Implementation of Neural Networks on Massive Memory Organizations [1]

Manavendra Misra and V.K. Prasanna Kumar

Technical Report
CENG 89-30

Department of EE-Systems
SAL 344 - MC 0781 University of Southern California
Los Angeles, California 90089-0781

November 10, 1989

[1]This research was supported in part by the National Science Foundation under grant IRI-8710836. M. Misra is supported by a Pre-Doctoral Merit Fellowship from the USC Graduate School.

Abstract

Artificial Neural Networks (ANN's) possess image recognition properties that are of particular interest to researchers in the image processing community. Early ANN simulations on serial computers however, were too slow to be of practical significance. The SIMD architecture presented here has n PE's and n^2 memory modules arranged in an $n \times n$ array. This massive memory is used to store connection weights. A fully connected, single layer neural network model with n neurons can be mapped onto the architecture in a very natural fashion. An update in this case requires $(n + 2)$ time steps. Although there have been previous attempts to simulate fully connected neural networks on systolic arrays [7], these methods do not exploit sparsity if the network is not fully connected. We show how models with sparse connectivity among neurons can be simulated in $O(\sqrt{n + e})$ time, where e is the number of non-zero weights in the connection matrix. Preprocessing is carried out on the connection matrix of the sparse network resulting in data movement that has an optimal asymptotic time complexity and a small constant factor. The proposed architecture can also be used for the efficient simulation of multilayer networks with a Back Propagation learning scheme. Massive memory organizations are shown to be more attractive for neural network simulations than other known architectures.

1 Introduction

The inspiration to develop artificial neural networks stems from a desire to mimic the immense computational power of the human brain in artificial systems. This biological inspiration has translated into models called Artificial Neural Networks (ANN's) that are characterized by many simple processing elements, interconnected by weighted links.

Models based on the neural paradigm have been shown to possess computational properties that conventional models do not provide naturally. Foremost amongst these is the ability to store images and recall them from partial information about the stored patterns. Memories that achieve such recall are called *associative memories* ([6],[19]). A memory is called *autoassociative* if its output is the stored pattern that most resembles the input key pattern. In a *heteroassociative* memory, on the other hand, a pair of patterns $\{X_i, Y_i\}$ is stored as an association. On being input a key pattern, X, such that X most resembles pattern X_i, the output of the memory is Y_i. Both autoassociative and heteroassociative memories have been designed using ANN's. The recall from some neural associative memories has been shown to be invariant to translations, rotations and distortions of the input pattern [12]. Such a pattern recognition system could be used as a component of a complete vision system.

The theory of Neural Nets has shown that ANN's can prove useful in a variety of real life applications. It now remains to implement such networks so that their full potential can be realized. Initially, neural networks were simulated in software on serial computers. These simulations however, proved to be extremely time consuming and it was realized that ANN's would have to be simulated on fine grain parallel machines that captured the inherent fine grain parallelism of the paradigm. A number of ANN implementations have thus been attempted on existing and specially designed parallel digital architectures.

In [7],[8], the authors have described a scheme to design special purpose systolic ring architectures to simulate neural nets. By recognizing that neural algorithms can be re-written as iterative matrix operations, the authors have been able to directly apply well known techniques for mapping iterative matrix algorithms onto systolic architectures. The method is shown to work for fully connected Hopfield Nets. To simulate a completely connected network with n neurons, the architecture consists of a ring array with n processing elements (PE's), each with a local memory of size n. Each update step, either in the search phase or in the learning phase takes n time steps. Multilayer ANN's can be simulated by this method by using the layer number instead of time as the iteration index and cascading linear arrays, one for each layer. The method however, works efficiently only for fully connected networks. Simulating sparsely connected networks requires the storing of zero weights for all the missing interconnections. A considerable amount of space and time is thus wasted. Also, the existence of wrap-around connections is an undesirable feature of these architectures.

H. T. Kung et al [9] have simulated feedforward neural networks implementing a Back-propagation learning scheme on the CMU Warp machine. The Warp is a programmable systolic machine with 10 powerful PE's and thus provides a coarse grain of parallelism on

which ANN's are simulated. Two implementations of the back-propagation algorithm are described in [9]. In the first implementation, a network partitioning scheme is used and each cell simulates one vertical slice of the network. This network partitioning scheme however, is not very efficient for simulating large networks as each PE has to store all the weights associated with the neurons it is simulating. In the data partitioning scheme, each PE simulates the whole network on a different set of training patterns in parallel. The weights in this case are stored in an external memory and are pumped through the PE's. The main drawback in the data partitioning approach is that it is far removed from the neural paradigm. The Warp is unable to provide the fine grain parallelism desired by neural network simulations with its small number of PE's. The data partitioning scheme is essentially pipelining which works well during the training phase but would be inefficient if a single pattern were to be classified.

A simulation of multilayer ANN's running the Backpropagation learning algorithm on the Connection Machine CM-2 is presented in [25]. The authors have used the physical, nearest neighbor links of the 2D mesh connections in the CM-2 to carry out communication. Each PE stores a vertical slice of the network –the activation values of the neurons in that vertical column, along with the weights of the links coming into these neurons. The activation values for a particular layer are rotated through the PE's and a *multiply-accumulate-rotate* iteration, a process quite similar to the one described in [7], is carried out. Update of weights according to the Backpropagation rule also requires a sum of products to be computed and so, can be carried out similarly.

Other approaches to simulating ANN's on digital hardware are described in [17], [22], [24].

Recently, several researchers have considered alternate interconnection schemes for parallel computations [23], [21], [1]. In this paper, we present techniques for simulating ANN's on one such organization, the Reduced Mesh of Trees organization. A Reduced Mesh of Trees (RMOT) of size n is an SIMD architecture with n PE's and n^2 memory modules arranged in an $n \times n$ array. The i^{th} PE has access to the memory modules in the i^{th} row and the i^{th} column of the memory array. The RMOT has been shown to be very efficient for applications requiring dense data transfer operations [1]. It can provide optimal performance for many image and graph algorithms. A fully connected, single layer neural network model with n neurons can be mapped onto an RMOT of size n in a very natural fashion. An update in this case requires $(n+2)$ time steps. We show how models with sparse connectivity among neurons can be simulated on an RMOT of size $\sqrt{n+e}$ in $O(\sqrt{n+e})$ time, where e is the number of non-zero weights in the connection matrix of the network. Preprocessing is carried out on the connection matrix of the given sparse network resulting in data movement that has an optimal asymptotic time complexity and a small constant factor.

Since an implementation of the RMOT will have a fixed number of PE's, we show how algorithms designed for an RMOT with n PE's translate onto an RMOT with p PE's ($p < n$) with a slight degradation of performance. An RMOT with 16 PE's is being built by a research group at the University of Southern California. The advantage of the

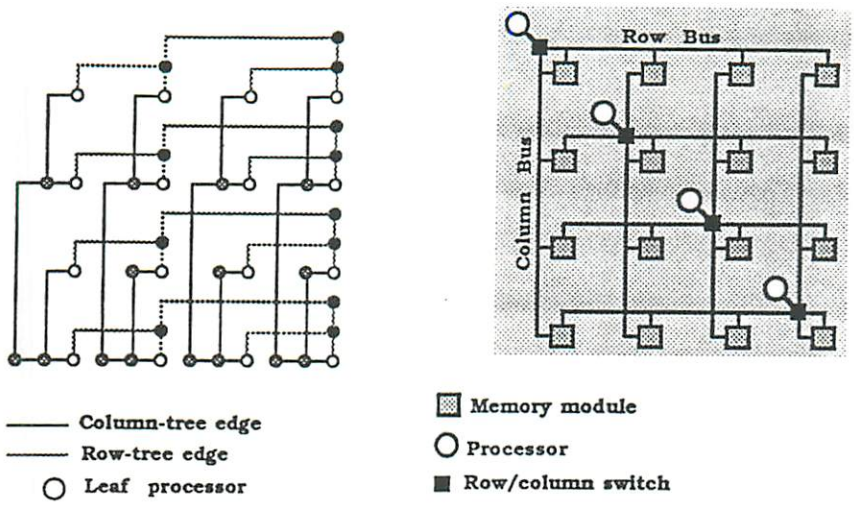

Figure 1: Organization of (a) a 4×4 MOT (b) an RMOT with 4 PE's.

