A VML Based Implementation of a Neural Network Library on General Purpose Parallel Processors

H. Byun
S. K. Kortesis
Elias N. Houstis

Purdue University, enh@cs.purdue.edu

Report Number:
92-024
A VML BASED IMPLEMENTATION
OF A NEURAL NETWORK LIBRARY ON
GENERAL PURPOSE PARALLEL PROCESSORS

H. Byun, S.K. Kortesis and E.N. Houstis
Computer Science Department
Purdue University
West Lafayette, IN 47907
Technical Report CSD-TR-92-024
CAPO Report CER-92-11
April 1992
A VML Based Implementation of a Neural Network Library on General Purpose Parallel Processors

H. Byun, S.K. Kortesis and E.N. Houstis

April 8, 1992

Contents

1 Introduction

2 Neural Network Library
  2.1 Backpropagation
  2.2 Feedback Associative Networks
  2.3 Competitive Networks
  2.4 Probabilistic
  2.5 Adaptive Resonance

3 Mapping VML Primitives to Parallel Machines
  3.1 Overview of Parallel Matrix Multiplication Algorithms
  3.2 Parallelization of matrix multiplication operations
  3.3 Dense Matrix x Vector Multiplication
  3.4 Parallelization of matrix multiplication operations
  3.5 Band Matrix x Band Matrix Multiplication

4 Parallel Neural Network Library
  4.1 Hopfield Net (HOP)
    4.1.1 Euler processing equations
    4.1.2 Runge-Kutta processing equations
  4.2 Multilayer back propagation (MBPN)
  4.3 Counter Propagation (CPN)
  4.4 Learning Vector Quantization (LVQ)
  4.5 Adaptive Ring
1 Introduction

The objectives of this effort is to identify and implement a number of virtual machine language primitives on various parallel machines and use them to implement ANN as an extension of some algorithmic language. For the implementation of matrix-vector and matrix-matrix multiplication operations for dense matrices, we have adopted the algorithms reported in [Abae 91]. Section 3.1 includes the description of these algorithms, together with performance results on nCUBE II and appropriate references. Furthermore in this report, we list the processing equations of a number of ANN in terms of a high level definition of the VML primitives. The performance of the parallel implementation of the Hopfield (HOP) and Multiple Back Propagation Network (MBPN) is reported for nCUBE II and Intel iPSC/860 machines.

This report is organized as follows: Section 2 lists a set of primitive functions suitable for the implementation of ANN. This set coincides with the one chosen by the ESPRIT Galatea project [Tayl 91]. Section 3, describes the neural network library whose parallel implementation we are considering. We follow the presentation in [HNC 90]. Section 3 describes the algorithms and their performance for the implementation of dense matrix-vector VML operations on the NCUBE parallel machine. This is part of the publication [Aboe 91]. In Section 4, we make an attempt to formulate the processing equations of the neural networks considered in a matrix-vector and matrix-matrix form. Finally, in Section 5 we present some preliminary data of the performance of HOP neural net used to solve a scheduling problem and MBPN for a simple test problem.

A Virtual Machine Language (VML) for Implementing Neural Nets

In this section we list a number of arithmetic operations and their high level definitions to be implemented as an extension of an existing algorithmic
language. These extensions will be implemented for a variety of targeting parallel architectures. The parallel implementation of some of them is already available for nCUBE II and Intel machines. We are implementing the rest on the above machines. This set has been adopted by the ESPRIT Galatea project as virtual machine language for a NN Software environment under development.

Table 2.1. Virtual machine languages arithmetic operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Synopsis</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R = add (s, t)</td>
<td>scalar addition</td>
</tr>
<tr>
<td>afm</td>
<td>R = afm (f, M)</td>
<td>apply function f to each element of matrix</td>
</tr>
<tr>
<td>av</td>
<td>r = av (M)</td>
<td>calculate average of elements of matrix</td>
</tr>
<tr>
<td>cp</td>
<td>R = cp (M)</td>
<td>copy (sub) matrix</td>
</tr>
<tr>
<td>cv</td>
<td>s = cv (M[i,j])</td>
<td>matrix element to scalar conversion</td>
</tr>
<tr>
<td>cvi</td>
<td>s = cvi (t)</td>
<td>scalar convert to integer part</td>
</tr>
<tr>
<td>d</td>
<td>R = d (M, d)</td>
<td>divide M by d element by element</td>
</tr>
<tr>
<td>decr</td>
<td>r = decre (s)</td>
<td>scalar decrement</td>
</tr>
<tr>
<td>det</td>
<td>s = det (M)</td>
<td>calculate determinant</td>
</tr>
<tr>
<td>div</td>
<td>r = div (s, t)</td>
<td>scalar division</td>
</tr>
<tr>
<td>ea</td>
<td>R = ea (M, N)</td>
<td>addition of M to N element by element</td>
</tr>
<tr>
<td>em</td>
<td>R = em (M, N)</td>
<td>multiplication of M by N element by element</td>
</tr>
<tr>
<td>es</td>
<td>R = es (M, N)</td>
<td>substraction of N from M element by element</td>
</tr>
<tr>
<td>id</td>
<td>M = id</td>
<td>set matrix to identify</td>
</tr>
<tr>
<td>idiv</td>
<td>r = idiv (s, t)</td>
<td>integer scalar division</td>
</tr>
<tr>
<td>imod</td>
<td>r = imod (s, t)</td>
<td>integer modulus division</td>
</tr>
<tr>
<td>incr</td>
<td>r = incr (s)</td>
<td>scalar increment</td>
</tr>
<tr>
<td>inv</td>
<td>R = inv (M)</td>
<td>produce inverse of M</td>
</tr>
<tr>
<td>m</td>
<td>R = m (M, m)</td>
<td>multiply M by m element by element</td>
</tr>
<tr>
<td>max</td>
<td>s = max (M)</td>
<td>find maximum of elements of matrix</td>
</tr>
<tr>
<td>maxi</td>
<td>r = maxi (M, s, t)</td>
<td>find maximum of elements returning indices</td>
</tr>
<tr>
<td>min</td>
<td>s = min (M)</td>
<td>find minimum of elements of matrix</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>mini</td>
<td>find minimum of elements returning indices</td>
<td></td>
</tr>
<tr>
<td>mt</td>
<td>matrix by matrix transpose ((M \times N')) multiplication</td>
<td></td>
</tr>
<tr>
<td>mu</td>
<td>matrix by matrix multiplication</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>scalar multiplication</td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>scalar change sign</td>
<td></td>
</tr>
<tr>
<td>norm</td>
<td>normalise: divide by (\sqrt{\text{sum of squares}})</td>
<td></td>
</tr>
<tr>
<td>ra</td>
<td>randomise matrix within range (s) to (t)</td>
<td></td>
</tr>
<tr>
<td>rec</td>
<td>scalar reciprocal</td>
<td></td>
</tr>
<tr>
<td>rms</td>
<td>root mean square</td>
<td></td>
</tr>
<tr>
<td>rnd</td>
<td>random number in range (s) to (t)</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>scalar assignment</td>
<td></td>
</tr>
<tr>
<td>sgn</td>
<td>signum: (\text{sgn} (x) = x &lt; 0 ? -1 : 1)</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>scalar subtraction</td>
<td></td>
</tr>
<tr>
<td>sv</td>
<td>set values to (s)</td>
<td></td>
</tr>
<tr>
<td>tr</td>
<td>produce transpose of (M)</td>
<td></td>
</tr>
</tbody>
</table>

2 Neural Network Library

The objective of this effort is to identify and implement on general purpose vector and multiprocessor machines, a subset of Neurosoftware that is currently commercially available for NN emulators or co-processors. This library contains software modules available in the HNC neurosoftware library. Our initial effort will concentrate on a subset of such modules (see figure 1). For completeness, we present a short description of these modules and define the acronyms used to refer to them following the presentation in reference [HNC 90].

2.1 Backpropagation

Backpropagation (BPN) networks comprise one major neural network family. BPN implements a feedforward mapping which is determined by its weights. Backpropagation learns by comparing the actual outputs produced using its current weights with the desired outputs for the mapping it is supposed to implement. It uses the differences, or errors, to adjust its weights in order to reduce the average error. BPN must be provided with the desired output corresponding to each input, which makes it a supervised learning network. BPN networks are very homogeneous in that all PEs have basically the same transfer function regardless of their position in the network.
Backpropagation networks are very versatile, because their transfer functions can implement a wide variety of mappings with appropriate weights. With a little creativity in representing the problem, BPN can be applied to many tasks which do not at first appear to be BPN-type problems. For example, backpropagation can be used for classification by making the outputs represent the correct class of the input, or for noise reduction by making the outputs less noisy versions of noisy inputs. Backpropagation is currently the most widely used family of neural network paradigms.

The multiple layer implementation of BPN is referred throughout as MBPN.

