but the massive parallelism and distributed computing.

Because of the wide range of ANN architectures and training algorithms, the requirements to the computational systems are different. The variety of applications also have different requirements to speed and flexibility to alter the implementation.

1.3 Hardware for ANN Simulations – Different Approaches

The design issues discussed in the previous section, lead to systems ranging from hardware analogous in complexity to traditional random access memories (RAMs), to conventional serial computers [87]. Various machines have been designed and optimised for computing with ANNs. These machines can be divided into special purpose neurocomputers, which are dedicated to one ANN model implemented directly in hardware, and fully programmable computers capable of implementing the whole spectrum of ANN paradigms. A neurocomputer, whether general or special purpose, is essentially a parallel array of interconnected processing elements that operate concurrently. In addition to neurocomputers, machines in the categories of cellular arrays, general parallel computers and supercomputers have been found suitable for ANN simulations. These computers differ greatly in size, cost, speed, processor interconnection network, and degree of parallelism. As a result of the enormous research activity in the field of neurocomputing the last few years, research groups in USA, Japan, and Europe have operational parallel neurocomputers, and several parallel processor array architectures for ANNs are under development.

1.4 RENNS, a REconfigurable Neural Network Server

RENN S, REconfigurable Neural Network Server, is a programmable neurocomputer designed at the Norwegian Institute of Technology. The first version of RENNS is a configuration of 16 high performance processing modules, which makes it a moderately parallel computer. Each module contains a 32-bit floating point digital signal processor, the TMS320C30 [83], memory, and a communication network interface. In addition to a scal-
able interprocessor communication system, other design goals are minimum communication overhead, and close to linear speed-up when new modules are added.

1.4.1 Why a Reconfigurable Neurocomputer

The main objective has been to design a neurocomputer suitable for experimenting with different system architectures for simulation of ANNs [48, 50]. This has motivated the design of a reconfigurable multiprocessor. The simulation of an ANN application can be approached in various ways. For a specific application it is possible to choose among several ANN paradigms and training algorithms. These paradigms and algorithms can then be mapped to available computer architectures. RENNS is a multiprocessor which can be reconfigured to form a range of communication architectures, and in particular those suggested for ANNs. This allows the user to evaluate different configurations without building new hardware, hence reducing development time and cost. Such experiments can be helpful when searching for better computing structures for a given application. RENNS can also be used in prototyping. A trained ANN for a given application can be optimised for implementation on a special target computer system, which can be VLSI, programmable gate arrays or simply a PC. A more efficient simulation can often be achieved by minimising the number of neurons, reducing the precision of the weight and state values, and sometimes also the complexity of the arithmetic operations can be reduced.

1.4.2 Performance Analysis of RENNS

RENNS can be configured in various ways. The configuration is of great importance in the testing and evaluation process of RENNS, and in evaluating different communication architectures for the various ANN models.

In this work I have concentrated on implementations of recurrent associative memories, or more precisely, variants of Hopfield ANNs, which are single layer feed-back networks. For this type of ANNs, the weight and bias values are calculated off-line, prior to normal operation, based on the patterns to be stored in the network. Two different ways of calculating the weights and biases are implemented. Because of the regular structure of the Hopfield ANN, the scaling properties of the RENNS can be analysed.

Hopfield ANNs are single layer fully connected networks. Little can be done in the mapping onto a parallel architecture to reduce the communication requirements of the implementation [61]. For increased performance, parallelisation strategies should be searched for, that pipeline the computations and communication. To investigate this, several implementations of the Hopfield ANN will be compared, using both a ring architecture and a
two-dimensional grid.

1.5 Contributions of this Work

RENNS was build as a teamwork at the Computer Systems Group, NTH. I have been working mainly with the communication subsystem in a three persons group. The first phase comprised a top-down design of the communication subsystem, and to determine the interface to the main board. This was accomplished in cooperation with those taking care of the main board. The next step was refining the design of the reprogrammable logic on the communication subsystem, and layout of the circuit board. My part in this work has been to do the printed circuit board layout, and the design of the logic in two of the LCAs (Logic Cell Arrays). In fact, it required two versions of the circuit board, and additional modifications of a few traces, to get the hardware to function properly. This was due to circumstances hard to predict, and some unnecessary errors. The programmable logic on the board can be divided into three main parts; the communication channel controllers, the FIFO controller, and the routing of data (multiplexer and demultiplexer). I have implemented the logic in the multiplexer and demultiplexer circuits, which main purpose are to direct byte-wide busses through the communication subsystem. Some additional logic for initialisation of the communication subsystem also went into these two circuits.

Programming of ANN test applications and evaluation of the machine have been done as an individual work. I have been working with implementations of variants of Hopfield neural networks for pattern recognition. Prior to running and optimising complete ANN applications, there has been much work on fault detection, by writing test programs and tracing errors with a logic analyser.

Being a reconfigurable neurocomputer, RENNS is not only a powerful and scalable multiprocessor, but also a platform for practical experimentations with different communication architectures. The design decisions for the communication subsystem, in particular the decision to have asynchronous communication, results in a quite complicated subsystem. The experiences from the implementation and test of the communication subsystem will be thoroughly discussed.

By implementation of recurrent associative memories, peak performance and scaling properties are measured for recall of simple patterns. Especially the performance for large networks, utilising all available memory, is investigated. To improve the low storage capacity in Hopfield ANNs, I have tried out an alternative method for determining the weights and thresholds (suggested by Li et al [52]). The improved method introduces computationally heavy operations on large matrices. Parallelisation methods for these operations are proposed and implemented. The recall phase of the Hopfield ANN has been subject to a study
of parallelisation and mapping strategies on RENNS. Several parallelisation strategies are proposed and tested. These ranges from simple one-dimensional ring configurations with non-overlapping communication and computation, to more sophisticated parallelisation strategies, like pipelining and two-dimensional communication architectures.

The RENNS architecture and implementation were first presented at the *ICANN92, International Conference on Artificial Neural Networks*, in Brighton, September 1992 [48]. The following publication, presented at the *Fifth ISMM Conference on Parallel and Distributed Computing and Systems*, Pittsburgh, Pennsylvania, October 1992 [50], had more focus on hardware issues and reconfigurability. The work presented at the *Nordic Symposium on Neural Networks and Advanced Applications*, Bergen, October 1993 [49], also included program development.

### 1.6 Outline of this Thesis

This thesis can be divided into three main parts. The first part is a study of ANN models, parallelisations strategies, neurocomputers, and considerations of neurocomputer design. Then follows a description of the RENNS architecture and implementation. The third part comprises performance analysis, including parallel programming, simulation results, speed measurements, and conclusions.

In the following chapter, the parallelism in neural networks is discussed, together with different mapping strategies for parallel implementations of ANNs. The chapter starts with a short survey of the most common ANN models and gives a few mathematical formulations, to provide some background material for the discussions on parallelisation and mapping. Chapter 3 is a review of neurocomputers, and in Chapter 4 some of the issues that arise in designing and implementing a general-purpose neurocomputer are discussed.

Based on the study in Chapter 2, 3 and 4, the architecture and implementation for a reconfigurable neural network server is outlined in Chapter 5. The design of RENNS has been divided into two main parts; the processing and the communication subsystem. The major design goal for the communication part is to minimise communication overhead. Also, most of the reconfiguration facilities lie within the communication subsystem. Since this is also the part of the RENNS project where I have been most involved, an own chapter is assigned for description of the communication subsystem. Chapter 6 includes design goals for the communication subsystem, trying and failing, and a more detailed description of the implemented logic. Reconfiguration facilities and possible configurations are considered in Chapter 7.

Chapter 8, 9 and 10 cover implementations of recurrent associative ANNs and performance
analyses. First, a more thoroughly description of the Hopfield type ANNs is given. From the network model and mathematical formulations, parallel algorithms for the RENNS are developed in Chapter 8. Chapter 9 goes into more detail on the programming, memory mapping and communication configuration. Simulation results and speed-up measurements are given in Chapter 10.

Finally, in Chapter 11, the results and experiences from the design, implementation and performance analyses of RENNS are summarised, with suggested enhancements.

In an approach to give an overview of the total RENNS computer system, additional information on the parts more peripheral to my work can be found in Appendix A, which gives an overview of the host interface, the system software and the programming environment.

For more details on the parts I have been most involved, which are the communication subsystem board and the multiplexer and demultiplexer LCAs, see Appendix B, [79] and [78].

Program listings and result tables (for the results presented graphically in Chapter 10) can be found in Appendix C and D.
Chapter 2

Parallel Processing in ANN Simulations

A thorough theoretical understanding of explicit and inherent parallelism in neural network models can be basis for design of simulation hardware and the development of efficient mapping and partitioning schemes. Some models are inherently suitable to parallel processing, for example the multilayer perceptron, while other neural network models, like the asynchronous Hopfield model, become parallelisable after modification of the original models [45, pages 237–238].

2.1 ANN Paradigms, A Short Survey

At least 50 different ANN models have been proposed, of which more than 10 are in common use [29].

The short term memory in an ANN consists of the states of the neurons, while the long term memory comprises the values of the weighted interconnections. The weights and the bias values are determined through a learning process, which is accomplished prior to normal operation, and can be either supervised or unsupervised. In unsupervised learning, the weights and bias values are organised based on the input patterns presented to the network. If the learning is supervised, the weights and biases are adjusted according to the correct output values for each input pattern. Some networks have the ability of continuous learning, by adapting to new information, and gradually “forgetting” what was previously learned. In other networks, like Hopfield and Hamming networks, the weights and biases are not determined through iterations. Instead the network values are calculated in advance,
based on the set of input patterns to be stored in the network. These networks have been classified as fixed-weights networks [45].

**Single Layer Networks**

ANN models with a single feed-forward layer of neurons were introduced already in 1958, primarily variants of the single layer perceptron [70]. Other examples of feed-forward single layer networks are Hamming net [54], and Kohonen net or Self-organising Feature Maps [41, 42, 43].

Recurrent networks have their outputs fed back to the inputs. Various kinds of recurrent associative memories exist, where the Hopfield net [35, 36, 37] is the best-known.

Single layer ANNs are used in a range of applications, like pattern recognition, and as classifiers and associative memories. A fundamental weakness of single layer ANNs, is that they cannot solve problems where the decision regions are arbitrarily shaped. As an example, the XOR problem cannot be solved by a single layer network, because the decision regions are not linearly separable [71, pages 318–322].

**Multilayer Networks**

Multilayer networks overcome the serious limitation of single layer networks because arbitrarily shaped decision regions can be formed by adding one, two or three hidden layers. The capabilities of multilayer networks come from the nonlinearities used within the nodes. The determination of the weight and bias values in these networks is generally more complicated than for single layer networks.

Several multilayer architectures have been proposed. Among these are the multilayer perceptron [71, pages 318–364], with a range of learning algorithms, Bidirectional Associative Memories (BAM) [44], and different versions of Adaptive Resonance Theory (ART) networks [6, 7]. More complex and biologically plausible networks exist, like Neocognitron [18, 19, 20], which is also classified as a hierarchical neural network. Interesting properties have been obtained by combining two different single layer networks, like Counter-Propagation network [28], which consists of a Self-organising Feature Map [42] and a Grossberg Outstar [26].
Multilayer Perceptron

The best-known and most frequently used ANN model is the multilayer perceptron. A multilayer perceptron consists of one input and one output layer, and one to several hidden layers. In general, the number of neurons in each layer is varying. The output of a neuron is broadcasted to all neurons in the following layer. A more detailed illustration of a multilayer perceptron consisting of three layers is shown in Figure 2.1. Only the weight layers are counted as a layer. The nonlinearity can be any of those in Figure 1.2.

\[ y_l = f\left(\sum_{k=0}^{N_2-1} w_{kl}'' x_k'' - \theta_l''\right), \quad 0 \leq l \leq M - 1 \]
\[ x_k'' = f\left(\sum_{i=0}^{N_1-1} w_{ik} x_i' - \theta_k'\right), \quad 0 \leq k \leq N_2 - 1 \]
\[ x_j' = f\left(\sum_{i=0}^{N-1} w_{ij} x_i - \theta_j\right), \quad 0 \leq j \leq N_1 - 1 \]

**Figure 2.1: A three layer perceptron.**

For a given layer, the following expression describes the calculation process:

\[ o = F(W x) \]  

(2.1)

where \( x \) is the input vector to and \( W \) is the weight matrix for that layer. The threshold (see also Equation 1.1) is not expressed explicitly in Equation 2.1, but can be implemented...
as one extra input neuron which has always the value of one. The threshold will then be the weight from that neuron, and is an adjustable value, like the other weights.

Updating the weights is most often based on the delta learning rule, where the delta learning signal for a given neuron is defined as:

\[ \delta_i = [t_i - f(w_i^t x)] f'(w_i^t x) \quad (2.2) \]

\( t_i \) is the target value for that neuron, and the term \( f'(w_i^t x) \) is the derivative of the activation function \( f(w_i^t x) \). Because the derivative of the activation function is used, the delta rule is only valid for first order differentiable functions. This learning rule is derived from the condition of least squared error between the actual output and the desired output of a neuron.

The Backpropagation learning Algorithm

The most frequently used learning algorithm is backpropagation [71, pages 318–364]. A sequence of input and target pattern pairs is presented, and the weights are changed until the network has converged. Many repetitions may be necessary, and it may also be required to adjust some of the updating parameters and repeat the whole learning process.

Because a target value is available for each neuron in the output layer, adjusting the associated weights is easily accomplished, using the rule presented in Equation 2.2:

\[ \delta = o (1 - o) (t - o) \quad (2.3) \]

where \( t \) is the target vector for the output layer. If the sigmoid activation function is used, the derivative becomes the simple expression, \( o(1 - o) \).

The weight change for the connection between neuron \( k \) in the hidden layer, and neuron \( l \) in the output layer, can be calculated as follows:

\[ \Delta w_{kl} = \eta \delta_l o_k \quad (2.4) \]

The training rate coefficient, \( \eta \), is typically between 0.01 and 1.0.

Hidden layers are not associated with a target vector. For the neurons in the hidden layers, the \( \delta \) values are calculated by propagating the \( \delta \)s back from the output layer through the
weights to the previous layers, in the same manner as the forward pass. Backpropagation of the error can then be written as:

\[ \delta_i = \delta_j w_j^T \odot [o_i \odot (I - o_i)] \] (2.5)

The operator, \( \odot \) (suggested by Wasserman in [91]), is defined to indicate component-by-component multiplication of the two vectors. For updating the weights, Equation 2.4 is used for all layers.

A machine dedicated to neural network simulations, should aim for an efficient implementation of both the computation of the activations value (Equation 2.1), the backpropagated error (Equation 2.3 and 2.5), and updating of the weights (Equation 2.4).

Statistical Learning Algorithms

Other important learning algorithms for the multilayer perceptron are statistical methods, like Boltzmann and Cauchy training. Pseudorandom changes in the weights are made, retaining those changes that result in improvements. If a weight change results in increasing the error, the probability of accepting that change is calculated from a Boltzmann or Cauchy distribution. A random number from a uniform distribution between zero and one is selected, and the weight change is accepted if the calculated probability is greater than the random number.

2.2 A Unifying Mathematical Framework for Various ANN Models

Most neural network algorithms involve primarily operations that are repetitive and regular. Various unifying mathematical formulations have been developed [87, 47, 14].

As shown in Table 2.1, the various models can be presented by a unifying mathematical framework expressed in basic matrix operations, such as inner-product, outer-product, and matrix multiplications.

The recall phase, or the forward propagation, are for most models calculation of the inner product between the weight matrix and the input vector associated with each layer. During training, the weights are adjusted proportionally to the state value of the neuron, and often
<table>
<thead>
<tr>
<th>Neural Network Model</th>
<th>Propagation Rule</th>
<th>Weights Derivation</th>
<th>Error Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Hopfield [35]</td>
<td>$Net = \Sigma XW$</td>
<td>$w_{ij} = x_i x_j$</td>
<td></td>
</tr>
<tr>
<td>Bidirectional Associative Memory [44]</td>
<td>$Net = \Sigma XW$</td>
<td>$w_{ij} = x_{1i} x_{2j}$</td>
<td></td>
</tr>
<tr>
<td>Single Layer Perceptron [71]</td>
<td>$Net = \Sigma XW$</td>
<td>$\Delta w_{ij} = \eta x_i e_j$</td>
<td>$e_j = t_j x_j$</td>
</tr>
<tr>
<td>Back-propagation [71]</td>
<td>$Net = \Sigma XW$</td>
<td>$\Delta w_{ij} = \eta x_i e_j$</td>
<td>$e_{j_o} = f'(Net)(t_j - x_j)$</td>
</tr>
<tr>
<td>Counter Propagation [28]</td>
<td>$Net = \Sigma SW$</td>
<td>$\Delta w_{ij1} = \eta e_{j1}$</td>
<td>$e_{j_1} = x_i w_{ij1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Delta w_{ij2} = \eta e_{j2}$</td>
<td>$e_{j_2} = x_i w_{ij2}$</td>
</tr>
<tr>
<td>Self-organising Map [43]</td>
<td>$Net = \Sigma XW$</td>
<td>$\Delta w_{ij} = \eta e_j$</td>
<td>$e_j = x_i w_{ij}$</td>
</tr>
<tr>
<td>Adaptive Resonance Theory [6]</td>
<td>$Net = \Sigma XW_f$</td>
<td>$\frac{w_{ijf}}{w_{ijb}x_i}$</td>
<td>$e_j = x_i w_{ij}$</td>
</tr>
<tr>
<td>Neocognitron [18]</td>
<td>$Net = \frac{1 + x_s W_x}{1 + x_s W_h} - 1$</td>
<td>$\Delta w_{ij} = \eta e_j$ or $e_j = x_i w_{ij}$</td>
<td>$e_j = f'(Net)(t_j - s_j)$</td>
</tr>
</tbody>
</table>

Notation:
- $w_{ij} =$ weight from neuron$_i$ to neuron$_j$, or from input$_i$ to neuron$_j$
- $x_i =$ state from neuron$_i$, or input$_i$
- $e_j =$ output error of neuron$_j$
- $t_j =$ target output of neuron$_j$
- $f() =$ nonlinear function
- $\eta =$ constant less than 1

Table 2.1: A mathematical framework for various ANN models.
also proportionally to some error measure. For supervised learning, the error is calculated from the difference between the desired and actual output. In the unsupervised case, the error is derived from the difference between the present and the previous state value, based on an optimisation criteria.

2.3 Levels of Parallelism in ANN Algorithms

In a parallel ANN implementation, there are several ways of utilising the inherent parallelism in the model. These are parallel processing with respect to synapses (weights), neurons, layers, training patterns, and training sessions [64, 99]. Which of the dimensions of parallelism to be chosen in any particular implementation, depends on the ANN model, the hardware platform, and the timing constraints of the application.

Neuron Parallelism

An ANN model can be viewed as a directed graph, where the neurons represent the nodes or computations, and the connections represent the arcs or the data dependencies between the computations. The perhaps most obvious form of parallelism in an ANN is the parallel processing performed by all neurons in the network. If one processor is assigned to each neuron, in the forward phase, all the processors in the first layer can start their calculations simultaneously. The processors in the adjacent layers start their calculations when they get their data from the previous layer. Since all processors in one layer have equal processing time, the total processing time is proportional to the calculation time for one processor in the layer with most incoming data.

This form of parallelism corresponds to viewing the calculations as matrix operations and assigning each row in the weight matrix to a processor.

To reduce the number of processors required, only the neurons in one layer can do their calculations in parallel, and the layers are time-multiplexed in a one dimensional computer architecture.

Weight Parallelism

The most fundamental operation in an ANN is multiply-accumulate by weighting the neuron’s inputs and accumulating the result. A study of the calculations in the neurons
Chapter 2. Parallel Processing in ANN Simulations

reveals another parallelisation scheme. The calculations performed in the neurons are:

$$a_j = f(\text{Net}_j) = f\left(\sum_{i=0}^{n-1} w_{ij} x_i\right)$$

The time spent on the nonlinear function is constant, but the summation can be reformulated to:

$$\text{Net}_j = \sum_{i=0}^{n/2-1} w_{ij} x_j + \sum_{i=n/2}^{n-1} w_{ij} x_j$$

One processor can be assigned to each partial sum, thus reducing the calculation time to nearly half. In the extreme case, one processor can be assigned to each multiplication. Then the $n$ partial results must be added, which can be done in $\log_2 n$ steps.

This fine grained parallelism requires a high number of processors. For most applications, using this high number of processors is not feasible. Instead, a number of nodes or virtual processors can be mapped to a significantly smaller number of physical processors connected in various ways.

**Pipelining the Layers**

In multilayer networks, it is possible to pipeline the layers to speed up the learning, by having more than one training pattern going through the network simultaneously. If the model has a backward pass, as for backpropagation learning, it is possible to fold the pipeline back again. Backpropagation learning is done in three stages: forward pass, backpropagation of an error signal, and accumulation of the weight changes. The weights are updated after each iteration of the complete training set. Figure 2.2 shows a diagram of a pipelined three layer backpropagation network.

In the forward pass, the state vector in each layer is distributed among several processors, together with the corresponding part of the weight matrix. The output vector of one layer is broadcasted to all processors in the next layer. When the processors in the last layer have finished their computations, the outputs are compared to the target vector, and the processors assigned to the backpropagation pass can start their processing. The neuron state values for each layer in the forward pass are used in the calculation of the delta values, and are also forwarded to the processors in the backpropagation pass. The third set of processors accumulate weight changes for updating the weight matrices between iterations of the training set.
Training Session and Training Pattern Parallelism

In most of the ANN models, training is accomplished by updating the weights according to a set of training patterns. The weight adjustments can be done either after each training pattern, or per epoch, which is between each presentation of the complete training set. An advantage with epoch updating is that it can be efficiently parallelised. A group of processors keep a full copy of the network together with a nonoverlapping subset of the training patterns. The calculated weight adjustments are exchanged between the groups of processors for each iteration of the training set. Training pattern parallelism gives little communication overhead and almost linear speedup with the number of processors.

Training session parallelism means starting several independent training sessions on different processors. The sessions may differ in initial values for the weights, learning rates and other network parameters, and even train networks of different topologies.

These two types of parallelism are of interest only during the training phase of a network.
2.4 Mapping Neural Networks to Parallel Architectures

The parallelism in an arbitrary ANN model can be expressed in a fine grained data dependency graph, where the computations in each node are one multiply-accumulate (MAC), as shown in Figure 2.3. This graph represents the recall phase of a two-layer feed-forward network retrieving a single pattern. For recurrent networks, the dependency graph is also regular and can be represented by a grid model corresponding to Figure 2.3. For multiple patterns, a cubic data dependency graph for matrix-matrix multiplication is required, thus exploiting both weight and training pattern parallelism.

Figure 2.3: Data dependency graph for multilayer ANNs.

Based on the data dependency graph, a systematic mapping methodology can be found, to map ANN models to a general systolic processor array [46, 45]. For mapping to a
systolic array, the design of a locally linked dependency graph is a critical step. The multidimensional graph is mapped through a linear projection to a processor space of one dimension less than the graph. Then a scheduling scheme specifies the sequence of operations for the processors.

The main goal for a parallelising scheme is to balance computational and communication power, which is the idea of systolic computation. By pumping the data into a series of processors, and mapping the communication according to the problem, many processors can usually be fully utilised.

### 2.4.1 Ring Architectures

A ring of processors is probably the simplest systolic architecture. Based on a unifying mathematical formulation, Kung and Hwang [47] show that a programmable universal ring systolic array is well suited as a target architecture for a wide variety of ANNs.

The architecture exploit neuron parallelism per layer or iteration by simulating one layer in a feed-forward or feedback pass at a time. An important consideration in the architectural design has been to ensure that both the recall phase and learning phase can share the same storage and processing hardware. The proposed ring systolic array for the retrieval phase can easily be adapted to apply to the backpropagation rule and updating of the weights.

For all three operations, each processor is keeping one column vector of the weight matrix. In the retrieval phase, the activation values are circulating in the ring, and new activation values are calculated from the inner product of the weights and the activation values from the previous iteration. In addition, each processor stores a threshold value and computes the nonlinear function. For weight updating, a function \( g_i(\cdot) \) of the local state \((o_i, \delta_i)\) is first computed. Then \( h_j(\cdot) \), a function of the state in neuron \( j \) \((o_j, w_{ij})\), is produced and propagated cyclically to all other processors in the ring, while the weights are updated according to \( \Delta w_{ij} = g_i(\cdot)h_j(\cdot) \). For backpropagation networks, updating the weights is combined with calculations of the \( \delta \) values for the next layer (in the backward direction). The new \( \delta \) values are accumulated by circulation. One of the advantages using this method is that the transposed weight matrix need not to be stored or calculated.

### 2.4.2 Mesh Architectures

In the general systematic mapping methodology discussed above, the target architecture has one dimension less than the data dependency graph. By linear projection, the graph can be mapped onto 1, 2, or 3 dimensional parallel array architectures.
Mesh architectures comprise 2-dimensional communication topologies, where the processors exchange data between more than two of its neighbours. The most widely used mesh architecture has four communication lines per processing module (quadratic mesh), but also hexagonal mesh topologies exist.

An example of a quadratic mesh architecture designed especially for ANN simulations, is the PLA, Plannar Lattice Architecture [16, 15], which represents a straightforward mapping of a neural network to a mesh architecture. A virtual processing element is associated with each synapse (SP) and each neuron (CP). The virtual processing elements are organised in a \((i + j) \times (j)\) matrix, where \(i\) is the size of the input vectors and \(j\) is the number of neurons in the network. The CP's are placed along the \(i + j, j\) diagonal, and perform multiply-accumulate and execute the nonlinear function. The other positions in the matrix are occupied by SP's, which only do multiply-accumulate.

The forward pass of the PLA consists of three steps. First, all CP's send their activation value to the SP's in the same column, while simultaneously the input vector is asserted on the upper positions in the \(i\) leftmost columns. The activation or input values are multiplied with the weight in the SP's, and the results are accumulated along the rows of the matrix. The accumulation process is initialised by the SP's in the leftmost and rightmost columns, and the results are propagated towards the CP. When the partial sums from both sides are received by the CP, the new activation value can be calculated. This scheme is repeated for all layers in the network. In the back-propagation pass, the rows and columns swap function. The \(\delta\) values are distributed along the rows and the weighted \(\delta\) values are accumulated in the columns.

The algorithm is mapped to a smaller matrix of physical processors by having each physical processor emulating a rectangle of virtual processors. The upper row of processors need to communicate with the host computer, to get the input and target vectors, and to send the resulting outputs. The load balancing problems are addressed by an algorithm, which executes row and column permutations, before the mapping to a physical processor matrix. The performance of PLA is reported to be almost proportional to the number of processors [15].

A related architecture, TLA, Toroidal Lattice Architecture, has the same structure of SP's and CP's. The main difference is that in TLA, the processors are connected in horizontal and vertical rings, instead of being connected in a mesh by four bidirectional links [17, 14, 15]. The number of communication links is then halved. TLA has been implemented on \(4 \times 4\) Transputers, and has, like PLA, proved good scaling properties.
2.5 Parallelisation of Different ANN Models

ANN models are inherently parallel, and exploit at least neuron and weight parallelism. Which parallelisation scheme is most efficient, depends on the neural network topology, learning rule, and other application dependent properties. Equally important is that the parallelisation and mapping strategy is suited for the target hardware.

For optimum speed, the parallelisation and mapping method is highly application dependent. Another issue is to develop efficient general parallelisation and mapping methodologies. In many cases it would be advantageous for this task to be accomplished automatically. Work has been done in finding generic mapping strategies for a specific target computer [3, 4].

Several unifying mathematical formulations for ANN models have been developed, which aid in the process of finding general parallelisation and mapping methods. It is shown that most neural network models can be parallelised on a ring [47] or on a quadratic grid architecture [16].

In [61], Murre concludes that communication overhead is the decisive factor in the efficiency of scaled-up processor networks, but for fully or randomly connected neural networks processor topologies can only contribute towards a small reduction in communication overhead. What is more promising, is that the global regularity in the connectivity pattern found with modular neural networks may be used to implement more efficient routing schemes. Modular neural networks are defined as networks that can be decomposed into a number of subnetworks or modules that have dense internal interconnections, but where these subnets are not all interconnected. Modular systems are therefore good candidates for large scale neural network implementations, where the processor topology, e.g. grid, line, hypercube etc. may strongly influence the communication overhead. Murre also argues that modular neural networks are the most realistic from a psychological and biological perspective.

Hopfield networks and Boltzmann machines

Associative memories like Hopfield networks [35, 36, 37] and Boltzmann machines [33] have full connectivity between the neurons. For each iteration, the activation values from one neuron are sent to all others. For a network with $n$ neurons mapped to a linear processor topology, this broadcasting takes $n - 1$ steps. Using 4 bidirectional links per processor in a grid architecture, the communication takes $(n - 1)/4$ steps. With more communication channels per processor, the time to broadcast activation values for each iteration can only be reduced with a constant factor, which is the number of communication links per processor.
The calculations for the Hopfield network are proportional to the number of weights, $n^2$. The amount of data communicated in each iteration, with an efficient broadcast scheme, is proportional to the number of neurons, $n$. Although only a small reduction in communication overhead can be achieved by selecting more sophisticated processor topologies, significant speedup can be gained by parallelising the network.

Dividing the neurons and their weights equally among the processors in a systolic ring array is one of the simplest parallelisation techniques for the Hopfield networks and similar models. A small reduction in communication time should theoretically be obtained by selecting a mesh architecture, like TLA or PLA. One of the implementational problems with this class of networks, is that the number of weights are square the number of neurons. Since the weights are stored distributedly, more storage capacity can be obtained by adding more processors to a scalable architecture, or it may be necessary with secondary storage for the weights.

Boltzmann machines can be implemented in a similar manner. In a Boltzmann machine, the activation values are determined from a stochastic function. The large number of iterations make a parallel simulation mandatory.

**Hamming Net**

The computations in a Hamming net can be divided into two stages. The first is a one layer feed forward network to calculate Hamming distances between the input pattern and the previously stored patterns. The neuron representing the pattern closest to the input pattern is the most responding neuron, and is determined in a MAXNET. There are several possible implementations of MAXNET. One is a feed-forward tree structure (shown in [54]), another is through feed back and lateral inhibition [98, page 394]. A third alternative is to use a broadcast scheme. Which method is preferred, depends on the size of the network, the number of processors, the complexity of the processors, and the communication architecture.

To fully exploit the parallelism in a feed forward MAXNET, a large number of processors is required, but then the computations can be pipelined in $1 + \log_2 n$ steps. If the most responding neuron is found through broadcasting, the time spent on computations and communication is proportional to $n/p + p$, where $p$ is the number of processors.

For the iterative scheme, the feed forward stage and the iterations can time-multiplex the same collection of processors, which can be connected in various ways, like in a ring or a grid.
Bidirectional Associative Memory

Bidirectional Associative Memory [44] (BAM) consists of two layers of neurons, where the outputs from the second layer are fed back and broadcasted to the first layer. The two layers operate alternately in a resonance fashion until the network reaches a stable state. Since only one layer is "active" at a time, the two layers can time-multiplex the available processors, and the discussion on parallelisation methods for Hopfield nets is also valid for BAM.

Self-organising Feature Maps

Self-organisation Feature Maps, SOFM [42], convert patterns of arbitrary dimensionality into responses of one or two-dimensional arrays of neurons. The network is organised in a single layer feed-forward network, plus a MAXNET [54, 98], and is trained according to a winner-take-all or neighbourhood learning rule. The best-matching neuron, \( i \), can be determined by \( \min \|w_i - x\| \), where the neuron giving the smallest value must be picked out. This selection is computed in the same structure as the MAXNET. After the feed forward pass, the best-matching neuron is picked out, and a selection of the weights in a neighbourhood around that neuron is updated. Both neuron and weight parallelism for SOFM are discussed, leading to an array processor for neuron parallelism, and a vector processor for weight parallelism [80, 81]. Systolic ring and TLA/PLA are universal architectures, which can also be used for SOFM. The learning processed in SOFM can be pipelined by dividing the neurons equally among the processors in a ring, having the input vectors and the value and position of the locally calculated best-matching neurons circulating in the ring [8]. For pure recall, an obvious way to achieve fine grained parallelism, is to pipeline the computation of the neuron outputs and the MAXNET.

In the NERV neurocomputer [55], which is a bus based architecture, the VME-bus has been modified to fast computation of the winner-take-all rule.

