Efficiently Modeling Neural Networks on Massively Parallel Computers

Robert M. Farber
Los Alamos National Laboratory
Los Alamos, N.M.
87544

Neural networks are a very useful tool for analyzing and modeling complex real world systems. Applying neural network simulations to real world problems generally involves large amounts of data and massive amounts of computation. To efficiently handle the computational requirements of large problems, we have implemented at Los Alamos a highly efficient neural network compiler for serial computers, vector computers, vector parallel computers, and fine grain SIMD computers such as the CM-2 connection machine. This paper will describe the mapping used by the compiler to implement feed-forward backpropagation neural networks (D. Rummelhart and J. McClelland 1986) for a SIMD (Single Instruction Multiple Data) architecture parallel computer. Thinking Machines Corporation has benchmarked our code at 1.3 billion interconnects per second (approximately 3 gigaflops) on a 64,000 processor CM-2 connection machine (Singer 1990). This mapping is applicable to other SIMD computers and can be implemented on MIMD computers such as the CM-5 connection machine. Our mapping has virtually no communications overhead with the exception of the communications required for a global summation across the processors (which has a sub-linear runtime growth on the order of O(log(number of processors))). We can efficiently model very large neural networks which have many neurons and interconnects and our mapping can be extend to arbitrarily large networks (within memory limitations) by merging the memory space of separate processors with fast adjacent processor inter-processor communications. This paper will consider the simulation of only feed forward neural network although this method is extendible to recurrent networks.

A simple XOR network can be seen in Fig 1. This network (or any feed-forward neural network) is "trained" as follows: First, the outputs for each example of a "training set" of examples are calculated for a given set of network parameters (neuron thresholds and connection weights). This can be seen for the XOR problem of fig 1 in eqn. 1.1 - 1.4. In these equations W(a,b) means the connection weight from a to b and g() is a user specified linear or non-linear function. The fitness of the calculated outputs (and

$$H = H_{threshold} + W(I_1,H) * I_1 + W(I_2,H) * I_2 \qquad \text{Eqn 1.1}$$

$$O = O_{threshold} + W(I_1,O) * I_1 + W(I_2,O) * I_2 \qquad \text{Eqn 1.2}$$

$$O \mathrel{+}= g(H) * W(H,O) \qquad \text{Eqn 1.3}$$

$$O = g(O) \qquad \text{Eqn 1.4}$$

hence the network parameters) is determined by some function of the known and calculated outputs. A common fitness function is the sum of the square of the differences as shown in eqn. 2. The parameters of the network are then adjusted by some nonlinear

$$\text{Fitness} = \sum_{0}^{num\_examples} (known\_output - calculated\_output)^2 \qquad \text{Eqn 2}$$

minimization scheme such as powell's method or conjugant gradient (Press *et. al.*_1988). The network is continually adjusted and re-evaluated until a "best fit" is found. The neural network is then said to be "trained". If the number of examples is small relative to the

number of network parameters, then the network can "memorize" the training set. In other words, there are so many parameters in the network that it "memorizes" the training set. Unfortunately, neural networks which are over parameterized generally predict poorly on examples which were not in the training set. Hence most neural networks are trained with a number of examples far larger than the number of network parameters. This "overloading" of the network is done to force the network to "generalize" a solution from the training set. It is then hoped that the network will then predict well on data which was not in the training set. The literature abounds with important problems where neural networks have been shown to be good predictors. For example, neural networks can be used to predict time series with orders of magnitude increases in accuracy over conventional methods (Lapedes and Farber 1987 and Lapedes and Farber 1987). Neural networks have also been shown to be highly accurate predictors of coding regions for short regions of DNA (Farber *et. al.* 1992 and Lapedes and Farber 1989). We can see that the runtime growth for evaluating a neural network during training is on the order of $O(m*n)$ where m is the number of network parameters and n is the number of examples. From our discussion we can see that n generally dominates the runtime growth.