RMOT over other architectures used to simulate ANN's is that it is implementable using current technology. Also, the paper presents a method to simulate sparse networks on the RMOT that is more efficient than simulations on other architectures.

The rest of this paper is organized as follows: Section 2 presents a description of the RMOT architecture. Section 3 presents the neural models addressed by this paper. Section 4 describes how ANN's can be simulated on the RMOT. Section 5 presents the various implementation issues. Section 6 concludes the paper.

2 The Reduced Mesh of Trees Architecture

The organization of an $n \times n$ Mesh of Trees (MOT) is shown in Fig. 1. It consists of an $n \times n$ array of processing elements (PE's) in which each row and each column of PE's form the leaves of a binary tree. The root and the internal nodes of each tree are also PE's. The reduced VLSI architecture considered in this paper consists of n PE's each having row and column access to an $n \times n$ array of memory modules, such that PE_i can access the modules in the i^{th} row and i^{th} column of the array. The memory module in the i^{th} row and the j^{th} column of the array is denoted by M_{ij}. Note that M_{ij} acts as a shared space between PE_i and PE_j. The proposed organization can be looked upon as a Mesh of Trees organization, in which the n^2 leaf PE's are replaced by n^2 memory locations, and each row (column) tree is replaced by a single PE with a row (column) bus. This organization has been called a Reduced Mesh of Trees (RMOT) [1]. The organization of such an architecture is shown in Fig. 1 for $n = 4$. The PE's have arithmetic/logic capabilities, and all their memory registers are $O(\log n)$ bit wide. Also, each memory module consists of a fixed number of $O(\log n)$ bit registers.

Notation: For simplicity and compactness, a memory row will be denoted by RM and a memory column will be denoted by CM. The i^{th} memory row (column) is denoted by RM_i (CM_i). Also, the j^{th} memory module in RM_i will be denoted by $RM_i[j](CM_i[j])$. Unless stated otherwise, it is assumed that the memory modules are indexed according

to the *row-major* indexing scheme. In this scheme, memory module $M_{i,j}$ is given the index $i * n + j$.

Memory Access and Operation Modes: The access of memory locations is done by the PE's. In normal operation, each PE can read or write one unit of data from a single memory location in its row or column. A single bit is used to indicate whether the access is *row-access* or *column-access*. Memory contention is avoided by allowing all PE's to do either a row-access or a column-access, but not both, in one cycle. Each PE specifies a $\log n$ bit address to select the memory module to be accessed. In addition, if each memory module contains k registers or memory locations, then each PE specifies a $\log k$ bit address to select a register within the module.

It should be noted that other research groups have been working with similar orthogonal arrays of PE's ([21], [23]).

3 Neural Network Models

In this section, we first describe the model of an individual neuron and then the model of a general neural network. Finally, in the last subsection, we provide a brief insight into the learning mechanisms incorporated in ANN's with special emphasis on Backpropagation.

3.1 Model of the Neuron

The computational model of the neuron used in ANN's is an abstraction of the characteristic properties of the biological neuron. The earliest neural model was developed in the 1940's by McCulloch and Pitts (Figure 2). The McCulloch-Pitts Neuron [6] is a simplistic two state device. It forms a weighted sum of its inputs and yields a binary output depending on whether the weighted sum is greater than or less than a threshold θ.

$$a_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n} w_{ij} a_j > \theta_i \\ 0 & \text{if } \sum_{j=1}^{n} w_{ij} a_j < \theta_i \end{cases} \tag{1}$$

where a_i is the activation of the i^{th} neuron, w_{ij} is the weight of the connection from neuron j to neuron i and θ_i is the threshold of the i^{th} neuron. To more closely mimic the biological model, this transfer function could be replaced by a continuous, monotonically non-decreasing function which better matches biological data. One such function that is often used is the Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-cx}} \tag{2}$$

where c is a constant. Neurons with continuous transfer functions are called Graded Response Neurons [5].

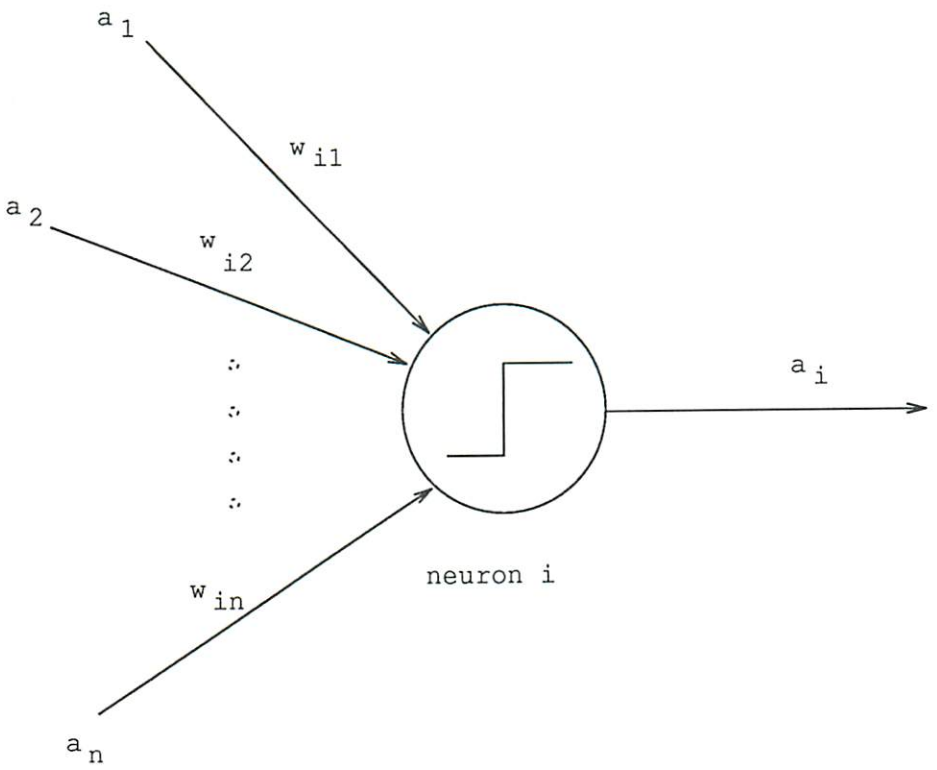

Figure 2: The McCulloch-Pitts Neuron

The method proposed here can simulate most models of ANN's with minor modifications.

3.2 The General Model

A number of ANN models have been proposed in literature [11]. These models can be differentiated on the basis of:

- Whether the network is a single or a multilayer network.
- Whether it is a feedforward network or it has feedback.
- Whether the network incorporates learning [1] or not.

The computations involved in most ANN models however, conform to a common form. The neural networks addressed in this paper adhere to the following general model. A neural network consists of interconnected simple neurons. The input signals received by a neuron are multiplied by appropriate weights and summed to yield the overall input to the neuron. The output of the neuron is produced by applying a function f, called the activation function, to the weighted sum.