Self-organising Feature Maps have been implemented on RENNS in one-dimensional ring and tree configurations [63].

Backpropagation Networks

The most popular ANN model is multilayer perceptron with backpropagation learning. The learning process can be extremely time consuming, but parallelisation down to one processing element per weight can be utilised. In addition, the layers and stages can be pipelined, as shown in Figure 2.2.
For both the unifying ring architecture [47], and for the quadratic grid architectures PLA and TLA [15], backpropagation is implemented to illustrate the methods. Much effort is made to develop architectures that are well suited for the backward pass, as well as the forward stage.

Implementations of backpropagation networks have been reported for a large number of supercomputers, neurocomputers, and other target machines. In addition to the hardware independent parallelisation strategies discussed so far, a range of parallelising methods tailored to specific parallel computers have been proposed. In [75], five different implementations of the backpropagation algorithm on the Connection Machine (CM-2) are described and compared, ranging from straightforward implementation on a grid architecture, to sophisticated implementations utilising the many features of CM-2 [97]. The backpropagation algorithm can easily be formulated as a set of matrix operations, and an implementation on the Cray Y-MP, which is optimised for matrix operations on large data sets, is described in [66]. Another fast SIMD implementation of backpropagation is described in [92] for the IBM GF11 experimental supercomputer. It is argued [92] that SIMD architectures are best utilised by training set parallel implementations, because all the processors execute the same instruction and the communication requirements are reduced. For simulation of backpropagation on the Emma-2 hierarchical general purpose parallel computer, three job distribution algorithms are offered: training set parallel, training pattern parallel, and functional decomposition. Other parallel computers for which simulation results on backpropagation have been reported are Warp [68], RAP [59], CNAPS [27]. Backpropagation on RENNS is presented in [77].

**Neocognitron**

The ANN models discussed so far are single layer networks or multilayer networks with massive interconnection between layers. The neocognitron is a hierarchical network, with sparse and localised connectivity between the layers [18, 19, 20]. The neocognitron is regarded as the most biologically plausible neural network. The applications reported so far have concentrated on recognition of handwritten and printed characters and numbers, independent of size, position, and noise distortion.

In neocognitron, each layer consists of two sublayers, which is again partitioned into planes. A plane contains a 2-dimensional matrix of neurons, or cells. The number of planes in each layer and the size of each plane is dimensioned for the recognition of characters and numbers. The network contains a total of more than 50 thousand processing elements (cells) and more than 14 million connections.

In a pair of sublayers, the first contains neurons of a type called S-cells ("S" for simple). The S-cells receive inputs from the previous layer, except the cells in the first layer, which
get their inputs from the input image. The cells in each plane in the S-layers have the same transfer function and the same weights. Each plane in the S-layer performs a local feature analysis operation at each location in the input image. The processing elements are, during learning, tuned to a specific spatial pattern feature, which increases in complexity with increasing layers. The neocognitron can be trained using either supervised training or self-organisation.

The first layer extracts simple features such as line segments and edges at various orientations. The higher level layers deal with more complex features, composed of line segments and arcs. If a particular feature has been detected, the cell has the value of one, otherwise the state of the cell is zero.

Once the S-cells in a layer have responded to their inputs, the output signals are sent to the appropriate planes in the second sublayer within the same layer, which are built from C-cells ("C" for complex). The C-cells in the second sublayer have fixed input weights, which are highest in the middle of the receptive field and fall off towards the edges. The C-cells are insensitive to small variations and distortions in the input pattern, and increase the robustness of the total network.

Training of the neocognitron can be carried out by using either supervised or unsupervised learning. In supervised mode, the training is done one layer at a time. An input pattern is presented to the S-cells of the first layer, and the weights are adjusted according to a target vector for that layer, in a similar manner as for training of perceptrons. The input patterns used during training are the same as the desired weight pattern for that plane after training. This means that the training patterns contain the simple features to be detected by the first layer. For the following layers, the input vector is the outputs from the C-cells of the previous layer, and the target vector presented contains more complicated patterns as one ascends through the layers. Training of the next layer does not start until training of the previous layer is finished. The weight sharing mechanism in the S-cells means that when one of the cells in a plane has its weights updated, all the other cells in the same plane immediately adopt these new weights.

In the self-organising mode, a competition mechanism similar to the Kohonenen's self-organising feature maps is used. As inputs, the net is simply presented with a large number of character examples. With supervised learning, constructing a good training pattern set is difficult and time consuming. With unsupervised learning, where the training processes progress automatically, the result is a network with somewhat less ability to recognise deformed patterns. An improved learning algorithm with a combination of unsupervised and supervised learning has been proposed [20].

The computation time to recognise one alphanumeric character is 3.3 s in average on a SUN SPARCstation, compared to 1.5 s for the ten-numeric-character recognition system [19]. If supervised learning is used, one of the advantages is short training time, only 13
minutes on the SUN SPARCstation. This is extremely short compared to other training methods such as backpropagation. Reported training time for character recognition using backpropagation was 3 days [19].

Possible parallelisation strategies for Neocognitron will be different for the learning and the recall phase. During recall, the computations are performed in a feed forward manner, where the outputs from one sublayer are inputs to the next. The computations can easily be pipelined, where one sublayer is one stage in the pipeline. One or more processors can be assigned to each sublayer. For the parallelisation of each sublayer, the most complex connectivity pattern is between complex cells in one layer and the simple cells in the next. The processing of these connections should be finer partitioned than for the other stage. Through the network, the number of neurons and connections in each layer is reduced. The processor assignment should reflect this by using more processors for the first stages.

For unsupervised learning, one pair of layers is trained at a time, based on the organisation of the previous layer. The parallelisation methods suggested for SOFM will also be valid for Neocognitron. In case of supervised learning, a parallel learning process becomes feasible because the output pattern of each plane is determined prior to the training, and each plane is trained with different training sets. A pair of layers can be trained in parallel in the same manner as the planes in the simple layers.

2.6 The Efficiency of the Parallelisation

The efficiency of a parallel implementation is highly dependent on the application, as well as the target hardware. Theoretical studies on this subject can provide useful information prior to hardware design. Benchmarks are often used in performance evaluation for more standardised test applications.

2.6.1 Benchmarks

Two commonly used benchmarks in ANN simulations are NETtalk [74] and Bignet [69]. These are large and realistic problems, which also illustrate the difference of two ANN applications, although based on the same ANN model. Benchmarks are not restricted to speed comparisons, but also used in evaluation of algorithms, in terms of learning speed, quality of ultimate learning, ability to generalise etc. For comparisons of hardware implementations, the application or benchmark can influence the simulation results and favour some parallelising and mapping schemes above others.
2.6. **The Efficiency of the Parallelisation**

**NETtalk**

NETtalk is a two-layer feed-forward network, and is often used for speed measurements of backpropagation learning. The network consists of 203 inputs, 60 to 120 nodes in the hidden layer, and 26 outputs. The network is relatively small, but is characterised by its large number of training patterns, which is about 20000. There are different number of neurons in each layer, with "fan-in" towards the outputs. The network is used for text to phoneme translation, given a string of letters as input. Each input vector consists of seven consecutive letters from a training text. The central letter in this sequence is the letter for which the phonemic output is produced. The letters on each side provide the necessary context to help determine the pronunciation. The network has a set of 29 neurons for each input letter, one for each letter in the English alphabet, and three for punctuation characters, making a total $29 \cdot 7 = 203$ input units. Typically 60 to 120 nodes are used in the hidden layer. The 26 output nodes encodes various articulatory features [74].

**Bignet**

Bignet is another two-layer feed-forward network. As the name indicates, it is a large network with 130-260 inputs, 130-260 nodes in the hidden layer, and 17000 outputs. This network is used for extracting convolution rules in proteins. The outputs are meant to give a three-dimensional image of the expected convolution in a protein. The relatively small training set consists of 26 well-known proteins. The network has finished its learning when it can predict the convolution of 100 control proteins with a 95% success rate [69].

### 2.6.2 Comparison of Communication Topologies

By studying an ANN model carefully, the number of arithmetic operations and communication steps can be derived theoretically. By developing a simulation model of a general processing element and the communication links, a study of the parallelisation of ANN models can be done prior to building the hardware. Although several implementational issues are not considered, such theoretical considerations can give valuable information on the scaling properties of various computer architectures.

The optimum communication topology for a given application minimises the communication overhead. In [2], broadcast bus, ring, and mesh architectures have been investigated, to optimally trade off the processing time and the number of processors, for a neural network with totally interconnected layers. The processing time includes both calculation and transmission time. The calculation time decreases as the number of processing elements
increases in every topology. The study shows that for broadcast bus and ring architectures, the total transmission time increases monotonically as new processing elements are added, but for the mesh, there exists an optimum number of processing elements.

To speed up the computation, the stages in a neural network simulation can be pipelined. This can be done with a topology based on several rings, as will be discussed in an example later in this thesis. For a mesh topology, it is shown [82] that instead of mapping a multilayer neural network to a single mesh, the communication network can be extended and pipelined, and thereby creating a more scalable topology of processing elements.

A similar study done by Øian [99], also shows that the application highly influences the efficiency of the mapping. Here a processor model is developed, and the performance of four different computer architectures is analysed, using backpropagation as an example. The parallel architectures investigated are ring (based on the mapping in [47]), grid (based on the PLANNS architecture [16]), pipelined processing, and a training set parallel implementation on a bus based architecture. Speed comparisons have been done for the implementations of both the NETtalk and Bignet benchmarks. Not surprisingly, simulation results show that when the number of processors grows large, bus based communication has the poorest scaling properties. For the grid architecture, the simulation time decreases monotonically with the number of processors, independent of the size of the neural network. The large number of training patterns in NETtalk favours training set parallel and pipelined implementations. These architectures have best performance for NETtalk when the number of processors is less than or equal the number of neurons. The ring architecture shows inefficient because of the uneven number of neurons in the layers. For Bignet, the performance of the grid architecture are close to maximum processor utilisation when the number of processors grows large, while for up to around 20 processors, the ring have the shortest simulation time.

2.6.3 Maximum Speedup and Optimal Size of a Processor Network

In [61], a performance analysis is presented that focus on the achievable speedup of a neural network implementation, and the optimal size of a processor network. The iteration time is decomposed into a computing part, constant communication overhead, and communication overhead proportional to the amount of distributed data. The optimal number of processors can be found by taking the first derivative of the iteration time with respect to the number of processors, and setting it equal to zero. The maximum speedup of the network can be found by considering the ratio between the iteration time on a single processor and the iteration time of the entire processor network, as the number of processors grows very large.
It is concluded that in randomly or fully connected neural networks that are large relative to the processor network, the achievable speedup is proportional to the size of the processor network. With large processor networks, more than half the processing power can be lost in communication, regardless of the processor topology. Limiting the neuron fan-in in those networks lead to decrease in performance because of the decrease in processing load. For a modular network on a torus, with limited neuron interconnect, implementations have a markedly better speedup of $O(\sqrt{p})$, compared to $O(1)$ for randomly connected networks.

### 2.6.4 Prototyping

Although unifying formulations and mapping methods for neural network models are proposed, there is a myriad of solutions for implementing ANNs. As the comparison studies show, the simulation results are influenced by the combination of the neural network size and other parameters, as well as the target computer architecture. Results obtained from simulations are concerned with uncertainties. In practise, the bandwidth of the communication channels may not be high enough to feed the processors for operation at full speed. Another uncertainty is whether the processors can utilise data from more than one channel during the same clock cycle. These questions can only be addressed by a hardware implementation of the models. With a reconfigurable computer, like RENNS, several issues concerned with the parallelisation of ANNs can be studied.
Chapter 3

Neurocomputers

One important difference between ANN research today and what was possible 30 years ago, is the huge improvement in technological capabilities [24], and the field of neurocomputers is rapidly progressing. A neurocomputer is essentially a parallel array of interconnected processing elements that operate concurrently. A variety of commercially available and research neurocomputers have been presented, and many more parallel processor array architectures are under development. Neurocomputers can be divided into special purpose neurocomputers, which are dedicated to one ANN model implemented directly in hardware, and fully programmable computers that can implement the whole spectrum of ANN paradigms.

In [87], the spectrum of computer architectures used in ANN simulations is classified according to a taxonomy proposed by Seitz [73]. This classification is shown in Figure 3.1. Lowest in complexity are RAMs. Next are special purpose neurocomputers. General purpose neurocomputers are placed in the middle zone. For comparison, the two subclasses of cellular arrays, computational arrays and systolic arrays, are shown. The last two zones cover coarse grained computers with complex processing units, like conventional parallel and serial computers, and supercomputers. Some classes are slightly overlapping, as the figure shows. A such classification will continuously be subject to adjustments because of rapid improvements in the underlying technology, and development of new technologies.

3.1 Special Purpose Neurocomputers

The demand for high speed in ANN simulations has motivated the design of special purpose neurocomputers, which are hardwired implementations of a specific neural network
architecture, or a pretrained ANN for a given application. Implementations of Kohonen and Hopfield associative memories have dominated because of their architectural simplicity compared to for example backpropagation, but also fully trained more complex networks have been integrated into VLSI.

Special purpose neurocomputer hardware can be broadly categorised by implementation technology into silicon, optical, and molecular. The most common implementation technology is digital VLSI, but analog and hybrid techniques have a huge potential for artificial neural systems. Alternatively, it is possible to do the computations analogous, and interface digitally to the processing device. This also simplifies interfacing to external memory storage and I/O [60].

The high performance in special purpose neurocomputers is achieved at the penalty of programmability and flexibility.

3.2 Programmable Parallel Computers Used in ANN Simulations

General purpose neurocomputers cover a range of parallel architectures, which are highly different with respect to size, cost, speed, processor interconnection network, and degree of parallelism. In addition, general parallel computers with high numerical capability, can be suited for ANN simulations.
3.2. Programmable Parallel Computers Used in ANN Simulations

The simplest and cheapest general purpose neurocomputers are commercial coprocessors, which are single boards that plug into a PC bus or interfaces to a workstation. These boards contain from one to several processors and local storage for neurons and interconnections. Examples are the TRW MARK III [29], and the ANZA and ANZA Plus systems [30].

Parallel processor arrays are more powerful systems, with many processing units connected in a regular topology. Processor arrays are typically built from several identical boards equipped with one or a group of processors, which can either be a commercially available or a custom designed processor. The processor array is usually operating as a back-end system, or server, and is connected to a host through some kind of bus based interface, for distribution of program code and data, and communication with the users. As a host driving the processor array, various computers are used, from a PC upwards. Figure 3.2 illustrates a digital neurocomputer system consisting of a host computer, interface unit, interconnection network, and a processor array.

![Diagram](image)

Figure 3.2: A general digital neurocomputer system (from [45]).

The degree of parallelism, or granularity, can be used as a characterisation criteria for the computers used in ANN simulations, but first a suitable classification scheme must be found. In [64], the following definitions are suggested (n is the number of processing
elements, PEs):

- Massively Parallel \( n \geq 2^{12} \)
- Highly Parallel \( 2^8 \leq n < 2^{12} \)
- Moderately Parallel \( 2^4 \leq n < 2^8 \)
- Barely Parallel \( 2^0 \leq n < 2^4 \)

This characterisation can be augmented with a description of the complexity of each building block. The bit-length of the natural data type for the PE is suggested as a metric, with a limit between simple and complex PEs around 12-bit. Alternatively, the numerical performance can be used. One problem is that since processing capability is rapidly increasing, this measure should be continuously updated.

In the following, a short survey of neurocomputers is given, classified according to the granularity scheme above. Because of the large and still growing number of such computers, a few computers are selected to represent the main categories.

### 3.2.1 Massively Parallel General-purpose Computers

In massively parallel computers, the number of PEs is so high that it is impossible to treat each processing unit individually, with its own instruction flow, which is the concept of a SIMD computer. None of the computers designed especially for ANN simulations can be characterised as massively parallel, according to the classification scheme above, but not surprisingly, the best-known general-purpose massively parallel computers have shown high performance in ANN simulations.

The Connection Machine (CM, CM-2) [32, 38], manufactured by Thinking Machines Corporation, is for the moment the most massively parallel machine built, with up to 1 M PEs. In addition to a large number of processors, two strong points are its powerful hypercube connection for general communication, and the multidimensional torus structure for problems with regular communication demands. The PEs are simple, bit serial processors. CM-2 is, in addition, equipped with a floating point processor per 32 bit-serial processors. CM-2 is one of the most popular parallel computers for implementing ANN algorithms. Most of the implementations so far concern backpropagation, but various parallelisation strategies have been utilised in different applications [75, 97].

Many massively parallel arrays use bit-serial PEs. The processing time on these computers grows linearly with the data length. This may be regarded as a serious disadvantage when using 32- or 64-bit floating point numbers, but bit-serial data paths greatly simplify the communication in massively parallel computers.
3.2. Programmable Parallel Computers Used in ANN Simulations

High numerical performance in floating-point calculations has been given high priority in some more recent massively parallel computers. In the second version of MasPar, MP-2, [56], 32-bit register based RISC processing elements have replaced the processing elements of the MasPar MP-1, which have only a 4-bit integer arithmetic-logic unit. ANN implementations on the MasPar are described in [25]. This trend towards more powerful processors also shows in the latest version of the Connection Machine, CM-5, which consists of up to 16 K SPARC processors [38]. Another difference between CM-5 and its predecessors, is that in CM-5, the processors operate in a MIMD fashion, to get a more flexible machine, and overcome the inefficiency with independent branching in SIMD architectures. For efficient interprocessor communication, the processors are still synchronised, with the same clock distributed to all the processors.

3.2.2 Highly Parallel and Moderately Parallel Neurocomputers

Despite the highly parallel nature of neural networks, most of the existing neurocomputers consist of less than 1 K processors. There is a wide range of different system solutions with respect to communication architecture, complexity of the processing elements, and whether commercially available hardware is used, or specialised logic is developed.

CNAPS (Connected Network of Adaptive Processors), manufactured by Adaptive solutions, USA, is one of the first architectures developed especially for ANNs [27]. It is a SIMD machine built from proprietary VLSI chips, each containing 64 simple DSP-like Processor Nodes (PNs). The interconnection scheme is based on broadcast buses. Each PN is connected to three 8-bit global buses, which are input bus, output bus, and command bus. The architecture can handle 1, 8, and 16 bit weights, and 8 and 16 bit activation values. A special hardware mechanism is developed for efficient representation of sparse connectivity. The CNAPS chips can be connected in various ways, in one or two dimensions, to adapt to the applications. The output bus from one set of PNs can be used as input bus for another.

REMAP³ (Real-Time, Embedded, Modular, Action-oriented, Parallel Processor Project) [53], is a Swedish research project aimed at obtaining a massively parallel computer architecture, which can be adjusted to the applications. A prototype of a software configurable processor array is built. The computer consists of a number of computing modules, which are each a SIMD computer on its own, containing an array of bit-serial processing elements. Reconfigurability is obtained by implementing the PEs in field programmable gate arrays (FPGA). The inter-PE communication is simple, with broadcast and nearest neighbour connections.

More complex processing elements are used in the GF11 computer, which is an experimental SIMD machine built at IBM Research Laboratory [92]. The GF11 consists of 566 32-bit
processing elements, each with a capability of 20 MFLOPS, making a total of 11 GFLOPS (as the name tells). It was not built primarily for ANNs, but has shown to be one of the most powerful computers for ANN simulations. Each processing element has 256 registers, SRAM and DRAM, and is connected in a Benes network together with a disk system.

These first examples of highly parallel computers are SIMD machines. Among the computers containing from $2^8$ to $2^{12}$ processors are also MIMD architectures, which are typically built from more complex PEs.

Ariel is a multiprocessor architecture designed by Texas Instruments, Dallas, USA [13]. It is based on a hierarchical network of up to several thousand coarse-grained processing modules. A module is composed of a large primary storage of 32 M 32-bit wide words, a 32-bit digital signal processor (Texas Instruments TMS320C30), a 32-bit general purpose processor (Motorola MC68030), several high speed communication ports, and a dedicated disk memory unit. Ariel modules are interconnected by a physical bus hierarchy. Fifteen processing modules are grouped together on a common backplane bus. An expansion module is included within each group, to link groups of modules. At the next level, groups of 15 expansion modules can be connected to a common bus, to form a hierarchy of 225 modules. Groups of 225 modules can also be connected. This global bus hierarchy is called the broadcast bus and carries the communication between the modules. The designers argue that bus-oriented communication provide reasonably uniform performance over variations of neural networks, but is not optimum for networks that have uniform connectivity and data flow. It is also concluded that it is the computing power, not the communication bandwidth, that will limit the performance of a neural network simulator.

An early project developing hardware for ANNs, was the design of the NETSIM Neurocomputer, through a collaboration of Texas Instruments (UK) and Cambridge University. The NETSIM system consists of a collection of low cost neural network emulator cards physically connected in a 3-dimensional array [21]. Each NETSIM card consists of both a commercially available processor (80188), and a custom designed chip for fast multiply-accumulate operations, and for addressing the large quantity of memory. The communication of data is pipelined with the computations, for communication to add only a minimum overhead to the solution time. One possible configuration is a system of 125 modules in a 5x5x5 array.

Another high performance machine for computational intensive applications, is the Warp Computer developed at Carnegie Mellon University in 1984-87 [1]. Ten high performance custom designed processors are connected in a linear systolic array. Each cell/processor has a microprogrammable controller, a floating point multiplier and a floating point adder both with a capacity of 5 MFLOPS, and local memory. Communication between adjacent cells can be conducted in parallel over two independent channels, a left-to-right X channel and a bidirectional Y channel. The computational speed of 17 MCUPS on NETtalk [68], was at the time reported (1987) the highest performance achieved for ANN learning.
3.3. Other Architectures

Most moderately parallel neurocomputers have been designed around a number of general commercially available microprocessors. Most popular are digital signal processors (DSPs) and some RISC processors. These are designed for fast sum-of-product computations, and are well suited for ANN simulations, especially since the later generations feature high performance floating point processing.

The RAP (Ring Array Processor) is a multi-DSP system for layered network calculations, developed at the International Computer Science Institute, Berkeley, California [59]. The PEs are built around a TMS320C30 DSP, the same processor as used in the Ariel neurocomputer [13]. Each PE is equipped with a significant amount of local memory, but is otherwise kept simple to allow space for four nodes per circuit board. Field Programmable Gate Array (FPGA) technology is used where possible, for compact design and to allow future enhancements. Both the memory controller and the interprocessor communication are implemented in FPGAs. The PEs are connected in a low-latency ring. The design strategy has been to achieve the computational power by using several high-performance floating-point processors, not from complex interconnection schemes or sophisticated peripheral circuits. The performance is 128 MFLOPS per board. A 20-node system (5 boards) can achieve 200-300 MCPS for feed-forward propagation, and 30-60 MCUPS for full learning cycle in a two-layer network (matrix size 128×128) [59].

In this category Transputers can be included. Transputers are single chip 32-bit microprocessors with on-chip RAM and four bidirectional communication links. By connecting these links, several topologies can be realised, but 2-dimensional mesh architectures are most common. The current generation of Transputers is capable of at least 10 MFLOPS and have 16 K bytes of memory [57]. The interprocessor communication links have the capacity of only 20 bits per second, which may be a limitation. Lately, through-routing without processor involvement has been supported. Various Transputer based machines exist, in configurations with from a few (4 or 8) to several thousands processing elements [86, 3, 17]. In [94], the issue of processor topologies are addressed by testing a parallelisation algorithm on a 64-Transputer system configured as a ring, torus, and hypercube.

3.3 Other Architectures

Supercomputers, most often pipelined vector processors, contribute to the various hardware solutions for ANN simulations. Supercomputers are optimised for matrix operations on large data sets, and exist in variants with from 1 to 16 processors. High speed is achieved by having special register banks (vector registers), which support pipelining of functional units. They also have very short clock cycles (typically 4-8 ns), and use multiport memory to increase memory bandwidth. Supercomputers do not take advantage of the inherent
parallelism and distributed computing in ANNs, and are usually not cost-effective solutions for such applications.

Best known is the Cray series of computers. An implementation of backpropagation on Cray-2 resulted in 7 MCUPS [66]. The performance on a Cray X-MP was reported to 50 MCPS in the DARPA neural network study [11].

In the last few years, several Japanese neurocomputers have been designed. When Hitachi announced their wafer scale integration neural network, it was the world's fastest neurocomputer. It is constructed of eight silicon wafers, each containing 144 neurons, and is used in conjunction with a workstation. The learning speed is reported to be 2300 MCUPS for a 1152 neuron backpropagation network [95].

The probably most powerful general-purpose neurocomputer today is SYNAPSE-1 from Siemens [90], which has peak performance of $5.1 \times 10^9$ CPS. SYNAPSE-1 consists of four separate boards, one that performs the computational intensive operations, another executing the remaining operations, a high bandwidth memory board for the weights, and a controller board for the coordination of the other ones. The computational power is provided by a 2-dimensional systolic array of 8 neural signal processors, MA16, which are full custom VLSI. Each MA16 itself contains 4 processing modules, and is accompanied by a local memory for intermediate data. Unlike most of the neurocomputers, which are built from a number of modules containing a processor and local memory, SYNAPSE-1 has a separate weight memory. The computer can be scaled up according to the requirements of the application, by adding more MA16 boards and memory boards.
Chapter 4

Hardware Design Considerations

ANN simulations require a huge number of computations and communications. Although speed is of primary concern in the design of a neurocomputer, there is a tradeoff between performance and cost. For a research computer this means to get the optimum computing power from the available resources. For neurocomputers, which are tailored to one specific application area, another tradeoff is flexibility versus speed. The flexibility reflects the computers ability to adapt to various ANN models and applications.

One design goal for RENNS is that it should be flexible enough to support most ANN models. In addition, the communication should be reconfigurable for experimentations with different communication architectures. For optimal design, a balance of computing speed, memory capacity, and communication bandwidth, must be ensured.

In this chapter, the choice of processors, memory requirements, and the interprocessor communication will be discussed.

4.1 The Processors

Although there are several general-purpose programmable neurocomputers built from custom VLSI [1, 21, 90], the most flexible and cheapest solution is to select a commercially available processor.

It has been shown that fast matrix and vector operations are fundamental in ANN simulations. In addition, the processing elements must calculate (or look-up) the nonlinear functions associated with the neurons.
Several Digital Signal Processors (DSPs) and RISC processors offer high numerical capacity, and their performance is rapidly increasing with new products. In 1990, the most powerful DSPs had the performance of 30–40 MFLOPS (Texas Instruments TMS320C30 and Motorola DSP96002), a computing capacity that is now doubled. The main objections against using DSPs are their relatively small address space, the need for expensive high-speed memory to utilise the computing capacity, and a larger instruction set than necessary for ANN simulations.

Besides having high numerical performance and a large address space for the weight memory, the processors must support the learning and weight updating phase with sufficiently large numerical precision.

### 4.1.1 Numerical Precision

Processors have a varying degree of numerical precision, from bit serial up to floating point formats of more than 100 bits. DSPs range from 8-bit integer to 32-bit floating-point processors.

A majority of ANNs require floating-point performance [45, page 386]. It is the weight updating that require high precision. Representing weights at an insufficient resolution may cause convergence problems. Knowledge about the required resolution should be utilised in hardware implementations, to minimise the number of bits in the representations, thus saving space in VLSI circuits.

In [34], Holt and Hwang present an analysis of the finite precision error of neural network hardware implementations. The goal is to have the same learning convergence and accuracy as that of 32-bit floating-point at limited precision fixed-point. The required number of bits in the representations depends on the accepted error, which is divided into soft convergence with error (square error function) less than $2^{-3}$, and hard convergence with error less than $2^{-4}$. The activation values, which refers to the precision of the inputs, targets, and outputs, require a precision of 8–9 bits for soft convergence, and 9–10 bits for hard convergence. For the weight precision, which are weights, biases, weight updates, and delta values, the minimum resolution is 13–15 bits for soft convergence and 15–16 bits for hard convergence.

In [21] and [27], it is also concluded that 8 and 16 bits are sufficient for the activation values and weights, respectively.

Others have suggested that 32-bit resolution is necessary for some nets to guarantee convergence. According to Kung [45, page 386] word length of 32-bit floating-point is often required in signal processing applications, while a word length of 8 to 16 bits fixed-point
would suffice for many image processing applications.

In [30, page 128–131], Hecht-Nielsen concludes that one of the most commonly used algorithms, backpropagation, is very sensitive to the precision and number range used. This is due to shallow slopes and flat areas in the backpropagation error surfaces, and a weight resolution of 32-bit floating-point is suggested.

Although the exact calculations of Holt and Hwang show that the required accuracy can be obtained with limited resolution, a general neurocomputer should support the numerical precision required by a range of ANN applications. This will be ensured by using 32-bit floating-point processors.

### 4.2 Storage Capacity

A typical ANN processing unit consists of local memory in addition to the computing part. A study made by DARPA [11] shows that the ratio between storage capacity and processing speed in biological neural networks is independent of the network size. This is illustrated in Figure 4.1, where the storage capacity (total number of connections) and the processing speed (connections per second) for some small living creatures is plotted into a diagram. For comparison, the same characteristics for a few computers are also marked on the figure.

These results indicate that the amount of memory on each processing unit should be balanced to match the processing speed, thus attempting to fit the resource demands of the applications. Assuming that the linear relationship is valid for neurocomputers, using this as a guideline for determining the amount of memory will give a balanced system also for scaled-up versions of a neurocomputer.

### 4.3 Communication Bandwidth

ANN simulations are usually computational bound, but still an efficient communication scheme is essential for not slowing the processors. The communication structure and bandwidth become more important as the number of processors increase.

The communication bandwidth, which should balance the processing speed depends on:

- the number of communication channels attached to each processing module,
Figure 4.1: Storage capacity versus computing speed in neural networks (from DARPA study [11]).

- the bandwidth of the individual communication channels, and
- the communication network interface on each module.

If the processing part and the communication network interface on a module can operate in a pipelined manner, this additional parallelism can reduce communication overhead. The interface must then provide enough buffer capacity to avoid the application to become communication bound.

In the existing parallel neurocomputers, the communication interface is of various complexity. If custom designed processors are used, the communication interface can be integrated into the chip. This is also the case with some commercially available processors, like Transputers [57], and the last generation of DSPs from Texas Instruments (TMS320C40) [84]. Early generations of Transputers suffered from low floating-point performance and communication bandwidth. The more recently announced INMOS T9000 is specified to 25 MFLOPS and an effective data rate of up to 17.6 Mbyte/second on each of the four communication channels. The TMS320C40 can do up to 50 MFLOPS and the six parallel ports have each a capacity of 20 Mbyte/second. The on-chip FIFOs and the six-channel DMA coprocessor allow for concurrent I/O and CPU operations.

For more sophisticated communication schemes, or to further minimise processor involvement, a separate communication interface must be designed.
4.4 Processor Topologies

The choice of certain structured processor interconnections can significantly enhance the performance of parallel processing systems [45, 61].

Several processor topologies have been proposed for and used in ANN simulations. The most common are broadcast bus, line and ring architectures, 2- and 3-dimensional mesh networks, and hypercubes. For the mesh, the edges can be wrapped around in a toroidal architecture. Figure 4.2 illustrates the most common processor topologies.

![Diagram of processor topologies](image)

Figure 4.2: Some of the most common processor topologies used in ANN simulations.

Generally, the processor topologies suggested for and used in neurocomputers are regular
interconnection networks, which reflect the regular structures of the ANNs. A common solution in neurocomputer is to have one processor network for high-performance communication of large amounts of data, and in addition a common bus or other network for general communication and control information, including synchronisation and diagnostics. MasPar [56] and CM-2 [32] have both a global network for irregular communication, and fast regular interconnect for problems with regular communication demands.

4.4.1 Broadcast Communication

Broadcast bus is the simplest multiprocessor communication scheme, and is an efficient interconnect architecture for ANNs with large fan-in or fan-out.

If training example parallelism is used, broadcast communication is a good choice, as the resulting weight matrix for each training example must be distributed to all the other processors involved, either by batch or epoch updating.