This means that contrary to what one would first expect, the most efficient method of mapping neural networks on to a massively parallel machine is not one neuron per processor. Rather, the most efficient method is to map *one example to each processor*. By using this mapping for SIMD or MIMD (Multiple Instruction Multiple Data) parallel computers, it is possible to get number_of_example operations done in each instruction cycle of the machine by having each processor evaluate the network for it's example. Hence, we effectively get no change in our runtime for a problem which has one example over a problem which has 250,000 examples. In reality, there will be a small increase in the runtime as the number of examples exceeds the number of processors. However this increase is on the order O(number_of_examples/number_of_processors) and is very small for the large numbers of processors in current SIMD machines. Thus, we get essentially large training sets for free. This allows neural networks to be applied to problems of a size and complexity not possible using serial machines. Our mapping can also be used to efficiently implement neural networks on vector computers. However, the runtime growth is much more strongly affected by the number of examples (effectively, the number of processors is small). Thus conventional vector machines such as a CRAY cannot achieve the reduction in the runtime growth possible with a SIMD machine containing a large number of processors. This analysis is overly simplistic since there are complex trade-offs between cycle time, vector pipeline length, and the number of processors. The bottom line is that given access to both vector machines and highly parallel SIMD/MIMD machines, we use vector machines for medium sized problems (generally less than 8,000 examples) and parallel machines for larger problems (from 8,000 examples to 10^6 examples).

The overall computational efficiency of a parallel computer can be high only as long as the associated communications overhead for the problem is low. Otherwise the parallel processors will spend all their time waiting for data. Using our mapping onto SIMD hardware, we will show that it is possible to avoid any communications overhead by mapping neural networks onto the parallel machine via the one example per processor approach. In our implementation on the CM-2 connection machine, the only communications required (with one minor exception) are global broadcast and local processor to processor communications. Since both of these operations occur in one clock cycle on the

connection machine, they provide no delay over a simple memory fetch. Hence the rate limiting step is how fast the parallel hardware of the connection machine can do floating point operations. In other words, our mapping turns the training of neural networks into a parallel algorithm which is limited by the computational rate of the hardware and not by communications overhead.

The mapping onto the CM-2 for the XOR architecture of fig 1 can be seen in fig 2-4. As can be seen in fig 2, the front-end computer contains all the network parameters and the SIMD processors contain all examples and temporary storage for the network. We can see the initial calculation of the hidden neuron (given in eqn 1.1) as it would be executed in parallel in Fig 3 - 4. The feedforward pass is initiated by broadcasting the neuron threshold from the front-end computer to all processors (see fig 3). The connection weight $W(I_1,H)$ is then broadcast to all processors with the instruction to multiply it by the local memory location containing the value of I_1 and add it to the local memory location containing the value of the hidden neuron (see fig 4). Since each SIMD processor contains one example, we get the number_of_examples instructions done per instruction cycle with no communications overhead. Similarly the calculations of eqn 1.2 - 1.4 occur using only global broadcast and local processor memory. It is clear that we are able to calculate the outputs for all the training examples for the XOR architecture or any arbitrary neural network, without communications delays, using only global broadcast communications. (The evaluation of recurrent networks is dependent upon how the back connections are to be evaluated. It is possible to do a purely parallel implementation for SIMD architectures using our mapping (see Pineda 1988 for the mathematical description). Other recurrent implementations may require a MIMD architecture as the required number of conditional operations would result in an extremely inefficient use of the SIMD processors per machine cycle.) The next step is to evaluate how the calculated outputs fit the known outputs. To do this the front-end issues an instruction to subtract the known output from the calculated output and square the result. Since all memory values are in local processor memory there is no communications overhead. The front-end then issues an instruction to calculate the summation over all processors of the squared differences. On the CM-2, the global summation instruction is provided by Thinking Machines Corporation and is optimized for their hardware. However, the global summation instruction has a runtime growth which is approximately O(log(n)) where n is the number of processors. Fig 5 diagrams how a O(log(n)) runtime growth could be achieved for a global summation. Since the run-time growth of this instruction is sub-linear with respect to the number of processors (or number of examples for our problem), it does not provide a significant decrease in the runtime performance. All other network calculations required for backpropagation occur in a similar manner and have no communications overhead except for that required by the global summation over processors.