The update step can be formally described as

$$a_i^{k+1} = f_i(\sum_{j=1}^{n} w_{ij} a_j^k) \tag{3}$$

The neurons in the network could form a single layer with feedback connections or could form the input, output and hidden layers of a multilayer network. In a single layer network, a neuron computes its new activation value after receiving its inputs, and communicates the new value onto neurons its output connects to. In a multilayer network, the activation values are communicated to the next layer. A forward pass of data from the input layer to the output layer, which does not involve changes of weights, is referred to as a *recall operation* or the *search phase*. Learning can either be executed in the forward pass by carrying out additional computations in the neurons or may require a separate pass of data in the opposite direction (as in the Backpropagation model).

3.3 Learning

Learning is defined as the modification of synaptic weights so as to encode a pattern into the ANN. Learning can either be supervised or unsupervised. In unsupervised learning (eg. Hebbian Learning), a weight of a link is updated based on local information available to the neurons connected by the link. Supervised learning, on the other hand, requires the presence of an external "teacher". The teacher modifies the weights based on the error between a desired response and the actual response to an input.

[1] Learning is defined to be the updating of synaptic weights.

One of the most popular learning schemes for multilayer neural networks is the Backpropagation Algorithm [19]. Backpropagation is a supervised learning mechanism which minimizes the mean squared error between the desired and actual output values. One of the reasons that it is often used to solve real life problems is that it is computationally very cost effective. A training pattern is input to the input layer of the multilayer ANN. Let the actual output of the j^{th} neuron of the output layer be represented by a_j. Let the desired (or target) output at that neuron be t_j. Then, $(t_j - a_j)$ defines the error ϵ_j at that neuron. The change in weight w_{ij} is given by:

$$\Delta w_{ij} = \eta \delta_i a_j \tag{4}$$

where w_{ij} is the weight of the connection from neuron j to neuron i, η is the learning rate and δ_j is the error signal. The error signal is defined as follows. If neuron j is an output unit, then:

$$\delta_j = \epsilon_j f_j'(x_j) \tag{5}$$

x_j is the weighted sum of inputs to neuron j and f_j' is the derivative of the activation function. The error signals for the hidden units are computed recursively:

$$\delta_j = f_j'(x_j) \sum_k \delta_k w_{kj} \tag{6}$$

There are two phases to the Backpropagation Algorithm. In the forward pass, the training pattern is input to the network and activations of the neurons are updated till the output emerges at the output layer. This output is compared with the desired output for that pattern and the error signals are propagated back through the network and the weights are updated. The computational complexity of the backward phase is the same as that of the forward phase.

4 Simulating Neural Networks

In this section, we present a description of how ANN's are simulated on the RMOT. We first describe how fully connected single layer networks are simulated. This is followed by a description of how sparsely connected networks are simulated. Finally, we show how multilayer ANN's with the Backpropagation Algorithm are simulated.

4.1 Simulating Fully Connected Networks

This section describes how a fully connected, single layer neural network is simulated on the RMOT.

4.1.1 Mapping

A fully connected single layer net with four neurons is shown in Fig.3.

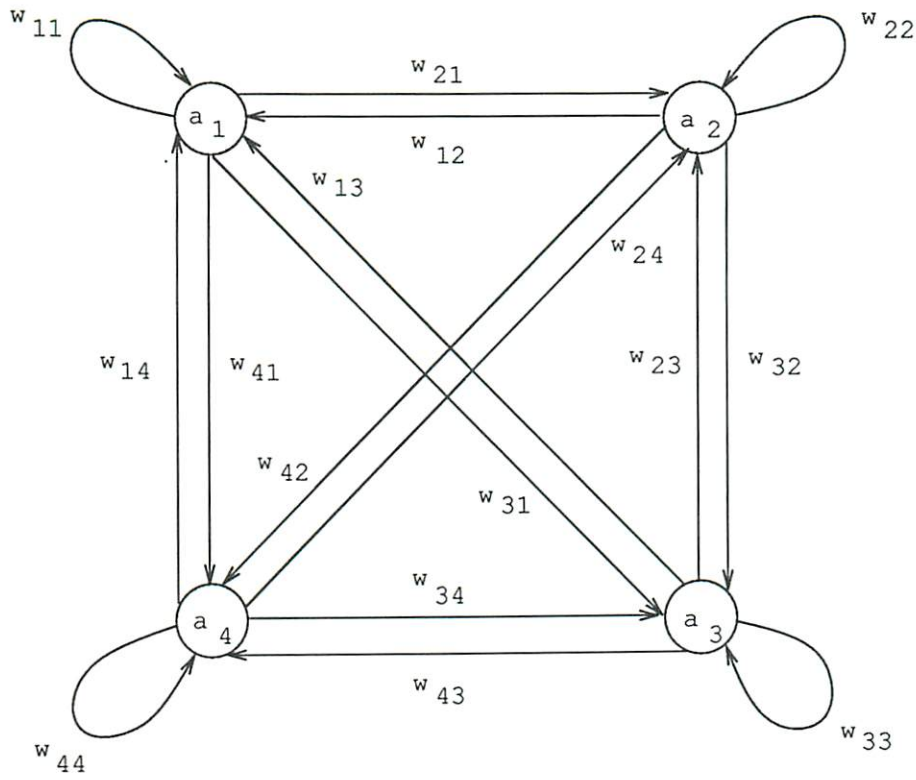

Figure 3: A fully connected single layer net with four neurons.

An RMOT of size n is used to simulate such a net with n neurons. Each of the neurons in the net has n connection weights associated with it. PE i of the RMOT stores the current activation value of the i^{th} neuron. The connection weights are stored in the $n \times n$ array of memory modules in row major order. Connection weight w_{ij} is stored in memory module M_{ij}. Thus, the mapping of the activation values and connection weights onto the RMOT is done in the most natural fashion.

4.1.2 Update of Activation Values

As shown in Eq. 3, an update of the activation value of the i^{th} neuron can be written as

$$a_i^{k+1} = f_i(\sum_{j=1}^{n} w_{ij} a_j^k) \tag{7}$$

where k is the iteration index and f_i is the activation function (step function, sigmoid function etc.).

Equation 7 can be re-written as

$$a_i^{k+1} = f_i(x_i^k) \tag{8}$$

$$x_i^k = \mathbf{W}.\mathbf{a^k} \tag{9}$$

where $\mathbf{W}$ is the $n \times n$ matrix of connection weights and $\mathbf{a^k}$ is the $n \times 1$ vector storing the activation values during the k^{th} iteration. The update of activation values can therefore, be thought of as a matrix-vector multiplication.

The computation represented by Eq.7 can be carried out in three steps:

1. The activation value of the j^{th} neuron, a_j, is communicated to all memory modules that contain the weights of the connections emanating from neuron j (ie. all w_{ij}'s).
2. Computing $\sum_{j=1}^{n} w_{ij} a_j$ for each neuron i.
3. Update of the neuron's activation value by applying the activation function to the weighted sum.

The mapping of activation values and connection weights onto the RMOT described in Section 4.1.1 facilitates the above three steps. They are carried out as follows.