2.2 Feedback Associative Networks

Another neurosoftware family is the feedback associative networks. These networks are made up of a single functional layer, which is highly self-interconnected. All of these networks have similar connection geometry and processing equations. They fall into two classes: binary and continuous. Most of them, including Hopfield and BAM (Bidirectional Associate Memory), have both binary and continuous versions which are closely related. One of the most useful features of many feedback associative networks is their automatic minimization of system "energy" which guarantees convergence of the states and also makes them applicable to optimization problems. The feedback associative networks are even more homogeneous than the BPN family, for not only do the PEs have the same transfer function, but all are equivalent with respect to the connection structure.

HOP (Hopfield network) is a continuous-valued associative network. In addition to its main processing layer, it also has an input layer. Its main processing layer is fully connected. The Hopfield network is most often used for optimization and associative memory problems. Neither the HOP nor the BAM neurosoftware has a learning law.

BAM is a binary-valued associative network. Its main processing layer is divided into two parts (slabs). Each PE on each slab is fully connected to each PE on the other slab, but not to any PEs on its own slab. (The BAM can be thought of as a Hopfield network with the connections between PEs on the same slab deleted; alternatively, the Hopfield network can be thought of as a BAM in which the two slabs are the same.) The BAM has no input layer, although one could be defined for it analogously to the Hopfield network's.

The GAN (Generalized Associative Network) combines many features
of the HOP and BAN networks into a single network. The GAN provides additional control over network topology not found in the other feedback associative networks.

2.3 Competitive Networks

One strategy often used by neural networks is vector quantization—representing large numbers of vectors by a smaller set of prototypes stored as PE weight vectors. The important task for a network which uses vector quantization is to find a set of weight vectors which represent the input vectors in a suitable manner. Networks using this approach are called prototype-based networks.

The competitive or Kohonen learning networks represent one type of the so-called prototype-based networks. There is some fixed number of PEs with modifiable weight vectors. For each network input, a subset of the PEs is allowed to modify its weight vectors so they become either more or less like the input vector. The process is called competitive because PEs "compete" against each other for the right to modify their weights. As learning progresses, the weight vectors differentiate and spread out so that each weight vector has its region in the input space in which some inputs are closer to it than to any other weight vector. Each weight vector becomes the prototype example for inputs in that region. Note that this is an unsupervised learning procedure.

The first network in this family is counter propagation (CPN). CPN has two functional layers. The first uses a form of Kohonen learning in which only the PE whose weight vector is most like (i.e., closest to) the input vector modifies its weights. This PE's weight vector is adjusted to become more like (closer to) the input vector. CPN is designed to learn mappings, but in a very different way than BPN. Apart from the Kohonen learning, the processing of this network works as follows. Only the PE whose weight vector is closest to the input vector can output a non-zero value. The PEs in the second functional layer output different values depending on which one of the first layer PEs outputs a non-zero value, and the values they output are determined by their own weights. The vector of second layer PE outputs is the output of the network. CPN like BPN, is a supervised learning network, meaning that it requires a desired output vector corresponding to each input vector.

LVQ (Learning Vector Quantization) is much like the first layer of CPN; however, it is used for pattern classification. Input vectors are associated with classes, and so are first layer PEs. LVQ uses a form of supervised
learning in the sense that the actual class of the input must be supplied to the network during training. If the PE with the closest weight vector is of the same class as the input vector, then as in CPN, it is moved closer to the input vector. But if the PE is of a different class, it is moved away from the input vector. This processing tends to result in the PEs associated with each class staking out a region where the input vectors in that class tend to come from. The network is used by determining which PE’s weight vector is closes to an input vector whose class is unknown, then assigning to the input vector the same class as the PE.

ELVQ (Extended Learning Vector Quantization) has two layers, the first of which is exactly like LVQ. Its second layer has one PE for every class. The network is used by assigning to an input vector whose class is unknown the class associated with whichever second layer PE has the largest output. The second layer PE outputs are determined by a weighted voting among first layer PEs in the same class. The PEs which are closer to the input have a greater vote. Thus where LVP gives the single closest PE the whole vote, ELVQ allows all PEs to participate in assigning the class.

RING (Adaptive Ring) is another form of self-organizing map in which the topological ordering is not a two-dimensional grid, but a one-dimensional ring. It is useful for some optimization problems, such as the traveling salesman problem.

2.4 Probabilistic

Another family of networks is the so-called probabilistic networks. Their main characteristic is that they use probabilistic criteria to advance from one stage to the other. We will implement the Simulation Annealing (SA) and Boltzmann Machine (BM) techniques.

2.5 Adaptive Resonance

Another family of networks that uses the strategy of keeping prototypes for pattern classification is the ART (Adaptive Resonance Theory) family. ART1 is designed for binary input patterns and ART2 for continuous-valued patterns.

ART does unsupervised pattern classification or clustering, in the sense that it forms its own pattern classes and classifies new input vectors as either being in the same class as some inputs previously seen, or as being the first example of a new class. Basically, if an input vector is not sufficiently like
Figure 1: Parallel Netware (S = Supervised, U = Unsupervised, N = None, 
Y = Yes, A = Analog, B = Binary.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adaptive</td>
</tr>
<tr>
<td><strong>Backpropagation</strong></td>
<td></td>
</tr>
<tr>
<td>BNP</td>
<td>Y</td>
</tr>
<tr>
<td>MBPN</td>
<td>Y</td>
</tr>
<tr>
<td><strong>Competitive</strong></td>
<td></td>
</tr>
<tr>
<td>CPN</td>
<td>Y</td>
</tr>
<tr>
<td>LVQ</td>
<td>Y</td>
</tr>
<tr>
<td>Adaptive Ring</td>
<td>Y</td>
</tr>
<tr>
<td><strong>Adaptive Resonance</strong></td>
<td></td>
</tr>
<tr>
<td>ART 1</td>
<td>Y</td>
</tr>
<tr>
<td><strong>Probabilistic</strong></td>
<td></td>
</tr>
<tr>
<td>SA</td>
<td>N</td>
</tr>
<tr>
<td>BM</td>
<td>N</td>
</tr>
<tr>
<td><strong>Association</strong></td>
<td></td>
</tr>
<tr>
<td>HOP</td>
<td>N</td>
</tr>
<tr>
<td>BAM</td>
<td>N</td>
</tr>
<tr>
<td>GAN</td>
<td>Y</td>
</tr>
</tbody>
</table>

any stored prototype, it is stored as the prototype for a new class. Otherwise it is classified as the same class as the nearest prototype, and that prototype may be modified. ART networks are complex and unhomogeneous feedback networks.

3 Mapping VML Primitives to Parallel Machines

The most important objectives in designing algorithms/software for multiprocessor systems include the minimization of i) the so called edge contention (one or more links are shared between more than one paths in the computational graph of the algorithm [Bokh 90] and [Chri 90]), ii) the amount of data transferred between processors, and iii) the synchronization delay. It has been observed that the minimization of the cost functions corresponding to the above three design objectives depend on the way the underlying computation graph is decomposed which constitutes an NP-complete
problem for general computational graphs [Gare 79].

For the case of well structured computations, special purpose algorithm/architecture pairs were suggested known as systolic arrays [Kung 82], [Mold 82], [Mira 84], and [Chen 87]. These architectures consist of simple processing elements (PEs) which are capable of performing one arithmetic operation. In systolic computations, the decrease of edge contention and synchronization is achieved by mapping the computation graph into a systolic array such that the correct data are in the correct place at the appropriate time.

In this paper we propose systolic type techniques to parallelize primitive linear algebra operations applied to sparse data. The set of these operations include multiplication of banded matrices and banded matrix-vector operations. A source of sparse data structures is the discretization of Partial Differential Equations (PDE) with well known finite element and difference techniques. We are using the above implementation of matrix VML operators to implement ANN. Unlike the sparse matrix operations, the dense matrix-vector and matrix-matrix have been studied extensively [Fox 87], [Cher 88], [Bern 89]. In section 2, we review some of the proposed ideas for the parallelization of matrix VML operators and their complexity on various architectures. For comparison purposes, section 3 presents the performance of parallel matrix multiplication VML operators for dense matrices. The experimental results indicate an efficiency of up to 98 % for matrix-vector operations and 94 % for matrix-matrix operations on a 64 processors configuration nCUBE II with one Mbyte of memory per processor. In section 4, we present our proposed algorithms for the implementation of sparse matrix-matrix multiplication operators on distributed memory machines with mesh and ring interconnection topologies. These algorithms depend on the bandwidth of the matrices involved. Our preliminary limited experimental data based on block banded matrices indicate 92 % efficiency on the nCUBE II.