Our mapping of one example per processor also allows networks with large numbers of parameters to be trained. We can see in Fig 6 that the worst-case memory growth for a fully interconnected recursive neural network is on the order $O(n^2)$; where n is the number of neurons. Since the network parameters (neuron thresholds and connections weights) are the same for all examples and hence for all the SIMD processors, it is makes sense to store them in one common block of memory and broadcast them to all other processors. This makes for an ideal mapping onto the CM-2 hardware as the $O(n^2)$ network parameters can be stored in the large virtual memory space of the front-end computer and broadcast to the SIMD processors. This frees the limited memo-

ry available to each CM processor to be used for the storage of the example input(s) and output(s) and intermediate values of the calculations.

It is the memory available to each SIMD processor which limits the size of the neural network, the size of individual training examples, and the amount of training data which can be evaluated. If the SIMD hardware has fast adjacent processor communications it is possible to efficiently merge the memory of adjacent processors to allow arbitrarily large training examples and neural networks to be evaluated. (It is possible to use a fast I/O memory device for the SIMD processors such as the CM-2 data vault to allow essentially unlimited network sizes and number of examples. However, we have not found it necessary to go to such extremes to train complex networks with even 10^5 to 10^6 examples.) This means that the memory map of individual SIMD processors will differ. However, we can merge the memory space of different processors by defining a special memory location in the memory map of all the SIMD processors to be a memory data bus. If we consider the example of fig 4 in evaluating an example of the XOR network, we would see a mapping onto the SIMD processor as seen in fig 7. If a value is required in the first processor which is in the memory of the second processor, it is copied to the common memory bus location and transferred via adjacent processor communications to the first processor. The arithmetic operation then proceeds on the first processor. Data shifts between adjacent processors on the CM-2 connection machine occur in one machine cycle (which is as much as 10^3 time faster than using the router communications). Thus we incur minimal communications overhead when using merged processors. However, merging processors introduces inefficiencies other than in moving data between the memory space of separate processors. In the case of merging two processors, only half of the SIMD processors can be active per computational instruction cycle. Similarly only 1/3 of the processors would be active if three processors were merged together and so forth. The advantage of the merged memory model is that arbitrarily large neural networks and data sets (which normally would be impossible to evaluate due to memory limitations) can be evaluated and in a manner transparent to the user. Since the number of processors which have to be merged to provide adequate memory storage is quite small in most cases, the performance loss is quite acceptable.

At Los Alamos, we have been using the mappings described above within the context of a neural network compiler since 1988. The details of the compiler are too numerous to present here. However, the compiler implements a paradigm familiar to any code developer as seen in fig 8. Aside from receiving the problem specification (the neural network architecture, initial parameter values, and training data) the compiler does all the remaining steps automatically for the destination machine including "writing" of the neural network program. Fig 9 shows how data moves through a complete neural network simulation. We can see that the neural network can be specified interactively by a graphical interface or by a machine generated file. The graphical interface allows a user to merge sub-networks "trained" to task into a large complex network. The sub-networks parameters may be locked to preserve the functionality of the sub-network or they may be "equivalenced" to force the unique sub-network parameters to maintain identical values during training. Of course the user may "unlock" the network parameters at will to allow "tweaking" of the parameter for the particular problem. The user may also automatically generate the network architecture so that the neural network may be modified so that various "pruning" or "growing" heuristics may be used. The training set data is presented to the compiler as either floating point or single bit boolean values. This allows

the compiler to minimize floating point operations for the individual training set and can provide significant increases in computational throughput. The data manipulation prior to the compiler can be a non-trivial task. We have had intermediate amounts of data exceeding 60 gigabytes which had to be pre-processed prior to presentation to the compiler.