- **Broadcast:** The k^{th} iteration begins with each PE j broadcasting the stored activation value, a_j^k, to all memory modules in the column j. The broadcast of data to modules in a column can be done in just one time step by using the column busses in the RMOT. At the end of the broadcast step, the memory module storing w_{ij} also has a_j.
- **Multiply and Add:** In this phase, a PE accesses each memory module in its row, multiplies the two values (w_{ij} and a_j) and adds the product to a partial sum register (PS). At the end of this phase, the register has the sum $\sum_{j=1}^{n} w_{ij} a_j^k$. Since there are n memory modules in each row, this phase takes n time steps. The steps of this phase are shown in Fig.4.

```
begin
 for each PE i in parallel do
  begin
   PS := 0;
   for j:= 1 to n do
    begin
     read w_ij and a_j from memory module j in row i;
     PS := PS + (w_ij * a_j)
    end
  end
end.
```

Figure 4: The Multiply and Add phase

- **Activation Function Application:** After the multiply and add phase, the register PS has the required weighted sum. The activation function f_i is then applied to this sum to yield the new activation value of that neuron.

The complete update step therefore takes $n + 2$ time steps.

4.1.3 An Example: Simulation of Hopfield Networks

A Hopfield Net ([5],[6]) is a single layer neural network which does not incorporate learning during run time. The interconnection weights are pre-computed depending upon the patterns to be stored in the associative memory being implemented. The calculations of weights is done using Hebb's learning law. The two conditions that need to be satisfied by a Hopfield Network are:

1. $w_{ij} = w_{ji}$ for all i, j.
2. $w_{ii} = 0$ for all i.

Once the interconnection weights have been computed off-line, they are mapped onto the array of memory modules as described. The diagonal memory modules, M_{ii}'s, store 0's and the weights are symmetric about the main diagonal. Hopfield Nets can therefore be simulated by this technique in a straightforward manner.

4.2 Simulating Sparsely Connected Networks

If a single layer ANN has a few links missing, it is possible to carry out the search phase processing in the manner described in the previous section. In this approach, a zero is stored as the weight of a missing connection. If, however, the number of missing connections is very large, a considerable amount of space and time is wasted in storing and computing with the zero weights. An efficient method of carrying out search phase computations for sparse neural networks is presented in this section.

4.2.1 Initial Data Mapping

Let the neural network that is to be simulated have n neurons and e non-zero connections. An RMOT having $\sqrt{n+e}$ PE's and a $\sqrt{n+e} \times \sqrt{n+e}$ array of memory modules is used to simulate the network. The memory locations in a module are either used as data registers or they are used as routing registers. The initial values of the components of the vector a are assigned to the first n PE's in row major order.

Given the sparse connection-weight matrix for the network, the non-zero entries are mapped onto the memory array in snake-like column major order. This ensures that entries belonging to a particular column of the weight matrix form a connected region (Figure 5). In each iteration of the algorithm, the $w_{ij} * a_j$ products corresponding to a row of the weight matrix have to be collected and summed. This requires the realization of a permutation that arranges the elements in snake-like row-major order. Each data element to be routed has three routing tags associated with it. The three routing tags required to achieve this permutation are calculated for each entry and stored in the routing registers. The computation of routing tags is discussed in Section 4.2.2.

Recall that an iteration of the algorithm consists of the computation

$$a_i^{k+1} = f_i(\sum_{j=1}^{n} w_{ij} a_j^k) \qquad \text{for } 1 \leq i \leq n$$

During an iteration:

1. Each a_j is sent to all the memory modules which contain elements from the j^{th} column of the weight matrix.

2. The product of w_{ij} (the element in the memory module) and a_j is to be routed to a module such that in the new distribution of elements, the products corresponding to a row form a connected region.

3. The data in each new connected region are summed to yield the weighted sum.

4. This sum is sent to the appropriate memory module. An application of the activation function on this weighted sum yields the updated activation value.

4.2.2 Data Routing

During each iteration, three kinds of data routing problems arise:

1. The broadcast of a_j to all elements of the j^{th} column of the connection weight matrix.

2. Transformation from a snake-like column major order distribution to a snake-like row major order distribution.

3. Transportation of the sums $(\sum_{j=1}^{n} w_{ij} a_j^k)$ to the memory modules that store the components of a.

$$W = \begin{bmatrix} x & 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & x & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & 0 & 0 & x \\ x & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

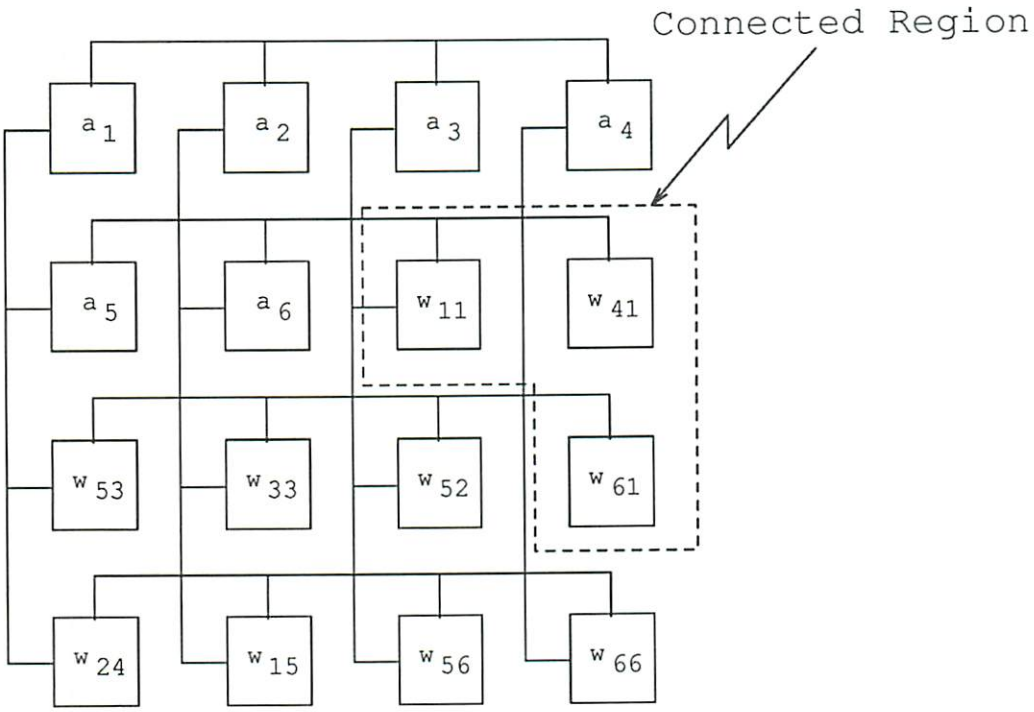

Figure 5: An example of mapping non-zero elements in snake-like column major order

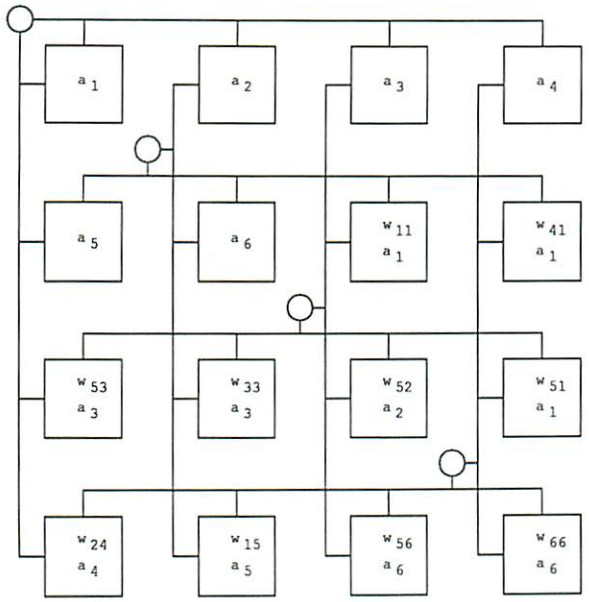

Figure 6: The distribution before the transformation to snake-like row major order