### 3.1 Overview of Parallel Matrix Multiplication Algorithms

The development and implementation of scalable and portable scientific algorithms across a number of parallel machines is an important and challenging problem. One of the approaches that have been extensively explored is the VML approach. The basic idea is to identify a kernel of high level primitive mathematical operations, implement them on a variety of machines, and use them to develop more complex applications on these targeting machines. Two compute bound linear algebra operations are the matrix-vector and matrix-matrix multiplications. In this section we review some of the
ideas proposed for their parallelization in the case of dense data structures and various architectures.

Fox et al in [Fox 87] proposed techniques for the multiplication of matrices decomposed into square or rectangular subblocks on hypercube architectures. These blocks are distributed between the processors. The product matrix is distributed among the processors in the same fashion. The algorithms exploit the mesh architecture embedded in any hypercube architecture. They also use broadcasting for communicating some of these data blocks. The algorithms presented in section 3 and 4 avoid any broadcasting, and attempt to implement the required communication locally among neighboring processors.

Dekel, Nassimi, and Sahni in [Deke 81] proposed a matrix multiplication algorithm for cube connected and perfect shuffle computers. They use \( N^2 m \) processors to multiply two \( N \times N \) matrices in \( O\left(\frac{N}{m} + \log m\right) \) time. They also show how \( m^2, 1 \leq m \leq N \), processors can be used to multiply two \( N \times N \) matrices in \( O\left(\frac{N^2}{m} + m\left(\frac{N}{m}\right)^{2.61}\right) \) time. This method is efficient for multiplying dense matrices, but, it appears to be inefficient for sparse BLAS 2 and 3 operations.

Johnson [John 85] presented algorithms for dense matrix multiplication and for Gauss-Jordan and Gaussian elimination. His algorithm can run on any boolean cube or torus computers. It achieves a 100% processor utilization except for a latency period \( T_{\text{latency}} = O(n) \) on an n cube system. In [John 89], Johnson et al presented a data parallel matrix multiplication algorithm which was implemented on the Connection Machine CM-2. They report 5.8 GFLOPS overall performance.

Independently Cherakasky et al in [Cher 88], Berntsen in [Bern 89] and Aboelaze [Aboe 89] improved Fox's algorithm for dense matrix multiplication, reducing the time complexity from

\[
T = \frac{2N}{P} \tau + \frac{2N^2}{\sqrt{P}} t_{\text{transf}} + \sqrt{P}(\sqrt{P} - 1)t_{\text{start}}
\]

to

\[
T = \frac{2N}{P} \tau + \frac{2N^2}{\sqrt{P}} t_{\text{transf}} + (\sqrt{P} - 1)t_{\text{start}}
\]

where \( P \) is the number of processors, \( \tau \) is the time for one addition and multiplication, and \( t_{\text{transf}}, t_{\text{start}} \) are machine dependent communication parameters. Berntsen's second idea was to partition the hypercube into a set of subcubes and using the cascaded sum algorithm to add up the contributions.
to the product matrix. His idea also reduced the asymptotic communication to \( \frac{N^2}{P^2} \) at the expense of having \( \frac{N^2}{P^2} \) extra bytes of memory per processor.

The algorithms for dense matrices presented in [Fox 88], [Cher 88], [Bern 89], and [Aboe 89] require \( P \) and \( \sqrt{P} \) iteration steps to compute \( c = c + Ab \) and \( C = C + AB \) respectively; each iteration step requires \( \frac{N}{P} t_{\text{transf}} + t_{\text{start}} \) and \( \frac{N^2}{P} t_{\text{transf}} + t_{\text{start}} \) communication time respectively. In this report, we present two algorithms for operation on band matrix \( A \in \mathbb{R}^{N \times N} \) with bandwidth \( w \). The first algorithm is to multiply \( A \) by \( b \), where \( b \in \mathbb{R}^{N} \). The second algorithm is to multiply \( A \) by \( B \), where \( B \in \mathbb{R}^{N \times N} \), with bandwidth \( \delta \). The first algorithm requires \( w \) iteration steps with each iteration requiring \( \frac{N}{P} t_{\text{transf}} + t_{\text{start}} \) communication time. The second algorithm requires \( w + \delta - 1 \) iteration steps with each iteration step requiring \( \frac{N}{P} t_{\text{transf}} \min(w, \delta) \) communication time.

### 3.2 Parallelization of matrix multiplication operations

In this section we are considering a parallel implementation of matrix-vector matrix-matrix operations on a wrap around linear array and grid of \( P \) processors respectively. These operations involve the matrix-vector operation \( c = \beta c + \alpha A b \), and matrix-matrix operation \( C = \beta C + \alpha A B \) where \( A, B, C \) are matrices of dimensions \( M \)-by-\( K \), \( K \)-by-\( N \) and \( M \)-by-\( N \) respectively, \( b, c \) are column vectors of compatible dimensions, and \( \alpha, \beta \) real scalars. Our current implementation applies only to square matrices \( (N = M = K) \). For the complexity analysis and performance evaluation of the proposed parallel implementation of the above VML primitives we assume i) \( \tau \) denotes the time to perform a floating-point multiply or add, ii) \( \gamma + \delta w \) is the time of transferring \( w \) words in a interconnection network (\( \gamma, \delta \) are machine dependent parameters) iii) the fixed speedup is defined as \( S(N, P) = \frac{T_{p}}{T_{P}} \), where \( T_{P} \) is the execution time of the computation in a \( P \) processors machine, iv) scaled speedup is computed [Gust 88] by:

\[
S_{\text{Up2}} = \frac{M \text{ flops using } P \text{ processors}}{M \text{ flops using single processor}}
\]  

or:

\[
S_{\text{Up2}} = P \times \frac{T_{\text{Work done by } P \text{ processors}} - T_{\text{Work wouldn't done by serial processor}}}{T_{\text{Work done by } P \text{ processor}}}
\]

\[ (3.2) \]
where $T_{\text{Work\_done\_by\_P\_processors}}$ indicates the total elapse time using $P$ processors, and $T_{\text{Work\_wouldn't\_done\_by\_serial\_processors}}$ indicates the overhead due to communication and synchronization.

### 3.3 Dense Matrix $\times$ Vector Multiplication

First, we consider the implementation of the operation $c = \beta c + \alpha A b$ on a linear wrap around array of $P$ processors. We assume that the input data $A$ and $b$ are decomposed into submatrices $A_{i,j} \in \mathbb{R}^{N/P \times N/P}$ and subvectors $c_i, b_i \in \mathbb{R}^{N/P}$ respectively. Each processor, $i$, contains the block row $\{A_{i,j}\}_{j=1}^{P}$ and subvectors $b_i, c_i$ and computes the updated subvector $c_i$ using $A_{i,(i+1)\mod P}$ submatrix, and receive $b_i$ from $(i + 1)\mod P$ processor, one can easily show that complexity of this computation is

$$T_P = P\left(\frac{N^2 r}{P^2} + \gamma + \frac{N}{P}\right)$$

and $T_1 = N^2 r$. Furthermore, the space complexity is $O\left(\frac{N^2}{P^2} + \frac{2N}{P}\right)$.

The algorithm was implemented on the nCUBE II. Table 1 indicates the performance for different sizes of dense matrices measured in Mflops together with the speedup obtained. We use the three different ways defined above to measure the speedup. Its performance is very close to optimal.

**Table 1: Measured Mflops and SpeedUp for dense matrix-vector multiplication using 64 processors, and matrices of size $N = 360, 640, 1600$**

<table>
<thead>
<tr>
<th>$\frac{N}{64} \times N$ / node</th>
<th>Mflops 1 p</th>
<th>Mflops 64 p</th>
<th>$S(N, 64)$</th>
<th>$S_{\text{C_SpUp1}}$</th>
<th>$S_{\text{C_SpUp2}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 x 320</td>
<td>.439</td>
<td>7.359</td>
<td>16.46</td>
<td>16.76</td>
<td>35.12</td>
</tr>
<tr>
<td>10 x 640</td>
<td>.446</td>
<td>17.048</td>
<td>38.14</td>
<td>38.22</td>
<td>46.63</td>
</tr>
<tr>
<td>25 x 1600</td>
<td>.447</td>
<td>28.292</td>
<td>(*)</td>
<td>63.29</td>
<td>59.50</td>
</tr>
</tbody>
</table>

(*) $T_1$ can not be computed due to limited memory on a single nCUBE processor

### Dense Matrix $\times$ Matrix Multiplication

Second, we consider the implementation of the matrix-matrix operation

$C = \alpha C + \beta A B$ on a wrap around grid of $P$ processors. We assume that
the input data $A, B$ are decomposed into submatrices $A_{i,j}, B_{i,j} \in \mathbb{R}^{\sqrt{P} \times \sqrt{P}}$ which are stored in each processor $(i,j)$. The product submatrix $C_{i,j}$ is computed in $\sqrt{P}$ interactions. If we suppress the block indexes, then the computation carried out by each processor in the $k$th iteration consists of sending $B, C$ to processors $(i, (j-1) \mod P)$ and $((i+1) \mod P, j)$ respectively, computing $C = C + AB$, receiving $B$ and $C$ from processors $(i, (j+1) \mod \sqrt{P})$ and $((i-1) \mod \sqrt{P}, j)$ respectively. It can be shown that the time complexity of the algorithm is

$$T_P = \sqrt{P}\left\{\frac{N^3}{P^2} + 2(\gamma + \delta \frac{N^2}{P})\right\}$$

(3.4)

and the space complexity is $O(N^3/P)$. The described computation was implemented on the nCUBE II and its performance is depicted in Table 2. Again we see a close to optimal behavior. We will use these data to compare the performance of VML matrix operations for sparse matrices.