The compiler takes the network/data specification and generates an intermediate language "program". This program then goes through a dependency analysis and is presented to a "compiler" which creates a relocatable instruction stream which is then passed to the loader linker. For the CM-2 the loader/linker creates an appropriate memory map (including merging multiple processors together) and creates a state machine instruction stream which is then executed once the data is loaded into the connection machine.

The compiler automatically calls the user specified optimization code as well as user functions specifying arbitrary neuron types. The user code on the front-end computer sees the compiler generated calculation of the forward pass, error propagation and calculation of the gradient (if possible) for the destination machine given the specified training set and neural network architecture. The user can then call these routines from their optimization code. Since algorithms for nonlinear or multidimensional optimization are quite complex and are either difficult or impossible to implement efficiently on a SIMD processor array, they are instead executed serially on the front-end computer. This allows the use of optimization algorithms such as conjugant gradient, powell's method, steepest descents or some other algorithm written in the users favorite language. This use of the front-end provides advantages on the connection machine. For example, the optimization code can "twiddle" network parameters with a 100 ns clock instead of the μs cycle time of the connection machine processors. In addition, some of the work done in the optimization code can be gotten "for free" due to the asynchronous operation of the front-end computer and the SIMD array of processors.

In summary, we have been able to exploit the gigaflop capabilities of the connection machine to train arbitrary feed forward neural networks on large, complex, and noisy data sets with examples on the order of hundreds of thousands to millions. We have done this with a neural network compiler which implements an extremely efficient mapping to SIMD architecture parallel computers. The mapping allows efficient use of the computational facilities of the parallel hardware with virtually no communications overhead. Arbitrarily large networks can be implemented by using the large virtual address space of the front end computer and by merging the memory space of adjacent SIMD processors together via fast local inter-processor communications. Thinking Machines Corporation has acknowledged that our implementation is considerably faster than other known implementations and that our implementation "has either constant time behavior or linear time dependence with respect to the number of training patterns, depending on the size of the connection machine used" (Singer 1990). Since the number of examples is the dominating factor in the runtime growth of training, our method allows the use of the CM-2 Connection Machine for real world problems of a complexity not possible using other computational hardware.

This work was done under the auspices of the U. S. Department of Energy and was partially funded by a grant from the National Institutes of Health (GM 40789-03). We express our gratitude for the hospitality of the Santa Fe Institute where part of the work was performed. We also acknowledge the help and support of Alan Lapedes who has been an integral part of the design and use of this work and without whom this work would have been impossible.

Fig 1: An XOR Neural Network

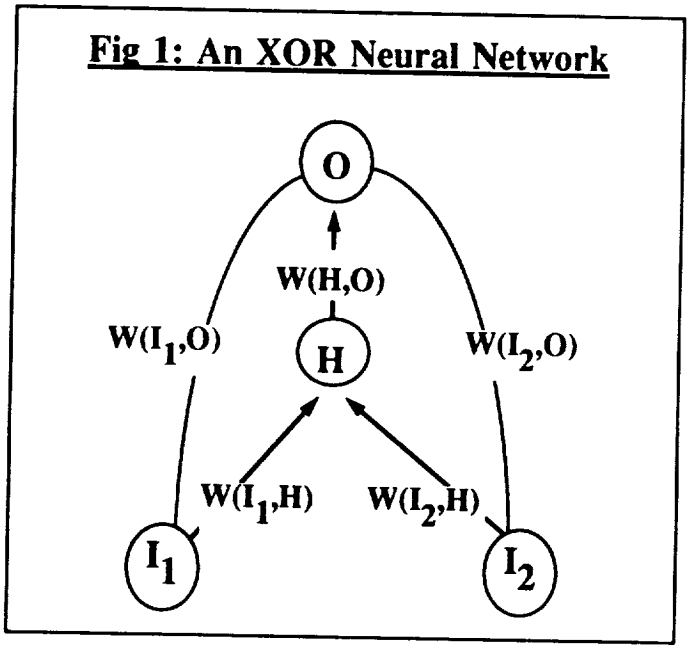


Fig 2: Mapping onto the Connection Machine

The Front End Computer (Generally a SUN) holds all variables for the neural network in *virtual memory*. This allows essentially unlimited neural network sizes and connectivity. The Front End also contains the energy minimization code written in a high level language like C. We generally use conjugant gradient although the user has complete flexibility to use his own code.