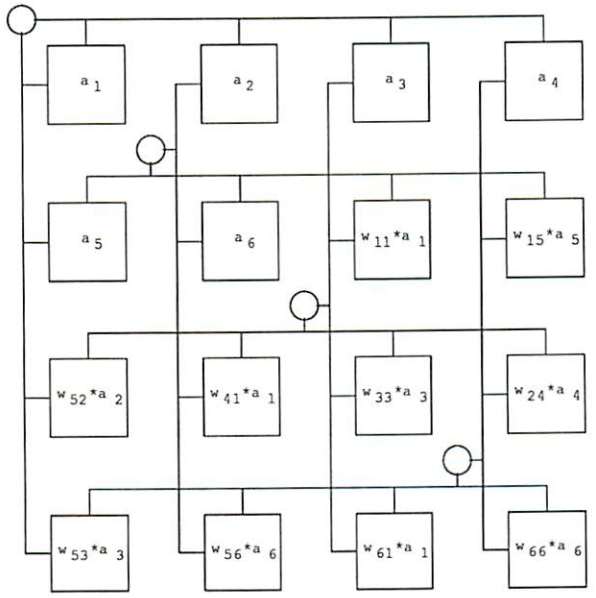

Figure 7: The distribution after the transformation to snake-like row major order

We develop an efficient method that uses preprocessing done on the structure of the weight matrix to solve the above routing problems in $O(\sqrt{n+e})$ time per iteration [13]. We describe the technique to be used to route data before we look at how each routing problem is solved.

The problem of data-transport among memory modules is essentially that of realizing a permutation of the elements contained in the modules. More formally, if M_{ij} has to send data D_{ij} to $M_{i^*j^*}$, then the permutation to be realized is $\pi : (i,j) \rightarrow (i^*,j^*)$. An approach towards realizing such a permutation is to apply the following two steps. In the first step, the elements are moved within their columns till they are in their respective destination rows. In the second step, the elements are moved within their rows till they are in their correct positions. This method, however, could result in many elements accumulating in one module at the end of the first step (eg. if all the elements of a column have the same destination row, they will all end up in the same module). To avoid this kind of congestion, the elements are first permuted within their rows in such a manner that when the permutations along the columns are carried out, no two elements end up in the same module [16].

The 'three-phase' routing method can therefore be described as follows:

Phase I :

Permute the elements within their rows so as to avoid congestion in Phase II.

Phase II :

Permute the elements within columns so as to get them to their destination rows.

Phase III :

Permute the elements within their destination rows so as to get them to their final positions.

Three-phase routing of data is pictorially represented in Figure 8, which can be identified as the Clos Interconnection Network [2], [3].

The rectangular boxes in the figure represent a permutation of a particular row or column in the memory array. For the above routing scheme to be able to realize any permutation of elements, each of the row and column sub-permutations should be able to realize any desired rearrangement. This is indeed possible in $\sqrt{n+e}$ steps (for a row permutation: the PE accesses each module sequentially and writes the data there, in its destination module in that row). The overall routing process therefore takes $3(\sqrt{n+e})$ steps. To decide on what permutations to carry out in the rows and columns, the overall permutation is represented as a $\sqrt{n+e}$-regular, bipartite graph, G_π. A bipartite graph is a graph G whose set of vertices is the disjoint union of two subsets A and B such that each edge of G is from a vertex in A to a vertex in B. A matching for G is a set of edges M, no two of which are incident to the same vertex. A matching M is complete iff the number of edges in M equals the number of vertices in A.

The vertices of G_π represent the rows of the memory array. In this case, the set A represents the rows before the permutation is carried out and B represents the rows after the permutation. An edge in the graph represents the transport of a data element from one row to another in accordance with the permutation to be realized. The

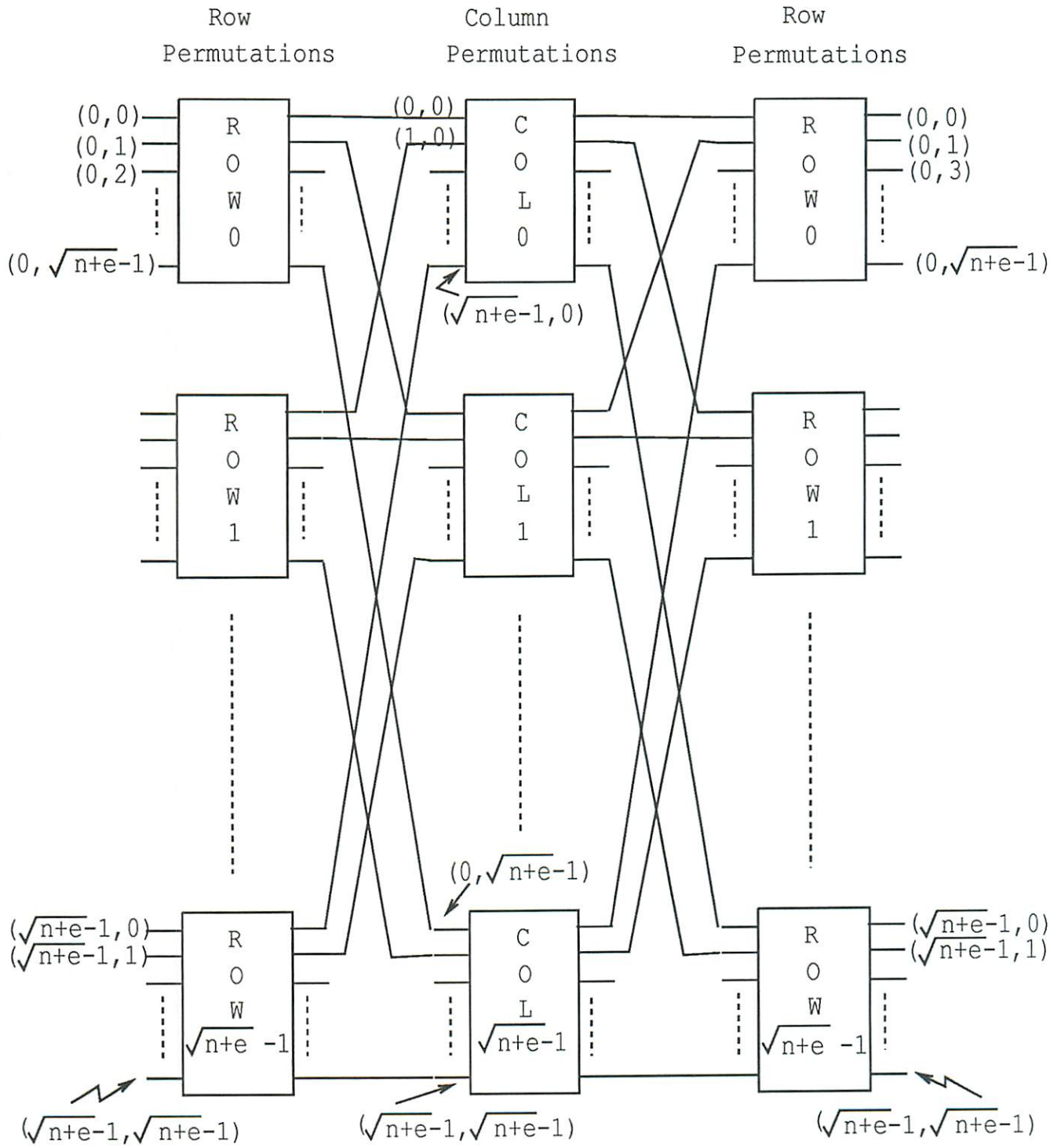

Figure 8: Permuting the contents of the PE's

$\sqrt{n+e}$ *disjoint* complete matchings of this graph are determined. The existence of these matchings is guaranteed by Hall's Theorem [4]. The elements that constitute the first matching are to be routed through the first column. This means that Phase I must route these elements to the first column and Phase III must route them from the first column to their destinations. This process is repeated $\sqrt{n+e}$ times, once for each complete matching, thus determining the elements to be routed through each column. This process also determines the permutations to be carried out in the rows as part of Phases I and III. Each phase of routing requires one routing register per data element which contains the destination address of the module where that element has to be moved to, in that row/column. Each piece of data therefore has three routing registers associated with it: RR_I, RR_{II} and RR_{III}. The computation of these routing tags can be done in $O((n+e)[\log(n+e)]^2)$ time on a serial computer [10] and in $O(\sqrt{n+e})$ time on a $\sqrt{n+e} \times \sqrt{n+e}$ mesh connected computer [14].