For neuron and weight parallelism, the communication is more localised, and it is less obvious how broadcast communication can be utilised. One possible mapping scheme using neuron parallelism is to compute one layer at a time. In the forward pass, each row of the weight matrix is mapped onto one processor. In each step of the matrix-vector multiplication, one processor broadcasts its activation value to all the other processors. The processors then multiply the received value with a weight value from the corresponding column of the weight matrix, and accumulate the result. If there is a backpropagation phase, the delta values can be broadcasted in the same manner as the activation values, and multiplied with a weight matrix column stored in each processor. These numbers are summed across all the processors instead of within each processor as in the forward phase. Summing across all the processors requires a number of broadcasting steps corresponding to the number of neurons in the layer.

For broadcast bus, the number of processors that can be attached to a single bus is restricted due to electrical constraints. This limits the system's scalability. This limitation can be overcome by connecting several busses in a hierarchical structure, or in other ways clustering the processors, to keep the heaviest part of the communication within a group of processors. In the Ariel neurocomputer, the modules are connected in a physical bus hierarchy [13]. In BSP400, 16 processors are grouped on each board, where each board is connected to a common bus [31]. The modules in CNAPS communicate through broadcast busses, allowing the modules to be connected in two dimensions, to mimic the layered structure of many ANNs [27]. In the Mod 2 neurocomputer [60], the processing subsystems called hardware layers are organised in groups of six, and share six busses for local communication. Four additional busses are used for global communication.
4.4.2 Ring and Linear Array Architectures

The reason for choosing ring and linear array architectures is to avoid the possible bottleneck of a single bus, and to take advantage of the distributed nature of neural networks.

In a ring, each processor can communicate with its left or right, or both neighbours, depending on whether the ring is unidirectional or bidirectional. The last processor has the first as its right neighbour, and vice versa. It is shown that most neural network algorithms can be mapped onto a ring architecture [47]. Linear array is simply a line of processors, with bidirectional communication between the right and left neighbours, except at the end points.

Many neurocomputers are ring architectures because of the simple and efficient communication, and the possibility to operate in systolic mode. Examples are NEP [10], RAP [59], MUSIC [89], and SMART [51]. The latter allow operation in both broadcast and systolic mode. A parallel machine for computational intensive applications that has shown high performance in ANN simulations, is the Warp Computer [1, 68] developed at Carnegie Mellon University, with its commercial counterpart iWarp [67]. The Warp machine is a linear systolic array. Communication between adjacent processors can be conducted in parallel over two independent channels, one unidirectional and the other bidirectional.

4.4.3 Two-Dimensional Mesh Structures

Two-dimensional mesh architectures are flat communication topologies where the processors communicate with more than two of their neighbours. Most common is four communication lines per processor, but six and eight lines per processor are also used.

The quadratic mesh is most widely used. The communication channels can be either bidirectional, or the edges can be wrapped around to reduce the communication requirements to unidirectional channels. This topology is especially attractive to Transputers, because of their four integrated bidirectional communication channels. Two efficient mapping strategies, TLA (Toroidal Lattice Architecture) and PLA (Plannar Lattice Architecture) [15], have been developed for quadratic mesh. A prototype of TLA has been implemented on 4×4 Transputers, and an implementation on a grid of some thousands powerful RISC processors is planned [15].

DAP, Distributed Array Processor, is a series of commercial 2-dimensional fine grained SIMD computers. In addition to the mesh, row and column highways are used to speed up long distance communication. Various ANNs implementations on DAP are reported in [65].
The regular interconnection network in MasPar, $xnet$, is a 2-dimensional architecture, which allows each processor direct communication with eight of its nearest neighbours to the north, south, east and west, and in the diagonal directions [56].

### 4.4.4 Higher Dimensional Architectures

The NETSIM neurocomputer system consists of a collection of low cost neural network emulator cards physically connected in a 3-dimensional array [21]. A separate communication chip on the board passes data packets to the destination along the shortest route at a speed of 10 Mbit/s. The idea is to view a neural system as a number of subnets, where a subnet should be retained in one board. A 3-dimensional system was chosen as a compromise between average forwarding distance and complexity. It is also being consistent with the layered structure of many ANN problems.

Multidimensional architectures, like hypercubes, allow a much more general communication pattern than any of the previously discussed architectures. The distance between two processors is maximum $\log_2 N$ steps, where $N$ is the number of processors.

### 4.4.5 Communication by General Routing

Some massively parallel architectures provide support for general routing, e.g. CM-2 [32] by packet switching, and MasPar [56] by a multistage crossbar switch. Utilising these facilities is a straightforward method on these computers, but general routing is normally much slower than regular communication and not well suited to the communication patterns in ANNs, maybe with the exceptions of sparsely connected networks.

### 4.4.6 Other Communication Schemes

Although not built especially for ANN simulations, GF11 [92], IBMs experimental machine, was for some period one of the fastest neurocomputers announced. In GF11, the processors and a disk system are connected in a Benes network, which is a nonblocking crossbar switch.
4.4.7 Speeding Up the Simulations Through a Pipelined Architecture

For increased performance, the stages in a neural network simulation can be pipelined. This can be done with a topology based on several rings, as will be discussed in an example later in this thesis. For a mesh topology, it is shown in [82] that instead of mapping a multilayer neural network onto a single mesh, the communication network can be extended and pipelined, thereby creating a more scalable processor topology.

4.4.8 Comparison of Processor Topologies

The optimum communication topology for a given application minimises the communication overhead. Broadcast bus, ring and mesh architectures have been investigated to optimally trade off the processing time and the number of processors for a neural network with totally interconnected layers [2]. The processing time includes both calculation and transmission time. As expected, the calculation time decreases as the number of processing elements increases in every topology. For broadcast bus and ring architectures, the transmission time increases monotonically as new processing elements are added, but in case of the mesh there exist an optimum number of processing elements.

4.5 Implementation Technologies

For both the computational and communication part of a neurocomputer, there are several alternative implementation technologies. A processor based solution, where the computations and possibly also the communication part is solved in software, is most flexible. The processor can be either commercially available or custom designed. Speed improvements can usually be achieved by tailoring a system to a specific application, using specialised circuits.

As a compromise between a dedicated and a software based solution, reconfigurable logic has become a powerful alternative. Dynamically reconfigurable logic facilitates fast and easy in-system reprogramming of the circuits. An increasing variety of technologies, architectures, and vendors exist in this market [5, 39, 72]. Field Programmable Gate Arrays (FPGAs) like Xilinx LCAs (Logic Cell Arrays) [93] are most flexible and powerful, but the clock rate and delays are highly dependent on the design.

In addition to high speed, one of the major design goals for RENNS is flexibility to exper-
iment with different communication architectures. Dynamically reconfigurable hardware as implementation technology should therefore be considered for parts of the system. In a prototype, utilising FPGA technology gives flexibility within the limits of the complexity of the circuits used, and reduces development time and cost.

4.6 Performance Criteria

4.6.1 Measurements and Metrics

Raw computing power is often given in FLOPS (Floating Point Operations per Second), but this information does not consider communication overhead and the sequential fraction of a program. This metric can only serve as an upper limit for the performance. The utilisation of this potential can be highly varying with different ANN models and problem sizes.

The most common metrics for speed measurements in ANN simulations are connections processed per second (CPS), and connections updated per second (CUPS), for the learning phase. Updating the weights typically takes 2-5 times longer than the mapping from input to output. For these measurements to be of more value than another upper performance limit, the ANN model, problem size, and nonlinear function should be given as additional information. Comparing the CPS measure with the multiply-accumulate or FLOPS measure, show to which degree the computer architecture suits the the application, and opposite.

Table 4.1 shows speed characteristics for a selection of various computer systems used in ANN simulations.

The computers differ in size and cost, and they are also affected by the technology available at the time the system was being built. These factors highly influences the speed measurements given in CPS or CUPS. Instead the number of CPS or CUPS can be compared to the raw computing speed in FLOPS. These relationships give information on the computer system's suitability to the ANN application, and reflect the balancing of computational capacity, and memory and communication bandwidth. These numbers, too, are not necessarily good metrics, unless accompanied with information on the application from which the measurements were obtained. The CPS/CUPS measured from running a benchmark can be significantly different from the results achieved from applications tailored to the computer system.

The numerical precision also influences the speed. CNAPS, with 256 custom designed chips on four boards, shows a very high performance. The weight and activation value resolution
### 4.6. Performance Criteria

<table>
<thead>
<tr>
<th>Computer system</th>
<th>Communication architecture</th>
<th>MFLOPS</th>
<th>MCPS</th>
<th>MCUPS</th>
<th>Utilisation of computing power</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-2 64 K PE Conf. [75]</td>
<td>Array Processor in Hypertorus</td>
<td>10 000</td>
<td>1 300</td>
<td>325</td>
<td>7.7 FLOPS/CPS 30.8 FLOPS/CUPS</td>
</tr>
<tr>
<td>GF11 566 PEs [92]</td>
<td>3 stages Benes Network</td>
<td>11 000</td>
<td>901</td>
<td>12.2 FLOPS/CUPS</td>
<td></td>
</tr>
<tr>
<td>CNAPS 256 PE Conf. [27]</td>
<td>Various, 3 Broadcast Buses</td>
<td>5 000</td>
<td>1 000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAP 20 PE Conf. [59]</td>
<td>Ring</td>
<td>640</td>
<td>300</td>
<td>60</td>
<td>2.1 FLOPS/CPS 10.7 FLOPS/CUPS</td>
</tr>
<tr>
<td>Warp 10 PEs [68]</td>
<td>Linear Array Processor</td>
<td>100</td>
<td>17</td>
<td></td>
<td>5.9 FLOPS/CUPS</td>
</tr>
<tr>
<td>SMART 8 PE Conf. [51]</td>
<td>Ring and Broadcast</td>
<td>160</td>
<td>76.8</td>
<td>19.2</td>
<td>2.1 FLOPS/CPS 8.3 FLOPS/CUPS</td>
</tr>
<tr>
<td>MUSIC 45 PE Conf. [89]</td>
<td>Ring</td>
<td>2 700</td>
<td>427</td>
<td>203</td>
<td>6.3 FLOPS/CPS 13.3 FLOPS/CUPS</td>
</tr>
<tr>
<td>SYNAPSE-1 4.8 PE Conf. [90]</td>
<td>2-D Systolic Array</td>
<td>5 100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Speed measurements for ANN simulations on different computer systems.

is 16 and 8 bits, and the computer has not floating-point facilities, like the other computer systems listed in Table 4.1.

The CPS measure can be achieved in various ways. For the recall phase of a trained backpropagation network, communication can be reduced to zero by handling a number of patterns in parallel, giving one pattern as input to each processor. If the nonlinearity is a simple function or efficiently implemented, the CPS will be approximately half the number of FLOPS. For other models, like the Hopfield ANN, the neurons are usually evenly distributed among the processors. Included in the CPS measure can then be loading the input pattern, calculating the outputs, and convergence check, which requires all-to-all communication. For the computers listed in Table 4.1, the FLOPS/CPS varies from 2.1 to more than 7.
The efficiency of the weight updating is also susceptible to how much of the I/O is included in the measurements. The FLOPS/CUPS measure varies from around 10 to 30. Updating the weights is typically 3 to 7 times more time-consuming than for reading the weight values.

### 4.6.2 Speedup and Efficiency

In the design of a parallel computer we strive for scalable performance, which is close to linear speedup when adding more modules to the system.

Important in modelling the speedup is the uniprocessor timing, \( t_1 \), and the execution time for the same problem on \( p \) processors, \( t_p \).

A straightforward method for speedup measurements is to express speedup as the relationship between uniprocessor and multiprocessor timing:

\[
s(p) = \frac{t_1}{t_p}
\]

(4.1)

where we want \( s(p) \) as close to \( p \) as possible.

The efficiency of the implementation is defined as:

\[
e = \frac{s(p)}{p}
\]

(4.2)

In a study of speedup and efficiency of various parallel computers solving the same numerical problem, the efficiency is varying from 0.03 for the Intel Delta with 512 processors, to above 0.9 for some computers with six or less processors [38, page 115].

### 4.6.3 Benchmarks

Benchmarks in ANN simulations can be used in qualitative comparisons of algorithms, but they are also important in speed measurement. Performance measurements are likely to be different when achieved from customising an ANN to a parallel architecture, compared to running a standardised application. The two benchmarks NETtalk [74] and Bignet [69] described in Section 2.6.1 are often used in speed measurements.
Chapter 5

The RENNS Architecture and Implementation

At the Norwegian Institute of Technology, a first version of the RENNS computer system is now operative. The first section in this chapter focuses on the RENNS architecture, how the system was planned at top level, independent of the implementation. The second part of this chapter concerns the implementation, with a description of the logic and the components that form the RENNS modules, including motivation for the choice of components and other implementational issues.

5.1 Design Goals

ANN problems are computational intensive. Hence it can be argued that the processing part should be given priority above complex interconnection schemes or sophisticated peripheral units. Although high speed is of primary concern for the RENNS computer system, a major design goal is to facilitate experimentation with different communication architectures. High degree of flexibility is important for research purposes, and in prototyping.

For ANN simulations on a coarse grained multiprocessor, the communicated data are mainly vectors of various lengths. Because of this, the communication network is optimised for transmission of and operations on data streams. Point-to-point physical connections are used between modules to obtain a scalable and flexible communication system.
5.2 Building Blocks

The two main building blocks of the RENNS architecture are *processing modules* and their attached *data streams*. Both are optimised to operate on vectors. A processing module accepts vectors as input, performs vector operations, and produces new vectors as output. The vector is fetched from either the local memory on the module or from a data stream. A data stream functions as the communication channel between two or more processing modules, and is reconfigurable to several modes of operation.

Figure 5.1 shows a processing module with its associated data streams. The number of data streams attached to each processing module is not fixed, and may vary between implementations.

![Diagram of a processing module with data streams](image)

*Figure 5.1: A processing module with its attached data streams (from [50]).*

The data streams can operate in all modes shown in Figure 5.2, which are the following:

- Receiving.
- Sending.
- Bypassing, one incoming data stream is redirected to an outgoing data stream.
- Broadcasting the same information on several or all data streams.

![Diagram of different configurations for data stream operation](image)

*Figure 5.2: Different configurations for data stream operation (from [50]).*

Through the data streams, the processing modules can be connected in various communication structures. Depending on the parallelisation and mapping of a particular ANN, the interprocessor network can reflect the communication requirements. RENNS can be configured into several variations of ring and mesh architectures, as well as other processor
topologies, like hypercube. It also opens for experimenting with different communication architectures. A possible configuration is to have two data streams receiving input vectors into the processing module, and four other data streams to broadcast the resulting vectors to other modules.

How these flexible building blocks are realised, and their limitations, will be discussed in the following sections.

5.3 The Modules

A RENNS module consists of a processing and a communication subsystem. In this implementation, the number of data streams attached to each module is limited to eight data streams. The different operational modes shown in Figure 5.1 are all implemented, but there are restrictions on the independent operation of the streams. A block diagram of a RENNS module and its data streams is shown in Figure 5.3.

Figure 5.3: Block diagram of a RENNS module (from [48]).

Besides the modules, a typical configuration will have different support boards, including interface boards to the host computer and special communication boards, which is discussed in Appendix A.
5.4 The Processors

5.4.1 Choosing a Processor

In ANN simulations, the computations are the core of the task to be performed and should execute as fast as possible. Several processors were evaluated with respect to the following factors:

- high floating-point performance,
- availability,
- integration in system,
- address space, and
- cost.

The choice of using floating-point processors excluded several processors at an early stage. Floating-point processors are for many problems faster and easier to program than fixed-point processors. In Section 4.1.1, it was concluded that up to 32-bit floating-point precision is necessary during training of some ANNs. For these reasons, the processors considered were all fast 32-bit floating-point DSPs or RISC processors. Transputers were excluded because they could not match the speed of the DSPs and other RISCs. The T800 was capable of a maximum of 2 MFLOPS clocked at 30 MHz. The DSP32C from AT&T, Fujitsu MB86232, TMS320C30 from Texas Instruments, and Motorola DSP96002 were studied more thoroughly. All these processors had high floating-point performance and an address space of more than 1 M 32-bit words. Both AT&T DSP32C and Motorola DSP96002 were considerably more expensive than the two others. The Motorola DSP96002 is the most powerful processor in this selection, with many registers, and a large and clean instruction set. Unfortunately, the processor was not available at the time detailed design of the modules was started.

5.4.2 The TMS320C30

The TMS320C30 from Texas Instruments is used in the first version of the RENNS for several reasons:
5.4. The Processors

- It has a high floating point performance. Clocked at 33 MHz the processor can achieve 33 MFLOPS for inner products.

- Efficient matrix and vector operations are supported.

- It can support up to 16 M 32-bit words of external memory and has fast, internal RAM banks, as well as a small instruction cache.

- It has two external buses, which makes it easier to utilise data from the main memory and the communications channels simultaneously.

- Affordable, the cost per processor was approximately 200 US $.

A short description of some features of the TMS320C30 follows. More information on the processor can be found in the TMS320C3x User’s Guide [83].

Cycle Time and Performance

The processor has a 60-ns cycle time. The performance of 33 MFLOPS is obtained by having two functional units operating in parallel, one for floating-point multiplication and another for floating-point addition. Most of the instructions are executed in one cycle (16.7 MIPS), but a deep pipeline causes unconditional jumps and subroutine calls and returns to take 4 to 5 cycles.

Internal Architecture

The data types supported are 32-bit floating-point and 24-bit fixed-point numbers. Because of several internal busses, parallel multiplications and additions/subtractions can be performed on four operands in a single cycle. Two auxiliary register arithmetic units operate in parallel with the ALU and are used for address generation.

The internal memory banks add significant capacity to the total memory bandwidth, with one 4 K word (32-bit) single cycle dual access ROM block, two 1 K word single cycle dual access RAM blocks, and a 64 word instruction cache. Dual access means that the internal memory can be read or written twice per instruction cycle.
External Interface

The TMS320C30 has a dual bus architecture, the primary bus and the expansion bus. Both busses transfer 32-bit wide data. The primary bus has 24 address lines whereas the expansion bus has 13. For RENNS, the parallel bus operation can be utilised to divide the total bus bandwidth between the main memory and the communication network. The main memory is accessed through the primary bus, while the communication subsystem is attached to the expansion bus.

The external memory capacity is up to 16 M 32-bit words (64 Mbyte). With no wait-states, and both the busses operating in parallel, the maximum attainable external bandwidth is approximately 128 Mbyte/s. Read cycles at full speed on the primary bus take one instruction cycle (60 ns) per read operation. Write operations and write after read cannot be done in one clock cycle and require 2 or 3 cycles, respectively. Read and write operations on the expansion bus are slower and always take a minimum of two clock cycles.

Wait-states can be generated either internally by programming a wait-state generator, or externally by giving the control of the ready-signals to the peripheral units.

Strict timing constraints in the processor require SRAM with access time of maximum 20 ns, to operate at full speed.

For DRAM accesses, it is necessary to do some address decoding to determine if an address lies within the same bank as the previous address. Otherwise an extra wait-state must be inserted. To obtain full speed using DRAM will not be possible, unless one has some knowledge about the memory access patterns. Then the addresses can be generated one cycle in advance, to compensate for the strict timing constraints of the processor. Extra logic is required to distinguish between a sequence of successive addresses and a random addressing scheme. The fundamental operation in ANN simulations is computing of inner products, and the access pattern will most often be successive addresses, fetching either a matrix or a vector from memory. It may be advantageous taking the effort to design proprietary logic to achieve full speed read operations on large amounts of cheap DRAM. Currently, DRAM accesses require minimum one wait-state.

Peripherals

The processor has 2 serial ports that support 8/16/32-bit transfers. In addition, the processor is equipped with two 32-bit timers, which can be used for time measurements, two general-purpose external flags, and four external interrupts. One of the interrupts will be used by the communication subsystem.
5.5 Memory Configuration

Most ANNs require a substantial number of variables, such as weight matrices, state vectors, biases, accumulated errors etc. The memory capacity should be sufficiently high for most applications, and balance the processing power of the processor. The memory bandwidth should ideally be able to support the processors operating at full speed.

The amount of memory sufficient for most ANN models is not easy to predict. The DARPA study [11] discussed in Section 4.2 shows a linear relationship between the processing speed and the memory capacity in biological neural networks. The storage capacity in biological neural systems is measured in number of connections, which are the long term memory. The DARPA study is used as a guide to determine the amount of memory on the modules.

Since the operating speed of the modules is dependent on memory access time, the memory speed will also influence the amount of memory needed, to balance the computing power. For not to slow the processors, a large amount of SRAM is required. Such amounts of SRAM are not economically feasible, and DRAM is used for the large quantities of memory. It may still be possible to have zero wait-states for sequential DRAM access. This will be rather complicated, because the processor must be notified early in the cycle (8 ns after valid address) whether to insert an extra wait-state or not. In this first prototype, it was decided not to add this feature, for a simpler design. As a consequence, at least one wait-state must be inserted for each memory read operation. An additional wait-state is inserted when an address causes a memory bank switch.

Each processing module is equipped with one TMS320C30 DSP with a peak performance of 33 MFLOPS. For most simulations, the weight precision will be 32-bit. If DRAM is used for weight storage, at least two clock cycles are used to read one 32-bit word from weight memory, slowing the processor with at least a factor of two for inner product computations. The external bandwidth will then be approximately 16 M 32-bit words per second, indicating that the number of weights on a module should be in the order of 16 Mwords, or 64 Mbyte, which is a large amount of memory.

As a compromise between large storage capacity and high memory bandwidth, the RENNNS modules have a memory hierarchy where the most frequently used information is tried placed in fastest possible memory. The memory hierarchy, consisting of DRAM, SRAM, and on-chip RAM, is shown in Figure 5.4.

The current implementation supports 4–16 Mbyte of dynamic RAM and 256 Kbyte–1 Mbyte of static RAM. The DRAM is placed in four banks of either 256 Kwords or 1 Mwords each. For the DRAM, standard SIMM-modules with static column are used. This storage capacity is less than the suggestions based on the DARPA study, but it is the maximum amount of memory afforded in this prototype. The 50-ns DRAM can follow a
speed of up to 2 clock cycles per operation. The internal memory banks, two 1 Kwords RAM blocks, allow two operations per clock cycle, and add significant capacity to the total memory bandwidth.

The DRAM is intended for storage of the large data structures such as weight matrices and state vectors. Most operations on these structures will access data in a sequential manner, which increases the memory bandwidth. Programs and scratch variables may be stored in external or internal static RAM, which does not exhibit performance penalties for nonsequential access patterns.

### 5.6 Clock Distribution

At an early stage, it was decided to have separate clocks on each module, and an asynchronous communication protocol, to avoid problems with clock distribution. It was considered too complicated to distribute the same clock to processors at a physical distance of some tens of centimetres apart. The synchronisation problem is then moved to the interprocessor communication. An advantage with an asynchronous design is increased flexibility. New modules with higher performance can be directly connected.

In the RAP neurocomputer [59], the modules operate synchronously. A common clock is distributed using a special cable carrying a differential ECL signal. The CM-5 [38] has up to 1024 modules operating in MIMD or SPMD fashion, but with the same clock distributed to all modules, for more efficient communication. This clock distribution is patented.
5.7 Interprocessor Communication

The design of fast, asynchronous interprocessor communication is a challenge. The communication bandwidth should be greater than the need for data exchange in most algorithms. Also, the reconfiguration abilities are implemented in the communication subsystem. During the design phase, it was decided to separate the processing and communication functionality both logically and physically, and an interface between the two subsystems was agreed upon. The communication subsystem is an independent board connected to the processor subsystem as a "piggy-back" board. An advantage is that new and older versions of both the processing and communication subsystems can coexist in the computer, as long as the interface between the two parts is unchanged.

The communication subsystem will be described into more detail in the following chapter.

5.8 The Processing Subsystem

In addition to the DSP and memory on the main board, the processing modules are equipped with other necessary functions. These are DRAM controller, EPROMs that contain start-up programs and the monitor program, serial interface for connection to a terminal, interrupt controller, "Watchdog"-timer, and a VME-bus interface. Figure 5.5 shows a block diagram of the processing subsystem of the RENNS module.

![Diagram of the processing subsystem](image)

Figure 5.5: The processing subsystem.

The backplane VME bus is used for distribution of program and data, connection to a LAN, and for controlling the processing modules.
Figure 5.6 shows a number of RENNS modules in a rack. The modules are connected through the backplane VME bus and the intermodule communication links (on the back). The cable that connects the module’s serial ports is a temporary arrangement for test purposes.

Figure 5.6: RENNS modules in a rack.
Chapter 6

The Communication Subsystem

6.1 Design Goals

Prior to designing the communication subsystem, several design goals were identified [50]:

- It should operate with a minimum of processor involvement.
- The communication bandwidth should at least match the bandwidth of the processor bus.
- All the operational modes defined for the data streams should be supported, and be dynamically reconfigurable during and between simulations.
- The subsystem should be optimised for communication in a ring bus, but not exclude other types of network topologies.

These points define the requirements to both speed and flexibility. High speed is achieved through a combination of fast circuits and careful design. For economical reasons, the fastest available circuits have not been selected for all functions. For many components used in RENNS, faster versions became available during the design period, but too late to be considered. The following description of the communication subsystem will therefore focus on the functionality. Improvements by using faster components and new technologies will be discussed later.
6.2 First Approach, with Serial Intermodule Communication

An issue considered early in the design phase, was the architecture and implementation technology for the interprocessor communication. Bus based solutions were discussed, along with a study of advanced commercial system bus products. Because of the shortcomings of bus based communication, alternative solutions were considered.

An interesting possibility was to use the TAXI (Transparent Asynchronous Xmitter – Receiver Interface) chip, for fast serial communication. The TAXI gives flexibility to form a range of communication topologies, and fits well with the decision of having asynchronous interprocessor communication.

The TAXI links are unidirectional serial busses, where data are transferred over a +5 V ECL serial link, which can be either coaxial cable or twisted pair. With some extra interfacing, optic fiber can replace the cable. The chip has a parallel TTL bus interface of twelve bits. Up to nine bits can be used for data, the rest are command bits. The chip provides automatic multiplexing between data and commands, by having one strobe for each type of information. Although the data has to be converted between parallel and serial format, the TAXI concept has satisfying performance. The throughput over a TAXI link were specified to 32–100 Mbit/s, or 4–12.5 Mbyte/s.

It was planned to equip each RENNS module with four transmitters and four receivers, the same number of channels as in the current implementation. A TAXI chip, connectors, and necessary logic and buffer capacity, form a small I/O module. The first sketches of the processor board had slots for eight I/O modules mounted vertically on the main board. From the processor, the TAXI modules can be accessed as I/O ports. To transfer word-wide (32-bit) data, packing and unpacking is required. Each module should have been equipped with some buffer capacity, to guarantee no loss of data if the I/O module in the other end is responding too slowly.

The TAXI chip is a small 28-pin package, but because of the need for additional logic and buffering capacity, the modules became quite complex. Another disappointment was to realize that the connectors for both coax and twisted pair were very expensive, and took nearly as much space as the TAXI chip itself. The high costs for the communication hardware indicated that alternative solutions should be considered. Serial communication using TAXIs is well suited for flexible communication topologies, especially if there is some distance between the units. In the RENNS case, the modules will be placed in a rack, and the distance between to communicating modules will be maximum 60-70 cm. Most of the cables will connect neighbour modules in a distance of about 10 cm. The use of serial links for this purpose cannot be justified, and the TAXI approach was given up. For scaled
up versions of the RENNS, serial communication can be an alternative between modules longer distances apart.

6.3 Testing the Bandwidth of Flat Ribbon Cable

Because the primary problem with the TAXIs was the expensive connectors and cables, the next idea was to try a simple and cheap solution, using flat ribbon cable and byte-wide communication.

To ensure high enough bandwidth and noise robustness with flat ribbon cable, this was tested in laboratory experiments. The test hardware basically consisted of an FPGA, an oscillator, and two 12-pin connectors, to make a loop with one flat ribbon cable. Byte-wide data were transferred over cables of various lengths, from 15 cm to 1 m, at different speed. The sender logic in the FPGA generated patterns verified at the receiving end, and the errors were counted. For short cable lengths, about 15 cm, error-free transmission was measured for frequencies up to 25 MHz. For cable lengths up to 1 m, 20 MHz was a limit. This transmission medium was robust enough for the noise in the hardware laboratory environment.

6.4 Current Implementation

After these successful experiments, it was decided to use flat ribbon cable, and to have the same number of I/O channels as proposed above, which was four incoming and four outgoing data streams.

For practical and economical reasons, it was decided to have one single I/O module containing eight data stream connectors, FIFO memory and the necessary control logic. The structure of the communication subsystem is shown in Figure 6.1. (The figure is taken from [50].)

The I/O module is connected to the lower half of the main board as a "piggy-back"-board. The communication subsystem printed circuit board is an eight-layer board. It is a full length Eurocard, but has the non-standard height of half a full height board, which gives the dimensions of 239.4mm × 116.8mm. Four of the eight layers are signal layers, the other are +5 V and ground layers, which also contain the distribution of clock and strobe signals. The clock distribution is designed for clock frequencies up to 50 MHz, for not to exclude the possibility to sample asynchronous incoming data with a rate of twice the data-rate of
the incoming data. The bottom layer, next to the processor board, is a ground layer. This ground layer was added in the second version of the communication subsystem PCB, to reduce the influence of noise generated from high speed components on the main board.

The interface between the processor and the communication subsystem is asynchronous, and the processor accesses the communication subsystem through memory-mapped FIFO-buffers on the expansion bus. Data is sent by writing to one of the FIFOs. The communication subsystem takes care of the data transmission without further processor involvement. Incoming data on any of the data streams are notified the processor either by an interrupt or by setting flags. The latter require the processor to poll a status register.

Up to eight data streams can be connected to other modules. The data streams are independent, unidirectional (half-duplex) and byte-wide. Routing facilities are needed to direct data between the FIFO-buffers and the specified data streams.

More details on the communication subsystem are documented in the following manuals, "RENNS Communication Subsystem – Hardware Guide – All you never want to know about the RENNS Communication Subsystem" [79], "RENNS Communication Subsystem – Configuration Guide – Description of the existing configurations" [78], and "RENNS Communication Subsystem – User’s Guide" [76].
6.5 Use of Reconfigurable Logic

A design criterion for the communication subsystem was to balance the bandwidth of the processor bus. Another criterion was that communication should take place with a minimum processor involvement. The bandwidth requirements ruled out a programmed implementation of the communication functionality, and led to a dedicated hardware solution.

As much as possible of the logic is implemented in field programmable gate arrays (FPGA). The use of field programmable logic allows certain design changes to be made fast, and without extra cost, which is not easily done when using standard logic or mask programmed circuits.

The wide range of ANN applications has different communication demands. Use of field programmable gate arrays gives another level of flexibility for experimentation with different solutions.

All the control logic in the communication subsystem is implemented in in-system reprogrammable Logic Cell Arrays (LCAs), manufactured by Xilinx [93]. The architecture of an LCA is similar to that of other gate arrays, with an interior matrix of logic blocks, and a surrounding ring of I/O interface blocks. Interconnect resources occupy the channels between the rows and columns of blocks. The functions of the LCA configurable logic blocks and I/O blocks, and their interconnection, are controlled by a configuration program stored in on-chip SRAM. The configuration program is either automatically loaded from an external memory (PROM), on power-up or on command, or the LCAs are programmed by a microprocessor, as a part of system initialisation. Configuration of the RENNS communication subsystem can be done in both these modes, at system start-up, or on command during operation.

The performance of an LCA is determined by the circuit speed grade and the utilisation of the circuit, which is design dependent. The circuit speed grade is specified by the maximum toggle rate for a logic-block storage element configured as a toggle flip-flop. All the LCAs used in the communication subsystem are of the 3000-series, with speed grade of 125 MHz. These circuits were, at the time the components were ordered, the most cost effective devices for this application. The 3000-series logic blocks contain two flip-flops, and can form either one logic function of five variables, or two functions of four inputs each. All the I/O blocks have input and output latches. The 3000-series comprises circuits with logic densities from 2000 to 9000 gates. See [93] for a more thoroughly architectural description of the LCAs.
6.6 Processor Interface

The communication subsystem is connected to the main board via a 96-pin DIN-connector. This interface includes most of the signals on the processor expansion bus and several +5 V and ground pins. The following signals are available to, and used in, the communication subsystem:

- Address bus (A0 – A12).
- Data bus (D0 – D31).
- Memory-strobe from the processor (MSTRB).
- I/O-strobe from the processor (IOSTRB).
- Read/write-indicator from the processor (XR/W).
- Reset-signal for I/O devices (IO0–RESET).
- I/O-interrupt to the processor (IO–INT).
- Main processor clock, inverted (H3).