Table 2: Measured Mflops and SpeedUp for matrix-matrix multiplication using 64 processors, and $N = 160, 280, 360, 560$

<table>
<thead>
<tr>
<th>$N \times N$ / node</th>
<th>Mflops</th>
<th>Mflops 64</th>
<th>$S(N, P)$</th>
<th>Scl.SpUp1</th>
<th>Scl.SpUp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 x 20</td>
<td>0.440</td>
<td>22.870</td>
<td>51.991</td>
<td>51.977</td>
<td>55.150</td>
</tr>
<tr>
<td>35 x 35</td>
<td>0.441</td>
<td>25.117</td>
<td>55.966</td>
<td>56.954</td>
<td>58.873</td>
</tr>
<tr>
<td>45 x 45</td>
<td>0.441</td>
<td>25.794</td>
<td>58.437</td>
<td>58.489</td>
<td>59.976</td>
</tr>
<tr>
<td>70 x 70</td>
<td>0.441</td>
<td>26.064</td>
<td>60.351</td>
<td>60.462</td>
<td>61.373</td>
</tr>
</tbody>
</table>

3.4 Parallelization of matrix multiplication operations

First we consider the parallelization of the operation $C = C + A b$ on a linear array of $P$ processors when $A$ is a bounded matrix with $w_1, w_2$ upper and lower bandwidths. Throughout the paper we assume that matrices are stored using the sparse scheme [Rice 83]. For simplicity we assume that $N = P$. The proposed implementation is based on a decomposition of matrix $A$ into an upper $U$ (including the diagonal of $A$) and lower $L$ triangular matrices such as $A = L + U$. Furthermore, we assume that row $\{a_{i,j}\}_{j=1}^N$ and $b_j$ are stored in processor $i$. Then the vector $c$ can be expressed as $c = c + Lb + Ub$. The products $Ub$ and $Lb$ are computed within $w_1 + 1$
and \( w_2 \) iterations respectively. The computation involved is described by
the following pseudo code:

**Phase 1: Multiply the Upper triangular \( U \) by \( b \)**

```plaintext
temp := d
For each PE \( i \) do in parallel
   For \( j := 0 \) to \( w_2 \)
      if \((i + j) =< P\) then
         begin
            if \( i = 1 \) then do nothing
            else Send \( d \) to PE \( i-1 \)
            \( c := c + a(i, j+i) * d \)
            if \( i = P \) then do nothing
            else Receive \( d \) from PE \( i+1 \)
         end
         endif
      end
   end
end
```

**Phase 2: Multiply the Lower triangular \( L \) by \( b \)**

```plaintext
For each PE \( i \) do in parallel
   begin
   \( d := temp \)
   For \( j := 1 \) to \( w_2 \)
      if \( i < j \) then
         begin
            if \( i = P \) then do nothing
            else Send \( d \) to PE \( i + 1 \)
            if \( i = 1 \) then do nothing
            else Receive \( d \) from PE \( i - 1 \)
            \( c := c + a(i, i-j) * d \)
         end
         endif
      end
   end
```

14
Without any loss of generality we assume A has K non-zero elements, and \( N \gg w_1 + w_2 + 1 \). It can be shown that the time complexity is:

\[
T_P = \frac{K}{P} + (w_1 + w_2 + 1) \times \left\{ \gamma + \delta \frac{N}{P} \right\} \tag{4.1}
\]

The memory space required for each subdomain is: \( O(\frac{K}{P} + 3\frac{N}{P}) \)

### 3.5 Band Matrix x Band Matrix Multiplication

Second, we consider the implementation of \( C = aC + \beta AB \), on a grid of P processors when A, B are banded matrices with \( u_1, u_2 \) upper bandwidths, \( l_1, l_2 \) lower bandwidths respectively and \( \alpha, \beta \) real scalars. Again we describe the realization for \( N = P \). The case \( N \gg P \) is straightforward generalization. The processor \( i \) computes column \( C_i \) of matrix \( C \) and holds one row of matrix \( A \) (denoted by \( A_i \)) and a column of matrix \( B \) (denoted by \( B_i \)). The implementation proposed for this operation is described by the following algorithm:

**Algorithm**

The algorithm consists of two phases as in band-matrix vector multiplication. In the first phase, each PE starts by calculating \( C_{ii} = A_{i} \times B_{i} \), then each PE \( i \) passes \( B_{i} \) to PE \( i-1 \); this phase is repeated \( u_1 + u_2 + 1 \) times. In the second phase each PE restores \( B_{i} \), and passes it to PE \( i+1 \), this phase is repeated \( l_1 + l_2 \) times.

**Phase 1**

\[
\text{temp} := b
\]

For each PE \( i \) do in parallel /* each PE contain \( a = A_{i} \), \( b = B_{i} \) */

For \( j := 0 \) to \( u_1 + u_2 \)

if \( (i + j =< N) \) then

begin

if \( (i = 1) \) do nothing 
else Send \( b \) to PE \( i-1 \)
\( c(i, i+j) := c(i, i+j) + a \times b \)
if \( (i = P) \) then do nothing 
else Receive \( b \) from PE \( i+1 \)

endif

endfor

endfor

endfor
Phase 2

\[ b := \text{temp} \]

For Each PE \( i \) in parallel do

For \( j := 1 \) to \( 11 + 12 \) do

if \( (i > j) \) then

begin

if \( (i = P) \) then do nothing

else send \( b \) to PE \( i+1 \)

if \( (i = 1) \) then do nothing

else receive \( b \) from PE \( i-1 \)

\[ c(i, i-j) := c(i, i-j) + a \times b \]

endif

endfor

endfor

Complexity Analysis

Without lost of generality we assume that \( K_1, K_2 \) are the number of non-zero elements for the matrices \( A, B \) respectively and denote by \( w_1 = u_1 + l_1 + 1 \) and \( w_2 = u_2 + l_2 + 1 \) then we can show that

\[
T_P = \min(K_1w_2, K_2w_1) \frac{1}{P} + \left( \gamma + \delta \frac{N}{P} \min(w_1, w_2) \right) \tag{4.2}
\]

The above realizations have been implemented on the NCUBE 6400. Tables 3 and 4 indicate the performance of BLAS 2 computation for a block tridiagonal matrix where each block is dense. In these experiments each processor has the same computation to perform. The results indicate very satisfactory performance for these type of data.

Table 3: Measured Total Elapsed time (in Seconds) for block tridiagonal matrices, each block is of size \( n \times n \), where \( n = 8, 16, 32, 64 \)

<table>
<thead>
<tr>
<th>matrix size / node</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
<th>64 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 24</td>
<td>3.9E-3</td>
<td>4.1E-3</td>
<td>4.1E-3</td>
<td>4.2E-3</td>
<td>4.1E-3</td>
</tr>
<tr>
<td>16 x 48</td>
<td>1.18E-2</td>
<td>1.24E-2</td>
<td>1.27E-2</td>
<td>1.28E-2</td>
<td>1.2E-2</td>
</tr>
<tr>
<td>64 x 192</td>
<td>0.1689</td>
<td>0.1756</td>
<td>0.1794</td>
<td>0.1813</td>
<td>0.1822</td>
</tr>
</tbody>
</table>
Table 4: Measured Mflops for block tridiagonal matrices, each block is of size $n \times n$, where $n = 8, 16, 32, 64$

<table>
<thead>
<tr>
<th>matrix size / node</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
<th>64 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 24</td>
<td>0.239</td>
<td>0.494</td>
<td>1.004</td>
<td>2.0265</td>
<td>4.122</td>
</tr>
<tr>
<td>16 x 48</td>
<td>0.309</td>
<td>0.635</td>
<td>1.284</td>
<td>2.589</td>
<td>5.198</td>
</tr>
<tr>
<td>32 x 96</td>
<td>0.334</td>
<td>0.688</td>
<td>1.393</td>
<td>2.802</td>
<td>5.620</td>
</tr>
<tr>
<td>64 x 192</td>
<td>0.343</td>
<td>0.704</td>
<td>1.425</td>
<td>2.868</td>
<td>5.752</td>
</tr>
</tbody>
</table>

4 Parallel Neural Network Library

In this section, we make an attempt to formulate the neural nets included in Figure 1 in a matrix-vector form. For this presentation, we have adopted the notation and description adopted in [HNC 90].