Using the procedures shown in Fig.9, the broadcast, re-distribution and update problems are solved as follows.

A. BROADCAST

At compile-time, given the sparse connection matrix (W), the mapping of the non-zero elements onto the memory array in snake-like column major order is computed. This gives the connected regions in the array. An element of a has to be broadcast to each of these connected regions. To do this, a *leader* module is identified in each connected region and the element of a is transported to it using the procedure ROUTE. This part of BROADCAST is completed in $3(\sqrt{n+e})$ time steps. The leader module is either the leftmost or the rightmost module of the top row of the connected region. The leader module has two limit registers associated with it, one indicating how far the connected region extends along its row and the other showing how far down along its column the region extends. The data in the leader module can be copied into all the modules in the connected region in three stages. In the first stage, the data is broadcast to modules in the same column within the connected region. In the second stage, data is broadcast to connected modules in the same row and finally, a third stage broadcasts to connected modules in the same column again. Each stage can take up to $O(\sqrt{n+e})$ time and so the total time required for copying data within a connected region is $\leq 3(\sqrt{n+e})$. The procedure BROADCAST, therefore, consists of calling the procedure ROUTE on the elements of a and then carrying out the data movement described above. The time required for one such broadcast operation is $\leq 6(\sqrt{n+e})$.

After the memory modules receive the appropriate elements of a, the PE's compute the products $w_{ij} * a_j$

B. RE-DISTRIBUTE

The problem can be stated as follows:

> Given a distribution of elements in snake-like column major order, re-distribute them in snake-like row major order.

```
procedure PHASE(RR, phase-number)
 begin
   if phase-number=I or phase-number=III then
     for each PE(i) in parallel do
       for j := 1 to √(n + e) do
         begin
           read data and its destination, RR(i,j), from M_ij;
           write data in destination module in the same row
         end
   else
     for each PE(j) in parallel do
       for i := 1 to √(n + e) do
         begin
           read data and its destination, RR(i,j), from M_ij;
           write data in destination module in the same column
         end
 end.

procedure ROUTE
 begin
   PHASE(RR_I, I)
   PHASE(RR_II, II)
   PHASE(RR_III, III)
 end.
```

Figure 9: The procedures ROUTE and PHASE

At compile-time, the sparse connection weight matrix (W) is available. The mapping of the non-zero entries in snake like column major order as well as in snake like row major order can therefore be computed. The routing problem is then reduced to realizing a permutation of the contents of the modules that converts the distribution from one to the other. This permutation is realized using the three phase routing technique described above.

The procedure RE-DISTRIBUTE is quite similar to ROUTE. Due to the similarities of the procedures, details are omitted. The time required for RE-DISTRIBUTE is $3(\sqrt{n+e})$.

C. UPDATE

The solution of this problem is similar to the solution of the broadcast problem. During preprocessing, when the snake-like row major ordering of the non-zero entries is computed, the leader modules of the connected regions so formed are also determined. The sums of the connected regions come to these leader modules and so it is known beforehand which modules have to send data to the modules storing the components of a. The routing registers of these modules are set accordingly. The procedure UPDATE first forms the weighted sums and gets them to the leader modules. These weighted sums are then routed to the appropriate modules using ROUTE and the components of a are updated by the application of the function f_i. It is easy to verify that the time required for UPDATE is $\leq 6(\sqrt{n+e})$.

4.2.3 The Algorithm

The complete algorithm to update activation values of the neurons of a sparsely connected neural network is presented in this section. The procedures described in Section 4.2 are used in the algorithm. In the pre-processing stage shown in Figure 10, the memory array is set up to perform the iterations. The iterations are performed as shown in Figure 11.

4.3 Simulating Multilayer Networks

In this section, we describe how multilayer ANN's employing a Backpropagation Learning scheme are simulated on an RMOT. Initially, we assume that each layer of the network has an equal number of neurons and all possible connections between layers exist. The method can be modified for other cases too. The computations involved in the search phase of a multilayer network are similar to those for a single layer ANN. The major difference is that layer numbers are used as iteration indices instead of time steps.

4.3.1 Notation and Mapping

The ANN is assumed to have N layers, numbered from 1 to N with m neurons per layer. The neurons in each layer are numbered from 1 to m. The i^{th} neuron in the l^{th} layer has an activation value labeled by a_i^l $(1 \leq l \leq N, 1 \leq i \leq m)$. The weight from the j^{th} neuron

1. Compute the mapping of the non-zero elements of W onto the memory array in snake-like column major order and store it.

2. Identify the "leader" modules in the connected regions formed in step 1. Set the limit registers in these modules to define the boundaries of the connected regions.

3. Compute the mapping of the non-zero elements of W onto the memory array in snake-like row major order and store it.

4. Identify the leader modules in the connected regions formed in step 3. Set the limit registers in these modules to define the boundaries of the connected regions. Set the routing registers of the modules that correspond to the update step so as to get these data to the modules storing the components of a.

5. Store initial components of a^0 in the first n modules in row major order. Compute the routing tags to route these elements to the leader modules identified in step 2 and store them in the routing registers corresponding to the broadcast step.

6. Map the non-zero elements of W onto the memory array in snake-like column major order according to the mapping computed in step 1. Compute the routing tags for these modules for routing data so as to achieve a permutation from the mapping in step 1 to the mapping in step 3.

Figure 10: Preprocessing Steps

repeat

1. Broadcast the elements of a using the procedure BROADCAST. The routing registers, already set, determine which modules the elements go to.

2. Use RE-DISTRIBUTE to get the product terms in the new connected regions.

3. Use UPDATE to form the weighted sums, route them to the modules storing the components of a and update the activation values by applying the activation function to the weighted sum.

until (convergence).

Figure 11: Iteration Steps

in the k^{th} layer to the i^{th} neuron in the $(k+1)^{st}$ layer is labeled as w_{ij}^k $(1 \leq i,j \leq m$, $1 \leq k \leq N-1)$.

The mapping of the activation values and the connection weights is done in a manner similar to that in Section 4.1. We notice that updates are done on a layer by layer basis and parallelism is only required between the neurons of a particular layer. Thus, it is sufficient to use an RMOT with m PE's. The i^{th} PE stores the activation values $a_i^1, a_i^2, \ldots, a_i^N$ in its registers. The derivatives, f_i', are computed along with the activation values and are stored in the PE's too. The PE's also have registers allocated for storing the δ_i's for each neuron during the backward pass. The memory module M_{ij} stores $w_{ij}^1, w_{ij}^2, \ldots, w_{ij}^{N-1}$.