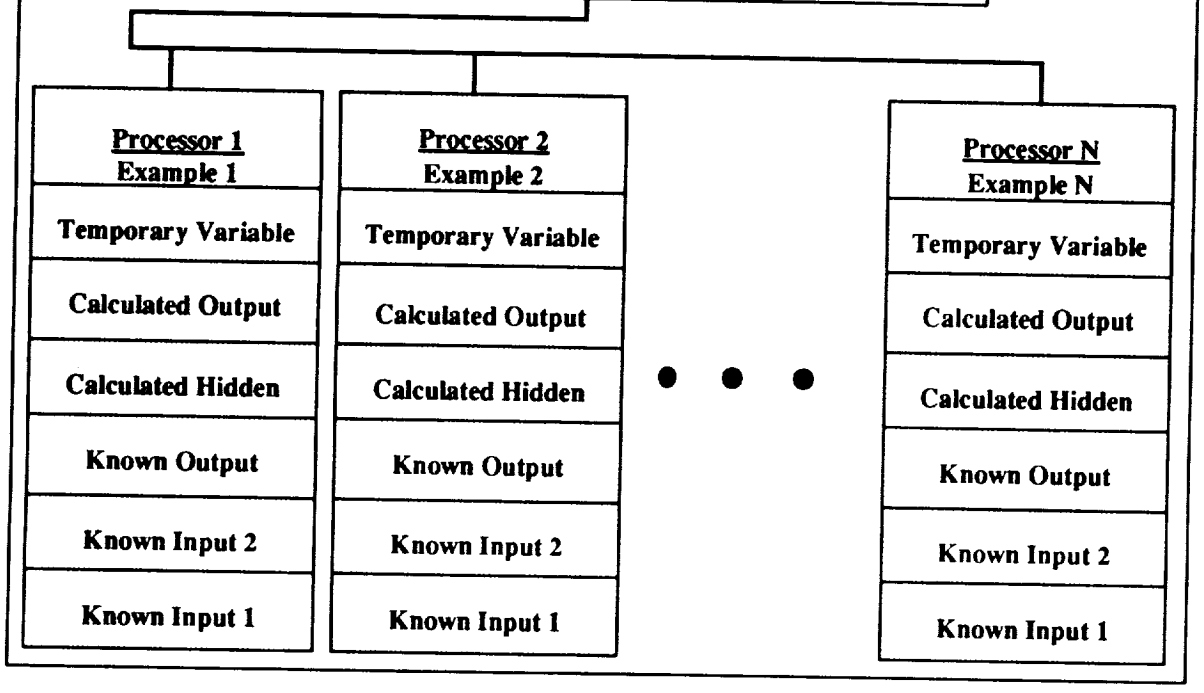


Fig 3: Example of a global broadcast

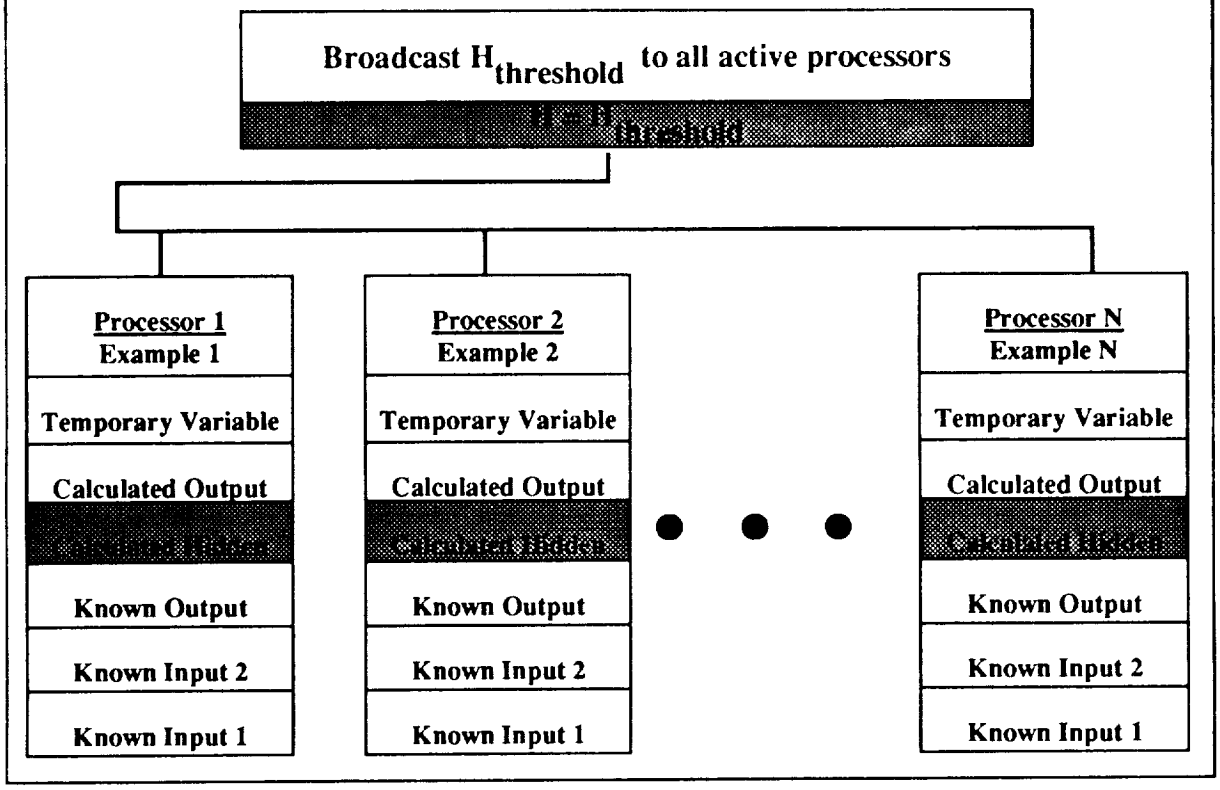


Fig 4: Calculation of Hidden Neuron for XOR Network. All operations are local except for the global broadcast of connections $W(I_1,H)$ and $W(I_2,H)$.