4.1 Hopfield Net (HOP)

The Hopfield network has been useful in performing content-addressable recall and solving optimization problems. A standard Hopfield network consists of two processing slabs (slab = groups of PEs with the same processing equations). A slab can be viewed as a layer. Following [HNC 90] description, we will divide each Hopfield PE into two components ($U$ and $V$) resulting in 3 slabs, where the first slab is the bias slab ($I$). Next we describe the processing equations based on various numerical differentiation rules. Throughout we make an attempt to use a matrix vector formulation of these equations.

4.1.1 Euler processing equations

$\vec{u}^{\text{old}}, \vec{u}^{\text{new}}$ — neural net input vectors to PEs
$
\vec{T}$ — bias vector
$T$ — weight matrix where $T_{ij}$ is the weight associated with the $j$th input to the $i$th Hopfield PE
$U_0$ — effective steepness parameter
$\tau$ — decay constant
$\Delta t$ — time increment
$V$ — output vector from PEs
$g$ — activation function
The equations and steps used to determine the output of Hopfield PEs are

i) Input $T, \Delta t, \tau, u_0, T$

ii) Initialization of $\overline{u}^{\text{old}}$.

iii) Iterate to compute $\psi = g(u_i/u_o)$.

iv) Iterate to compute $u_{\text{new}} = \overline{u}^{\text{old}} + \Delta t(T\overline{V} + I - \overline{u}^{\text{old}}/\tau)$

4.1.2 Runge-Kutta processing equations

In this case the equations and steps used to determine the output of Hopfield PEs are:

i) Same input as in Euler's case.

ii) Initialization of $\overline{u}^{\text{old}}$.

iii) Iterate to compute $\psi = g(u_i/u_o)$.

iv) Iterate to compute the input to Hopfield PEs.

The equations used in processing $U$ slab are

$$U_{\text{new}} = \overline{U}^{\text{old}} + I + \frac{\Delta t}{6} (\overline{K}_1 + 2\overline{K}_2 + 2\overline{K}_3 + \overline{K}_4)$$

where

$$\overline{K}_1 = T\overline{V}_1 - \overline{U}^{\text{old}}/r,$$

$$\overline{K}_2 = T\overline{V}_2 - (\overline{U}^{\text{old}} + \frac{\Delta t}{2}\overline{K}_1)/r,$$

$$\overline{K}_3 = T\overline{V}_3 - (\overline{U}^{\text{old}} + \frac{\Delta t}{2}\overline{K}_2)/r,$$

and

$$\overline{K}_4 = T\overline{V}_4 - (\overline{U}^{\text{old}} + \Delta t\overline{K}_3)/r.$$ 

The processing equations used in the $V$ slab are:

$$\overline{V}_1 = g(U_i), \quad \overline{V}_2 = g(U_i + 0.5\Delta t(\overline{K}_1); i),$$

$$\overline{V}_3 = g(U_i + 0.5\Delta t(\overline{K}_3); i),$$

and

$$\overline{V}_4 = g(U_i + \Delta t(\overline{K}_3); i).$$

The notation is defined in section 4.1.1.
4.2 Multilayer back propagation (MBPN)

From manual "Description, features, applications and network structure".

Processing Equations

Let's denote for \( \ell \)th hidden or output slab:

\[
\begin{align*}
N_\ell & \quad \text{Number of neurons;} \\
\bar{U}_\ell & \quad \text{The input vector;} \\
I_\ell & \quad \text{External bias;} \\
W^{\ell} & \quad \text{Weight matrix between } \ell - 1 \text{ and } \ell \text{ slabs;} \\
W^0 & \quad \text{Weight matrix between input and output slabs;} \\
\bar{V}^{\ell} & \quad \text{The output vector which for input slab it's denoted by } \bar{V}^0.
\end{align*}
\]

Also, throughout we denote by

\[
\bar{a} = (a_i) \quad \text{vector of elements } a_i; \\
(\bar{a})_i \quad \text{the element } i \text{ of vector } \bar{a}; \\
\bar{a} \times \bar{b} \quad \text{the cross product;} \\
\bar{a} \cdot \bar{b} \quad \text{the inner product;} \\
A^t \quad \text{transpose of matrix } A.
\]

The processing equations and steps that implement the functionality of the network are:

i) Input vector for neurons in slab \( \ell \)

\[
\bar{u}^{\ell} = I^{\ell} + w^{\ell}V^{\ell-1} + cW^0V^0
\]

where \( c = 1 \) if slab \( \ell \) is the output slab and the connections from the input slab to the output slab are enabled \( (w^0 \neq 0) \), otherwise \( c = 0 \).

ii) Compute output vector slab \( \ell \)

\[
\bar{V} = (g(u_i))
\]
iii) Learning rule for the output slab ($\ell = 0$).

\[
\delta^\ell = (g'(U^\ell)) \times (\bar{t} - \bar{v}^\ell),
\]
\[
\Delta W^\ell = \alpha_t (\delta^\ell \cdot (\bar{v}^{\ell-1})^t),
\]

and

\[
W^\ell_{\text{new}} = W^\ell_{\text{old}} + \Delta W^\ell
\]

where

$g'$ = The derivative of the activation function,

$\alpha_t$ = The learning rate for the $\ell$th slab,

$\bar{t}$ = The learning input vector.

iv) Learning rule for the hidden slab.

\[
\delta^\ell = (g'(U^\ell)) \times ((W^{\ell+1})^t \delta^{\ell+1}),
\]
\[
\Delta W^\ell = \alpha_t (\delta^\ell \cdot (\bar{v}^{\ell-1})^t),
\]

and

\[
W^\ell_{\text{new}} = W^\ell_{\text{old}} + \Delta W^\ell
\]

In the implementation of MBPN one of the following two operations are performed to determine the gradient direction: batching or smoothing. The processing equations for batching are:

\[
\Delta W^\ell_{\text{new}} = \alpha_t (\delta^\ell (\bar{v}^{\ell-1})^t) + \Delta W^\ell_{\text{old}}
\]
\[
W^\ell_{\text{new}} = \begin{cases} 
W^\ell_{\text{old}} + \Delta W^\ell_{\text{new}}/\text{count} & \text{if count = batchsize} \\
W^\ell_{\text{old}} & \text{otherwise}
\end{cases}
\]

The processing equations for smoothing are

\[
\Delta W^\ell_{\text{new}} = (1 - \beta_t) \alpha_t (\delta^\ell \cdot (\bar{v}^{\ell-1})^t) + \beta_t \Delta W^\ell_{\text{old}}
\]

and

\[
W^\ell_{\text{new}} = W^\ell_{\text{old}} + \Delta W^\ell_{\text{new}}
\]

where $\beta_t$ is the smoothing factor for slab $\ell$. 

20
4.3 Counter Propagation (CPN)

The outer propagation network is designed to adaptively learn a mapping between a set of input and output vectors from examples of the mapping's action. CPN is useful in solving problems that require the ability to learn a mathematical mapping by adaptation in response to examples of correct mappings.

Network Structure

This architecture consists of four slabs: the input slab, the training slab, the Kohonen slab and the Grossberg slab. The input slab is fully connected to the Kohonen slab, which in turn is fully connected to the Grossberg slab. An adaptive weight is associated with each input connection for PEs on the Kohonen and the Grossberg slabs. In addition, each Grossberg PE receives one input from its corresponding training slab PE. This input is used only during training and has no associated weight.

Processing Equations

The equations below are used to update the CPN PE states and weights. The following definitions are used in the discussion:
\( N \) = Number of Kohonen PEs;
\( n \) = Number of input PEs;
\( m \) = Number of output PEs;
\( \vec{x} \) = PE values of the input slab \((x_1, \ldots, x_n)\);
\( \vec{y} \) = PE values of the training slab \((y_1, \ldots, y_m)\);
\( \vec{z} \) = PE values of the Kohonen slab \((z_1, \ldots, z_N)\);
\( \vec{y}' \) = PE values of the Grossberg slab \((y_1, \ldots, y_m)\);
\( \vec{w}_i \) = Weight vector of the \( i \)th Kohonen PE \((w_{i1}, \ldots, w_{in})\);
\( \vec{w}_j \) = Weight vector of the \( j \)th Grossberg PE \((w_{j1}, \ldots, w_{jN})\);
\( b_i \) = Bias for the \( i \)th Kohonen PE;
\( p_i \) = Win frequency for the \( i \)th Kohonen PE.