4.3.2 Search Phase

The computations for the update of the activation values for the search phase are given by:

$$a_i^l = f_i(\sum_{j=1}^{m} w_{ij}^{l-1} a_j^{l-1}) \tag{10}$$

This can be re-written as

$$a_i^l = f_i(x_i^l) \tag{11}$$

where (x_i^l) represents the weighted sum of the inputs to neuron i of layer l. Thus, the computations of derivatives, $f_i'(x_i)$, to be used in the learning phase, can also be done at this time.

Equation 10 is quite similar to Equation 7 and so, the search phase computations can be carried out in a manner quite similar to Sec. 4.1. Details are omitted because of the similarity.

4.3.3 Learning Phase

The Learning Phase is composed of the following steps:

- Computation of error signals for the output layer. In terms of our notation, the computation is given by

$$\delta_i^N = (t_i - a_i^N) f_i'(x_i^N) \qquad 1 \leq i \leq m \tag{12}$$

  This computation can be done locally within each neuron.

- Update of weights going from layer $N-1$ to layer N. This computation is given by

$$w_{ij}^{N-1} \leftarrow w_{ij}^{N-1} + \eta \delta_i^N a_j^{N-1} \tag{13}$$

  This will require the broadcast of δ_i^N down a row and then each PE computes the new weight for each of the modules in its column.

- For all other layers (on a layer by layer basis), computation of error signals and the update of weights. This is done according to

$$\delta_i^l = f_i'(x_i^l) \sum_k \delta_k^{l+1} w_{ki}^l \tag{14}$$

$$\Delta w_{ij}^{l-1} = \eta \delta_i^l a_j^{l-1} \tag{15}$$

The calculation of δ_i^l is similar to a matrix vector multiplication operation and so can be done in a manner similar to that of Section 4.1. However, the broadcast of δ_i^{l+1} is done along a row and and the sums are formed for the elements of a column.

5 Implementation Issues

The algorithms presented in the previous section all require an RMOT whose size depends on the size of the ANN being simulated. However, it is of interest to us to introduce a modification of the RMOT such that the number of processors is fixed. Any computation which can be performed on an RMOT with n PE's in $T(n)$ time, can be performed on a fixed-size RMOT with $p \leq n$ PE's in $O(nT(n)/p)$ time [1]. This result is proved as follows: Denote the RMOT with n processors by RMOT(n) and the RMOT with $p \leq n$ processors by RMOT(p). The memory rows of RMOT(n) are partitioned into p *row groups* (RG's). Similarly, the column rows are partitioned into p *column groups* (CG's) and the PE's of RMOT(n) are partitioned into p *processor groups* (PG's). RG_i (CG_i) is mapped onto the i^{th} memory row (column) or RMOT(p) and PE_i of RMOT(p) simulates PG_i. Since each PE in RMOT(p) simulates n/p PE's from RMOT(n), each step which takes $O(1)$ time on RMOT(n) takes $O(n/p)$ time on RMOT(p). If a computation takes $T(n)$ time on RMOT(n), it means that each PE in RMOT(n) performs at most $T(n)$ operations in such a computation. For such a computation, each PE in RMOT(p) must perform at most $(n/p)T(n)$ operations.

The RMOT is a realizable architecture within the framework of current VLSI technology. A research group at the University of Southern California has recently started constructing a prototype of such a parallel machine [15]. The machine being built is to have 16 PE's and will be expandable to 64 PE's. Each PE is a powerful $i860$ microprocessor, a new RISC μP released by Intel. Each memory module is to have a capacity of 256K bits. The observation that a memory module is either accessed from the row bus or the column bus, but not both simultaneously, has led to the replacement of the two port memory modules of the design by single port modules. There is a 2 to 1 multiplexing of the address lines and a demultiplexing of the data. An extension of this design will involve a VLSI implementation, either on a single chip or a chip set.

Another architecture that uses single port memories and 2×2 switches and runs all RMOT algorithms with no loss in time is presented in [20]. This shared memory version of the RMOT is shown in Figure 12. The memory modules in Figure 12 are labeled by their labels M_{ij} from the original RMOT. All the switches in the architecture

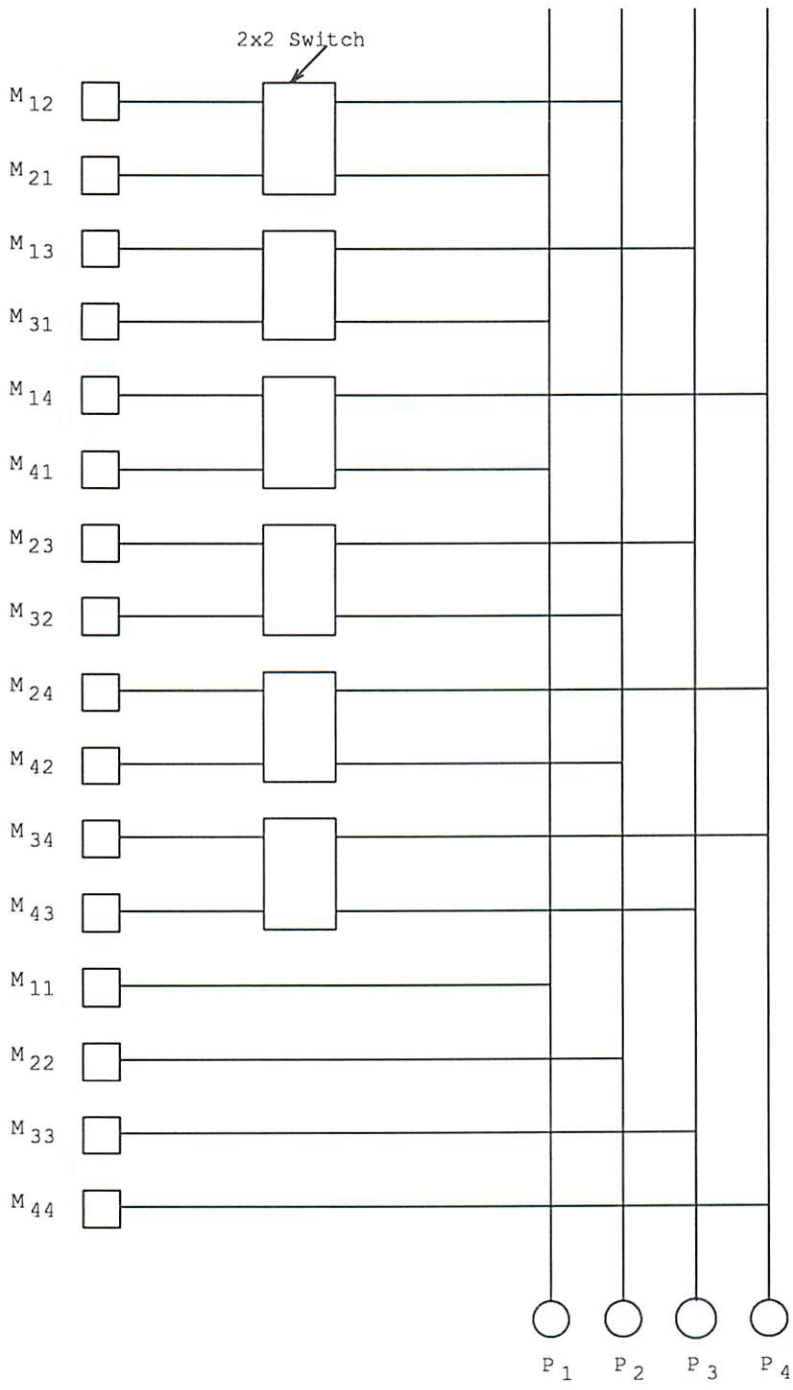

Figure 12: A shared memory version of the RMOT that uses single port memory modules.

are controlled by a single bit. A straight switch setting of the switches corresponds to column access in the original architecture while a crossed setting corresponds to row access. The above two implementations of the RMOT are equivalent in their capabilities and one can be got from the other by replacing the 2×2 switches by appropriately placed mux-demux combinations or vice versa.