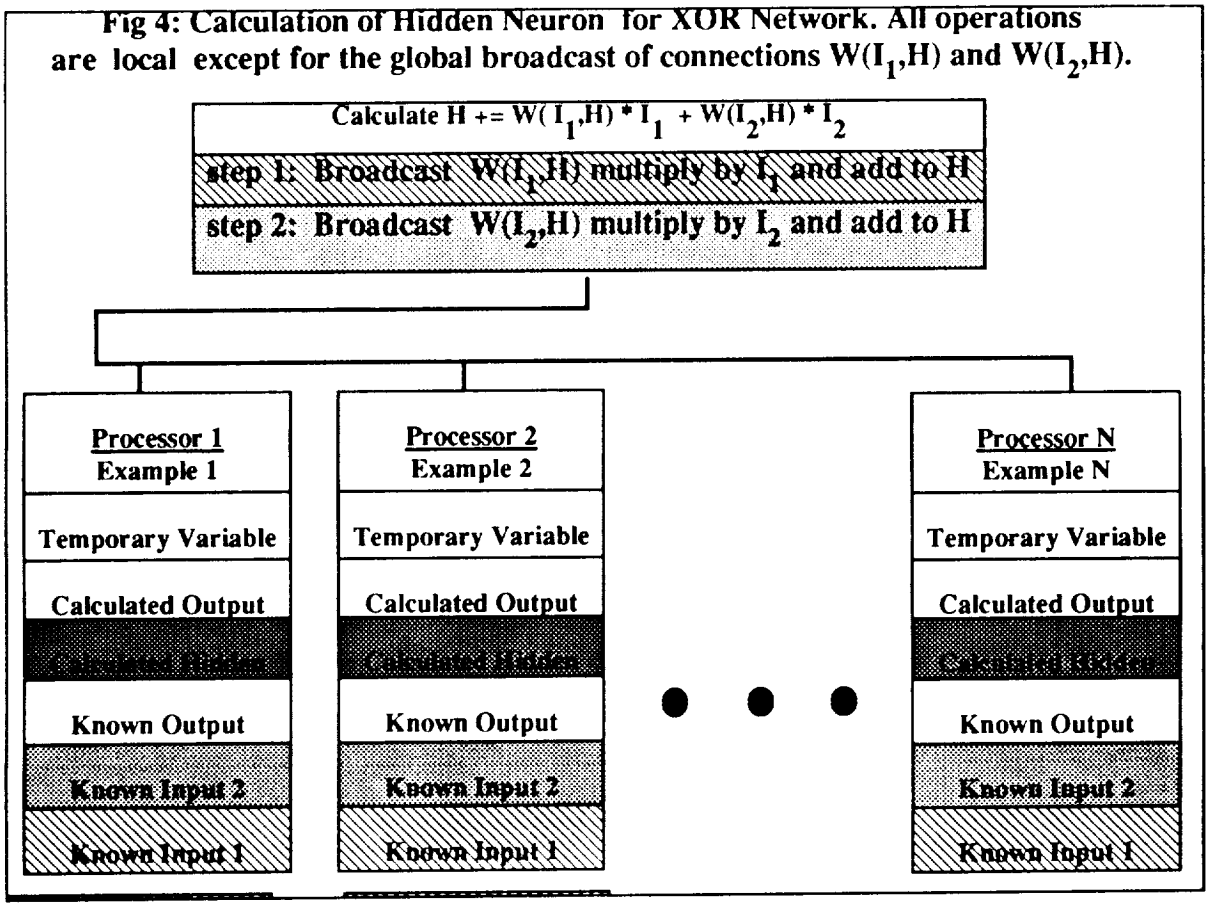


Fig 5: log(n) Runtime Growth of Global Summation

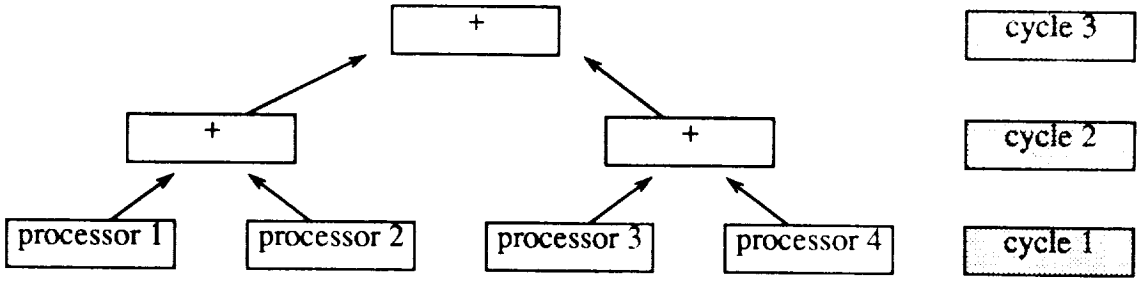
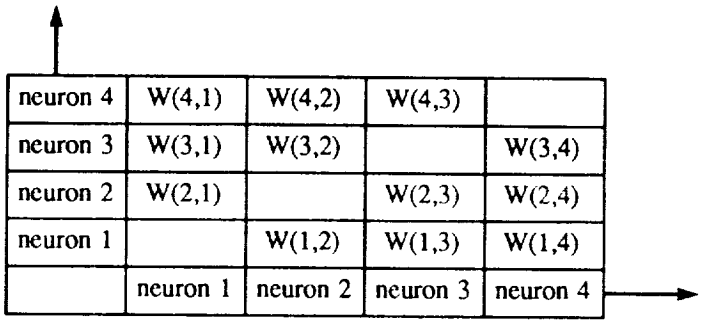