The expansion bus data lines are connected to four byte-wide bus-transceivers on the communication subsystem, to minimise the load on the data bus, and to ensure that the data lines are driven by only one source at a time.

The address bus and control lines are directly connected to the decode hardware, and to the control logic is the LCAs. The processor’s access to the different parts of the communication subsystem is handled by the decode hardware in fast PALs, which generates chip select signals to the LCAs, and read/write-strobes to the FIFOs. The speed of the FIFO chip-select decoder is critical for the processor to access the FIFOs without wait-states. A 5-ns speed grade device is used.

6.7 Data Streams

The flat ribbon cable has twelve lines, which carry the following signals:

- 8 data bits
- 1 command bit
- strobe
- hold
- ground

Byte-wide data is transferred via the data streams. The packing and unpacking between this format and the 32-bit wide processor data bus is done by the logic accessing the FIFOs. The one control bit is used to signal that the rest of the word should be handled as a command. After valid data is placed on the stream, the strobe signal is asserted, which is also used by the receiver to latch the incoming data. If the receiver is unable to respond to new data, the hold signal is asserted to notify the sender. The ground signal shields the strobe from noise that may be generated from the data lines.

6.8 FIFO Memory

The asynchronous interface between the modules implies that buffering capacity is necessary in the communication subsystem. It was decided to equip the system with FIFOs, instead of trying to obtain the same functionality with dual port RAM or fast SRAM with additional logic. Because of the 32-bit wide data bus on the processor side, the FIFO banks must be 32-bit wide. A FIFO bank is formed by four 9-bit FIFOs in parallel. The number of FIFO banks to use was an issue of discussion in the design phase. With less than eight FIFO banks for the eight communication channels, data transfer through the channels must somehow timeshare the FIFO banks. Fast FIFOs, especially those with programmable flags, are expensive. Also, a FIFO bank of four chips requires a significant part of the board area. The decision of having four FIFO banks on the board is therefore a trade-off between speed, flexibility, space and cost. Two FIFO banks are used for data flow from the communication links to the processor, and the two other for data transfer in the opposite direction, see Figure 6.2. The directions "in" and "out" are defined with reference to the processor.

The FIFOs, MT52C9010 and MT52C9012, the latter with programmable flags, are 1 K deep, but can be replaced with pin-compatible deeper FIFOs. The FIFO flags are used to avoid attempts to write to a full FIFO or read from an empty one. Often it is advantageous to know more about the FIFO fill grade than just empty or full, e.g. to allow reading a vector without checking the FIFO flags between each word. This is solved by using FIFOs with programmable flags, like the MT52C9012, which has almost-empty and almost-full flags, and an empty-full flag to be used in combination with the two other flags. It is
sufficient that only one FIFO in each bank (the least significant) has programmable flags. Currently, FIFOs of 20-ns speed grade are used. It is referred to the MICRON data book [58] for further information on the FIFOs.

6.9 Control Logic

The control logic can be divided into three separate tasks:

- Routing of data (multiplexing/demultiplexing) between the four FIFO banks and the eight data streams.
- FIFO control.
- Data stream control, which includes sending, receiving, bypassing data, decoding of address information, and to take action when the receiving unit is not ready to respond.

It was decided to implement all the logic in field programmable gate arrays, more specific Xilinx LCAs, to take advantage of the reconfigurability, and not least important, experiences in using this technology.
6.9.1 Partitioning of the Control Logic

The required functionality can be partitioned into several FPGAs of various complexity. Factors that influence this partitioning are:

- I/O-requirements versus logic complexity.
- Architectural constraints on clock distribution.
- Improved reconfigurability through independent implementation of the functional blocks.
- Cost.

An early proposal was to partition the logic between four complex LCAs, which means one LCA per channel, and use the XC3090 in 132-pin PGA (Pin Grid Array) packages. The routing of the data busses between the FIFOs and the communication channels could then be done by tri-state logic in the LCA I/O blocks. These circuits also have enough pins and logic blocks to include the FIFO access control. This logic partitioning is probably the least complex and most flexible, but is also an expensive alternative, due to the high costs of such powerful circuits. Instead, a solution with several smaller circuits was selected, as the costs were reduced by a factor of between two and three compared to the first alternative. Currently, the communication subsystem includes seven LCAs, which are:

- Four Data Stream Controllers implemented in XC3042 84-pin PLCC (Plastic Lead Chip Carrier) packages.
- One FIFO Controller implemented in one XC3064 84-pin PLCC package.
- One Multiplexer (MUX) and one demultiplexer (DMUX) implemented in XC3020 64-pin PLCC packages.

Figure 6.3 shows the main components and data paths in the communication subsystem. A picture of the communication subsystem board including the components is shown in Figure 6.4.

It was experienced later that the logic partitioning into seven LCAs lead to some problems. Only two I/O lines on the MUX and DMUX LCAs were available for setup data from the FIFO controller. One of these lines is used as chip select, the other is a serial data line. As a consequence, there is no line left for a clock enable signal, to freeze the contents in the registers when the hold signal on a stream is asserted. This problem was solved by an
Figure 6.3: Components and data paths in the communication subsystem (from [79]).
extra level of registers in the channel controllers. Using LCAs with more I/O pins for the MUX and DMUX, would have simplified the protocol between the FIFO Controller and the multiplexer circuits, but there was not space left on the board for larger circuits.

6.9.2 Data Paths through the Communication Subsystem

The Multiplexer (MUX) directs data from the four channel controllers to the two incoming FIFO banks, resulting in a 4-2 multiplexer. The MUX is built from two separate 4-1 multiplexers, one for each of the two FIFO banks. The Demultiplexer (DMUX) has the opposite function, to direct data streams from the outgoing FIFO banks to the channel controllers. Data from one of the two FIFO banks are selected for each of the four communication channels. The directioning of data between the FIFOs and channel controllers is shown in Figure 6.5.

The setup of the MUX and DMUX is done serially from the FIFO controller. The connection between the FIFO controller and the multiplexer circuits is only one line going through the MUX to the DMUX. This is done because there are no pins left on the two circuits that can be accessed directly from the processor. A redirection of the data paths
Figure 6.5: The data paths between the communication links and the FIFOs.

through the MUX/DMUX requires commanding the FIFO controller to change the setup by signalling the serial line.

6.9.3 FIFO Control

The FIFO controller is the most complex LCA in the system, and is accessible from the processor through an 8-bit wide data path. The FIFO controller is responsible for the following tasks:

- Control of FIFO access signals (read/write-strobes and reset) and FIFO flags.
- Control of the transmission between the FIFO banks and the channel controllers.

The FIFO controller has some additional functions:

- Setup of the data paths through the MUX and DMUX.
- Maintain status information about the communication subsystem, which can be read by the processor.
- Execute commands issued by the processor writing to the command registers.
6.9. Control Logic

Since field programmable logic is used for the FIFO controller, only the I/O pins are fixed, the rest is programmable. This section contains a description of a possible FIFO controller design. More details on the FIFO controller is given in the documentation manuals [79, 78].

The FIFO controller is responsible for transferring data from the outgoing FIFO banks through the DMUX to the data streams, and in the opposite direction, through the MUX. To do this, the FIFO controller has access to the read strobes for each FIFO in the outgoing FIFO banks, and the write strobes for the FIFOs in the incoming direction. Data present in the outgoing FIFO banks should be automatically transferred to the data streams. The FIFO controller has access to the three FIFO flags for each FIFO bank, and uses this information to decide the operations to perform. To read an empty FIFO, or to write to a full one, is prohibited.

On the data stream side of the FIFO banks, the same data bus is connected to all the four FIFOs in the bank. The four FIFOs are accessed in sequence, controlled by the read/write signals from the FIFO controller. When transferring data from a FIFO bank into a channel controller, the FIFO read strobes are asserted in a sequential manner to unpack data to a 9-bit format.

The interface between the FIFO controller and each channel controller consists of four signals. In the proposed protocol, two signals, CHANNEL-READY and READ-ENABLE, control the transmission of data from the FIFOs to the channel controller. Data is transferred when the channel controller is ready (CHANNEL-READY active). The READ-ENABLE is asserted for the transmission of each byte. Incoming data from the data streams are forwarded to the FIFOs, if there is space available in the FIFO banks. The two signals, FIFO-READY and WRITE-ENABLE, control this transmission. Figure 6.6 and Figure 6.7 show the data paths, components and control signal involved for incoming and outgoing data through the communication subsystem.

The FIFO controller registers can be accessed directly from the processor. These registers include several 8-bit command registers and a FIFO flag status registers. With the use of the four least significant address lines (A0–A3) and two chip-select signals, one for reading and the other for writing, up to 32 registers can be implemented in the FIFO controller.

6.9.4 Data Stream Control

The four channel controllers are responsible for transmitting and receiving data to and from other modules in the network. A channel controller LCA has three possible input and output data streams:
Figure 6.6: Outgoing data to the data streams.

- to/from other RENNS modules,
- to/from one of the neighbour channel controllers, and
- to/from the FIFO banks.

The functionality in the data stream controllers is defined by the logic in the LCAs. All the operational modes of Figure 5.2 are implemented, but the connectivity between the data streams is somewhat constrained. There is full connectivity between the data streams internally in a channel controller, limited connectivity with one neighbour controller, and no direct contact with the two others.

The implementation of the controller logic will for most configurations contain multiplexers for directing one incoming and one outgoing data stream from a source to a specified
destination. Some fast decoding logic is also needed to respond to incoming commands and data. The bidirectional links between the channel controllers have been omitted in the first version of the channel controller logic, to simplify design and test. For the setup of the internal multiplexers, the processor has access to registers in the channel controllers through a 4-bit wide data path. Figure 6.8 shows the possible data paths through one channel controller. The channel controllers have logic resources available for manipulating data to and from the stream.
Figure 6.8: Data paths through the channel controller.

6.10 Configuring the LCAs

The LCAs have three pins, which during programming of the circuits determine which of several configuration modes will be used. The LCAs in the communication subsystem can be configured in three different ways:

- In master parallel mode from an EPROM at start-up time.
- In master serial mode from the Xilinx download cable, connected to the board from a PC or workstation.
- In master serial mode from the processor through the decode hardware.

In all three cases, the FIFO controller is responsible for the configuration of all LCAs in the communication subsystem. Depending on which configuration mode is selected, the FIFO controller receives the configuration data either in parallel (byte-wide) from an EPROM, or serially from the download cable or the decode logic. The decode logic emulates the protocol used when configuring an LCA from a serial PROM, but the serial data are instead generated from the processor by writing to two different addresses in the expansion bus address space. The FIFO controller configuration mode is selected by setting three dip-switches. All three modes use the same serial daisy-chain for configuration of the other
LCAs, which are permanently set in slave mode. The configuration sequence then becomes FIFO controller, MUX, DMUX, and channel controller 1 through 4. This strange use of the decode logic for configuring the communication subsystem allows for reprogramming the LCAs between and during simulations, without physically replacing the EPROMs. The processor has direct access to the dedicated reconfiguration-pin of the LCAs, which is necessary for reconfiguration during simulations.

6.11 Synchronisation

A RENNS module has two clocks, one processor clock and another clock for the communication subsystem. In this way, transmission speed is only limited by the maximum clock rate of the logic in the LCAs. Communication subsystems at different modules are operating at the same clock frequency, but the clocks are generally not in phase. To handle incoming data, the channel controllers sample the data streams at a double data-rate, compared to the operational frequency for the rest of the communication subsystem. There is a two-level deep FIFO buffer in the channel controllers, for not to loose incoming data.

The logic for a reliable asynchronous protocol turned out to be much more complicated than anticipated, leading to many errors that have been hard to identify and solve. It may have been worth the effort to implement a common clock for the communication subsystem.

Another way to address the synchronisation problems is to divide the FIFO buffer capacity between the two asynchronous interfaces. This requires FIFO buffering both at the processor interface and in connection with the incoming data streams. Data into the module can then be strobed directly into a FIFO bank. Using one 9-bit wide FIFO on each incoming and and outgoing data stream, a total of 8 FIFOs are required for the data stream interface. The buffering capacity in the processor interface can then be reduced to one 32-bit wide (36-bit including parity bits) FIFO bank in each direction. The total number of FIFOs is unchanged. The necessary control logic for such solution is not studied in detail, but is not assumed to be more complex than for the present system.

6.12 Communication Protocols

6.12.1 Signalling on the Data Stream

The communication on the data streams was intended to follow a simple asynchronous protocol. When there is valid data to the stream, the active low strobe signal is asserted, unless
the hold signal indicates that the receiver is not ready. A problem with an asynchronous protocol, is that the incoming data must be sampled with a high enough rate, to ensure that no data are lost. For the same reason, it is a time constraint on the activation of the hold signal. These problems have been solved by operating the data streams at the double data-rate as for the rest of the communication subsystem. The communication subsystem is equipped with an oscillator of the double frequency, which is divided down in the LCAs for those functions that operate on normal speed. When the channel controllers detect an active hold signal, all the steps, from reading the FIFOs to writing the data streams, must be stopped immediately, to ensure that data are not overwritten in any stage of this pipeline. The activation of the receiver hold signal and to latch the data in the pipeline are the most time critical operations for the communication subsystem. Currently, a clock rate below the initial design goals is used to handle this complex logic. Higher speed can be achieved by replacing the channel controllers with circuits of higher speed grade. Circuits with equivalent functionality are now available in speed grades two and a half time faster than those used. A more detailed description of the signalling on the data streams and the channel controller logic, can be found in the documentations [79, 78, 76].

### 6.12.2 Token Ring Protocol

The communication protocol first implemented is based on one-dimensional rings. A module can be connected to one or more rings. Only one module at a time is allowed to transmit data to a ring, while the other modules are receiving and/or retransmitting the data. A token is circulating to determine the node which is allowed to transmit. When there is data to be transmitted to a data stream, the controller must wait for the token and remove the token from the ring, before data can be transmitted. When transmission of a vector is completed, the token is inserted back to the ring. A vector consists of 32-bit words. First comes a header-word with the address of the receiver, then follows a variable length data field, and finally the tail-word, which is the new token.

All data on a ring are handled as vectors. A module is only allowed to transmit one vector before it has to reinsert the token back to the ring. This ensures that all the modules in the ring have the opportunity to transmit before one module can transmit twice.

More discussions on communication protocols can be found in [62]. The vector format is detailed in [76].
6.12.3 Addressing on the Ring Bus

The data stream, or group of data streams, that will receive data from the bus must be given a unique address. Each module has an identifier, which can be set from a dip-switch on the front panel. This identifier can be used as an address and be programmed into one of the channel controller registers, or other addressing schemes can be used. A four bit register in the channel controller holds the address. The width of channel controller registers (4 bits), limits the number of unique addresses in a ring to 16. The registers are set from a program at initialisation of the communication subsystem, and can be changed on demand during simulations.

6.13 Balancing the Processor Speed

The maximum bandwidth of the processor expansion bus is 120 Mbyte per second. This maximum speed is achieved for read operations, by reading one 32-bit word per clock cycle. Currently, the peak performance of one data stream is 5 Mbyte per second. A configuration of four incoming data streams is able to feed the communication subsystem with 20 Mbyte per second, but the number of FIFO banks, two banks in each direction, limits the number of simultaneously active data streams to two. Thus, the peak performance of one communication subsystem is 10 Mbyte per second.

The communication bandwidth of the processor expansion bus is higher than the bandwidth of the communication subsystem. In the current implementation, the communication subsystem can possibly slow the processors.
Chapter 7

Configuring the Communication Network

7.1 Reconfiguration on Three Levels

All the logic between the FIFOs and the data streams is implemented in field programmable gate arrays (FPGAs). This includes converting data between byte- and word-length formats, and routing the data to the appropriate data stream. Normally, the logic in the controllers will include registers accessible from the processor, to define the actual function of the communication subsystem.

To meet the communication demands of different ANN algorithms, the communication network between the RENNS modules is reconfigurable on three levels; The data-path level, the logic level, and the command level. The normal procedure is to change the network topology when a different algorithm is loaded, but it is also possible to change the topology dynamically during a simulation.

The reconfigurability of the communication subsystem can be divided into the command level and the logic level. The configuration on these levels can be changed dynamically, for adaptations to changing functional requirements.

Figure 7.1 shows the hierarchy of the three reconfiguration levels. Changes on the lowest level, the data-path level, are most fundamental, while reconfiguration at the top level, the command level, is fastest and most easily done.
7.1.1 Data-path Level

Reconfiguration of the communication network topology will be handled by a programmable communication switch. Through such a switch it is possible to connect the data streams in various ways, thereby forming different network architectures. The extended reconfigurability provided by this level is not necessary for reconfiguration on the logic and/or the command level. The configuration switch [96] is not yet completed. Currently, reconfiguration of the data path level is done manually.

7.1.2 Logic Level

The reprogrammability of the communication subsystem is achieved by use of field programmable logic. Whenever a change in the functionality of the data streams is needed, the LCAs can be reprogrammed. The logic level implements the data paths through the communication subsystem, from the FIFOs to the data streams, and is defined by the configuration of the LCAs. Communication based on circuit switching, packet switching, or a combination, can be implemented. The components used in the current version do not have the complexity to let the two fundamentally different communication schemes coexist, but the logic can instead be changed dynamically, on demand, by reprogramming the circuits. See Section 6.5 for more details on reconfiguration of the LCAs and the available configuration modes.
7.1.3 Command level

The different LCAs in 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. Byprogramming these registers, different operation patterns of the data streams can be implemented. Operation patterns include receiving, sending, bypassing, broadcasting and combinations of these (see Section 5.2).

In the current implementation, the logic level includes the following command registers; FIFO-flag-setup, I/O-interrupt-source, I/O-interrupt-mask, FIFO-selection-setup, and data-stream-setup. These registers can be both read and written by the processor. In addition, the processor can read the current state of the FIFO flags.

7.1.4 Possible Configurations

By utilising the three reconfiguration levels, different network topologies can be built. Simple topologies comprise ring and mesh networks. Hybrid configurations, i.e. mixing different topologies, can also be implemented. The most direct way to alter a configuration is to program the command registers in the different FPGAs. If more fundamental changes in the communication topology are required, the data-path level and/or the logic level must be reconfigured.

7.2 One-Dimensional Ring Architectures

One-dimensional ring is one of the simplest scalable interconnection architectures. All well-known ANN paradigms can be simulated on a systolic ring architecture. Using a ring, the computational power from the high-performance processors can be utilised in a simple communication topology.

7.2.1 Token Ring

In Section 6.12, a token ring communication protocol was proposed for sending vectors of arbitrary length between modules. The modules to receive data are determined by setting an address in the channel controllers. When the node sees its own address in the header
field of the vector, the data on the ring are tapped off into one of the incoming FIFO banks. The data are simultaneously forwarded further on the ring.

For sending of a vector to more than one module in a ring, a group of modules can be formed by giving the modules the same address. To distribute data from one layer to the next in a multilayer neural network, one group comprises the first layer modules, another group contains the modules assigned to the next layer. A pure broadcast scheme is implemented by setting the same address in all nodes in the ring. Figure 7.2 shows one module with channel controller two and four attached to separate rings. An example configuration is one ring set up with unique addresses and the other for broadcast communication.

![Figure 7.2: Internal data paths in a module connected to two rings.](image)

An arbitrary number of modules can be connected in rings, as shown in Figure 7.3.

![Figure 7.3: RENNS modules connected in two one-dimensional rings.](image)
7.2.2 Direct Communication

Direct connections can be set up by connecting two or more modules. With one sender and one or more receivers, the flat ribbon cable form a line from the sender through the receiving channels. With only one sender, a token is not necessary. This communication can use the same logic as the token ring, as long as there is only one sender on the same line. A ring of direct connections is shown in Figure 7.4. Communication between processors not directly connected must go through the processors.

Figure 7.4: RENNS modules set up with direct connections.

7.3 Architectures Based on Several Rings

General Parallel Architectures Based on Rings

Broadcast between the layers in a multilayered network can be efficiently implemented by having one ring per layer, as shown in Figure 7.5. The modules assigned to the same layer are listening to the same address. For the hidden layers, two bidirectional communication channels and all four FIFO banks in a module are used.

A Specialised Configuration for Pipelined Backpropagation Learning

An example of application development on RENNS is the implementation of a multilayer perceptron with pipelined backpropagation learning, which was presented in section 2.3. The broadcast busses between the layers and the fold-back of the pipeline can be implemented by using several rings, as shown in Figure 7.6.
In this implementation, the outputs, the delta values (the $\delta$s) and the weight changes for each layer are distributed through separate rings. Each module can be attached to maximum four rings. Since there are only two FIFO banks in each direction, the FIFOs must be time-multiplexed if a module is connected to more than two rings. Another solution is to have two logical rings on one physical ring, and distinguish the rings by the addresses given to the modules.

For the solution illustrated in Figure 7.6, the distribution of outputs and $\delta$s can take place on two separate rings simultaneously, at full speed. In backpropagation, updating of the weights can be done per epoch, instead of after each presented pattern. The data paths in the communication subsystem can then be temporarily configured for the broadcasting of the weight changes between each batch, before resuming the calculations of outputs, $\delta$s and weight changes in a pipelined fashion. The communication resources, with four incoming and four outgoing channels per module, are sufficient to keep a fixed configuration at the data path level. The reconfiguration for updating the weights can be done at the command level, by changing the values in the setup registers in the communication logic.

### 7.4 Mesh Topologies

A two-dimensional mesh like the Toroidal Lattice Architecture (TLA) is also implemented by using several rings. A TLA is a two-dimensional mesh where the edges is wrapped around, and is well suited for implementation with rings (see Figure 4.2). Two-dimensional architectures fully utilise the FIFO capacity of two FIFO banks in each direction. Figure 7.7 shows the internal paths for a module configured for connection in a TLA, while Figure 7.8 illustrates RENNS modules connected in a TLA.

Alternatively, direct connections like the ring shown in Figure 7.4, can be used in a TLA. For a such solution to be efficient, the processor must do both calculations and the shuffling of data without using extra instructions, but instead utilise the FIFOs as fast memory.
Figure 7.6: Implementation of pipelined backpropagation learning using several rings.

Figure 7.7: Internal data paths in a module connected in a TLA.
In TLA, two of the four stream LCAs are idle. In Planar Lattice Architecture (PLA) [15], data is communicated bidirectionally along both the horizontal and vertical direction, and will occupy all the available communication links. A problem with PLA is that the communication channels must time-multiplex the FIFO banks. To facilitate this, the MUX and DMUX must be reconfigured before a change in communication direction. This is time consuming, and synchronisation problems may occur. Although reconfiguration at this level is fastest and easiest, it is not recommended to reconfigure the communication this regularly.
Figure 7.8: RENNS modules connected in TLA.
Chapter 8

Parallelisation of Recurrent Associative Memories

8.1 Motivation for the Choice of Test Problem

The far most popular ANN model used in performance analysis is multilayer perceptron with backpropagation learning. Benchmarks, like NETtalk and Bignet, are common in speed measurements. Also, other backpropagation networks, often tailored to the target computer, are used for this purpose. Backpropagation is paid much attention in performance analysis of RENNS.

In addition to backpropagation, around 10 other ANN models are in common use [29], and intensive research is in progress on development of ANN architectures. A class of neural networks, recurrent associative memories, have, despite their limitations, been offered much interest over several years. Recurrent associative ANNs are single layer networks, with feedback loops from each neuron to all the other. The input and stored patterns can be of either binary, bipolar, or continuous data. The structure of a recurrent associative network is shown in Figure 8.1.

Recall of a stored pattern is done by iterations until the outputs have stabilised. For large networks, the recall phase can be a time-consuming task. The Hopfield network is the best-known associative recurrent network. The recurrent associative memories implemented on RENNS for performance analysis are of Hopfield type. Different methods for calculating the weights and biases are investigated.

Recurrent associative ANNs are useful for recognition of patterns corrupted by statistical
noise. They also have interesting properties in efficiently finding close to optimal solutions to NP-complete (nondeterministic polynomial) optimisation problems [91, page 106]. The main disadvantage concerned with Hopfield networks is their low storage capacity. One neuron is needed for each element in the input pattern, and the number of weights are square the number of neurons. The weights represent the long term memory. If the memory gets overloaded, there is a possibility that the network fails to converge to the correct output pattern, and instead recalls spurious outputs. It may also happen that the network converges to stored pattern complements. The recommended storage load for a Hopfield network is only about 15% of the number of weights [54]. Another estimate for the storage capacity in Hopfield networks is:

\[
\text{Storage Capacity} = \frac{N}{2 \log_2 N} \tag{8.1}
\]

where \( N \) is the number of neurons [45]. Using Equation 8.1, the storage capacity of Hopfield neural networks of various sizes is shown Table 8.1.

The probably most important reason for the interest in recurrent associative networks is their potential for implementation in VLSI, both digital and analog. For VLSI implementation, the weights and thresholds are usually calculated off-line, prior to production of the chip. For research on models and algorithms, and application development, it is also necessary to work with these ANNs on general computers. Ideally, the weights and thresholds are calculated once, but it is more likely that stepwise refinement of the application is necessary.

For large networks, recall of patterns through many iterations is a time consuming and
<table>
<thead>
<tr>
<th>Pattern Size</th>
<th>Storage Capacity Number of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8=64</td>
<td>5</td>
</tr>
<tr>
<td>8x16=128</td>
<td>9</td>
</tr>
<tr>
<td>16x16=256</td>
<td>16</td>
</tr>
<tr>
<td>16x32=512</td>
<td>28</td>
</tr>
<tr>
<td>32x32=1024</td>
<td>51</td>
</tr>
<tr>
<td>32x64=2048</td>
<td>93</td>
</tr>
<tr>
<td>64x64=4096</td>
<td>170</td>
</tr>
</tbody>
</table>

Table 8.1: Storage capacity in Hopfield neural networks.

computationally demanding task. Depending on the procedure used for determining the weights and thresholds, this may involve operations on large matrices. For efficient application development, the calculation of the weights and thresholds, and the recall phase, should be parallelised on a neurocomputer. Another reason for using a neurocomputer is that the memory requirement for such problems is often too large to fit into the primary memory of a conventional computer.

When starting program development on RENNS, it is advantageous to first concentrate on simple problems. In recurrent associative memories, the network topology and the calculations are regular and not very complicated. Several performance parameters can be tested with respect to both the processing part and the interprocessor communication. The performance issues to be discussed are:

- Communication bandwidth, and the balancing of communication bandwidth versus processing speed.
- The flexibility to form various communication architectures, and their suitability to the application.
- Memory capacity, and the balancing of memory versus processing speed.
- Scalability.
- Identification of bottlenecks.
- Peak performance measured in connections per second (CPS).

Another reason for choosing a simple ANN model for test purposes is that the programming environment and host interface were at an early development stage when the first
experiments were started. The test programs were developed without the advantage of having an operating system and VME-bus interface software on the individual modules. The programs were loaded through the serial interface (RS232) on each module. Synchronisation of the modules was done through the high speed communication channels, instead of via the VME-bus and the host computer. While the programming environment and user interface is being developed, there is much work to be done in performance testing and refining the hardware and parallelisation techniques. The study of more complex ANN algorithms, parallelisation techniques, and communication architectures, will be further explored by coming Ph.D. students.

8.2 A Simple Pattern Recognition Problem

For input data to the recurrent associative memory, binary bitmap images of characters are used as pattern vectors. The binary bitmap images are generated from the font files in the X-windows user interface. The upper and lower case letters in the English alphabet, and the numbers from 0 to 9, are selected, which make a total of 62 patterns. Bitmaps of size $8 \times 16 = 128$ and $12 \times 24 = 288$ are scaled-up to pattern vectors of size 128, 256, 288, 512, 576, 1024, and 2048. These input patterns form an $m \times n$ pattern matrix $P$, where $m$ is the number of patterns, and $n$ is the number of picture elements in the input image.

The 62 pattern vectors are stored in the network. For recall, up to 40% statistical noise is added to the patterns, and these noisy patterns are presented to the network. In most of the test-runs, input data with 25% noise are used. Figure 8.2 shows an $12 \times 24$ example pattern, and the corresponding pattern corrupted with 25% noise.

This simple pattern recognition problem is computational demanding and requires a large amount of storage capacity for scaled-up input patterns. The problem should therefore be well suited for performance analysis of the target computer. As a step further in the development of recurrent associative memories, gray scale patterns can be stored and recalled. For the RENNS, which is built from 32-bit DSPs, the computational burden is similar for bitmap/bipolar and gray scale patterns.

8.3 Processor Architectures for Recurrent Associative Memories

Recurrent associative memories are single layer networks with full connectivity between all the neurons. As discussed in Section 2.6.3, sophisticated methods cannot be found for the
partitioning and mapping of the problem, to reduce interprocessor communication on a parallel computer [61]. The most straightforward mapping method, which is to give each processor equal sized parts of the neural network, should be well suited to this problem.

RENNs can be configured to form a range of processor architectures, but because of the regular structure of the Hopfield ANN, it is natural to reflect this in the processor topology. Simple and regular processor architectures are one-dimensional ring and two-dimensional mesh and toroid architectures. At the time the experiments were done, eight modules were up and tested, and the experimental applications were implemented on one-dimensional ring and two-dimensional toroid configurations.

8.4 Determining the Weights and Thresholds

Recurrent associative memories do not have an iterative learning phase for determining the weights. Instead, all the input patterns must be known in advance, and the weights and thresholds are calculated from the input pattern vectors.

In [30], Hecht-Nielsen has listed three basic goals in the design of a recurrent associative network. These are:

- Given any initial state, the network should always converge to some stable state.
• The stable state to which the network converges should be the one closest to the initial state, as measured by some metric.

• It should be possible to have as many stable states as desired.

There is no learning law associated with the transfer function of recurrent associative ANNs. The only restrictions on the real number values in the weight matrix, \( W = [w_{ij}] \), are that the matrix, \( W \), must be symmetric and have a 0 diagonal.

### 8.4.1 Methods for Specification of Weights and Thresholds

The Hopfield network is the best-known recurrent associative memory. In a Hopfield network, the determination of the weights is simple, and only involves calculation of the outer product of the input pattern vectors \([35, 54]\):

\[
    w_{ij} = \begin{cases}
        \sum_{s=0}^{m-1} x_i^s x_j^s, & i \neq j \\
        0, & i = j, 0 \leq i, j \leq n - 1
    \end{cases}
\] \quad (8.2)

In this formula, \( m \) is the number of patterns, \( w_{ij} \) is the connection weight from neuron \( i \) to neuron \( j \), and \( x_i^s \) element \( i \) of pattern \( s \). The resulting weight matrix, \( W \), is symmetric and have a 0 diagonal.

To calculate the weights using Equation 8.2 is not a very time-consuming task, even for large networks. The calculations are done prior to normal operation of the network. The thresholds can be set to zero. Alternatively,

\[
    \theta_i = -\frac{1}{2} \sum_{j=0}^{n-1} w_{ij}
\] \quad (8.3)

is suggested for the thresholds \([45]\).

A shortcoming of the Hopfield model is the low storage capacity, despite the high number of weights. In \([52]\), Li, Michel and Porod, describe an efficient analysis method, which can be used to completely determine the set of asymptotically stable equilibrium points, and the set of unstable equilibrium points, for a recurrent associative ANN. This network retains the basic structure of the Hopfield model, and the results from the analysis and synthesis of this network can be applied directly to the Hopfield model. The storage capacity and the convergence properties are improved compared to traditional Hopfield networks. A short description of the method and synthesis procedure is given in this section.
8.4. Determining the Weights and Thresholds

The recurrent associative ANNs can be described by a system of first-order ordinary linear differential equations which are defined on a closed hypercube:

\[
\frac{dx}{dt} = Tx + I
\]

with the constraints

\[-1 \leq x_i \leq 1, i = 1, \ldots, n\]

where \( x = (x_1, \ldots, x_n)^T \in D^n = \{ x \in \mathbb{R}^n : -1 \leq x_i \leq 1, i = 1, \ldots, n \} \), \( T = [T_{ij}] \) is an \( n \times n \) constant matrix, and \( I = (I_1, \ldots, I_n)^T \) is a constant vector. In addition to the system in Equation 8.4, the set \( B^n = \{ x \in \mathbb{R}^n : x_i = 1 \text{ or } -1, i = 1, \ldots, n \} \) must be defined.