Step 1

Processing for the Kohonen slab begins with calculating the Euclidean distance between the input vector \( \vec{x} \) and each Kohonen weight vector \( \vec{w}_i \)

\[ d_i = \| \vec{w}_i - \vec{x} \| = \sqrt{\sum (w_{i} - x_{i})^+ (w_{i} - x_{i})}. \]

Step 2

After \( d_i \) is calculated, subsequent Kohonen processing depends on whether training is enabled. If training is on, the PE with the smallest distance is determined according to

\[ z_i = \begin{cases} 
1 & \text{if } i \text{ is the smallest integer for which } d_i \leq d_j \text{ for all } j = 1 \ldots N \\
0 & \text{otherwise}
\end{cases} \]
Step 3
The Grossberg slab then uses $z_i$ to modify its weights. The equations used on the Grossberg slab are discussed later in this section.

Next, the Kohonen PE distances are adjusted by the bias values to yield the biased distances required by the conscience mechanism. This is done according to

$$d'_i = \begin{cases} 
d & \text{if } \text{Win Frequency}_i < T \\
d_i - b_i & \text{if } \text{Win Frequency}_i \geq T. 
\end{cases}$$

Step 4
The biased distances are then used to determine which PE will modify its weight vector. The selection of the biased winner is according to

$$z'_i = \begin{cases} 
1 & \text{if } i \text{ is the smallest integer for which } d'_i \leq d'_j \text{ for all } j = 1 \\
0 & \text{otherwise}
\end{cases}$$

Step 5
After the biased winner is selected, the weight vectors are modified according to

$$\bar{w}_i^{\text{new}} = \bar{w}_i^{\text{old}} + \alpha(\bar{z} - \bar{w}_i^{\text{old}})z'_i + \beta(\bar{z} - \bar{w}_i^{\text{old}})(1 - z'_i)$$

where $\alpha$ and $\beta$ are user defined parameters.

Step 6
The bias term is calculated according to

$$b_i = c \left( \frac{1}{N} - p_i \right)$$

where $c$ is user determined parameter.

Thus, as the Kohonen PEs near the equiprobable distribution, the conscience mechanism is automatically disabled.
Step 7

The actual win frequency for a Kohonen PE is calculated using a fading-window averaging process. This is accomplished by

\[ p_i^{\text{new}} = p_i^{\text{old}} + b(z'_i - p_i^{\text{old}}) \]

where \( b \) is a user parameter which determines the period over which the average is taken.

Step 8

If training is not enabled, the Kohonen output values \( v_i \) are determined according to the following equations. These equations define an interpolation mechanism when the parameter \( \text{Winners} \) is greater than one. Let \( q \) equal the desired number of winners, then define \( S \) as

\[ S = (i_1, i_2, \ldots, i_q) \]

where \( i_1, i_2, \ldots, i_q \) are the Kohonen PE indices such that \( d_{i_1}, d_{i_2}, \ldots, d_{i_q} \leq d_j \) for all \( j \in \{1, 2, \ldots, N\} - S \). The actual number of winners in \( S \) can be greater than the desired number if a tie occurs at \( d_{i_q} \). In this case all PEs with a distance of \( d_{i_q} \) are accepted, and \( q \) is incremented to reflect the number of indices in \( S \). The minimum distance must then be selected:

\[ d_0 = \min(d_i), i \in S. \]

Using \( d_0 \), each Kohonen PE output, \( z_i \) is calculated as follows:

\[ c_i = \begin{cases} \frac{d_0}{d_i} & \text{if } i \in S \text{ and } d_0 \neq 0 \\ 1 & \text{if } i \in S \text{ and } i \text{ is the smallest integer such that } d_i = d_0 = 0 \\ 0 & \text{if } i \in S \text{ and } d_i \neq d_0 = 0 \\ 0 & \text{if } i \in S \end{cases} \]

\[ f_i = c_i^T \]

\[ z_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \]
Step 9

The processing equations for the Grossberg slab are much simpler than those for the Kohonen slab. The output of the kth Grossberg PE is calculated by:

$$y'_k = u_k \cdot z = \sum_{i=1}^{N} u_{ki} z_i$$

where $z$ is a vector containing all the $z_i$ values.

If training is enabled, then:

$$u_{ki}^{new} = \begin{cases} u_{ki}^{old} + \alpha (y_k - u_{ki}^{old}) & \text{if } z_i = 1 \\ u_{ki}^{old} & \text{if } z_i = 0 \end{cases}$$

or

$$u_{ki}^{new} = u_{ki}^{old} + \alpha (y'_k - u_{ki}^{old}) \times z$$

where $\alpha$ is a network parameter and $y_k$ is the $k$th element of the training vector. Only the Grossberg weights associated with connections from the winning Kohonen PE (the only one for which $z_i = 1$) are modified. In the steady-state solution, this equation becomes:

$$u_{ki} = AVG(y_k)$$

where $AVG(y_k)$ is the average value of all $y_k$ values present when this weight was allowed to modify. This average uses an exponentially-decaying time window with decay rate determined by the parameter $\alpha$.

4.4 Learning Vector Quantization (LVQ)

The LVQ network is applicable to difficult pattern classification problems. Its adaptive capabilities allow it to be used in problems in which there is little a priori knowledge of the pattern class distributions. It has been used successfully to classify phonemes derived from continuous speech data.

The HNC version of Learning Vector Quantization to be implemented on parallel machines contains three slabs: the input slab, the Kohonen slab, and the training slab. The input slab is fully connected to the Kohonen slab. An adaptive weight is associated with each input to the Kohonen slab connections. The Kohonen slab is partitioned into groups of PEs. There is one group for each pattern class. Thus each Kohonen slab PE is assigned
to a particular pattern class. The number of PEs per class must be equal for all classes. The training slab contains one PE that is connected to each Kohonen slab PE. There are no weights associated with these connections. This training slab PE must contain the right class number of the input pattern vector.

Processing Equations

Step 1

The standard LVQ training mode begins with a calculation of the distance $d_i$, between each Kohonen PE's weight vector $\bar{w}_i$ and the $i$ input vector $\bar{x}$ according to

$$d_i = \| \bar{w}_i - \bar{x} \| = \sqrt{\sum_{j=1}^{n} (w_{ij} - x_j)^2}$$

where $n$ is the size of the input vector.

Step 2

The PE with the smallest distance is designated the network-wide winner. With conscience disabled, this PE adjusts its weight vector according to

$$\bar{w}_q^{new} = \begin{cases} 
\bar{w}_q^{old} + \alpha(\bar{x} - \bar{w}_q^{old}) & \text{if the network-wide winner is in correct class} \\
\bar{w}_q^{old} - \gamma(\bar{x} - \bar{w}_q^{old}) & \text{if the network-wide winner is in incorrect class}
\end{cases}$$

where $\alpha$ and $\gamma$ are the user-selected learning rates and the index $q$ designates the network-wide winner. This equation moves the network-wide winner's weight vector a small distance toward or away from the input vector along the line joining the current weight vector and the input vector.

Step 3

When conscience is enabled, the PEs assigned to the correct class (i.e., the class associated with the input vector) have their distances adjusted by a bias term according to the following equations:

$$d_i^c = d_i - b_i$$
where $c$ is a user parameter, $N$ is the number of Kohonen PEs per class, and $p_i$ is the relative win frequency for the $i$th PE. The win frequencies are calculated among the PEs assigned to a class; they are not network-wide.

**Step 4**

The biased distances are then used to calculate the in-class winner. If the indices $q$ and $s$ are used to designate the network-wide and in-class winners respectively, the weight vectors of these two PEs are adjusted according to

$$ b_i = c \left( \frac{1}{N} - p_i \right) $$

where $c$ is a user parameter, $N$ is the number of Kohonen PEs per class, and $p_i$ is the relative win frequency for the $i$th PE. The win frequencies are calculated among the PEs assigned to a class; they are not network-wide.

**Step 4**

The biased distances are then used to calculate the in-class winner. If the indices $q$ and $s$ are used to designate the network-wide and in-class winners respectively, the weight vectors of these two PEs are adjusted according to

$$ w_i^{new} = \begin{cases} 
  w_i^{old} & \text{if network-wide winner is in the correct class} \\
  w_i^{old} - \gamma (\bar{x} - w_i^{old}) & \text{if network-wide winner is not in correct class}
\end{cases} $$

$$ w_s^{new} = \begin{cases} 
  w_s^{old} + \alpha (\bar{x} - w_s^{old}) & \text{if in-class winner is in the correct class} \\
  w_s^{old} + \beta (\bar{x} - w_s^{old}) & \text{if in-class winner is not in correct class}
\end{cases} $$

where $\beta$ is a user selected learning rate. The network-wide and in-class winners may be the same PE, in which case the weight vector is adjusted by the $\alpha$ factor.