6 Conclusion

The RMOT is shown to be an attractive architecture for simulating Artificial Neural Networks. Fully connected single layer ANN's are mapped and simulated on an RMOT with n PE's, in a straight-forward manner in $(n+2)$ time steps. Single layer ANN's with sparse connectivity can be simulated on an RMOT with $\sqrt{n+e}$ PE's in $O(\sqrt{n+e})$ time if preprocessing is carried out on the connection weight matrix of the ANN. It is shown that multilayer ANN's that employ the Backpropagation Algorithm for learning patterns can be efficiently simulated on an RMOT too. In comparison to the linear systolic array simulation of fully connected ANN's [7], the RMOT implementation uses the same number of PE's and updates take the same amount of time. In addition, the RMOT is shown to have an efficient algorithm for handling sparse networks. A linear array simulation of sparse ANN's would use $O(n^2)$ storage as all the zero weights would have to be stored and an update would take $O(n)$ time. An $n \times n$ Mesh Connected Computer (MCC) could also be used to simulate fully connected ANN's in the same amount of time (using sort as a basic routine), but the MCC would have n^2 PE's as compared to only n PE's in the RMOT. Notice that $\Omega(n)$ time is needed to carry out these simulations. These results are tabulated in Tables 1, 2 and 3. A simple partitioning scheme is presented to run the proposed algorithms on a realizable RMOT that has a constant number $(p < n)$ of PE's. With attempts being made to actually build such massive memory machines, it will soon be possible to use ANN's for real life applications.

	# of PE's	Total Storage	Time per Update
Linear Array [7]	n	$O(n^2)$	$O(n)$
MCC	n^2	$O(n^2)$	$O(n)$
RMOT	n	$O(n^2)$	$O(n)$

Table 1: Simulation of Fully Connected Single Layer Networks

	# of PE's	Total Storage	Time per Update
Linear Array [7]	n	$O(n^2)$	$O(n)$
MCC	$n+e$	$O(n+e)$	$O(\sqrt{n+e})$
RMOT	$\sqrt{n+e}$	$O(n+e)$	$O(\sqrt{n+e})$

Table 2: Simulation of Sparsely Connected Single Layer Networks

	# of PE's	Total Storage	Time per Update
Linear Array [7]	$m \times N$	$O(m^2N)$	$O(mN)$
MCC	m^2	$O(m^2)$	$O(mN)$
RMOT	m	$O(m^2)$	$O(mN)$

Table 3: Simulation of a Multilayer network with N layers and m neurons per layer

References

[1] Hussein M. Alnuweiri,"Communication Efficient Parallel Architectures and Algorithms for Image Computations", Ph.D. Thesis, Dept. of EE-Systems, University of Southern California, Aug. 1989.

[2] V. E. Benes, " On Rearrangeable Three-Stage Connecting Networks", *B.S.T.J.*, vol.41, pp. 117-125, Sept. 1962.

[3] C. Clos, "A Study of Non-Blocking Switching Networks", *B.S.T.J.*, vol.32, pp. 406-424, 1953.

[4] D. Gale, "A Theorem on Flow in Networks", *Pacific Journal of Mathematics*, pp. 1073-1082, 1957.

[5] J.J. Hopfield,"Neurons with graded response have collective computational properties like those of two-state neurons,"*Proc. Natl. Acad. Sci. U.S.A.*, vol.81, pp. 3088-3092, 1984.

[6] J.J. Hopfield, "Neural Networks and physical systems with emergent collective computational abilities,"*Proc. Natl. Acad. Sci. U.S.A.*, vol.79, pp. 2554-2558,1982.

[7] S. Y. Kung, "Parallel Architectures for Artificial Neural Nets", *Proc. of the International Conf. on Systolic Arrays*, pp. 163-174, 1988.

[8] S. Y. Kung, J. N. Hwang, "A Unified Systolic Architecture for Artificial Neural Networks",*Journal of Parallel and Distributed Computing*, **6**, pp. 358-387, 1989.

[9] H. T. Kung, D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky " How we got 17 Million Connections per second", *Proc. of the International Conf. on Neural Nets*, Vol. 2, pp. 143-150, 1988.

[10] Gavriela Lev, Nicholas Pippenger and Leslie Valiant, "A fast Parallel Algorithm for Routing in Permutation Networks",*IEEE Transactions on Computers*, vol.c30, No.2, pp.93-100, Feb. 1981.

[11] R. P. Lippmann,"An Introduction to Computing with Neural Nets", *IEEE ASSP Mag.*, pp. 4-22, April 1987.

[12] C. von der Malsburg, "Pattern Recognition by Labeled Graph Matching", *Neural Networks*, vol 1, pp. 141-148, 1988.

[13] Manavendra Misra and V. K. Prasanna Kumar, "Efficient VLSI Implementation of Iterative Solutions to Sparse Linear Systems", *Proc. 3rd Int. Conf. on Systolic Arrays*, Killarney, Ireland, 1989.

[14] David Nassimi and Sartaj Sahni,"Parallel Algorithms to Set Up the Benes Permutation Network", *IEEE Transactions on Computers*, vol.c31, No.2, pp.148-154, Feb. 1982.

[15] D. K. Panda, Personal Communication.

[16] C. S. Raghavendra and V. K. Prasanna Kumar, "Permutations on ILLIAC -IV Type Networks", *IEEE Transactions on Computers*, vol.c37, No.7, pp. 662-669, July 1986.

[17] U. Ramacher, J. Beichter, "Systolic Architectures for Fast Emulation of Artificial Neural Networks", *Proc. 3rd Int. Conf. on Systolic Arrays*, Killarney, Ireland, 1989.

[18] C. R. Rosenberg, G. Blelloch, "Network Learning on the Connection Machine", *Proc. 10th Int. Joint Conf. on Artificial Intelligence*, Milan, Italy, 1987.

[19] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group,*Parallel Distributed Processing (PDP): Exploration in the Microstructure of Cognition (Vol. 1)*, MIT Press, Cambridge, Massachusetts, 1986.

[20] I. D. Scherson, "A Graph Theoretical Description of Spanning bus Hypercube Multiprocessing Systems", Technical Report, Dept. of EE, Princeton University, June 1989.

[21] I. D. Scherson, S. Sen,"Parallel Sorting on two-dimensional VLSI models of computation", *IEEE Trans. Comput.*, **C-38**(Feb. 1989), 238-249.

[22] Sherryl Tomboulian, "Introduction to a System for Implementing Neural Net Connections on SIMD Architectures", *Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, ICASE Report No. 88-3*, Jan. 1988.

[23] P. S. Tseng, K. Hwang, V. K. Prasanna Kumar, "A VLSI Based Multiprocessor Architecture for Implementing Parallel Algorithms", *Proc. Int. Conf. on Parallel Processing*, 1985.

[24] T. Watanabe, Y. Sugiyama, T. Kondo, Y. Kitamura,"Neural Network Simulation on a Massively Parallel Cellular Array Processor: AAP-2", *Proc. Int. Joint Conf. on Neural Networks*, Washington D. C., June, 1989.

[25] X. Zhang, M. Mckenna, J. P. Mesirov, D. Waltz,"An Efficient Implementation of the Backpropagation Algorithm on the Connection Machine CM-2", *Proceedings of NIPS*, 1989.