Then, given \( m \) pattern vectors \( \alpha_1, \ldots, \alpha_m \in B^n \), the design goals for the network can be formulated as follows:

1. \( \alpha_1, \ldots, \alpha_m \) should be asymptotically stable equilibrium points of the system in Equation 8.4.

2. The system should have no oscillatory solutions.

3. The total number of the spurious asymptotically stable equilibrium points, should be as small as possible.

4. The domain of attraction of each \( \alpha_i \) should be as large as possible.

8.4.2 Synthesis Procedure

The following synthesis procedure is used for an implementation of a Hopfield type neural network on RENNN.

1. Compute the \( n \times (m - 1) \) matrix:

\[
Y = [\alpha_1 - \alpha_m, \ldots, \alpha_{m-1} - \alpha_m].
\]

2. Perform singular value decomposition of \( Y \) and obtain the matrices \( U, V, \) and \( \Sigma \), such that \( Y = U\Sigma V^T \), where \( U \) and \( V \) are unitary matrices, and where \( \Sigma \) is a diagonal matrix with the singular values of \( Y \) on its diagonal. Let

\[
Y = [y_1, \ldots, y_{m-1}]
\]
\[ U = [u_1, \ldots, u_n] \]  

(8.6)

and

\[ k = \dim(\text{span}(y_1, \ldots, y_{m-1})). \]  

(8.7)

From the properties of singular value decomposition, we have that \( k = \text{rank}(\Sigma) \), \( \{u_1, \ldots, u_k\} \) is an orthonormal basis of \( \text{Span}(y_1, \ldots, y_{m-1}) \) and \( \{u_1, \ldots, u_n\} \) is an orthonormal basis of \( \mathbb{R}^n \).

3. Compute

\[ T^+ = [T^+_{ij}] = \sum_{i=1}^{k} u_i u_i^T \]  

(8.8)

\[ T^- = [T^-_{ij}] = \sum_{i=k+1}^{n} u_i u_i^T \]  

(8.9)

4. Chose a large positive value for the parameter \( \tau \) and compute

\[ T_\tau = T^+ - \tau T^- \]  

(8.10)

and

\[ I_\tau = \alpha_m - T_\tau \alpha_m \]  

(8.11)

Then all pattern vectors in \( L_\alpha \cap \mathbb{B}^n \), where \( L_\alpha = \text{Aspan}(\alpha_1, \ldots, \alpha_m) \), including \( \alpha_1, \ldots, \alpha_m \), will be stored as asymptotically stable equilibrium points in the system

\[ \frac{dx}{dt} = T_\tau x + I_\tau \]  

(8.12)

with constraints

\[ -1 \leq x_i \leq 1, i = 1, \ldots, n \]

5. If the parameter \( \tau \) is sufficiently large, the set of the asymptotically stable equilibrium points contained in \( \mathbb{B}^n \), is approximately equal to \( L_\alpha \cap \mathbb{B}^n \).

For the proof of this procedure, see [52].

Equation 8.12 can be simulated by use of the difference equation

\[ x((k + 1)h) = F(\Phi, x(kh) + \Gamma_\tau) \]  

(8.13)

with the constraints

\[ -1 \leq x_i \leq 1, i = 1, \ldots, n \]
where \( x(kh) \in \mathbf{D}^n \), \( h \) is the sample period, \( F : \mathbf{R}^n \rightarrow \mathbf{D}^n \), \( F(x) = (F_1(x_1), \ldots, F_n(x_n))^T \), with

\[
F_i(\rho) = \begin{cases} 
1, & \text{if } \rho > 1 \\
\rho, & \text{if } -1 \leq \rho \leq 11 \leq i \leq n \\
-1, & \text{if } \rho < -1 
\end{cases}
\]

and

\[
\Phi_r = \exp(hT_r) \\
\Gamma_r = \int_0^h \exp(\rho T_r) \, d\rho \cdot \mathbf{I}_r
\]

For \( 1 \leq k \leq n \), \( \mathbf{I}_k \) is the \( k \times k \) identity matrix, and

\[
T_r = U \begin{bmatrix} \mathbf{I}_k & 0 \\ 0 & -\tau \mathbf{I}_{n-k} \end{bmatrix} U^T
\]  

(8.14)

where \( U \) is the \( n \times n \) unitary matrix given in Equation 8.6, and \( k = \text{rank}(\Sigma) \) given in Equation 8.7.

The parameter matrix, \( \Phi_r \), which corresponds to the weight matrix in a Hopfield ANN, and the vector, \( \Gamma_r \), which corresponds to the thresholds, can be calculated using the factors from the SVD:

\[
\Phi_r = \exp(hT_r) = \exp \left( hU \begin{bmatrix} \mathbf{I}_k & 0 \\ 0 & -\tau \mathbf{I}_{n-k} \end{bmatrix} U^T \right) \\
= U \exp \left( h \begin{bmatrix} \mathbf{I}_k & 0 \\ 0 & -\tau \mathbf{I}_{n-k} \end{bmatrix} \right) U^T \\
= U \begin{bmatrix} e^h \mathbf{I}_k & 0 \\ 0 & e^{-\tau h} \mathbf{I}_{n-k} \end{bmatrix} U^T
\]  

(8.15)

and

\[
\Gamma_r = U \begin{bmatrix} \int_0^h e^\rho \, d\rho \mathbf{I}_k & 0 \\ 0 & \int_0^h e^{-\tau \rho} \, d\rho \mathbf{I}_{n-k} \end{bmatrix} U^T \mathbf{I}_r \\
= U \begin{bmatrix} (e^h - 1) \mathbf{I}_k & 0 \\ 0 & \frac{1}{-\tau} (e^{-\tau h}) \mathbf{I}_{n-k} \end{bmatrix} U^T \mathbf{I}_r \\
= G_r \mathbf{I}_r
\]  

(8.16)

Thus, by using the factors from the SVD, Equation 8.13 can be solved without involving matrix exponential functions. Matrix exponential functions are usually solved by iterative
methods, and are extremely computational demanding. For large matrices, even the multiplication of two matrices, is a time-consuming task. Multiplication of two $m \times n$ and $n \times p$ matrices requires $O(nmp)$ operations by sequential algorithms [40, pages 247–248].

8.4.3 Algorithms

Each of the 2-dimensional input patterns is a vector of size $n$, and the $m=62$ patterns form a target matrix of size $m \times n$. In the following, the notion targets and patterns will be used interchangingly. From the targets, the $n \times n$ weight matrix and the threshold vector of size $n$, are computed.

Calculation of the weights and biases based on the outer product method is straightforward. An algorithm, expressed in pseudocode, is shown in Figure 8.3

```plaintext
CalcWeights(matrix weights, vector biases, matrix targets)
begin
{ Initialise weight matrix with zeros }
zeros(weights, NO_OF_NEURONS, NO_OF_NEURONS);
for $k = 1$ to $k = NO_OF_PATTERNS$
  for $i = 1$ to $i = NO_OF_NEURONS$
    for $j = 1$ to $j = NO_OF_NEURONS$
      weights[$ij$] = weights[$ij$] + targets[$ki$] * targets[$kj$];
  end
for $i = 1$ to $i = NO_OF_NEURONS$
  temp = 0.0;
  for $j = 1$ to $j = NO_OF_NEURONS$
    temp = temp + weights[$ij$];
  biases[$i$] = -0.5 * temp;
end;
```

Figure 8.3: Sequential calculation of weights and thresholds in a Hopfield net, outer product method.

The method based on orthogonalisation of the pattern vectors overcome many of the limitations with the Hopfield ANN, but is also much more complicated and computational demanding. A procedure, SolveHop, for calculation of weights and thresholds has been developed, see Figure 8.4. The order of the calculations differ somewhat from the synthesis procedure, to minimise the memory needed for temporary storage of the large matrices. SolveHop takes as input the $m \times n$ pattern matrix and computes the $n \times n$ weight matrix and the bias vector of size $n$. 
SolveHop(matrix weights, vector biases, matrix targets)

begin

Calculation of \( Y \) from the target matrix;  
\hspace{2em} (Equation 8.5)  
Perform SVD on \( Y \), returns \( U \) and \( \Sigma \);  
Calculation of \( r \);  
\hspace{2em} (Equation 8.7)  
Calculation of \( \Gamma_\tau \);  
\hspace{2em} (Equation 8.11)  
Calculation of \( \Gamma_\tau \);  
\hspace{2em} (Equation 8.16)  
Calculation of weights, \( \Phi_\tau \);  
\hspace{2em} (Equation 8.15)

end;

Figure 8.4: Sequential calculation of weights and thresholds in a Hopfield net, orthogonalisation method.

Note that for the parameters to SolveHop, also the target matrix is declared as an array of floating point numbers. The TMS320C30 digital signal processor is optimised for floating point operations. It is often advantageous to use the floating point data type even for binary variables, if they are used together with floating point numbers in arithmetic operations. Type casting, either explicitly or implicitly, will slow the calculations. Using floating point numbers does not increase the memory requirements, because all data types in the TMS320C30 are of 32-bit size.

Singular Value Decomposition

First in SolveHop, the singular value decomposition of the \((m-1)\times n\) matrix \( Y \) is calculated, which can be accomplished by standard routines. The SVD produces a diagonal matrix \( \Sigma \), with the same dimension as \( Y \), and unitary matrices \( U \) and \( V \), so that \( Y = U \Sigma V^T \). The SVD used here is based on the routines found in "Numerical recipes in C" [12], which is again based on an algorithm for efficient computation of the SVD described by Golub and Van Loan [22]. In these algorithms, various transformations are applied to the input pattern matrix \( P \) in an iterative fashion, thus time consuming operations on the matrix \( P^T P \) are avoided. The SVD of an \( m \times n \) matrix, \( n \geq m \), is calculated in the following two steps:

1. Reduction of \( P \) to upper bidiagonal form, by applying Householder matrices. The rest of the computations are now performed on the upper quadratic part of the bidiagonal matrix.
2. Iterative diagonalisation of the bidiagonal matrix, by multiplication of a series of Givens transformations for rotations both in row planes and column planes. After a few iterations, the super-diagonal entries become negligible.

The routines for the singular value decomposition used here are modified to only return those parts of the SVD needed in the following calculations, to reduce storage requirements. The SVD then returns an \( n \times n \) matrix \( U \), and a vector \( \Sigma \) of size \( n \), where the first \( m - 1 \) elements are the singular values of \( Y \).

Matrix Exponential Functions Utilising Singular Value Decomposition

The matrices \( T_r \), \( G_r \), and \( W \), (Equation 8.14, 8.15 and 8.16) are calculated in the same manner, from \( U \), \( U^T \), and a diagonal matrix:

\[
U \begin{bmatrix} c1 I_k & 0 \\ 0 & c2 I_{n-k} \end{bmatrix} U^T
\]  

(8.17)

Instead of using the diagonal matrix in the calculations, the same information is given by \( c1 \), \( c2 \), and the rank \( k \). Figure 8.5 shows the steps in calculating the matrix exponential functions, based on singular value decomposition.

```
MultSVDFactors(matrix rm, matrix u, int rank, float c1, float c2)
begin
The matrix u is copied to the result matrix, rm;
The elements in the rank first columns of rm are multiplied by c1;
The elements in the (n–rank) last columns of rm are multiplied by c2;
Post-multiplication with the transposed of u;
end;
```

Figure 8.5: Algorithm for calculating matrix exponential functions for the Hopfield net.

### 8.4.4 Parallelisation

Parallelisation of the outer product method for the calculation of weights and thresholds is straightforward. The modules are given the input pattern matrix, and equal sized parts of the weights and thresholds are distributedly computed. No communication is required.
The alternative method described in Section 8.4.2, is extremely computational demanding, and should be implemented in parallel. Each of the \( p \) processors used in parallel simulation of the Hopfield ANN is given a unique number ranging from 0 to \( p - 1 \). The parallel algorithms are formulated so that the same code can be loaded to all the processors. The processors use their identification number to decide which part of the code to execute. Synchronisation of the processors, which will be done via the VME-bus and the host, are in these test programs administered by one of the modules, which acts a control processor. The synchronisation messages are sent through the communication network.

**Singular Value Decomposition**

Sequential simulations on one RENNS module have shown that for small problem sizes, calculation of the SVD takes more than half of the total time spent on determining the weights and thresholds. For larger problems, although the SVD takes several minutes, the rest of the calculations is a much more heavy computational burden. The SVD should be computed in parallel, but in this work, a parallel SVD has not been prioritised. The SVD can be parallelised by the same techniques as those suggested for the eigenvalue problem. A method is described in [23].

Since the SVD is calculated sequentially, the storage required for the matrix \( U \) on one processor, limits the problem size that can be solved. For to some degree overcome this limitation, one processor can be equipped with more DRAM, up to 4 M words.

**Other Matrix and Vector Operations in the Calculation of Weights and Thresholds**

After sequential calculation of the SVD, the resulting matrix, \( U \), and the rank, \( r \), are distributed among the \( p \) processors. One processor must have memory for the entire matrix \( U \), the other processors need space to keep their local part of \( U \). The other \( n \times n \) matrices, \( T_r, G_r, \) and \( W \), are computed in parallel, where each processor holds \( n^2/p \) elements of the matrices. When finishing parallel SolveHop, the weight matrix and the bias vector are evenly distributed among the processors. The computational load is tried divided equally among the processors, although it cannot be avoided that the control processor becomes more heavily loaded than the other.
Parallel Algorithm

A parallel algorithm, ParSolveHop, that computes the weights and thresholds on $p$ processors, is shown in Figure 8.6.

ParSolveHop(matrix weights, vector biases, matrix targets)
begin
  if (node = control processor) then begin
    read(Y);
    SVD(Y, U, Sigma);
    r=rank(Sigma);
  end;
  DistrMatrix(U, m, n);
  Distribute(rank);
  c1 = 1.0;
  c2 = -tau;
  ParMultSVDFactors(Ttau, U, c1, c2, r);
  DistrVector(alpha_m, n);
  MultMatrixVector(Itau, Ttau, alpha_m, m, n);
  Itau = alpha_m - Itau;
  MergeVector(Itau, n);
  c1 = exp(h) - 1;
  c2 (exp(-tau*h) - 1)/(-tau);
  ParMultSVDFactors(Gtau, U, c1, c2, r);
  MultMatrixVector(biases, Gtau, Itau, m, n);
  c1 = exp(h);
  c2 = exp(-tau * h);
  ParMultSVDFactors(weights, U, c1, c2, r);
end;

Figure 8.6: Parallel calculation of weights and thresholds in a Hopfield net, orthogonalisation method.

General Parallel Routines Used in SolveHop

A small set of general parallel routines for operation on matrices and vectors have been developed. These are based on an architecture with $p$ modules in one ring. Similar routines can be developed for other processor topologies.
DistrVector(vector v, int n): Broadcast of the vector v of length n from one processor to the others.

DistrMatrix(matrix a, int m, int n): Distribution of the $m \times n$ matrix a from the control processor to the other processors. Each processor gets $m/p$ consecutive rows. If the number of rows cannot be distributed evenly among the processors, the remaining $m(n \mod p)$ rows are distributed among the first $m(n \mod p)$ processors.

MergeVector(vector v, int n): Merges the parts of vector v and gives all processors a copy of the total vector of size n. The processors broadcast their parts of the vector together with their node identification. Then $p - 1$ subvectors are read and organised into the right order.

Matrix Exponential Functions Using the Factors from the SVD

For efficient calculation of $UDU^T$ on p processors, a parallel version of the routine MultSVD-Factors (Section 8.4.3) is developed for a ring architecture. This routine prerequisites matrix U evenly distributed among the processors, except the control processor, which has a copy of the whole matrix. Storage for the result-matrix of size $n^2/p$ must be allocated in advance. The calculations with the transposed of U, are accomplished by circulating the rows of U in the ring. The parallel routine is shown in Figure 8.7.

8.5 Recall Phase

8.5.1 Calculations

When a pattern is presented to the network, the network is allowed to iterate until it reaches a stable state. The outputs of the n neurons is then the recalled pattern vector. The recall phase in a single layer recurrent network can be described mathematically as:

$$Net_j = \sum_{i \neq j} w_{ij}Out_i + In_j + T_j$$

$$Out_j = f(Net_j)$$

The calculations are basically a loop that is iterated through until the output values are unchanged between the iterations. Each iteration involves calculation of the inner product
ParMultSVDFactors(matrix rm, matrix u, int rank, float c1, float c2)
begin
  for i = 1 to n do begin \{for each column of u^T (= row of u)\}
    if control processor then
      Send column to the other processors;
    else
      Receive column of u^T
    end;
  for j = 1 to n/p do begin \{for n/p rows of u\}
    for k = 1 to rank do \{for the rank first elements\}
      multiply and accumulate u(j,k) and u^T(k,i) and c1;
    for k = rank + 1 to n do \{for the (n - rank) last elements\}
      multiply and accumulate u(j,k) and u^T(k,i) and c2;
    Store result in rm(i,j);
  end;
end;

Figure 8.7: Parallel routine for calculating matrix exponential functions for the Hopfield net.

of the weight matrix and the output vector. The nonlinear function \( f \) applied to the inner product is for this problem chosen to be the hard-limit nonlinear function:

\[
Out_j = \begin{cases} 
1 & \text{if } Net_j > 0 \\
-1 & \text{if } Net_j < 0 \\
\text{unchanged} & \text{if } Net_j = 0 
\end{cases} \quad (8.18)
\]

If instead a saturated linear function is used, Equation 8.18 can be modified to:

\[
Out_j = \begin{cases} 
1 & \text{if } Net_j > U_j \\
-1 & \text{if } Net_j < -U_j \\
\text{unchanged} & -U_j \leq Net_j \leq U_j 
\end{cases} \quad (8.19)
\]

where \( U_j \) is the saturation limit.
8.5.2 Algorithm

A sequential implementation of the recall phase is described in pseudocode in Figure 8.8.

```
Recall(vector inputs, vector outputs, matrix weights, vector biases)
begin
  repeat
    for each neuron do begin
      for each input do
        multiply input by weight;
        add result to net value;
      end;
    compute output by applying hard-limit non-linearity to net;
    for each neuron do begin
      compare output with previous output, calculate difference;
      set output to new input;
    end;
  until unchanged;
end;
```

Figure 8.8: Sequential algorithm for the Hopfield net recall phase.

8.5.3 Parallelisation

The time spent on the recall phase depends on the dimension of the pattern vector and the number of iterations. For many applications, strict limitations on response time for each recall require a parallel implementation. A parallel algorithm for the recall phase is shown in Figure 8.9. Each processor simulates $n/p$ neurons. Initially, the input pattern vector is distributed to all the processors. From the inputs and weights, the $n/p$ new activation values are calculated. These are now the inputs for the next iteration, requiring all-to-all communication between the iterations.

The computational load is tried kept evenly distributed among the processors, although some extra load on the control processor cannot be avoided during initialisation and convergence check.
ParRecall(vector inputs, vector outputs, matrix weights, vector biases)
begin
  DistrVector(input, n);
  repeat
    for neuron = 1 to n/p do begin
      for input = 1 to n do
        multiply input and weight;
        add result to net value;
        add bias;
    end;
compute output by applying hard-limit non-linearity to net;
for neuron = 1 to n/p do begin
  compare output with previous output, calculate difference;
  set output to new input;
end;
MergeVector(input, n);
Calculate total difference;
until unchanged;
end;

Figure 8.9: Parallelisation of the Hopfield net recall phase.

8.6 Scaling the Problem

The input patterns are of size $8 \times 16 = 128$ or $12 \times 24 = 288$. To investigate the performance for different problem sizes, the input patterns are scaled in both dimensions. The scaling factor is limited by the memory required, in both the control processor and the other processors. The size of the weight matrix is square the size of the number of neurons. As an example, a binary input image of 32 by 32 pixels, which is a small picture, requires a Hopfield network of 1 K neurons and 1 M weights. For the double pattern size, 64 by 64 pixels, the Hopfield network size is increased to 4 K neurons and 16 M weights.

For analysis of the RENNS scaling properties, different sizes of the font recognition problem are implemented, see Table 8.2.
8.7 Storage Capacity and Recall Results

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Number of Neurons</th>
<th>Number of Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>8×16</td>
<td>128</td>
<td>16 384</td>
</tr>
<tr>
<td>16×16</td>
<td>256</td>
<td>65 536</td>
</tr>
<tr>
<td>12×24</td>
<td>288</td>
<td>82 944</td>
</tr>
<tr>
<td>16×32</td>
<td>512</td>
<td>262 144</td>
</tr>
<tr>
<td>24×24</td>
<td>576</td>
<td>331 776</td>
</tr>
<tr>
<td>32×32</td>
<td>1 024</td>
<td>1 048 576</td>
</tr>
<tr>
<td>24×48</td>
<td>1 152</td>
<td>1 327 104</td>
</tr>
<tr>
<td>32×64</td>
<td>2 048</td>
<td>4 194 304</td>
</tr>
</tbody>
</table>

Table 8.2: Scaling up the pattern recognition problem.

8.7 Storage Capacity and Recall Results

The test runs are done primarily to gather quantitative information on the processing speed, but some results regarding the quality of the algorithm are discussed in this section. In the tests, 62 bipolar patterns are stored in a Hopfield network. The patterns are generated from scaled-up font bitmaps of size 8×16=128 and 12×24=288. For recall, the patterns are corrupted with 25% statistical noise.

For a pattern size of e.g. 512 elements, the number of patterns that can be stored according to Table 8.1 (the outer product method) is limited to 28. To distinguish between 62 patterns, a Hopfield net with more than 1 M weights is required. The recall results, using the outer product method for calculating the weights and thresholds was not very successful, even for networks with more than 1 M weights. The network failed to recall the patterns, because many of the font images are too similar.

The method based on orthogonalisation of the input patterns for determining the weights and thresholds increases the storage capacity of the Hopfield net. Simulations show that with a pattern size of 12×24=288, the average error after recall is 19, and an average of 4 iterations per pattern is required. Doubling the pattern size to 576, the 62 patterns are recalled with a maximum error of 4, while the average error is reduced to 0. The number of iterations per pattern is reduced to maximum 5 and 4 in average. The scaled-up patterns give better results because of more weights in the network. This enhanced method for determining the weights and thresholds significantly improves the storage capacity of the Hopfield network, and the network’s robustness to noisy patterns. More results from using this method, also concerning gray scale patterns, are given in [52].
8.8 Memory Requirements for Scaled-up Problems

RENNs modules are normally equipped with 4 Mbyte of memory. Upgrading to 16 Mbyte is easily done by replacing the 1 Mbyte DRAM modules with pin compatible 4 Mbyte modules.

Test programs are run on a varying number of processors to analyse the RENNS scaling properties. For Hopfield networks, the memory required for calculation of the weights and biases limits the maximum problem size that can be solved. If the code, stack, one-dimensional vectors, and scalars are kept in SRAM, the DRAM can be reserved for the large matrices. The code occupy about 8 K words of memory, and an area of 1 K words is reserved for the program stack. The size of the data types in the TMS320C30 is 4 byte, which means that storage requirements are not reduced if characters or integers are used instead of floating point numbers.

The maximum pattern size that can be run on a single module is important for scaling analysis, because this shows the performance without communication overhead. For single module reference, the same program is run as in the multiprocessor case, except that the code concerned with the communication is not compiled.

Calculating the weights and biases from the outer product of the patterns, the total amount of DRAM on the modules corresponds to the size of the weight matrix, and hence also the problem size. Table 8.3 shows problem size range for a varying number of processors.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Maximum problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td>4</td>
<td>2048</td>
</tr>
<tr>
<td>8</td>
<td>2560</td>
</tr>
<tr>
<td>16</td>
<td>4096</td>
</tr>
</tbody>
</table>

Table 8.3: Problem size range for a varying number of processors.

To maximise the possible problem size, the pattern matrix is compressed by a factor of four for large patterns (above 1024). Each element is then represented by one byte, which is sufficient for most applications. The pattern matrix can then still fit into the SRAM together with the code and other variables. If the modules are equipped with 16 Mbyte of DRAM, the possible problem sizes in Table 8.3 can be increased by a factor of four.
The method based on orthogonalisation of the pattern matrix, requires storage of the matrix, $U$, in addition to the weights. Since the matrix, $U$, and the weight matrix are of equal size, a parallel version of this method requires double storage capacity compared to using the patterns directly. The memory requirements can be expressed as a function of $m$, $n$, and the numbers of processors, $p$. The storage needed by single variables like pointers, indexes, loop counters, etc, are included in a constant, $c$, which only occupy a negligible amount of memory. Table 8.4 shows the memory requirements for this variant of the Hopfield network.

<table>
<thead>
<tr>
<th>Variable/Data Structure</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern matrix</td>
<td>$m \cdot n$</td>
</tr>
<tr>
<td>matrix $U$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>scratch vector</td>
<td>$n$</td>
</tr>
<tr>
<td>weight matrix</td>
<td>$n^2$</td>
</tr>
<tr>
<td>bias vector</td>
<td>$n$</td>
</tr>
<tr>
<td>input vector</td>
<td>$n$</td>
</tr>
<tr>
<td>output vector</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Table 8.4: Memory used by matrices and vectors for the Hopfield net, orthogonalisation method.

The memory requirements can be minimised by using the same storage area for matrices and vectors not used simultaneously. For instance, the three $n \times n$ matrices $T_r$, $G_r$, and the weight matrix can use the same memory area. During calculation of weights and biases, the pattern matrix, the matrix $U$, the weight matrix (or $T_r$ or $G_r$) and the bias vector must coexist in the memory. Since the large $n \times n$ matrices contribute most significantly to the memory requirements, the maximum problem size on one single module is limited by the storage needed for two such matrices. The minimum total memory requirement as a function of number of neurons and processors can be expressed as:

$$2n^2 + (2 + m) n + c$$  \hspace{1cm} (8.20)

For the parallel Hopfield network, the matrices and vectors listed in Table 8.4 are divided equally among the processors. Equation 8.21 shows the minimum required data storage area for a parallel Hopfield network:

$$2n^2/p + (2 + m) n/p + c$$  \hspace{1cm} (8.21)

Because of sequential computation of the SVD, one processor must keep the entire $n \times n$
matrix, \( U \), in memory. This limits the maximum pattern size to 836. To overcome this limitation, one of the processors can be equipped with additional DRAM (up to 16 Mbyte).
Chapter 9

Implementation of Recurrent Associative Memories

Optimal speed and close to linear scaling are two major goals in programming a parallel computer, and can be approached by:

- Dividing the computational load equally among the processors.
- Optimal utilisation of the processor’s computational capacity.
- Optimal memory mapping.
- Selecting a processor topology suitable for the application.
- Minimising communication costs, avoiding the processors waiting for data.
- Minimising processor involvement in the communications, and utilising the logic in the communication subsystem.

These issues will be discussed throughout this chapter.

9.1 Load Balancing

Generally, a program consists of a sequential and a parallel part. For a Hopfield net the parallel part is the dominating, comprising inner product calculations, thresholding,
nonlinear function, and convergence check. Inner product calculation is proportional to \( n^2 \), the rest is proportional to \( n \). The total communication requirements for a parallel implementation depends on \( n \). The sequential part comprises loop and branch overhead, and waiting for the FIFOs to become ready for reading or writing. The sequential part should be kept as small as possible. To avoid having processors waiting for others to finish their computations, the parallelisable workload should be divided evenly among the processors. Figure 9.1 shows the ideal situation.

![Ideal workload distribution](image)

**Figure 9.1:** Ideal workload distribution.

If the number of processors is a factor of the problem size, the distribution of the neurons is straightforward. Otherwise, the processing concerned with the las\((n \mod p)\) neurons can be divided among \((n \mod p)\) modules. If the neuron vector cannot be split equally among the processors, the communicated vectors have variable length. Before reading the incoming data, the modules must check the sender identification number to know the amount of data to read. This requires a few extra instructions and contributes to the communication overhead.

### 9.2 Utilising the Computational Capacity of the Processors

The compiler generates code far from optimal, even when the optimisation options are used. To speed up the programs, much time can be spent in manually optimising the assembly language code. An efficient approach is to identify the computational intensive parts, and optimise these. The most time consuming operation in ANN simulations in general, and also in recurrent associative memories, is matrix-vector multiplication, in which multiply-accumulate (MAC) is identified as a fundamental operation.
For the TMS320C30, the maximum speed is one MAC per clock cycle, which at a 16 MHz clock frequency gives a period of 62.5 ns per MAC. The possibility to operate at this speed in practical applications depends on the number of memory-variables and register-variables involved in the computations, and the access pattern of the weight matrix. The conditions for achieving one MAC per clock cycle are the following:

- The processors' parallel multiply-add instruction must be used. For parallel multiply-add a maximum of two memory variables can be involved. The rest of the variables must reside in the CPU registers. The memory variables must be placed in storage accessible in one clock cycle. For RENNS, this means the on-chip RAM, SRAM or FIFO banks.

- There cannot be other operations in the inner loop, like for example type casting. All operands must be of floating-point type.

- No extra clock cycles must be spent on instruction fetching, which means either that the instructions must be cached internally in the processor, or the inner loop must consist of only one instruction.

A program example for full speed MAC is:

```c
for (temp=0.0, i=0; i<number_of_neurons; i++)
    temp += *weight+++ **input++;
```

which is compiled to:

```assembly
LDF 0.0, R0 ;Initialise pipeline
RPTS ;Repeat single instruction
ADDF R0, R2 ;One clock cycle if memory variables
|| MPYF *AR2++, *AR6++, R0 ;can be read without wait-states
ADDF R0, R2 ;Add last product
```

where AR6 is assigned to a pointer into the weight matrix, AR2 is a pointer to the input FIFO bank, and R2 is assigned the temporary variable.

Weight matrices above a certain size must be placed in DRAM. A DRAM read access requires from 1 to 7 wait-states. For page-mode read accesses, 1 wait-state is inserted. If
a read operation causes crossing of a page boundary, an extra wait-state is required. For a 1 M word DRAM memory configuration, the page sizes are 256 bytes. DRAM accesses that start during a refresh cycle may need as many as 7 wait-states.

Most Hopfield network applications have large weight matrices, which must be placed in DRAM. For inner-product calculation, one wait-state is minimum while reading from DRAM. It is therefore important that the weight matrix is organised to match the access pattern of the application. If the transposed weight matrix is used, the pointer into the weight matrix must be incremented with the number of columns for each access. The processor supports this in the dedicated 32-bit index registers. Inspection of the assembly code generated by the optimising compiler, has shown that the processor sometimes fails to recognise the possibility of using the index registers, when traversing the matrices. One extra instruction for incrementing the pointer is added to the inner loop, and the computations cannot be expressed in a single instruction. If the cache is not enabled, an overhead of two instruction fetches and the execution of the pointer increment, is added to the inner loop. Another wait-state is required if reading of the next weight causes crossing of a DRAM page boundary. In the worst case, each MAC takes four extra clock cycles; one for updating the weight matrix pointer, two for fetching the two instructions, and one because of the extra wait-state for crossing a page boundary. This shows the importance of storing the weight matrix according to the access pattern.

9.3 Memory Mapping

A RENNNS module is equipped with 128 K words of SRAM and 1 M words of DRAM, in addition to the two 1 K word blocks of on-chip RAM. The on-chip RAM has dual access per clock cycle, the SRAM and FIFOs can be accessed in one clock cycle, and the DRAM requires minimum two cycles per access. The goal is to keep as much as possible of the code and data in fast memory, at least the most frequently used parts.

The TMS320C30 compiler produces sections of relocatable code [85]. For the Hopfield network test applications, only the $n \times n$ matrices are located into DRAM. The rest of the program, which comprises the code, the global variables, and the vectors of size $n$ and $n/p$, is placed in SRAM. The sections' allocation into memory is specified in a linker command file (see Appendix C.3). For optimisation of the access time, the input and output vectors should be located in on-chip RAM, if the vectors are smaller than 1 K words.
9.4 Communication Costs

The transmission of data between two or more processors can be divided into three phases:

1. Writing to the output FIFO bank. Writing to the FIFOs takes two clock cycles, which are 125 ns per word.
   \[ t_{wf} = 125 \text{ ns} \] (9.1)

2. Transmission of byte-wide data over the communication links. At the time being, 5 Mbyte/s is the highest data rate that ensures reliable asynchronous communication. The time required to transfer one byte between two processors are 200 ns, while transferring a 32-bit word takes 800 ns. To reach a processor at a longer distance, 800 ns must be multiplied with the number of intermediate links to pass through.
   \[ t_{ch} = \frac{1}{5 \text{MHz}} = 200 \text{ ns} \] (9.2)

3. Reading the input FIFO bank, which can be accomplished at one clock cycle, or 62.5 ns, per word.
   \[ t_{rf} = 62.5 \text{ ns} \] (9.3)

The processor is involved in phase 1 and 3. The rest of the transmission and routing are taken care of by the communication subsystem.