**Step 5**

The win frequencies for each PE are calculated using a fading-window average. The equation for this is given by

$$ p_i^{new} = \begin{cases} 
  p_i^{old} + b(1.0 - p_i^{old}) & \text{if PE } i \text{ is the in-class winner} \\
  p_i^{old} + b(0.0 - p_i^{old}) & \text{if PE } i \text{ is not the in-class winner}
\end{cases} $$

**4.5 Adaptive Ring**

The adaptive ring network is suitable for optimization problems involving conflict between a mapping constraint and a topological constraint. For example, in the traveling salesman problem the requirement that the salesman visit every city forms the mapping constraint, and the requirement that the tour be a closed circuit of minimum length forms the topological constraint.
The adaptive ring neurosoftware can also operate in a disconnected mode, in which the connections between PEs are removed and the PEs are allowed to adapt independently. In this mode the network can be used to implement vector quantification and functions as a nearest-neighbor lookup table. In this mode, the adaptive ring network is similar to CPN, with the notable differences that there is no processing on the output layer as in CPN. Instead, the weights of the winning PE are mapped directly onto the output layer.

The adaptive ring network is a four-slab network that works as a single layer. The slabs include an input slab, a Kohonen slab, a competition slab and a boundary slab. The input slab is used to pass data to the Kohonen slab. It is fully connected to all Kohonen slab PEs. The Kohonen slab is a closed loop of Kohonen PEs with each PE connected to its two nearest neighbors. Each PE is also connected to the competition slab. The single PE competition slab polls the Kohonen PEs and determines which is closest to the input vector. The boundary slab allows a subset of the Kohonen slab PEs to be identified as "boundary" PEs. Such PEs do not adjust their weights as training progresses.

Processing Equations

Step 1

An input vector $\mathbf{y}_k$ is selected according to the PDE (Probability Density Function) that is being mapped. The Kohonen slab PE with the closest weight vector $\mathbf{x}_j$ is determined based on the reduced distance

$$d_{kj} = \| \mathbf{y}_k - \mathbf{x}_j \| \left( p_j + \frac{K}{N} \right)$$

where $K$ is a user-selected parameter and $p_j$ is the win frequency for the $j$th Kohonen PE. If $K$ is 0, then the actual distance between $\mathbf{y}_k$ and $\mathbf{x}_j$ is adjusted by the relative win frequency only. As $p_j$ decreases $d_{kj}$ increases. This allows PEs that are not winning very often to win, thereby forcing all Kohonen slab PEs to participate in the PDE mapping. Since $0 \leq p_j \leq 1$ a large value of $K$ will disable the attention mechanism.

Step 2

The relative win frequency estimate $p_j$ is calculated using a fading window averaging process according to
\[
\Delta p_j = \begin{cases} 
  b(1 - p_j) & \text{if the } j \text{th PE is the winner} \\
  b(0 - p_j) & \text{otherwise}
\end{cases}
\]

\[
p_{\text{n}}^\text{new} = p_{\text{j}}^\text{old} + \Delta p_j
\]

where \( b \) determines the size of the fading window.

**Step 3**

The distance metric used has the following form

\[
\| y_i - x_j \| = \sqrt{\sum_k (y_{ik} - x_{jk})^n}
\]

where \( n \) is a user-selected parameter. In most applications, \( n \) is set to 1 (Manhattan matrix) or \( w \) (Euclidean metric).

**Step 4**

Finally the weight vector \( \bar{x}_j \) of the winning PE is adjusted according to the learning law

\[
\Delta \bar{x}_j = \alpha (\bar{y}_k - \bar{x}_j) \beta_j
\]

\[
\bar{x}_{\text{j}}^\text{new} = \bar{x}_{\text{j}}^\text{old} + \Delta \bar{x}_j
\]

where \( \alpha \) is the learning rate for the winning PE. The term \( \beta_j \) is the value of the boundary slab input signal for this PE if the boundary slab is enabled. Otherwise \( \beta_j \) is set to 1 for all Kohonen slab PEs.

**Step 5**

The weight vectors of the PEs that neighbor the winner are also adjusted. These weights are either adjusted toward the input vector or the weight vector of the winning PE. The run-time flag \( \text{WinMap} \) determines which is used. The updating of these weight vectors are given by

\[
\Delta \bar{x}_i = \begin{cases} 
  \beta(\bar{x}_j - \bar{x}_i) B_i & \text{if } \text{WinMap} = 1 \\
  \beta(y_k - \bar{x}_i) B_i & \text{if } \text{WinMap} = 0
\end{cases}
\]

\[
\bar{x}_{\text{j}}^\text{new} = \bar{x}_{\text{j}}^\text{old} + \Delta \bar{x}_j
\]
where $\beta$ is the learning rate for the neighbors of the winning PE and $i = j \pm 1$. The value of $\beta$ is reduced as training progresses by multiplying it by a cooling factor after each iteration of the network.

### 4.6 Bi-directional Associative Memory (BAM)

The BAM network can be used to solve pattern recognition problems in a noisy environment and other applications where content addressability is important. BAM is a feedback neural network, which works as a single functional layer. This layer is divided into two slabs, referred to as the $X$ slab and the $Y$ slab. The $X$ slab is fully connected to the $Y$ slab, and the $Y$ slab is fully connected to the $X$ slab.

#### Processing Equations

**Step 1**

Each BAM PE has a state value of either 1 or -1. When learning is disabled, $Y$ slab PEs update their state values according to the equation:

\[
y_{j}^{new} = \begin{cases} 
1 & \text{if } \sum_{i}^{cX} w_{ji}x_{i} > 0 \\
y_{j}^{old} & \text{if } \sum_{i}^{cX} w_{ji}x_{i} = 0 \\
-1 & \text{if } \sum_{i}^{cX} w_{ji}x_{i} < 0
\end{cases}
\]

where $y_{j}$ is the state value of the $j$th slab PE, $x_{i}$ is the state value of the $i$th $X$ slab PE, $w_{ji}$ is the weight associated with the connection to the $j$th $Y$ slab PE from the $i$th $X$ slab PE, and $cX$ is the size of the $X$ slab.

Similarly, $X$ slab PEs update their state values according to the equation:

\[
x_{j}^{new} = \begin{cases} 
1 & \text{if } \sum_{i}^{cY} u_{ij}y_{i} > 0 \\
x_{j}^{old} & \text{if } \sum_{i}^{cY} u_{ij}y_{i} = 0 \\
-1 & \text{if } \sum_{i}^{cY} u_{ij}y_{i} < 0
\end{cases}
\]

where $u_{ij}$ is the weight associated with the connection to the $i$th $X$ slab PE from the $j$th $Y$ slab PE and $cY$ is the size of the $Y$ slab.

One iteration of the BAM network with learning disabled consists of a $Y$ slab update, followed by an $X$ slab update in accordance with the equations listed above. Typically the network is iterated until the $X$ and $Y$ state values do not change from one iteration to the next. This convergence is guaranteed to occur if the matrix $W$, made up of the weights $w_{ji}$ is the transpose of the matrix $V$, composed of weights $u_{ij}$. In other words, convergence occurs if $w_{ji} = u_{ij}$ for every $i$ and $j$. 

30
Step 2

When learning is enabled, the weights are updated according to an outer product rule. Using this rule, \( L \) associations \( \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_L, Y_L)\} \) where \( X_k \) and \( Y_k \) are column vectors are stored as follows:

\[
W = \sum_{k}^{L} Y_k X_k^T
\]

and

\[
V = \sum_{k}^{L} X_k Y_k^T.
\]

In other words,

\[
W_{ij} = \sum_{k}^{L} Y_{kj} X_{xi}
\]

and

\[
V_{ij} = W_{ji} \sum_{k}^{L} W_{kj} X_{ki}.
\]

One term in the sums given above is calculated on each iteration of the BAM when learning is enabled. The current state values in the \( X \) and \( Y \) slabs are used in the calculation. To create the weight matrices \( W \) and \( V \), the user should load each pair of associated vectors into the \( X \) and \( Y \) slabs and iterate the BAM once per pair.

The number of associations that can be stored with this simple learning technique is quite small and can be expected to be much less than \( \min(cX, cY) \).

4.7 Adaptive Resonance Theory I (ART1)

The adaptive resonance architecture provides pattern recognition where the sequence of input vectors is arbitrary, but continuous. As the values are presented, the network responds in real time with stable, self-organized pattern recognition codes. During recognition, the network matches invariant properties in the input pattern with exemplars in a recognition category. Thus learning is stable and adaptive, and can be buffered against noise and other irrelevant input.

ART1 is a feedback network that can be used in applications where the input noise level is very low so the binary patterns can be perfectly learned.
and/or classified. This includes applications where the exact nature of the patterns may not be known in advance and the input noise level is very low.

The HNC implementation of ART1 is a two-layer neural network with four slabs.