A transmission is initiated by the processor, by writing to an outgoing FIFO bank. An estimate for the time to set up a transmission for the token ring protocol is shown in Equation 9.4:

\[ t_{\text{setup}} = t_{wf} + l/2 \cdot t_{ch} + 4 \cdot t_{ch} + d \cdot t_{ch} + 4 \cdot t_{ch} \] (9.4)

The terms included are waiting for the token, which in average is half the number of links, \( l \), in the ring (not necessarily the same as the number of modules in the ring), the time to read the token, insert an address word, and the time to reach the destination node though the intermediate links, \( d \). The stages (registers) through the communication subsystem are neglected. It is also assumed that the ring is idle. The average waiting time for access to the ring is the time spent on communication of half the data.

After the address is written to the output FIFO, the data follows immediately, and the vector arrives at the receiving module at a speed of \( 4 \cdot t_{ch} = 800 \text{ns} \) per 32-bit word. To the total overhead the insertion of a new token must be added.
The costs by setting up a transmission must be seen in relation to the vector length. The overhead concerned with small vectors (one or a few words) is almost as large as the time spent by sending the data. For communication of large vectors, the setup costs and the pipeline stages in the communication subsystem become small in comparison to the transmission of the data. For a small patterns size of e.g. 128 on an 8-processor configuration, the communicated vector size is 128/8=16 words. The setup time for sending 16 words to a module at a distance of 4 links from the sender is approximately 24%, or 1/4, of the total transmission time. For a vector length of 1024/8=128, the setup time is only around 3% of the transmission time.

Since the FIFO access time is as fast as one or two processor clock cycles per access, there is a risk that the communication channels become a bottleneck. Neural network algorithms are compute bound, with computational load proportional to $n^2$, while the communications increase with $n$. The communication overhead will become less significant with increasing problem size. By overlapping the computation and the communication, the overhead represented by the communication will probably be reduced.

The computation time for a vector of length $n$ is:

$$t_{\text{comp}} = 2n^2 t_{\text{cycle}} = 125 n^2 \text{ ns}$$

(9.5)

for multiply-accumulate at maximum speed.

The communication cost for sending a vector of the same length to a neighbour processor includes setup of the communication, but after the first word is written to the output FIFO, the writing and the transmission take place concurrently. If the setup time is neglected, and the receiving processor waits until the total vector has arrived before reading the input FIFO bank, the communication time can be expressed as:

$$t_{\text{comm}} = 4n t_{\text{ch}} + n t_{\text{rf}}$$

(9.6)

For the large vectors common in Hopfield type neural networks, only a minor error is introduced by not concerning the setup time.

To minimise the communication costs, and avoid that the processors are waiting for data, the computations and communication phase can be pipelined. With overlapping computation and communication, the processors will not have to wait for input data if:

$$t_{\text{comp}} > t_{\text{comm}}$$

$$2n^2 t_{\text{cycle}} > n t_{\text{ch}}$$

(9.7)
This is less than the number of neurons likely to be distributed to each module in a Hopfield network. If the data go through several links to reach the destination module, this adds a constant term to the communication time. The pattern size limit, for which the processors must wait for data, is still less than the problem sizes common in Hopfield networks. For other ANN models, which may have large fan-in towards a layer, the communication can be a bottleneck.

The total number of MACs per iteration in the recall phase is \( n^2 \). The minimum amount of data communicated in a full broadcast is \( p \cdot n/p = n \), which is independent of the number of processors. As long as \( n >> p \), using more processors add little to the communication overhead, because the amount of data to be transmitted from each processor decreases correspondingly. For short vectors compared to the number of processors, the overhead concerned with administering the communication and the competition for the ring will become significant.

Another way to reduce communication overhead is to choose a topology that utilises more than one communication channel per module, like for instance a two-dimensional mesh. For the toroidal lattice architecture (TLA) on a two-dimensional mesh, the communication bandwidth is doubled.

### 9.5 Initialisation and Use of the Communication Network

#### 9.5.1 Configuration

As discussed in Section 4.4, regular processor topologies like ring and mesh are best suited for Hopfield type networks. In this section, it is therefore focussed on initialisation of a ring bus and a TLA (Toroidal Lattice Architecture).

A routine for configuring the communication subsystem must know

- which of the data streams to connect to the input and output FIFO banks,
• whether the node is enabled or not,
• an address to determine the vectors to receive, and
• the FIFO flag programmable offset.

At power-up, the communication subsystem first receive its configuration, and is then programmed according to application specific definitions. Later, both the logic and the setup can be changed on demand. The setup procedure involves selecting the data paths through the MUX and DMUX, setting the flag offset in the FIFOs and writing the address register and various command registers in the channel controllers. For a token ring protocol, the setup procedure is finished by having one module to set out the token. The token is written to an output FIFO, while asserting the type bit to indicate control information.

For changing the setup between or during simulations, a subset of the initialisation procedure can be run to save time. Some possible changes are quite time consuming. As an example, changing the FIFO flag offset for the incoming FIFO banks, requires the FIFO controller to run through a sequence of states to load the new value through the MUX. Setup definitions for a particular configuration are shown in Appendix C.6.

9.5.2 Communication

Sending data

The routine for sending data writes the necessary control and data information to the FIFO memory, then the communication subsystem does the rest of the transmission without further processor involvement. The FIFO flags are used while writing the output FIFOs. One possibility is to ensure that the FIFO bank is empty before writing the vector at full speed, requiring the length of the vector and control information not to be longer than the FIFO depth. Alternatively, the full flag can be checked before writing each word. The third option is utilising the programmable flags, to reduce the reading of the FIFO flags to only a few times per vector. If normally the FIFO banks are empty before sending a vector, the first method will be the fastest.

Receiving data

Receiving data is done by reading one of the two incoming FIFO banks (FIFO bank 3 or 4). A slow method of receiving data is polling the empty flag before reading each word
in the incoming vector. By waiting until the almost empty flag has gone inactive, a part of the vector can be read into the processor at full speed. If the vector length divided by the FIFO flag offset value has a fractional part, this remaining part of the vector is read while checking the empty flag for each word. It is sometimes necessary to include sender information in the vector. The module identifier of the sender will often be the first word of the vector, following directly after the header field.
Chapter 10

Simulation Results

The issues discussed in the previous chapter provide guidelines for implementation of neural networks on RENNS. Several parallelising and mapping strategies are tested to verify the theoretical considerations in Chapter 9, and to find optimal solutions for a set of applications.

The experiments are run on 1, 2, 4, and 8 processors, with pattern sizes ranging from 128 to the maximum problem size that can fit into the memory of the modules. For the one processor case the maximum pattern size is 1024. On eight modules, the maximum Hopfield network that can be simulated has pattern vector length of 2560. The enhanced method for determining the weights and thresholds, proposed by Li et al [52], requires extensive storage resources, which limit the problem size. Because of serial computation of the SVD, the pattern size on one module is limited to 836. For problem sizes above this limit, the weights and biases are calculated using a simple method based on the outer product of the patterns. The time required to determine the weight and bias values for two different methods are discussed in the end of this chapter.

The performance is measured in Connections Processed per Second, CPS or MCPS. The total number of iterations required for recall of 62 character patterns are multiplied with the pattern size, to find the total number of MACs, which corresponds to the number of connections processed. Dividing this number by the time, give the number of connections processed per second.

C-code for the experiments can be found in Appendix C. The simulation results are presented graphically. The same results are shown in table format in Appendix D.
10.1 Performance and Scale-up for Ring Bus, with Token Ring Protocol

10.1.1 Non-overlapping Computation and Communication

The perhaps simplest parallelising strategy is to split the neuron vector in \( p \) equally sized parts and distribute the parts among the processors. Each module then holds \( n/p \) neurons, with the corresponding part of the weight matrix. The modules are given a common broadcast address. The algorithm can be divided into non-overlapping communication and computation phases.

In the communication phase, the locally stored part of the input vector is broadcasted. If the vector length \( (n/p) \) is less than the FIFO depth, all modules write to their output FIFOs simultaneously, and start to check their input FIFOs. The vectors are sent to the broadcast address and received by all modules except the sender. The received vectors are read by the processor and merged into the total input pattern vector. With a token ring protocol, the sender of the arrived data cannot be predicted. To insert the incoming data correctly into the the input pattern, sender identification, e.g. the node number, is added to the vector.

The processors are now ready for the computation phase where the new activation values are calculated from the inner product between the input vector and the the locally stored part of the weight matrix. If the new activation values differ from the previous, another iteration is required. The convergence information is distributed in an all to all manner, and a new iteration starts with broadcasting of the activation values. Figure 10.1 shows the operations through one iteration.

Speed measurements in MCPS for 1 to 8 processors and problem sizes ranging from 128 to 1024 are shown in Figure 10.2.

Simulations show that the performance is highly dependent on the problem size. On one processor, the speed measurements range from 4.67 to 5.14 MCPS, depending on the problem size. This around 10% variation in the single processor case is mainly caused by loop and test overhead, which are more significant for small-scale problems. With linear scaleup, the performance on eight processors should be from 37.36 to 41.12 MCPS. For vector length of 128, the measured performance is 26.48 MCPS on eight processors, which is 70% of linear scaleup. This means that the communication overhead is around 30%, and will increase if more processors are added. For vector length of 1024, the overhead concerned with the communication is only 4%. The performance on eight processors for even larger problem sizes, 2048 and 2560, are measured to 40.60 and 40.82 MCPS.
Figure 10.1: Parallelisation with non-overlapping communication and computation.
The communication overhead becomes less significant for the large problem sizes common in recurrent associative memories. On eight processors, a speedup of 7.7 compared to the one-processor case is achieved for large patterns. For small patterns, communication costs and the sequential part of the program become more significant, reducing the speedup to 5.7.

Communication overhead influences the efficiency of the parallel implementation. Another important factor is how much of the executable code is proportional to the problem size, and therefore candidate for parallelisation. For a Hopfield network, the most time-consuming part is multiply-accumulate, which is proportional to square the number of neurons. A significant part of the execution time per iteration comes from the thresholding, nonlinear function, convergence check, updating the inputs for the next iteration, and communication. These parts are proportional to the number of neurons assigned to a module. Also contributing to the execution time is a constant time spent on the sequential parts of the programs. Included here are loop and branch overhead, and waiting for incoming data, or to transmit data. In Equation 10.1, execution time for the recall phase is split into fractions as a function of powers of n:

\[ t_{\text{recall}} = t_{\text{MAC}} \frac{n^2}{p} + t_{\text{lin}} \frac{n}{p} + t_{\text{const}} \]  \hspace{1cm} (10.1)

\( t_{\text{MAC}} \) in this experiment, with the weights in DRAM, is \( 2t_{\text{cycle}} = 2 \cdot 62.5 \text{ ns} = 125 \text{ ns} \). The computation time proportional to the number of neurons is complicated to estimate
and measure. Figure 10.3 and 10.4 shows the total execution time per iteration for the 128 and 1024 pattern sizes. The minimum time spent on multiply-accumulate is also shown.

![Figure 10.3: Execution time per iteration for small pattern sizes.](image1)

![Figure 10.4: Execution time per iteration for larger pattern sizes.](image2)

The minimum time for the multiply-accumulate phase per iteration is found from the formula

\[ t_{\text{MAC/iteration}} = \frac{n^2}{p} t_{\text{MAC}} = \frac{n^2}{p} 125 \text{ ns} \] (10.2)

Not included in Equation 10.2 are extra wait-states caused by crossing page boundaries, or DRAM accesses during refresh cycles.

As can be seen from Figure 10.3 and 10.4, only around half the execution time is spent on the computational demanding multiply-accumulate part, because of its very efficient
implementation. This also shows the importance of running multiply-accumulate at full speed. If the requirements for full speed multiply-accumulate cannot be met, the execution time is more than doubled. The single processor case illustrates the amount of time used on multiply-accumulate versus the other computations.

10.1.2 Pipelined Computation and Communication

The next experiment is an attempt to overlap communication and computation, to reduce the communication overhead. One broadcast ring is used, like in the previous test.

The modules first write their locally stored activation values to the output FIFOs, and then finish the calculations involving their part of the input vector. By that time, data from at least some of the other processors should have arrived, even for a small pattern size of 128 elements. To multiply the incoming data and the corresponding part of the weight matrix, sender information attached to the vectors is used. The convergence criterion is implemented as in the previously described experiment. Figure 10.5 illustrates the operations concerned with this strategy. Speed measurements are shown in Figure 10.6.

![Diagram](image)

**Figure 10.5:** Parallelisation with pipelined communication and computation.

Compared to the first experiment, the measured speed is significantly lower, which was not expected. As an example, for the eight-processor case and pattern size of 1024, the measured speed is only 82% of the corresponding value for the previous test. The explanation is twofold. First, and most important, is that the MACs are computed in an extra loop, which includes a test on the sender, and this adds overhead to the computations.
Figure 10.6: Speed measurements for pipelined communication and computation recall phase.

This loop overhead becomes more significant for small problem sizes, and increases with the number of processors. The processor has a deep instruction pipeline, which causes loops and branches to take 4 to 5 cycles. The overhead introduced by the extra loop is found from the one-processor case, and is 1–6%, depending on the pattern size. Second, the multiply-accumulate phase is less efficient, because the access pattern of the weight matrix introduces some extra wait-states. Although the time waiting for vectors to arrive, is reduced by this parallelising scheme, the sequential part of the program is larger, and increases with the number of processors.

10.1.3 Calculating and Communicating Partial Sums

A third parallelisation strategy for token ring is tried out. In contrast to the previous examples, the weight matrix is divided vertically among the processors. The processors compute concurrently the inner product of the weights and their part of the pattern vector. Then the locally calculated sums are broadcasted, and the total sum is computed, see Figure 10.7.

Since a word sent from a processor only need to be received by one of the modules, an addressing scheme with unique node addresses is best suited. The address to the destination node is attached to the transmitted vectors. To match the access pattern of the computations, the transposed weight matrix is stored.
As shown in Figure 10.8, the performance and scaling properties of this scheme lie between those of the two previous experiments. Extra computational load is concerned with this algorithm, because incoming data must be accumulated to the sums.

### 10.1.4 Comparison of Parallelising Strategies for Token Ring Protocol

The performance of the three variants of parallel recall is shown in Figure 10.9. Pattern size of 512 is selected for the comparison. Smaller patterns will cause more significant differences in speed and scaling properties. For larger patterns, the communication and computational overhead become less important, assumed that multiply-accumulate is run at full speed.
Figure 10.8: Speed measurements for recall phase based on computing and communicating partial sums.

Figure 10.9: Comparison of parallelising strategies for token ring protocol.
10.2 Comparing Direct Connections with Token Ring

Since Hopfield networks have full connectivity between the neurons, data from one module must be communicated to all the others, and the token ring protocol has shown to be efficient. However, solutions based on dedicated connections between neighbour modules are also investigated. In a such configuration, data from one module to another must go through the intermediate modules, and data is routed through the processor. Involving the processor in the communications is CPU-resource demanding. Advantages can be that the FIFOs can be used not only as communication buffers, but also as fast memory. It is possible to write incoming data simultaneously to both the local destination and to an output FIFO, by utilising parallel instructions. The communication time for each word between two neighbour processors is:

\[ t_{\text{comm}} = t_{\text{wf}} \cdot 4 t_{\text{ch}} \cdot t_{\text{if}} \]  

(10.3)

A vector based approach corresponding to the first experiment for the token ring protocol has been tested. With a ring of dedicated connections, the communication phase is accomplished in \( p - 1 \) stages, where each stage involves communication of \( n/p \) words between neighbour processors, see Figure 10.10.

The speed measurements in Figure 10.11 show that this usage of the communication links is less efficient than the token ring protocol.

Single Word Approach

A third scheme, based on doing all the computing involving an incoming data word before reading the next, has also been tested. The method is shown in Figure 10.12. An advantage is that the FIFOs are used not only as communication buffers, but also as fast memory in the computations.

Already before test-run, it was clear that this was not an efficient strategy. Each product is added to an element in the output vector stored in fast on-chip RAM. The pointer into this vector is incremented for each product, as shown in the following small piece of C-code:

```c
for (i=0; i<number_of_neurons; i++)
    *output++ = *weight++ *input;
```

The computation inside the loop involves three memory variables, and can be accomplished
in minimum two machine instructions. In the compiled code, three instructions are used in the inner loop, but this can be reduced to two by manual optimisation. Without copying the inner loop into the instruction cache, two instruction fetch cycles must be added to the execution time. This means that the computational intensive multiply-accumulate operation takes four times longer than for optimal speed. The weight matrix should be stored transposed, to avoid the extra instruction for incrementing the weight matrix pointer, and the extra wait-state by crossing a page boundary for pattern sizes of 256 and larger. In the worst case, five extra clock cycles are used per multiply-accumulate!

In Figure 10.13, the two lowest traces show measurements without storing the transposed weight matrix. Increased performance for larger pattern sizes is close to zero, because pattern sizes of 256 and larger cause crossing of a page boundary for each access. Improved performance is achieved by storing the weight matrix transposed, but still the speed is much less than for a vector based parallelising strategy (direct connections), which is shown for comparison.
10.3 Toroidal Lattice Architecture

The Toroidal Lattice Architecture is claimed to have close to linear scaling [14, 15], and should be well suited for regular algorithms with all-to-all communication. With the TLA on a two-dimensional grid, double communication bandwidth is achieved. Since the non-linear function and determination of convergence represent a considerable amount of computations, a permutation of the columns of the weight matrix is done, to ensure load balancing of these computations.

Figure 10.14 shows the distribution of the weight matrix among the processors in an example with 2 by 3 processors. The stages in the computations are illustrated in Figure 10.15.

The activation values are distributed along the vertical rings. Since the non-overlapping method with token ring protocol has proved to be most efficient so far, this strategy is used for the vertical communication. In horizontal direction, the local parts of the sums are calculated distributedly, and the total sum is found. Tests have been run to select the most efficient strategy for the horizontal communication, where the token ring based method described in Section 10.1.3, turned out to be best. Simulation results for TLA on 2×2 and 4×2 processors are shown in Figure 10.16.

The speed measurements as a function of problem size and number of processors approximate the results for the so far best method, which is non-overlapping communication and computation on a token ring. It was assumed that TLA, with double communication
10.3. Toroidal Lattice Architecture

![Diagram of Module 0, Module 1, and Module p-1](image)

Add Biases and Perform Non-Linear Function
Calculate Local Difference
Communication of Local Difference, total difference
If Total Difference \( \neq 0 \) GoTo 1)

**Figure 10.12:** Parallelisation with a computation order based on handling single words.

bandwidth, would be the most efficient solution. The results from TLA show that for performance and scale-up, the non-parallelisable part of a program is a more important limiting factor than the communication overhead. These findings are the same as for the pipelined approach. An advantage with TLA is that the FIFO buffering capacity that can be utilised is doubled. This is important if the amount of data to be received per iteration is larger than the FIFO depth.
Figure 10.13: Speed measurements for recall phase with direct connections, single word based algorithm.

Figure 10.14: Distribution of weight matrix for toroidal lattice architecture.
Figure 10.15: The stages in the computations for toroidal lattice architecture.
Figure 10.16: Speed measurements for recall phase with toroidal lattice architecture.

10.4 Optimal Memory Mapping

For optimum speed, data should be placed in the fastest memory large enough to hold the data. The memory hierarchy on a RENNS module consists of two 1 K words on-chip RAM, 128 K words of SRAM and 1 M words of DRAM. If the input and output pattern vectors are less than or equal 1 K, they are placed in the dual access on-chip RAM. For small patterns, or for a large number of processors, it may be possible to store the weight matrix in SRAM. This will double the multiply-accumulate speed. Figure 10.17 shows the speedup by keeping data in fastest possible memory.

For one and two processors, the upper problem size for which the weights can be kept in SRAM, is 256. For four and eight processors, this limit is increased to 512. The speedup of placing the weights is SRAM is close to 1.5.

10.5 Overhead Concerned with Testing the FIFO Flags

While reading from the incoming FIFO banks, the flags can be tested for each word, to check the contents of the FIFO bank. The FIFO flags are accessible through registers in the FIFO controller LCA. A read access to this register requires two wait-states, and
10.5. Overhead Concerned with Testing the FIFO Flags

![Graph showing speed-up from keeping data in fastest possible memory.](image)

Figure 10.17: Speed-up from keeping data in fastest possible memory.

considerable CPU-resources are used for this task. By utilising the programmable FIFO flags, the FIFOs can be tested to see if the total vector, or a part of it, has arrived. An amount of data corresponding to the offset of the programmable FIFO flags is read before next test of the flags.

Checking the FIFO-flags in a while-loop for each incoming word, increases the time to read each word with around 8 clock cycles, and adds 4-6% to the recall execution time. Figure 10.18 show the speedup by utilising the programmable FIFO flags, instead of checking the empty FIFO flag for each word.

![Chart showing speedup from utilising the programmable FIFO flags.](image)

Figure 10.18: Speedup from utilising the programmable FIFO flags.
10.6 Speed Measurements for the Calculation of Weights and Biases

If the weights and thresholds are calculated based on the outer product of the pattern vectors, the pattern matrix can be distributed to all the processors, and the determination of weights and biases is accomplished without any communication. The scaling is close to linear, see Figure 10.19.

![Figure 10.19: Time to determine weights and thresholds, outer product method.](image)

Using the method based on orthogonalisation of the patterns, which leads to an ANN overcoming many of the problems with the Hopfield network, is much more complicated and time consuming, and is therefore tried implemented in parallel. As described in Section 8.4.4, the SVD is implemented sequentially, while the other parts of the algorithm are parallelised.

Figure 10.20 and 10.21 show the time consumed by determining the weights and thresholds, for pattern size 128 and 512. The SVD is implemented sequentially, and adds a constant term to the execution time. Larger patterns than 512 can be solved by equipping one module with 4 M words DRAM, or by parallelising the SVD.
10.7 Peak Performance and Scale-up

The parallelisable part of Hopfield type ANNs are inner product calculation, thresholding, nonlinear function, and convergence check. The total communication is proportional to the number of neurons. The sequential fraction of a program, which should be kept as small as possible is loop and branch overhead, and waiting concerned with the communication.

The most computational intensive part is multiply-accumulate, which is proportional to the square of the pattern size, and is 40–60% of the total execution time, depending on the pattern size. Table 10.1 shows the scaling factor for different problem sizes.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small patterns</td>
</tr>
<tr>
<td></td>
<td>e.g. 128</td>
</tr>
<tr>
<td>2</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td>3.40</td>
</tr>
<tr>
<td>8</td>
<td>5.67</td>
</tr>
</tbody>
</table>

Table 10.1: System scaling for different problem sizes.

This shows that close to linear scaling is achieved for larger problems. Less good scale-up for smaller problem sizes is due to non-parallelisable parts of the program and communication costs, which become more significant for smaller patterns.

Measurements and calculations have shown that the time spent on waiting for data is 2–10% of the total execution time for small problems, while it is negligible compared to the total execution time for larger patterns. Loop and branches lead to nonsequential program execution and nonregular memory access patterns, and have a significant negative influence on the performance.
Chapter 11

Discussion and Conclusions

An eight-module version of the RENNS system is tested and analysed. Each module consists of one TMS320C30 DSP with a capacity of 33 MFLOPS, 128 K 32-bit words of SRAM, 1 M 32-bit words of DRAM, and a separate communication subsystem. Eight RENNS modules have a total maximum numerical capacity of 240 MFLOPS. The main objective has been to design a neurocomputer suitable for experimenting with different system architectures for simulation of ANNs. This is solved through a fairly complicated communication subsystem reconfigurable on three levels, which are the data-path, logic, and command level.

For performance analysis, Hopfield type neural networks on ring and two-dimensional mesh configurations are implemented. Two methods for calculation of weights and biases, with different requirements to the computational resources, are investigated. RENNS has shown high performance and good scaling properties for the various operations in Hopfield type neural networks.

11.1 Memory Capacity

A weight matrix of 1 M words can be stored in one module, if the code and the other data is placed in SRAM and on-chip RAM. For Hopfield networks, a pattern size of 32×32 can fit into one module, while the maximum size on 8 modules is approximately 50×50. Pattern sizes of 64×64 and larger is not uncommon in recurrent associative memories, which can be run on 16 modules. The memory capacity can be increased by adding more modules, or each module can be equipped with 4 M words of DRAM. The memory requirements for Hopfield networks increase with square the pattern size. For large problem sizes, adding
more modules to the system does not significantly increase the maximum pattern size. For this large Hopfield networks, secondary storage with a fast buffering mechanism should be considered.

The DRAM requires 1 wait-state for burst mode access and 2 wait-states otherwise. A speedup of around 1.5 is achieved by storing all data in memory accessible without wait-states. For Hopfield nets, this is only possible for pattern sizes less than 256-512.

### 11.2 Communication Bandwidth

The four bidirectional communication links, each with a capacity of 5 Mbyte/s, can theoretically feed the processor with a data rate of 20 Mbyte/s. Since the communication subsystem only has two FIFO banks in each direction, two incoming and outgoing channels can be handled simultaneously, limiting the communication bandwidth to 10 Mbyte/s per module. The communication bandwidth is less than the design goals, which was 20 Mbyte/s per channel. The speed limiting factor is the logic in the LCAs. Simulations have shown that for Hopfield nets, communication is less than 10% of the total execution time. For pattern sizes common in Hopfield networks, the communication bandwidth is not a speed limiting factor.

### 11.3 Reconfigurability

The communication architecture is reconfigurable at three levels. Experiences with changing the configuration at these three levels are achieved through this work.

Reconfiguring the physical connections between the processors is meant do be done by programming a communication switch. Since this switch is not yet implemented, reconfiguration at this level is done manually. To set up the different architectures investigated for recurrent associative memories, ring and two-dimensional mesh, require reconfiguration at this level. Experiments with one-dimensional ring and two-dimensional mesh architectures for a scalable application have shown that the significance of several performance limiting factors differ from assumptions based on theoretical considerations. For simulation of other ANN models, especially multilayered perceptrons, reconfigurability have been utilised to achieve higher performance by adapting the communication architecture to the structure of the application [62, 77].

The logic in the LCAs has gone through countless corrections and modifications during
design and testing of the communication subsystem. The LCAs receive their configuration
data from the processor. It is therefore possible to tailor the logic in the LCAs to each
program, within the limits of the complexity of the circuits. Use of reconfigurable logic has
been of great importance during design and development of the communication subsystem,
but apart from this, reconfigurability at this level is not utilised in this work.

Reconfiguration at the command level has been used extensively to set up the routing of
data between the stream LCAs and the FIFO banks, and to set the different command
registers in the stream LCAs. Configurations at this level are easily done. The possibility
to change the node addresses allows for quick setup of groups of modules with various com-
unication schemes, including broadcasting. For the toroidal lattice architecture (TLA),
different addressing schemes are used for the horizontal and the vertical rings, to tailor the
interprocessor communication to the communication requirements of the application.

11.4 Processor Topologies and Parallelising Strategies

The reconfigurability has been utilised to study different processor topologies and paral-
lelisation strategies for single layer recurrent fully connected networks. Of the processor
topologies and communication protocols investigated for the Hopfield net, one-dimensional
token ring with broadcast based communication, showed highest performance and best
scaling properties. For up to eight modules, a simple parallelising strategy with non-
overlapping computations and communication is the most efficient.

Pipelining the communication and computation phases was assumed to reduce the
waiting time concerned with the communication, and hence increase the performance.
Although the waiting time is reduced, more complex code a and larger non-parallelisable
part of the program instead reduce the performance compared to the simpler approach.

The time spent on waiting for data is 2–10% of the total execution time for small patterns,
while it is negligible compared to the total execution time for larger patterns. Loop and
branches lead to nonsequential program execution and unregular memory access patterns,
and have a high negative influence on the performance. This means that if an attempt
to reduce communication overhead introduces extra loops and tests in the inner product
calculations, the there is a chance that the total performance will be reduced. This is
confirmed in the attempt to overlap communication and computation.

Speed measurements for implementations of Hopfield networks on a two-dimensional grid
toroidal lattice architecture (TLA) approximate the results achieved for non-overlapping
communication and computation on a one-dimensional ring. Although the communication bandwidth is increased, this is not the most efficient architecture for the same reason as the pipelined approach.

For transmission of large vectors the time spent on communication is up to 1 ms per iteration. Although the communication time is less significant compared to the calculations, it should be reduced as much as possible. For large enough patterns, it is assumed to be a crossing point between performance of the non-overlapping algorithm and a pipelined approach or a mesh architecture. Simulations have shown that this intersection does not occur for problems sizes that fit into the 1 M word DRAM on the modules.

11.5 Peak Performance and Scale-up

Highest performance is measured for close to maximum pattern size on eight modules, with 40.82 MCPS for the character recognition problem. This means a utilisation of the numerical capacity of 5.88 FLOPS/CPS, which is within the range reported for other neurocomputers, see Section 4.6.1. The most computational intensive part is multiply-accumulate, which is proportional to the square of the pattern size, and is 40–60% of the total execution time, depending on the pattern size. The scaling factor for small problem sizes (e.g. vector length of 128) is 0.70–0.91. For larger patterns (e.g. 1024), the scaling factor increases to 0.96–0.99. Close to linear scaling is measured for larger problems. Less good scale-up is achieved for smaller problem sizes, due to non-parallelisable parts of the program and communication costs, which become more significant for smaller patterns.

11.6 Future Work

In the near future, a 16-module version will be operative and evaluated. An enhancement considered are adding subsystems optimised for vector operations to increase memory bandwidth.

Parallelising schemes for a range of ANN models should be developed and implemented on the RENNS computer system. It is believed that sophisticated processor topologies best can be utilised by modular neural networks [61]. These issues can be further investigated on a reconfigurable neurocomputer like RENNS. Continued work with recurrent associative neural networks should include storage and recall of 64×64 gray scale images. It will be a challenge to fully parallelise the enhanced method for calculation of weights and biases.
Bibliography


Appendix A

Host Interface, Software and Programming Environment

A.1 Overview

An objective for the RENNS project is to provide a complete environment for ANN simulations. It is a multiprocessor server, where the neural network services may be accessed from a host computer or over a LAN from a PC or workstation. Figure A.1 gives an overview of all parts of the computer system. A number of modules are grouped and connected to a common VME backplane bus. The high speed interprocessor communication data streams are shown to the left. A host computer with an Ethernet interface will provide remote access to RENNS.

A.2 Application Development

The development process of a neural network application can be viewed as follows:

Problem analysis will lead to one or several possible neural network architectures.

Evaluate algorithms to find the most suitable one. This is usually done by training different networks and measuring their performance. The developer must choose a network structure, the number of layers, the number of neurons, their interconnections, and a training algorithm. This stage also comprises experimenting with training parameters for the different algorithms.
Determine a **hardware architecture** that will efficiently solve the given problem. This architecture might be a special VLSI-device, special purpose computer, or a more traditional computer system. Closely connected to the choice of hardware architecture, is the strategy for mapping the neural network algorithm to the processing elements.

**Evaluate solution.** If the solution is not good enough, one will need to iterate through one or more of the above stages.

RENNs can be a useful tool during algorithm evaluation, and assist in the architecture determination process. RENNs will, as a general neural computer, support most of the known neural network algorithms. Thus, the algorithms can be evaluated for a given problem at a reasonable speed. By using the reconfiguration properties of the machine, RENNs can help in evaluating the possible hardware solutions, either by simulating it, or by being configured to directly match a proposed solution. It can also be (part of) the solution itself, either *as is*, or by forming a basis, from which unneeded features are removed, leading to a more cost-effective solution. The result can be a fixed, trained network, or an evolving network which learns during operation. For a fixed, trained network, the weight configuration can be used for implementation in VLSI devices. If the trained network is optimised with respect to size and precision, the hardware requirements to the application may be significantly reduced.

Application development is a tedious task due to many options and possible combinations, and requires powerful and specialised hardware. In contrast, an optimised, solution can often be run on a much simpler computer, for example a standard PC.
A suitable and user friendly programming environment is important for efficient use of the RENNS, especially for non computer science specialists. The users should be given the opportunity to concentrate on the problem, instead of on the implementation and details of the software and the underlying hardware. It is not likely that researchers on ANN models and applications would take the effort to start developing an application from scratch, and being concerned about the communication architecture and primitives for the target computer.