The four slabs used to implement ART1 are F0, F1, F2 and F3. The F0 slab is the input slab. It is connected one-to-one with the F1 slab. The short-term memory (STM) slabs F1 and F2 are used to store activation patterns and generated respond. They communicate with each other via top-down and bottom-up connections. F1 is fully connected with F2 and F2 is fully connected with F1. A set of adaptive weights is associated with both the top-down and the bottom-up connections. These weights implement the long-term memory (LTM) of the network. Slab F3 is a state slab that does no learning or network processing. F0 and F1 are Boolean (integer) arrays, F2 is a trivalent array, and F3 is a two-element array.

**Processing Equations**

The ART1 processing equations are simplified versions of the complete adaptive resonance theory equation set.

The following index sets are used throughout this section:

- \( I \) All active F0 PEs;
- \( V^{(j)} \) All F1 PEs with top-down weights from the \( j \)th F2 PE over the critical level;
- \( X \) All F1 PEs that are currently active;
- \( J \) All F2 PEs that are not inhibited for a given input pattern.

The input pattern is a binary vector with each element \( I_i \) defined as

\[
I_i = \begin{cases} 
\text{Active} & \text{if } i \in I \\
\text{Inactive} & \text{otherwise}
\end{cases}
\]

The output of each F1 slab PE is given by

\[
x_i = \begin{cases} 
\text{Active} & \text{if } i \in X \\
\text{Inactive} & \text{otherwise}
\end{cases}
\]
where $X$ is defined as

$$X = \begin{cases} I & \text{if all F2 PEs are inactive} \\ I \cap V^j & \text{if the } j \text{th F2 PE is active} \end{cases}$$

Only one F2 PE is active at any given time. The output states of the F2 slab PEs are given by

$$y_j = \begin{cases} \text{Active} & \text{if } T_j = \max\{T_k : k \in J\} \\ \text{Inactive} & \text{otherwise} \end{cases}$$

where $T_j$ is

$$T_j = \sum_{i \in X} w_{ij}.$$ 

A reset of the F2 slab active PE occurs if the ratio of active F1 PEs to active F0 PEs is less than the user-selected vigilance parameter $V$. The vigilance parameter should lie between 0 and 1 inclusive. Reset occurs if

$$\frac{||I \cap V^j||}{||I||} < V$$

where the magnitude of an index set is defined as the number of indices in the set.

If learning is enabled, the weights of the F1 and F2 slabs are modified when resonance occurs. The ART1 package implements the fast learning form of the weight modification equations. These are given by

$$x_{ij} = \begin{cases} \frac{L}{i-1+||x||} & \text{if } i \in X \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ji} = \begin{cases} 1 & \text{if } i \in X \\ 0 & \text{if } i \notin X \end{cases}$$

where $j$ is the index of the active F2 PE and $i$ is the index of an F1 PE.

An important part of the processing equations of ART1 is the initial values of the top-down and bottom-up weights. If these weights are not set to the proper initial values, the network will not function correctly.

The initial weight values must satisfy the following conditions:

$$0 < w_{ij} < \frac{L}{L - 1 + M}$$

$$\frac{B_1 - 1}{D_1} < w_{ji} < 1$$

where $M$ is number of F1 PEs and $B_1$ and $D_1$ are user-selected parameters.
5 Performance Evaluation of Parallel ANN

In this section we present some preliminary results of the performance of VML based implementation of HOP and MBPN networks on the nCUBE and Intel IPSC/2 (i860) parallel machines. The VML primitives used have been appropriately mapped to these machines. The HOP net considered was applied to solve the mapping of certain computations to MIMD parallel machines. This mapping is formulated at the geometric/topological data (grids or meshes) of the computations [Hons 91] as an optimization problem. The objective of the mapping strategy realized through a HOP net is to subdivide a grid or mesh in $P$ (number of processors) subdomains that have minimum interface length and equal number of grid points or elements. For the results presented here, we have considered a HOP net for the 2-way partitioning of an orthogonal $N \times N$ grid of an orthogonal region. First in Table 5, we present the total sequential time of three different processors.

<table>
<thead>
<tr>
<th>Connections per iteration</th>
<th>nCUBE II</th>
<th>i860</th>
<th>Sun 4/470</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8^4$</td>
<td>14.5</td>
<td>3.2</td>
<td>74.1</td>
</tr>
<tr>
<td>$16^4$</td>
<td>96.4</td>
<td>46.2</td>
<td>166.2</td>
</tr>
<tr>
<td>$32^4$</td>
<td>*</td>
<td>956.8</td>
<td>1,797.9</td>
</tr>
</tbody>
</table>

* memory limitation

Table 5. The execution time of a Hopfield net for the 2-way partitioning of $8 \times 8$, $16 \times 16$ and $32 \times 32$ grids.

Notice that the times for i860 are for scalar code. We intend to repeat these experiments in vector mode. In Table 6 we present the best execution time obtained for the above partitionings in two hypercube parallel machines, nCUBE II and Intel IPSC/2 based on i860. The Intel machine to our disposal has only 16 processors.
Table 6. The HOP net numerical simulation time and machine configurations used for obtaining 2-way partitionings of 8 \times 8, 16 \times 16, and 32 \times 32 orthogonal grids on the nCUBE and Intel IPSC/2.

It appears that the Intel time for a 2-way partitioning of 32 \times 32 can be improved if additional processors were available. For the benchmarking of parallel computers two additional indicators are usually computed. These are the fixed speedup \((S = T_1/T_P)\) and the corresponding efficiency \((e = S/P)\). Tables 7 and 8 present values for these indicators for Intel machine and the 2-way partitioning of 8 \times 8, 16 \times 16, and 32 \times 32 grids using a HOP net.

Table 7. The fixed speedups and corresponding efficiencies for three 2-way grid partitionings based on HOP net in Intel IPSC/2.
Table 8. The fixed speedups and corresponding efficiency for three 2-way grid partitionings obtained by a HOP net in nCUBE II.

For the benchmarking of neurocomputers, two additional measures are usually computed. These are MCPS (Million Connections Per Second) and MCUPS (Million Connections Updated Per Second). In Table 9, we list the MCUPS obtained by the three machines used so far. In the case of HOP net for the grid partitioning application, the

\[ \text{MCUPS} = (\text{number of iterations}) \times \left( \frac{N^4}{\text{time}} \right) \]

where \( N \times N \) is the size of the grid to be partitioned.

Table 9. The machine performance measured in MCUPS for three 2-way grid partitionings obtained by a HOP net.

The above results indicate that the Hopfield net numerical simulation, can be significantly speeded up by general purpose multiprocessors. In fact, we have observed close to optimal performance for large nets. It is worth noting that in all computations floating point arithmetic was used.
To test the performance of the VML based implementation of MBPN network, we considered a three layer BPM net and a very simple application. All the performance indicators presented in the previous tables were computed for this case too. The input of this application is a single vector of size $N$. Table 10 gives the fixed speedups and efficiencies for three different sizes of input vectors on the nCUBE II.

Here we observed that the speedups obtained are far from optimal. It is worth noticing that there are many alternatives for parallelizing the MBPN net other than the VML approach. Table 11 presents the best times on three machines to achieve $10^{-1}$ matching with a priori defined output.

Table 10. The speedups and efficiencies of parallel MBPN for three different size input data on the nCUBE.

<table>
<thead>
<tr>
<th>Processors</th>
<th>nCUBE 32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S$</td>
<td>$S/P$</td>
<td>$S$</td>
</tr>
<tr>
<td>2</td>
<td>1.71</td>
<td>.88</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td>2.61</td>
<td>.65</td>
<td>2.94</td>
</tr>
<tr>
<td>8</td>
<td>2.64</td>
<td>.33</td>
<td>5.34</td>
</tr>
<tr>
<td>16</td>
<td>1.81</td>
<td>.11</td>
<td>5.43</td>
</tr>
<tr>
<td>32</td>
<td>1.02</td>
<td>.03</td>
<td>2.03</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td>1.31</td>
</tr>
</tbody>
</table>

Table 11. The performance of nCUBE, iPSC/2 and Sun IPC for MBPN computations.

<table>
<thead>
<tr>
<th>Best total time in seconds</th>
<th>nCUBE</th>
<th>Sun 4/470</th>
<th>Sun IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>input size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.21</td>
<td>.47</td>
<td>.91</td>
</tr>
<tr>
<td>16</td>
<td>2.23</td>
<td>1.58</td>
<td>1.84</td>
</tr>
<tr>
<td>32</td>
<td>4.51</td>
<td>5.95</td>
<td>6.50</td>
</tr>
<tr>
<td>64</td>
<td>8.77</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>128</td>
<td>17.99</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>256</td>
<td>31.98</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* timings are effected by paging
6 References


[John 89] S. L. Johnsson, T. Harris, and Kapil K. Mathur, Matrix Multipli-


[Mira 84] W. L. Miranker and A. Winkler, Space-Time Representations of

[Mold 82] D. I. Moldovan, On the Analysis and Synthesis of VLSI Algo-
1121-1126.