The interest in ANNs has lead to the development of a range of software packages, both commercially and freely available. Most of these are software environments only, targeted to run on general sequential hardware platforms like PCs and workstations. Other are integrated hardware and software environments, which utilises the computing power of coprocessor boards or stand-alone neurocomputers. One of the earliest complete ANN computing systems, was the Computation Network Environment (CONE), software running on the Network Emulation Processor (NEP) [10]. This work was published already in 1987. In [88], Treleaven gives a short survey of some of the neural programming environments, but the main purpose of that paper is to present the PYGMALION Neural Network Programming Environment, which is a complete software environment that can be interfaced to various computers. Several publications concern software and integrated ANN simulation environments [86], [30].

The various programming environments for ANN development have many common features. The building blocks of a typical programming environment are shown in Figure A.2 (based on Treleaven [88]).

A programming environment for RENNS will in many ways resemble other ANN environments. However, RENNS has some special properties, the reconfigurability, which must be considered when designing the software.

A programming environment for the RENNS should be general, which means it should support a range of ANN architectures, and give the user the ability to create other models. The programming environment should support all stages in the application development process, which for the software part can be divided into the three following phases:

- design,
- training and testing, and
- application.
Additional requirements to the environment are to provide a user interface and the ability to control a simulation.

Unless the ANN is run in batch mode, every simulation environment provides an interactive run-time user interface. The user interface includes a command interface, and usually graphical tools to represent various states of the network. These allow the user to examine the state of the network by displaying variables, such as activation values in the layers, weight values, and for some ANNs, the calculated errors. Another important feature is to give the users the ability to examine the neural network during training. Training can be very time consuming, and to start training the network from the initialisation stage for each change, should not be necessary. If the programming environment allows the user to stop and restart training, save intermediate results, and change execution parameters, this
can speed up the training process significantly. Execution parameters to be changed can for instance be network structure, learning rate, and training set.

The user interface should be general and support the development process of a wide range of ANN models, and ideally also be flexible enough to give the user the opportunity to tailor the displayed information to the specific task. The monitoring of the ANNs should be provided on different detail levels. However, for those users who want to control and monitor the parallel execution, the user interface should provide some special information about RENNS. The two main categories of information can be further divided as follows:

1. General information on the ANN development processes:
   • Neuron states.
   • Weight values.
   • Error during training.
   • Training parameters.
   • Input patterns.
   • Output patterns.

2. Specific information on the computational tools:
   • The distribution of the ANNs among the processors.
   • The work load on the processors.
   • The configuration of the interprocessor communication network.

In some programming environments, the user interface and the simulation parts are often integrated within a single program. This is usually the case in environments based on PC platforms. However, some simulation environments support execution on a remote machine. When the execution is to take place on a remote machine, a client-server model is often used. The user interface is the client, while the remote machine runs the server routines. This is an appealing approach for RENNS. The intention is to control the ANN simulations from a remote machine, and to use RENNS as a server, where the ANN services can be accessed over a LAN. RENNS can then be operated from a standard workstation with a user-friendly graphical interface, and the users need not be located near the machine.

A programming environment for the RENNS, based on a client-server model, will consist of three main parts:

• The Client Interface, which can reside on a remote PC or workstation, or on the host computer.
- The Main Server, which will run on the host computer, and again be a client to the RENNS modules.

- The Node Servers.

To get a complete programming environment for the RENNS, one alternative is to develop a system for RENNS especially, but even a simple system is a project of its own. The probably most efficient approach is to interface an existing flexible and open system, like for example PYGMALION [88].

A.4 Mapping Programs to Multiple Modules

In relation to the Pygmalion Esprit project, research has been done to develop a generic strategy for mapping neural network models on Transputer based machines [3, 4]. The distribution strategy is capable of adapting to both a wide range of ANN models, and to the targeted Transputer-based machine. The mapping is divided into two phases, in the first phase the strategy identifies the type of parallelism in the specific ANN model. During the second phase, the strategy decides upon a particular distribution of the ANN model according to the target machine parallel characteristics, which are the number of Transputers, configuration flexibility, and communication mechanisms. A complete implementation of a prototype has been targeted to the PARSYS SN1000 Transputer-based machine with 48 Transputers. The mapper takes an ANN specification written in nC and produces a parallel description in 3L Parallel C, which was chosen as target language, together with the Tiny communication software package. nC is a machine-independent neural network specification language based on a subset of C [88]. An important feature of nC, is that the language comprises explicit identification of parallelism, both data parallelism and process parallelism.

A similar mapping strategy could be invented on RENNS, which can form regular architectures not unlike Transputers. An ANN description in nC can be mapped to a number of RENNS modules, using a target language developed for the RENNS together with a communication library.
A.5 Host Interface

Hardware

RENNS includes a host computer with Ethernet interface, which is the entrance from Unix workstations. The host computer provides remote file I/O, other Remote Procedure Calls (RPC), and remote login. RPC is implemented primarily to replace the PYGMALION simulation part with simulations on RENNS, but the client-server model is in general a convenient way to have interacting programs residing on different machines.

Early in the project, it was decided to use the Enet-1, which is based upon the Local Area Network Controller for Ethernet (LANCE) device and a Motorola 68020 microprocessor. The motivation for this choice, was the low cost of the Enet-1. Because of various problems with Enet-1, it was replaced with a SPARC node running UNIX. It is planned that the host node should act as a server for a client on a workstation, and that each of the RENNS modules again is server to a client on the SPARC.

An ANN description will be fed to the host node by a file or a script of commands, which initialise and control the simulation. The script would download the required programs on the modules, configure the communication, set up the mapping information, and start the execution on command from the user.

From the host interface board, access to the modules is given through a VME-bus. The interface board and the modules are connected to a common VME backplane bus. The number of modules is limited by the 16 slots of the backplane bus, which means that maximum 15 RENNS modules can be connected to one bus. To overcome this limitation, modules can be grouped, with only one module in the group connected to the VME bus. Alternatively another backplane bus can be added, but the extra costs of a such solution will not contribute to any speed-up.

System Software

System software is needed for both the host interface board and the modules, for control of the processing, resource management, and communication with external devices.

The system software on the modules must control and use

- the data streams,
Appendix A. Host Interface, Software and Programming Environment

- the serial links,
- the VME bus, and
- the console.

Until now, the only system software on the modules has been a monitor, which provides services for downloading programs, starting program execution, writing to and reading from a memory location or a memory area, and dumping the CPU registers. The user communicates with the monitor through an RS232 serial interface.

The original idea was to have the same operating system, Xinu, on both the host interface board and the modules. Xinu was selected because it has a small kernel, but offers most of the features needed, like multitasking. In addition, Xinu is well documented. Another important issue is that some versions of Xinu are free. For documentation on Xinu, see [9]. Although much work has been done in a students project to port Xinu to the Enet-1, with the new SPARC host interface board it was decided to run UNIX. Xinu is still planned used as operating system on the modules.

Processes on the modules must be able to communicate with each other using datagram or stream pseudo-devices. This will be done through the VME bus, which means that the software on the interface board must support this communication.

Fast interprocessor communication must be provided, and can be accomplished through a library of optimised communication routines operating on the high-speed links.

System software and programming environment for RENNS are discussed in [49].
Appendix B

Circuit Board Layout for the Communication Subsystem

The Communication Subsystem PCB is designed as a “piggy-back” board to the processor board, which has full height, full length Eurocard dimensions. The Communication Subsystem PBC is an eight layer board with dimensions 239.4x116.8 mm or 9425x4600 mil (the same width as the processor board, and half the height).

The communication subsystem PCB consists of the following eight layers:

- **Top layer**: Signal layer.
- **1. layer**: +5 V. Contains the processor clock (H3). In addition, some of the FIFO read and write strobes are routed in this layer.
- **2. layer**: Signal layer.
- **3. layer**: Signal layer.
- **4. layer**: GND. The communication subsystem clock (from an oscillator on the board) is distributed in this layer.
- **5. layer**: GND. Contains most of the strobe signals (FIFO read and write strobes, and IOSTRB and MSTRB from the processor bus).
- **6. layer**: Signal layer.
- **Bottom layer**: GND.

Figure B.1 shows the component placement on the communication subsystem board.
Figure B.1: Component placement on the communication subsystem printed circuit board
Appendix C

Program Listings

C.1 Definitions for the Hopfield Net

/*****************************/
* Definitions for Hopfield network, recognition of characters *
* August 9. 1994 *
* Lisbet Utne *
************************************************************************/
#define PATTERNS 62 /* Number of patterns */
#define WIDTH 24 /* Input pattern width */
#define HEIGHT 24 /* Input pattern height */

#define MAXIT 100 /* Max number of iterations during recall */
#define NOISE 25 /* Percent noise added to the patterns */
#define TAU 100 /* Parameter for calc. of weights and biases */
#define TS 0.15 /* Parameter for calc. of weights and biases */
#define MAXPROCS 16 /* Maximum number of processors */

#define INT_RAM1 ((float *) 0x809800)
#define INT_RAM2 ((float *) 0x809c00)
#define DRAM ((float *) 0x900000)

#define max(A,B) ((A) > (B) ? (A) : (B))
C.2 The Main Program

#include <stdio.h>
#include "hopfield.h"
#include "t_ringconf.h" /* or s_ringconf.h or tlaconf.h */
#include "ringExtern.h"
#include "comExtern.h"

extern long bitmap[ ]; /* Array which holds the input patterns */

int node;       /* Node ID */
int neurons;    /* Number of neurons */
int processors; /* Number of processors */
int series;     /* Interval between checking FIFO flags */
int flimit;     /* Offset value for programmable FIFO flags */
int no;         /* Number of neurons per processor */

float *int_ram1=INT_RAM1;
float *int_ram2=INT_RAM2;
float *dram=DRAM;
float biasvect[WIDTH*HEIGHT];
float *biases=&biasvect[0];
long int pmatr[PATTERNS*WIDTH*HEIGHT];

int *id=(int *) 0x804400;

/* Processor registers for counter/timer, */
/* used for time measurements */
volatile unsigned long *tctrl = (unsigned long *) 0x808020;
volatile unsigned long *tcount = (unsigned long *) 0x808024;
volatile unsigned long *tperiod = (unsigned long *) 0x808028;

void Noise(float *npattern, float *ipattern, int length, int percent)
/* Adds 'percent' noise to the pattern */
{
int no, invert, i;
float *npt;

npt = npattern;
for (i = 0; i < length; i++)
    *npt++ = *ipattern++;

if (percent == 0) return;
no = (int)(100/percent);
npt = npattern;
for (i = 0; i < length/no; i++)
{
    invert = rand()%no;
    if ((npt+invert) <= (npattern+length))
        if (*(npt+invert) == 1.0)
            *(npt+invert) = -1.0;
        else
            *(npt+invert) = 1.0;
    npt += no;
}

int Compare(float *pattern1, float *pattern2, int length)
/* Returns the number of differing bits in two patterns */
{
    int i;
    int diff = 0;

    for (i = 0; i < length; i++)
        if (*pattern1++ != *pattern2++)
            diff++;
    return diff;
}

main()
{
    int width=WIDTH;
    int height=HEIGHT;
    long int *patterns=&pmatr[0];
    float *weights=dram, *wp;
    float *inputs=int_rami;
    long int i, j, diff, inchar, temp;
    long int iterations, error, accerror, maxerror, acciter, maxiter;
float averror, aviter;
float acct_recall=0.0;
float cp, cps, mcps;
volatile unsigned long starttime;
volatile unsigned long t_recall;
volatile unsigned long t_calcweights;

printf("Hopfield: Start, Patternsize \%dx\%d\n",width,heigth);
printf("Number of processors: ");
processors=getchar()-'0';
printf("\n");

*tperiod = 0xffffffff; /* Max counter period */
*tctrl = 0x2c0; /* Reset and start counter */
accerror=maxerror=acciter=maxiter=0;
acc_trecall=0.0;
node = (*id & 0xf);
rings=1;
temp=no;
series=1;
while (temp > 254)
{
    temp/=2;
    series*=2;
}
flimit=temp/2;
ConfigureRings(flimit);

printf("Hopfield: Calculation of weights and biases...\n");

UnpackPattern(bitmap, patterns);
starttime=**tcount;
ParCalcweights(weights,biases,patterns);
t_calcweights=**tcount-starttime;
*tctrl = 0x2c0; /* Reset and start counter */

printf("Hopfield: Recall...\n");

for (i = 0; i < PATTERNS; i++)
{
    Noise(inputs, patterns+i*neurons, neurons, NOISE);
    starttime=**tcount;
    iterations=ParRecall(inputs,weights,biases);
acct_recall += (float)t_recall;
acciter += iterations;
maxiter = max(iterations, maxiter);
error = compare(patterns+i*neurons,inputs,neurons);
maxerror = max(error, maxerror);
acerror += error;
}
printf("Hopfield: Finished recall\n");
printf("Simulation results\n");
averror = acerror/PATTERNS;
aviter = acciter/PATTERNS;
printf("\nError: Max=%d  Av=%f Iterations: Max=%d  Av=%f
        Sum=%d\n", maxerror, averror, maxiter, aviter, acciter);

cp = neurons*neurons*acciter;
acct_recall /= 8000000.0;
t_calcweights /= 8000000.0;
cps = cp / acct_recall;
mcps = cps / 1000000.0;
printf("Execution time, Calc. weights and biases: %6.2f sec\n",
t_calcweights);
printf("Execution time, Recall: %8.4f sec\n", acct_recall);
printf("Performance %7.3f MCPS\n", mcps);
}

C.3 Memory Map and Section's Allocation into Memory

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
  SRAM:   org = 0x000040 len = 0x1FFBF /* SRAM */
  VECTORS: org = 0x000000 len = 0x40 /* Interrupt vectors */
  ROM:    org = 0x0f0040 len = 0xffc0 /* EPROM */
  RAM0:   org = 0x809800 len = 0x400 /* RAM BLOCK 0 */
  RAM1:   org = 0x809c00 len = 0x400 /* RAM BLOCK 1 */
  DRAM:   org = 0x900000 len = 0x100000 /* DRAM */

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
C.4 Parallel Recall, Different Parallelisation and Mapping Strategies

Token Ring, Non-overlapping Computation and Communication

For simplification of the code, the parts handling the case with the neurons unevenly distributed among the processors are omitted.
/* Bit masks for operations on the FIFO flag register in the */
/* FIFO Controller. */
#define IF_B_MASK 0x0f /* Input FIFO flag mask */
#define OF_B_MASK 0x0f /* Output FIFO flag mask */
#define IF_B_EMPTY 0x04 /* Input FIFO empty flag */
#define OF_B_EMPTY 0x04 /* Output FIFO empty flag */
#define IF_B_AE 0x02 /* Input FIFO almost empty flag */

extern int node; /* Node ID */
extern int neurons; /* Number of neurons */
extern int processors; /* Number of processors */
extern int series; /* Interval between checking FIFO flags */
extern int flimit; /* Offset value for FIFO flags */
extern int no; /* Number of neurons per processor */
extern float *int_ram2; /* Start address for on-chip RAM bank 2 */

int ParRecall(float *inputs, float *weights, float *biases)
{
    float *outputs = int_ram2;
    float *local_inputs;
    float *wpt, *bpt, *ipt, *opt, temp;
    int i, j, s, n, iterations, diff;
    int proc, sender;

    local_inputs = inputs + node * no;

    /* Iterate until outputs stabilized */
    iterations = 0;
    do
    {
        iterations++;
        wpt = weights;
        bpt = biases;
        opt = outputs;

        /* Calculate new outputs, store result in output */
        for (i = 0; i < no; i++)
        {
            for (ipt = inputs, temp = 0.0, j = 0; j < neurons; j++)
                temp += *wpt++ * *ipt++;
            temp += *bpt++;
        }
    }
    while (diff > 0.01); /* Add condition to terminate loop */
}

if (temp < 0.0) /* Hard-limit nonlinearity */
    *opt++ = -1.0;
else if (temp > 0.0)
    *opt++ = 1.0;
else
    *opt++ = *(local_inputs+i);
}

/* Calculate difference and update inputs for next iteration */
for (diff=0, ipt=local_inputs, opt=outputs, i=0; i<no; i++)
{
    if (*ipt != *opt)
        diff++;
    *ipt++ = *opt++;
}

/* Check total difference */
while ((fctrl->outflag & OF_B_MASK) != OF_B_EMPTY);
*osbptr = B_ADDR; /* Send address for broadcast comm. */
*ofbptri = diff; /* Send calculated difference */
*osbptr = TOKEN; /* Insert new token to the ring */
for (i=1; i<processors; i++)
{
    while ((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY);
    diff += *ifbptri; /* Accumulate received difference */
}

/* Broadcast result */
while ((fctrl->outflag & OF_B_MASK) != OF_B_EMPTY);
*osbptr = B_ADDR; /* Send address for broadcast comm. */
*ofbptri = node; /* Send result vector */
for (opt=outputs, n=0; n<no; n++)
*ofbptr = *opt++;
*osbptr = TOKEN; /* Insert new token to the ring */

/* Read result from the other processors */
for (i=1; i<processors; i++)
{
    while ((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY);
    sender = *ifbptri;
    for (ipt=inputs+offset+sender*no, s=0; s<series; s++)
    {
        while (!(fctrl->inflag & IF_B_AE));
    }
}
for (n=0; n<flimit*2; n++)
    *ipt++ = *ifbptr;
}
for (n=series*flimit*2; n<no; n++)
{
    while (((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY);
    *ipt++ = *ifbptr;
}
} while ((diff != 0) && (iterations < MAXIT));
return iterations;

Token Ring, Pipelined Computation and Communication

The definitions and external declaration part is identical with the previous function, and is therefore omitted.

**************************************************************************
* Hopfield net, parallel recall of one pattern
* Token ring communication protocol, one broadcast ring
* Pipelined computation and communication
* Input vector length: neurons/processors
* Output vector length: neurons/processors
* *
* July 23. 1994
* Lisbet Utne
**************************************************************************/

int ParRecall(float *inputs, float *weights, float *biases)
{
    float *outputs=int_ram2;
    float *wpt, *bpt, *ipt, *opt, temp;
    float *local_inputs, *received_inputs;
    int i, j, s, n, iterations, diff;
    int proc, totdiff, sender, offset;

    local_inputs=inputs;
    received_inputs=inputs+no;
    offset=node*no;
/* Iterate until outputs stabilized */
iterations = 0;
do {
    iterations++;
    /* Send local part of input vector */
    while (((fctrl->outflag & OF_B_MASK) != OF_B_EMPTY));
    *osbptr=B_ADDR;  /* Vector address field */
    *ofbptri = node;  /* Send node number */
    for (ipt=local_inputs, j=0; j<no; j++)
        *ofbptri=ipt++;  /* Send data */
    *osbptri=TOKEN;  /* Insert new token to the ring */

    /* Calculate new outputs, store result in output */
    /* MAC using own input data */
    for (opt=outputs, i=0; i<no; i++)
        {
            for (temp=0.0, ipt=local_inputs,
                wpt=weights+i*neurons+offset, j=0; j<no; j++)
                temp += *wpt++ * *ipt++;
            *opt++ = temp;
        }

    /* MAC while receiving data */
    for (proc=1; proc<processors; proc++)
        {
            while (((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY));
            sender = *ifbptri;  /* Read sending node */
            for (ipt=received_inputs, s=0; s<series; s++)
                {
                    while (!((fctrl->inflag & IF_B_AE))));  /* Wait */
                    for (j=0; j<flimit*2; j++)
                        *ipt++ = *ifbptri;  /* Read data */
                }
            for (opt=outputs, i=0; i<no; i++)
                {
                    for (temp=0.0, ipt=received_inputs,
                        wpt=weights+i*neurons+sender*no,
                        j=0; j<no; j++)
                        temp += *wpt++ * *ipt++;
                    *opt++ += temp;
            }
}
/* Add biases, perform non-linear function, */
/* Calculate local diff. and update inputs for next iteration */
for (diff=0, opt=outputs, bpt=biases, ipt=local_inputs,
     i=0; i<no; i++)
{
    *opt += *bpt++;

    if (*opt < 0.0) /* Hard-limit nonlinearity */
        *opt = -1.0;
    else if (*opt > 0.0)
        *opt = 1.0;
    else
        *opt = *ipt;

    if (*opt != *ipt)
        diff++;
    *ipt++ = *opt++;
}

/* Check total difference */
while ((fctrl->outflag & OF_B_MASK) != OF_B_EMPTY); /* Wait */
*osbptr = B_ADDR; /* Vector address field */
*ofbptri = diff;  /* Send difference */
*osbptr = TOKEN; /* Insert new token to the ring */
for (proc=1; proc<processors; proc++)
{
    while ((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY); /* Wait */
    diff += *ifbptri; /* Read diff. from the other processors */
}
while ((diff != 0) && (iterations < MAXIT));
return iterations;
}

Token Ring, Parallelisation by Computing and Communicating Partial Sums

The definitions and external declaration part is identical with the function in C.4, and is therefore omitted.
/*******************************************************************************
* Hopfield net, parallel recall of one pattern
* Token ring communication protocol
* One ring, where the nodes are given unique addresses
* Calculate partial sums in the processors,
 * find total sums between each iteration.
* Input vector length: neurons/processors
* Output vector length: neurons
* Requires transposed weight matrix.
* *
* August 5. 1994
* Lisbet Utne
*******************************************************************************/

/* Bit masks for operations on the FIFO flag register in the FIFO Controller. FIFO banks 2 and 4 are used for sending/receiving */
#define IF_MASK 0xf0
#define OF_MASK 0xf0
#define IF_EMPTY 0x40
#define OF_EMPTY 0x40
#define OF_FULL 0x20
#define IF_AF 0x20
#define OF_AF 0x30

Other definitions and external declarations are the same as for the function in C.4.

int ParRecall(float *inputs, float *weights, float *biases)
{
    float *outputs=int_ram2;
    float *wpt, *bpt, *ipt, *opt, temp;
    int i, j, m, n, iterations, diff;
    int proc, sender, s;

    /* Iterate until outputs stabilized */
    iterations = 0;
    do
    {
        iterations++;
        wpt = weights;
        bpt = biases;

        /* Calculate partial sums for new outputs */
while ((fctrl->outflag & OF_MASK) != OF_EMPTY);
for (proc=0; proc<processors; proc++)
{
  if (proc == node)
  {
    /* Calculate and store result in output */
    for (opt=outputs, i=0; i<no; i++)
      { 
      for (ipt=inputs, temp=0.0, j=0; j<no; j++)
        temp += *wpt++ * *ipt++;
      *opt++ = temp;
      }
  }
  else
  {
    /* Calculate and send partial sum */
    *osptr=(proc | 0x60) << 24; /* Send sum to node proc */
    for (i=0; i<no; i++)
      { 
      for (ipt=inputs, temp=0.0, j=0; j<no; j++)
        temp += *wpt++ * *ipt++;
      *ofptr = temp;
      }
    *osptr=TOKEN;
    }

  /* Read and accumulate partial sums */
  for (proc=1; proc<processors; proc++)
  {
    for (opt=outputs, i=0; i<series; i++)
      { 
      while (!((fctrl->inflag & IF_AE)); /* Wait for data */
        for (j=0; j<flimit*2; j++)
          *opt++ += *ifptr; /* Read data */
      }
  }

  /* Add biases, perform non-linear function, */
  /* Calculate difference and update inputs for next iteration */
  for (diff=0, opt=outputs, ipt=inputs, i=0; i<no; i++)
  { 
  *opt += *bpt++;
  }
if (*opt < 0.0) /* Hard-limit nonlinearity */
    *opt = -1.0;
else if (*opt > 0.0)
    *opt = 1.0;
else
    *opt = *ipt;

if (*ipt != *opt)
    diff++;
*ipt++ = *opt++;
"

/* Check total difference */
while (((fctrl->outflag & OF_B_MASK) != OF_B_EMPTY);
    *osbptr = B_ADDR; /* Vector address field */
    *ofbptr = diff; /* Send difference */
    *osbptr = TOKEN; /* Insert new token to the ring */
    for (proc=1; proc<processors; proc++)
    {
        while (((fctrl->inflag & IF_B_MASK) == IF_B_EMPTY);
            diff += *ifbptr; /* Read and accumulate difference */
    }
} while ((diff != 0) && (iterations < MAXIT));
return iterations;
"

Direct Connections, Vector Based Non-overlapping Computation and Communication

******************************************************************************
* Hopfield net, parallel recall of one pattern *                      *
* One ring formed of direct connections *                            *
* Input vector length: neurons *                                     *
* Output vector length: neurons/processors *                        *
*                                 *                                  *
* July 24. 1994 *                                                      *
* Lisbet Utne *                                                       *
******************************************************************************
#include "s_ringconf.h"

/* Bit masks for operations on the FIFO flag register in the FIFO Controller. FIFO banks 1 and 3 are used for sending/receiving */
#define IFMASK 0x0f
#define OFMASK 0x0f
#define IFEMPTY 0x04
#define IFFULL 0x02
#define OFEMPTY 0x04
#define IFAE 0x02
#define OFAF 0x03

Other definitions and external declarations, see function in C.4.

int ParRecall(float *inputs, float *weights, float *biases)
{
    float *outputs=init_ram2;
    float *wpt, *bpt, *ipt, *opt, temp;
    int i, j, n, s, iterations, diff;
    int proc, totdiff;
    int offset[MAXPROCS];

    /* Table for quick look-up of where in the input vector */
    /* incoming data should be placed */
    n=node;
    for (proc=0; proc<processors; proc++)
    {
        offset[proc]=n*no;
        if (--n<0) n=processors-1;
    }

    /* Iterate until outputs stabilized */
    iterations = 0;
    do
    {
        iterations++;
        wpt = weights;
        bpt = biases;
        opt = outputs;

        /* Communication phase */
        /* Send own part of input vector to neighbour */
while ((fctrl->outflag & OFMASK) != OFEMPTY); /* Wait */
for (ipt=inputs+node*no, j=0; j<no; j++)
  *ofptr=*ipt++;

/* Receiving and transmitting */
for (proc=1; proc<processors-1; proc++)
{
    while ((fctrl->outflag & OFMASK) != OFEMPTY);
    ipt=inputs+offset[proc];
    for (s=0; s<series; s++)
    {
        while (!((fctrl->inflag & IFAE)));
        for (j=0; j<flimit*2; j++)
            *ofptr = *ipt++ = *ifptr;
    }
}

/* Receive data from last processor */
ipt=inputs+offset[processors-1];
for (s=0; s<series; s++)
{
    while (!((fctrl->inflag & IFAE)));
    for (j=0; j<flimit*2; j++)
        *ipt++ = *ifptr;
}

/* MAC, add biases and perform non-linear function, */
/* store result in output */
for (i=0; i<no; i++)
{
    for (temp=0.0, ipt=inputs, j=0; j<neurons; j++)
    temp += *wpt++ * ipt++;
    temp += *bpt++;

    if (temp < 0.0) /* Hard-limit nonlinearity */
        *opt++ = -1.0;
    else if (temp > 0.0)
        *opt++ = 1.0;
    else
        *opt++ = *(inputs+offset[0]+i);
}

/* Calculate local diff. and update inputs for next iteration */
for (diff=0, ipt=inputs+offset[0], opt=outputs, i=0; i<no; i++)
{
   if (*opt != *ipt)
      diff++;
   *ipt++ = *opt++;
}

/* Check total difference, write and receive proc-1 times */
for (totdiff=diff, proc=1; proc<processors; proc++)
{
   while (((fctrl->outflag & OFMASK) != OFEMPTY));
   *ofptri = diff;
   while (((fctrl->inflag & IFMASK) == IFEMPTY));
   diff = *ifptri;
   totdiff+=diff;
}
diff=totdiff;
} while ((diff != 0) && (iterations < MAXIT));
return iterations;

Direct Connections, Single Word Approach, Utilising the FIFOs as Fast Storage

**************************************************************************
* Hopfield net, parallel recall of one pattern  
* One ring formed of direct connections 
* Do all processing concerned with that word immediately 
* after it has arrived. (Do not require temporary storage 
* of incoming data.) 
* 
* July 24. 1994 
* Lisbet Utne 
**************************************************************************

Definitions and external declarations are the same as for the previous function (C.4).

int ParRecall(float *inputs, float *weights, float *biases)
{ float *outputs=int_ram2;
    volatile float indata;
    int i, j, n, iterations, diff;
    int proc, totdiff;
    int offset[MAXPROCS];

    /* Table for quick look-up of where in the input vector
       incoming data should be placed */
    n=node;
    for (proc=0; proc<processors; proc++)
    {
        offset[proc]=n*max_no;
        if (--n<0) n=processors-1;
    }

    /* Iterate until outputs stabilized */
    iterations = 0;
    do
    {
        iterations++;
        /* Calculate new outputs, store result in output */
        for (opt=outputs, i=0; i<no; i++)
            *opt++ = 0.0;
        /* For all neurons in the processor */
        for (ipt=inputs+offset[0], i=0; i<no; i++, ipt++)
        {
            for (opt=outputs,wpt=weights+offset[0]+i,
                j=0; j<no; j++, wpt+=neurons)
                *opt++ += *wpt * *ipt;
            while ((fcntl->outflag & OFMASK) != OFEMPTY);
            *ofptr = *ipt;
        }

        /* Receive and retransmit data from the other processors-2
           processors */
        for (proc=1; proc<processors-1; proc++)
        {
            while ((fcntl->outflag & OFMASK) != OFEMPTY);
            for (i=0; i<length[proc]; i++)
            {
                while ((fcntl->inflag & IFMASK) == IFEMPTY);
            }
        }
    }
```c
*ofptr=indata=*ifptr;
    for (opt=outputs,wpt=weights+offset[proc]+i,
         j=0; j<no; j++,wpt+=neurons)
        *opt++ += *wpt * indata;
}

/* Receive from adjacent processor */
for (i=0; i<length[processors-1]; i++)
{
    while ((fctrl->inflag & IFMASK) == IFEMPTY);
    indata=*ifptr;
    for (opt=outputs,wpt=weights+offset[processors-1]+i,
         j=0; j<no; j++,wpt+=neurons)
        *opt++ += *wpt * indata;
}

/* Calculate nonlinearity and local difference */
for (diff=0,ipt=inputs+offset[0],bpt=biases,opt=outputs,
      i=0; i<no; i++)
{
    *opt += *bpt++;

    if (*opt < 0.0) /* Hard-limit nonlinearity */
        *opt = -1.0;
    else if (*opt > 0.0)
        *opt = 1.0;
    else
        *opt = *ipt;

    if (*opt != *ipt)
        diff++;
    *ipt++ = *opt++;
}

/* Check total difference */
for (totdiff=diff, proc=1; proc<processors; proc++)
{
    while (((fctrl->outflag & OFMASK) != OFEMPTY));
    *ofptr = diff; /* Write word */
    while (((fctrl->inflag & IFMASK) == IFEMPTY));
    diff = *ifptr;
```
```
totdiff+=diff;
}
diff=totdiff;
} while ((diff != 0) && (iterations < MAXIT));
return iterations;
}

Toroidal Lattice Architecture

Recall Phase

/***************************
* Hopfield net, parallel recall of one pattern
* 2-D toroidal mesh, token ring communication protocol
* Non-overlapping communication and computation
* Input vector length: neurons
* Output vector length: neurons/processors
* 
* August 9. 1994
* Lisbet Utne
***************************/

extern int neurons; /* Number of neurons */
extern int processors; /* Number of processors */
extern int series; /* Interval between checking FIFO flags */
extern int flimit; /* Programmable FIFO flag offset */
extern int no; /* Number of neurons per processor */

#include <stdio.h>
#include <math.h>
#include "hopfield.h"
#include "tlaconf.h"
#include "com_extern.h"
#include "tla_extern.h"

/* FIFO flag bit masks, horizontal communication, */
/* FIFO bank 1 and 3 */
#define IF_H_MASK 0x0f /* Input FIFO flag mask */
#define OF_H_MASK 0x0f /* Output FIFO flag mask */
#define IF_H_EMPTY 0x04 /* Input FIFO empty flag */
#define OF_H_EMPTY 0x04 /* Output FIFO empty flag */
#define IF_H_AE 0x02 /* Input FIFO almost empty flag */
#define OF_H_AF 0x03 /* Output FIFO almost empty flag */

/* FIFO flag bit masks, vertical communication, */
/* FIFO bank 2 and 4 */
#define IF_V_MASK 0xf0 /* Input FIFO flag mask */
#define OF_V_MASK 0xf0 /* Output FIFO flag mask */
#define IF_V_EMPTY 0x40 /* Input FIFO empty flag */
#define OF_V_EMPTY 0x40 /* Output FIFO empty flag */
#define IF_V_AE 0x20 /* Input FIFO almost empty flag */
#define OF_V_AE 0x30 /* Output FIFO almost empty flag */

extern int node; /* Node ID */
extern int Qproc; /* Number of processors in horizontal direction */
extern int Pproc; /* Number of processors in vertical direction */

/* Tables and variables that hold information on the distribution of the weight matrix */
extern int H[2][MAXPROCS*MAXPROCS], V[2][MAXPROCS];
extern int hb[2][MAXPROCS*MAXPROCS], he[2][MAXPROCS*MAXPROCS];
extern int vb[2][MAXPROCS], ve[2][MAXPROCS];
extern int rows, columns;

extern int h_series; /* Interval between checking FIFO flags */
extern int v_series; /* Interval between checking FIFO flags */
extern unsigned h_flimit; /* Offset value for programmable */
extern unsigned v_flimit; /* FIFO flags */

extern float *int_ram2; /* Start address for on-chip RAM bank 2 */

int ParRecall(float *inputs, float *weights, float *biases)
{
    float *outputs=int_ram2;
    int i, j, p, q, s, iterations, diff, sender;
    volatile unsigned long starttime;
    int offset[MAXPROCS];

    offset[0]=0;
    for (p=1; p<Pproc; p++)
        offset[p]=offset[p-1]+H[1][(p-1)*Qproc+(node%Qproc)];
    local_inputs=inputs+offset[node/Qproc];

/* Organises input vector into a contiguous vector for faster MAC */
lipt=local_inputs;
ipt=inputs+hb[1][node];
for (i=0; i<H[1][node]; i++)
    *lipt++ = *ipt++;

/* Iterate until outputs stabilized */
iterations = 0;
do {
    iterations++;
wpt=weights;
bpt = biases;

    /* Vertical broadcast of activation values */
    while ((fctrl->outflag & OF_V_MASK) != OF_V_EMPTY);
    *oc_v_ptr=B_ADDR;
    *of_v_ptr=node;
    for (ipt=local_inputs, i=0; i<H[1][node]; i++)
        *of_v_ptr=*ipt++;
    *oc_v_ptr=TOKEN;

    /* Read broadcasted activation values */
    for (p=1; p<Pproc; p++)
    {
        while ((fctrl->inflag & IF_V_MASK) == IF_V_EMPTY);
        sender=!*if_v_ptr;
        for (ipt=inputs+offset[sender/Qproc], s=0; s<v_series; s++)
            { 
                while (!((fctrl->inflag & IF_V_AE)));
                for (i=0; i<v_flimit*2; i++)
                    *ipt++ = *if_v_ptr;
            }
    }

    /* Calculate partial sums for new activation values */
    /* Store result in output vector */
    /* Broadcast partial sum horisontally */
    while ((fctrl->outflag & OF_H_MASK) != OF_H_EMPTY);
*oc_h_ptr=B_ADDR;

for (opt=outputs, i=0; i<rows; i++)
{
    for (ipt=inputs, temp=0.0, j=0; j<columns; j++)
        temp += *wpt++ * *ipt++;
    *of_h_ptr = *opt++ = temp;
}
*oc_h_ptr=TOKEN;

/* Read and accumulate partial sums */
for (q=1; q<Qproc; q++)
{
    for (opt=outputs, s=0; s<h_series; s++)
    {
        while (!(*fctrl->inflag & IF_H_AE));
        for (j=0; j<h_flimit*2; j++)
            *opt++ += *if_h_ptr;
    }
}

/* Add biases, perform non-linear function, */
/* Calculate difference and update inputs for next iteration */
for (diff=0, opt=outputs+hb[1][node]-vb[1][node/Qproc],
    ipt=local_inputs, i=0; i<H[1][node]; i++)
{
    *opt += *bpt++;

    if (*opt < 0.0) /* Hard-limit nonlinearity */
        *opt = -1.0;
    else if (*opt > 0.0)
        *opt = 1.0;
    else
        *opt = *ipt;

    if (*ipt != *opt)
        diff++;
    *ipt++ = *opt++;
}

/* Check total difference */
/* Horisontal sum */
while ((fctrl->outflag & IF_H_MASK) != IF_H_EMPTY);
*oc_h_ptr = B_ADDR;
*of_h_ptr = diff;
*oc_h_ptr = TOKEN;
for (q=1; q<Qproc; q++)
{
    /* Read and accumulate incoming difference */
    while ((fctrl->inflag & IF_H_MASK) == IF_H_EMPTY);
    diff += *if_h_ptr;
}
/* Vertical sum */
while ((fctrl->outflag & OF_V_MASK) != OF_V_EMPTY);
*oc_v_ptr = B_ADDR;
*of_v_ptr = diff;
*oc_v_ptr = TOKEN;
for (p=1; p<Pproc; p++)
{
    while ((fctrl->inflag & IF_V_MASK) == IF_V_EMPTY);
    diff += *if_v_ptr;
}
} while ((diff != 0) && (iterations < MAXIT));
return iterations;
}

Computation of Mapping and Column Permutations

*************************************************************************/
* Calculations of the mapping from an array of virtual *
* processors, VPs, to an array of physical processors, NPs.  *
* *
* June 19. 1994 *
* Lisbet Utne *
************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hopfield.h"
#include "tlaconf.h"

/* Tables and variables that hold information on the distribution *
 of the weight matrix */
extern int H[2][MAXPROCS*MAXPROCS], V[2][MAXPROCS];
extern int hb[2][MAXPROCS*MAXPROCS], he[2][MAXPROCS*MAXPROCS];
extern int vb[2][MAXPROCS], ve[2][MAXPROCS];

void Mapping(unsigned Qproc, unsigned Pproc, unsigned neurons)
{
    int vsup=0, hsup=0, b, h, v, u, c, n, x, y, q, p, k, j;
    int N[MAXPROCS], offset;

    /* Column adjustments of the input layer */
    b=neurons%Pproc;
    for (h=0; h<b; h++)
        H[0][h]=neurons/Qproc+1;
    for (h=b; h<Qproc; h++)
        H[0][h]=neurons/Qproc;
    hsup+=b;

    /* Row and column adjustments of the cell layer */
    b=neurons%Pproc;
    for (v=vsup; v<vsup+b; v++)
        V[1][v]=neurons/Pproc+1;
    for (v=vsup+b; v<vsup+Pproc; v++)
        V[1][v]=neurons/Pproc;
    vsup=vsup+b%Pproc;
    for (u=0; u<Pproc; u++)
    {
        c=(V[1][u])%Qproc;
        for (h=hsup; h<hsup+b; h++)
            H[1][u+Qproc+(h%Qproc)]=V[1][u]/Qproc+1;
        for (h=hsup+b; h<hsup+Qproc; h++)
            H[1][u+Qproc+(h%Qproc)]=V[1][u]/Qproc;
        hsup=(hsup+c)%Qproc;
    }

    for (q=0; q<Qproc; q++)
    {
        N[q]=H[0][q];
        for (p=0; p<Pproc; p++)
            N[q]+=H[1][p*Qproc+q];
    }

    /* Column mapping from the VP array onto the NP array */
    printf("Column mapping from the VP array onto the NP array\n");
    for (n=0; n<Qproc; n++)

\{
    for (hb[0][n]=0, j=0; j<=n-1; j++)
        hb[0][n]=hb[0][n]+H[0][j];
    he[0][n]=hb[0][n]+H[0][n]-1;
}\n
for (n=0; n<Pproc*Qproc; n++)
{  
    for (hb[1][n]=neurons, j=0; j<=n-1; j++)
        hb[1][n]=hb[1][n]+H[1][j];
    he[1][n]=hb[1][n]+H[1][n]-1;
}

/* Row mapping from the VP array onto the NP array */
for (n=0; n<Pproc; n++)

{  
    for (vb[1][n]=0, j=0; j<=n-1; j++)
        vb[1][n]=vb[1][n]+V[1][j];
    ve[1][n]=vb[1][n]+V[1][n]-1;
}

C.5 Calculation of Weights and Biases

Method Based on Outer Products

/***********************************************************************************/
* Computes weights and biases for Hopfield net                              *
*                                                                           *
* May 9, 1994                                                               *
* Lisbet Utne                                                              *
***********************************************************************************/
#include "hopfield.h"
extern int neurons; /* Number of neurons */
                    /* = number of columns in local weight matrix */
extern int no;     /* Number of neurons per processor */
                    /* = number of rows in local weight matrix */

void ParCalcweights(float *weights, float *biases, float *patterns)
{  
  float *wp, *pi, *pj, *bp;
  float temp;
  long int i;
  int j, k;

  for (wp=weights, i=0; i<no*neurons; i++)
    *wp++ = 0.0;

  for (k=0; k<PATTERNS; k++)
    {
      wp=weights;
      for (pi=patterns+k*neurons, i=0; i<no; i++, pi++)
        for (pj=patterns+k*neurons, j=0; j<neurons; j++)
          if (i != j)
            *wp++ += *pi * *pj++;
        else
          *wp++;
    }

  for (bp=biases, wp=weights, i=0; i<no; i++)
    {
      for (temp=0.0, j=0; j<neurons; j++)
        temp += *wp++;
      *bp++ = (-0.5)*temp;
    }
}

Method Based on Orthogonalisation of the Input Patterns

/***************************************************************************/
* Computes weights and biases for Hopfield net                          *
* Token ring communication protocol, one broadcast ring                *
*                                                                   *
* July 18. 1994                                                       *
* Lisbet Utne                                                        *
***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include "hopfield.h"
#include "matrix.h"
#include "t_ringconf.h" /* or s_ringconf.h or tlaconf.h */
#include "comsys.h"

extern int node;       /* Node ID */
extern int neurons;    /* Number of neurons */
extern int processors; /* Number of processors */
extern int series;     /* Interval between checking FIFO flags */
extern int flimit;     /* Offset value for programmable FIFO flags */
extern int no;         /* Number of neurons per processor */
extern float *int_ram1; /* Start address for on-chip RAM bank 1 */
extern float *int_ram2; /* Start address for on-chip RAM bank 2 */

extern void SVD(float **, int, int, float[], float **);

/* Matrices: */
float Y[NEURONS][PATTERNS-1];
float U[NEURONS][PATTERNS-1];
float SIGMA[NEURONS][NERONS];
float Ttau[NEURONS][NEURONS];
float Itau[NEURONS];
float W[NEURONS][NEURONS];
float b[NEURONS];
*/

void ParMultSVDFactors(float *rm, float *u, int rank,
        float c1, float c2)
{
    float *rmp, *up, *utcol, *utcolp, temp;
    int i, j, k;
    utcol=int_ram1;

    /* Circulate columns of U' = rows of U */
    for (i=0; i<neurons; i++)
    {
        if (node == CTRLPROC)
    {
            for (utcolp=utcol, up=u+i*neurons, j=0; j<neurons; j++)
                *utcolp++ = *up++;
            WriteVector(CTRLPROC,OFIFOB,ALL,neurons,utcol);
    }
}
else
ReadVector(IFIFOB,neurons,utcol);

    /* Calculate inner products */
    for (rmp=rn+i, up=u, j=0; j<no; j++, rmp+=neurons)
    {
        for (temp=0.0, utcolp=utcol, k=0; k<rank; k++)
            temp += *up++ * utcolp++ * c1;
        for (k=rank; k<neurons; k++)
            temp += *up++ * utcolp++ * c2;
        *rmp = temp;
    }
}

void ParSolveHop(float *weights, float *biases, float *targets)
{
    float *p, *lastp, *pp;
    float c1, c2, temp;
    int i, j, k, rank, sender;

    /************** Non-parallelised part *******************/
    if (node == CTRLPROC)
    {
        /* Calculate Y */
        p = targets;
        u = malloc((neurons*neurons)*sizeof(float));
        Zeros(u,neurons,neurons);
        for (i=0; i<PATTERNS-1; i++)
        {
            lastp = targets+(PATTERNS-1)*neurons;
            for (j=0; j<neurons; j++)
                *(u+j*neurons+i) = *p++ - *lastp++;
        }
        sigmap = malloc(neurons*sizeof(float));
        Zeros(sigmap,neurons+1,1);

        /* Singular value decomposition of Y */
        SVD(u,neurons,neurons,sigma,NULL);

        /* Find rank */
        for (rank=0, sigmap=sigma, i=0; i<PATTERNS-1; i++)
if (*sigmap++ != 0.0)
grank +=
} else
   u = malloc(neurons*no*sizeof(float));

/* Distribute matrix U */
if (node == CTRLPROC)
   for (i=max_no; i<neurons; i++)
      WriteVector(CTRLPROC,OFIFOB,ALL,neurons,u+i*neurons);
else
   {
      for (i=no; i<node*no; i++)
         ReadVector(IFIFOB,neurons,int_ram1);
      for (i=0; i<no; i++)
         ReadVector(IFIFOB,neurons,u+i*neurons);
      for (i=node*no+no; i<neurons; i++)
         ReadVector(IFIFOB,neurons,int_ram1);
   }

/* Distribute rank */
if (node == CTRLPROC)
   WriteVector(node,OFIFOB,ALL,1,&rank);
else
   ReadVector(IFIFOB,1,&rank);

/*************** Finished non-parallelised part ***************

/* Calculate Ttau from U, U' and TAU*/
ttau = weights;
c1=1.0;
c2=-TAU;
ParMultSVDFactors(ttau,u,rank,c1,c2);

/* Calculate Itau */
if (node == CTRLPROC)
   p = targets + (PATTERNS-1)*neurons;
else
   p = int_ram1;
if (node == CTRLPROC)
   WriteVector(node,OFIFOB,ALL,neurons,p);
else
   ReadVector(IFIFOB,neurons,p);
C.6 Configuring the Communication

How to configure the communication is shown in an example for a 2-dimensional grid connected in a toroidal architecture. The communication on the rings uses token ring protocol. The definitions and values for other topologies and protocols will follow the same structure.
Selecting Register Setup for a Particular Configuration

#include <stdio.h>
#include "t_comsys.h"
#include "comExtern.h"

#define ISTREAM_H 1 /* Input data stream in horizontal direction */
#define ISTREAM_V 3 /* Input data stream in vertical direction */
#define OSTREAM_H 1 /* Output data stream in horizontal direction */
#define OSTREAM_V 3 /* Output data stream in vertical direction */

#define IFIFO_H 3 /* Input FIFO in horizontal direction */
#define IFIFO_V 4 /* Input FIFO in vertical direction */
#define OFIFO_H 1 /* Output FIFO in horizontal direction */
#define OFIFO_V 2 /* Output FIFO in vertical direction */

#define NC Oxf /* Address for channels not active on the ring */
#define ALL Oxo /* Broadcast address */
#define B_ADDR (ALL | Oxo) << 24
#define TOKEN Oxc0000000

void CfgTLA(unsigned Qproc, unsigned Pproc,
             unsigned h_flimit, unsigned v_flimit)
{
    int *id=(int *) 0x804400;
    int node;
    int fregs[4];
    int sregs[12] = {NC,NC,NC,NC, 0,0,0,0, 0,0,0,0};
    int fflags[4];
    unsigned long *fvalue;

    node = (*id & Oxf);

    fregs[OFIFO_H-1] = OSTREAM_H; /* FIFO 1 --→ STREAM 1 */
    fregs[IFIFO_H-1] = ISTREAM_H; /* FIFO 3 --→ STREAM 1 */
    fregs[OFIFO_V-1] = OSTREAM_V; /* FIFO 2 --→ STREAM 3 */
    fregs[IFIFO_V-1] = ISTREAM_V; /* FIFO 4 --→ STREAM 3 */
    sregs[ISTREAM_H-1] = node; /* Unique addresses on horizontal rings */
    sregs[ISTREAM_V-1] = ALL; /* Same address for all nodes in vertical rings */
    sregs[3+ISTREAM_H] = 1; /* STREAM 1 Enabled */
    sregs[3+ISTREAM_V] = 1; /* STREAM 3 Enabled */
sregs[3+OSTREAM_H] = 1; /* STREAM 1 Enabled */
sregs[3+OSTREAM_V] = 1; /* STREAM 3 Enabled */
sregs[7+OSTREAM_H] = OFIFO_H; /* Token can be set out from OFIFO_H */
sregs[7+OSTREAM_V] = OFIFO_V; /* Token can be set out from OFIFO_V */

/* Determines which nodes will set out the token */
if (((node%Qproc) == 0) && (Qproc >= 4))
    sregs[3+OSTREAM_H] = 9;
else if ((node%(2*Qproc)) == 0)
    sregs[3+OSTREAM_H] = 9;

if (node < Qproc)
    sregs[3+OSTREAM_V] = 9;

/* Set values for the programmable FIFO flags */
fflags[OFIFO_H-1] = h_flimit;
fflags[OFIFO_V-1] = h_flimit;
fflags[IFIFO_H-1] = h_flimit;
fflags[IFIFO_V-1] = v_flimit;
fflags[IFIFO_V-1] = v_flimit;

xlat7seg(node); /* Show node ID in 7-segment display on front */

InitComsubsys(fregs,sregs,fflags);
    /* Write selected register values */

**Programming the Registers in the Communication Subsystem**

#define IO1_port ((volatile unsigned *) 0x801000)
#define SYS1_port ((volatile unsigned *) 0x801100)
#define IO2_port ((volatile unsigned *) 0x801200)
#define SYS2_port ((volatile unsigned *) 0x801300)
#define SYS_RESET ((volatile unsigned *) 0x804401)
#define CS_enable ((volatile unsigned *) 0x805500)
#define PROC_enable ((volatile unsigned *) 0x805700)
#define GAL_RESET ((volatile unsigned *) 0x805780)
#define STREAM1 ((volatile struct sc *) 0x805000)
#define STREAM2 ((volatile struct sc *) 0x805100)
#define STREAM3 ((volatile struct sc *) 0x805200)
#define STREAM4 ((volatile struct sc *) 0x805300)
#define FCTRL ((volatile struct fc *) 0x805600)
struct fc {
  unsigned long cmd;
  unsigned long mode;
  unsigned long modew;
  unsigned long moder;
  unsigned long pflag1;
  unsigned long pflag2;
  unsigned long pflag3;
  unsigned long pflag4;
  unsigned long outflag;
  unsigned long inflag;
};

struct sc {
  int cmd;
  int addr;
};

volatile struct fc *fctrl = FCTRL;
volatile struct sc *streams[4] = {STREAM1,STREAM2,STREAM3,STREAM4};

volatile unsigned *fifo1 = IO1_port;
volatile unsigned *fifo2 = IO2_port;
volatile unsigned *fifo3 = IO1_port;
volatile unsigned *fifo4 = IO2_port;
volatile unsigned *sysf1 = SYS1_port;
volatile unsigned *sysf2 = SYS2_port;

volatile unsigned *sys_reset = SYS_RESET;
volatile unsigned *com_cs = CS_enable;
volatile unsigned *com_proc = PROC_enable;
volatile unsigned *gal_reset = GAL_RESET;

/* 7seg.c - drive 7 segment display*/
static char *display = (char *)0x804400;
int xllattab[16] = { 0x03, 0xf3, 0x25, 0x61, 0xd1, 0x49, 0x09, 0xe3,
  0x01, 0xc1, 0x80, 0x18, 0xe0, 0x30, 0xe0c, 0x8c };

int modet[4][4]=
  {{0,0,0,0},{1,2,4,8},{0,0x10,0x20,0x30},{0,0x40,0x80,0xC0}};
int modewrt[2][4]= {{1,2,4,8},{0x10,0x20,0x40,0x80}};

int InitComsubsys(int fregs[4], int sregs[12], int flags[4])
{  
    /* fregs - FIFO-connections  
       0 - FIFO1  
       1 - FIFO2  
       2 - FIFO3  
       3 - FIFO4  
       */  

    sregs - values for the scrl registers
    0 - scrl1.adr  4 - scrl1.cmd  8 - FIFO# if scrl1 is master
    1 - scrl2.adr  5 - scrl2.cmd  9 - FIFO# if scrl2 is master
    2 - scrl3.adr  6 - scrl3.cmd 10 - FIFO# if scrl3 is master
    3 - scrl4.adr  7 - scrl4.cmd 11 - FIFO# if scrl4 is master

    fflags - values for the programmable FIFO flags
    0 - FIFO1
    1 - FIFO2
    2 - FIFO3
    3 - FIFO4
    */  

    volatile int x;
    unsigned long values[3];

    /* Program LCA's */
    *sys_reset = 0x1;
    *com_cs = 0x1;
    *sys_reset = 0x9;
    for (x=1;x<1000000;x++)
    {
        /* Reset GAL */
        *gal_reset = 0x1;
        /* Enable LCA chip selects */
        *com_cs = 0x1;

        /* Converting to the required bit-combinations for the  
        FIFO Controller register*/
        MakeRegValues(&fregs[0],&values[0]);

        /* Initialize FIFO-registers */
        fctrl->mode = values[0];
        fctrl->modew = values[1];
        fctrl->moder = values[2];
        fctrl->pflag1 = fflags[0];
fctrl->pflag2 = fflags[1];
fctrl->pflag3 = fflags[2];
fctrl->pflag4 = fflags[3];

/* Enable FIFO-reset */
fctrl->cmd = 0x1a;
for (x=1;((fctrl->cmd & 0xff) != 0x18) && (x<1000);x++) {

/* Reset and initialize FIFOs, dmux and mux */
fctrl->cmd = 0x99;
for (x=1;((fctrl->cmd & 0xff) != 0x98) && (x<1000);x++) {

/* Enable processor interface */
*com_proc = 0x1;
for (x=0;x<1;x++) {
  fctrl->cmd = 0x0;
}

/* Initialize Stream Controllers */
for (x=0;x<4;x++) {
  { 
    streams[x]->adr=sregs[x]; /* Init address-register */
    streams[x]->cmd=sregs[x+4]; /* Init command-register */
    if (sregs[x+4] == 0x9) /* Write Token if ring-master */
      switch(sregs[x+8]) {
        case 1:
          *sysf1 = 0xC0000000;
          break;
        case 2:
          *sysf2 = 0xC0000000;
          break;
      }
  }
}
}
Appendix D

Result Tables

The simulation results presented graphically in Chapter 10, are here shown in table format.

<table>
<thead>
<tr>
<th>Pattern size</th>
<th>Number of processors</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>128</td>
<td>4.67</td>
<td>8.48</td>
<td>15.88</td>
<td>26.48</td>
</tr>
<tr>
<td>256</td>
<td>4.93</td>
<td>9.38</td>
<td>18.14</td>
<td>33.45</td>
</tr>
<tr>
<td>512</td>
<td>5.07</td>
<td>9.89</td>
<td>19.16</td>
<td>37.45</td>
</tr>
<tr>
<td>1024</td>
<td>5.14</td>
<td>10.16</td>
<td>20.16</td>
<td>39.52</td>
</tr>
<tr>
<td>Linear (1024)</td>
<td>5.14</td>
<td>10.28</td>
<td>20.56</td>
<td>41.12</td>
</tr>
<tr>
<td></td>
<td>Processing speed (MCPS)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table D.1: Speed measurements for non-overlapping communication and computation recall phase.
### Table D.2: Execution time per iteration for small pattern sizes (128).

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total time</td>
<td>3.5</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>2.0</td>
</tr>
</tbody>
</table>

### Table D.3: Execution time per iteration for larger pattern sizes (1024).

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total time</td>
<td>200.0</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>132.0</td>
</tr>
</tbody>
</table>

### Table D.4: Speed measurements for pipelined communication and computation recall phase.

<table>
<thead>
<tr>
<th>Pattern size</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>4.38</td>
</tr>
<tr>
<td>256</td>
<td>4.76</td>
</tr>
<tr>
<td>512</td>
<td>4.98</td>
</tr>
<tr>
<td>1024</td>
<td>5.10</td>
</tr>
<tr>
<td>Linear (1024)</td>
<td>5.10</td>
</tr>
</tbody>
</table>

Processing speed (MCPS)
<table>
<thead>
<tr>
<th>Pattern size</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>4.51</td>
</tr>
<tr>
<td>256</td>
<td>4.86</td>
</tr>
<tr>
<td>512</td>
<td>5.02</td>
</tr>
<tr>
<td>Linear (512)</td>
<td>5.02</td>
</tr>
</tbody>
</table>

Processing speed (MCPS)

Table D.5: Speed measurements for recall phase based on computing and communicating partial sums.

<table>
<thead>
<tr>
<th>Parallelisation strategy</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Pipelined comm. and comp.</td>
<td>4.98</td>
</tr>
<tr>
<td>Calc. of partial sums</td>
<td>5.02</td>
</tr>
<tr>
<td>Non-overlapping comm. and comp.</td>
<td>5.07</td>
</tr>
</tbody>
</table>

Processing speed (MCPS)

Table D.6: Comparision of parallelisation strategies for token ring (pattern size of 512 elements).

<table>
<thead>
<tr>
<th>Parallelisation strategy</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128 Direct Connections</td>
<td>4.67</td>
</tr>
<tr>
<td>128 Token ring</td>
<td>4.67</td>
</tr>
<tr>
<td>1024 Direct Connections</td>
<td>5.14</td>
</tr>
<tr>
<td>1024 Token ring</td>
<td>5.14</td>
</tr>
</tbody>
</table>

Processing speed (MCPS)

Table D.7: Comparision of direct connections versus token ring protocol.
<table>
<thead>
<tr>
<th>Pattern size/strategy</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128 Single word</td>
<td>1.65</td>
</tr>
<tr>
<td>512 Single word</td>
<td>1.57</td>
</tr>
<tr>
<td>512 Transposed weight matrix</td>
<td>2.25</td>
</tr>
<tr>
<td>512 Vector based</td>
<td>4.96</td>
</tr>
<tr>
<td></td>
<td>Processing speed (MCPS)</td>
</tr>
</tbody>
</table>

Table D.8: Speed measurements for the recall phase with direct connections, single word based algorithm.

<table>
<thead>
<tr>
<th>Pattern size</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>4.67</td>
</tr>
<tr>
<td>256</td>
<td>4.93</td>
</tr>
<tr>
<td>512</td>
<td>5.07</td>
</tr>
<tr>
<td>1024</td>
<td>5.14</td>
</tr>
<tr>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>Linear (1024)</td>
<td>5.14</td>
</tr>
<tr>
<td></td>
<td>Processing speed (MCPS)</td>
</tr>
</tbody>
</table>

Table D.9: Speed measurements for toroidal lattice architecture (TLA), recall phase.

<table>
<thead>
<tr>
<th>Pattern size/location</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128 DRAM</td>
<td>4.67</td>
</tr>
<tr>
<td>128 SRAM</td>
<td>6.78</td>
</tr>
<tr>
<td>512 DRAM</td>
<td>5.07</td>
</tr>
<tr>
<td>512 SRAM</td>
<td></td>
</tr>
<tr>
<td>1024 DRAM</td>
<td>5.14</td>
</tr>
<tr>
<td></td>
<td>Processing speed (MCPS)</td>
</tr>
</tbody>
</table>

Table D.10: Speed-up from keeping data in fastest possible memory.
<table>
<thead>
<tr>
<th>Pattern size</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>1.25</td>
</tr>
<tr>
<td>256</td>
<td>4.76</td>
</tr>
<tr>
<td>512</td>
<td>56.54</td>
</tr>
<tr>
<td>1024</td>
<td>74.60</td>
</tr>
<tr>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>2560</td>
<td></td>
</tr>
</tbody>
</table>

Execution time (s)

Table D.11: Time to determine weights and thresholds, outer product method.

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Serial SVD</td>
<td>5.71</td>
</tr>
<tr>
<td>SolveHop</td>
<td>8.70</td>
</tr>
</tbody>
</table>

Execution time (s)

Table D.12: Time to determine weights and thresholds, orthogonalization method, small pattern (128).

<table>
<thead>
<tr>
<th>Part of program</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Serial SVD</td>
<td>104.8</td>
</tr>
<tr>
<td>SolveHop</td>
<td>272.8</td>
</tr>
</tbody>
</table>

Execution time (s)

Table D.13: Time to determine weights and thresholds, orthogonalization method, larger patterns (1024).
Index

addressing,
  communication, 105, 126

backpropagation, 14, 16, 18, 22, 25, 3, 43, 46
bandwidth,
  communication, 146, 43, 50, 81
  memory, 117, 50, 59
benmarks, 28, 52
Bidirectional Associative Memory, 16, 25
Bignet, 29
Boltzmann machines, 23
Boltzmann training, 15
broadcast bus, 29, 38, 46, 49

Cauchy training, 15

cellar arrays, 5
circuit board, 7
clock distribution, 37, 60, 79
communication,
  bandwidth, 146, 43, 81
communication overhead, 119, 128, 140, 148, 19, 29, 49
configuration, 121, 146, 6, 78, 83
conventional computers, 4
convergence, 126

CPS, 4, 50
CUPS, 4, 50

DARPA study, 43
data dependency graph, 20
data streams, 54, 66, 68, 73, 75, 81, 84
delta learning rule, 14, 22
demultiplexer, 73
digital signal processors, 42, 56
efficiency, 28, 52

epoch, 19, 46
error, 15
expansion bus, 58, 68

feed-forward networks, 12, 20, 24
FIFO flags, 122, 140
FIFOs, 119, 69, 75
FPGA, 37, 39, 49, 67, 83

Hamming net, 24
hidden layer, 13
higher dimensional architectures, 48
Hopfield net, 111, 16, 23, 6, 93, 98
hypercube, 36, 39, 45, 48

input patterns, 111, 14, 96

LCA, 146, 49, 70, 78, 7, 85
learning, 14, 15, 19, 21, 27
least squared error, 14
linear architectures, 47
linear projection, 21
load balancing, 107, 115, 126

mapping, 20, 21, 23
massively parallel computers, 36
MAXNET, 24
memory,
  access pattern, 118
  allocation, 118, 140
  bandwidth, 117, 50, 59
  capacity, 59
  organisation, 117
  requirements, 112
mesh, 22, 24, 29, 39, 45, 47, 49, 88
MIMD, 37, 38, 60
models, 3

Page 209
modular neural networks, 23, 31
multilayer,
  networks, 13, 18
  perceptron, 12, 13, 25
multiplexer, 73
Neocognitron, 16
neocognitron, 26
NETtalk, 29
neurocomputers, 26, 33
neuron parallelism, 17
neurons, 13, 17
noise, 94
nonlinear, 108
paradigms, 6
parallel instructions, 117
perceptron, 13, 16
pipeline, 120, 130, 18, 25, 49, 80, 87
PLA, 22, 47, 90
PLANNS, 22, 30
precision, 42
processor interface, 68, 75
processors, 41, 56
protocols, 126, 79, 85
prototyping, 31
random access memories, 5
recall, 107, 15, 20, 25, 93
reconfigurability, 146, 37, 49, 67, 70, 83
recurrent associative memories, 115
recurrent networks, 107, 115, 3, 93
RENNNS module, 54, 61
ring, 126, 21, 29, 39, 45, 47, 49, 80, 81, 85
scaling, 126, 143, 148
Self-organising Feature Maps, 16
self-organising feature maps, 25
sequential part, 128, 131
SIMD, 37
single layer networks, 12, 16
singular value decomposition, 103, 105, 142
special purpose neurocomputers, 33
speed, 143, 145, 148
speed measurements, 126
speedup, 19
speed-up, 52
SPMD, 60
statistical methods, 15
supercomputers, 26, 39
supervised, 17
synchronisation, 105, 60, 79
systolic array, 21, 24, 38, 40, 47
target, 102
thresholds, 106, 13, 142, 2, 97
TLA, 136, 137, 147, 22, 47, 88
TMS320C30, 103, 112, 118
topologies, 21–23, 29, 45, 83, 85, 96
toroidal, 22, 45, 88
training pattern parallelism, 19
training session parallelism, 19
Transputers, 22, 39, 47
two-dimensional architectures, 47
unsupervised, 17
virtual processors, 18, 22
VLSI, 34, 6, 94
weight parallelism, 17
weights, 106, 13, 142, 14, 17, 2, 97