Fig 6: $O(n^2)$ Memory Growth for a Fully Interconnected Neural Network



neuron 4	W(4,1)	W(4,2)	W(4,3)	
neuron 3	W(3,1)	W(3,2)		W(3,4)
neuron 2	W(2,1)		W(2,3)	W(2,4)
neuron 1		W(1,2)	W(1,3)	W(1,4)
	neuron 1	neuron 2	neuron 3	neuron 4

Fig 7: Calculation of Hidden Neuron (eqn 1.2) for XOR Network. All operations are local except for the global broadcast of the connections $W(I_1,H)$ and $W(I_2,H)$ and local inter-processor communications.

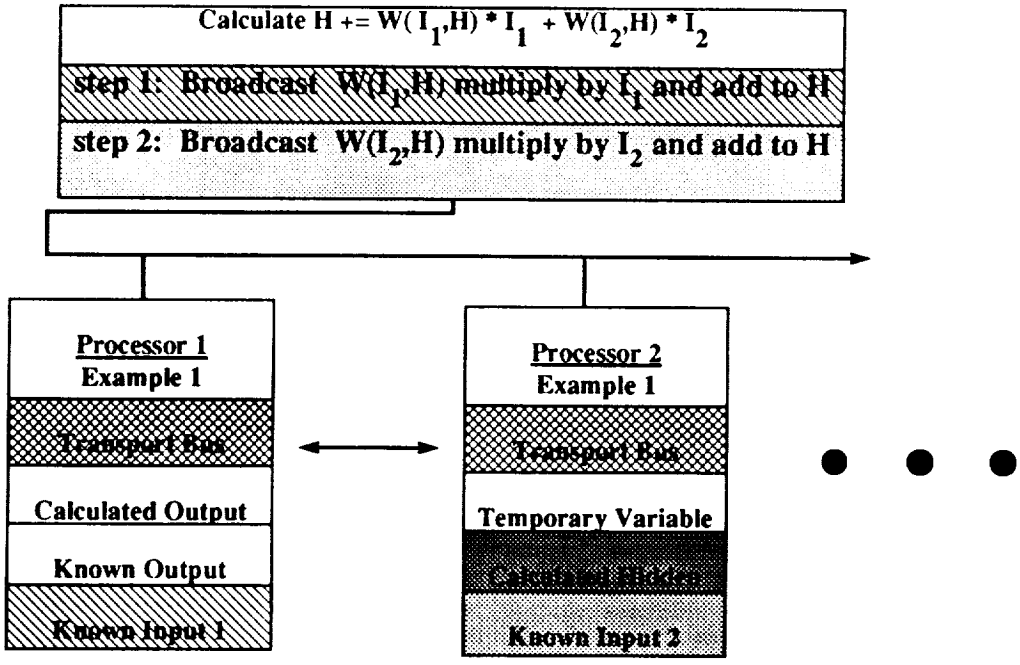


Fig 8: Compiler Paradigm

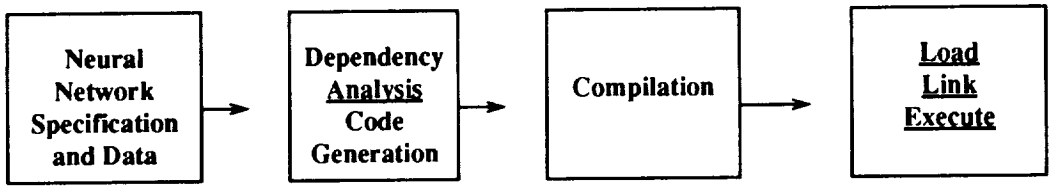
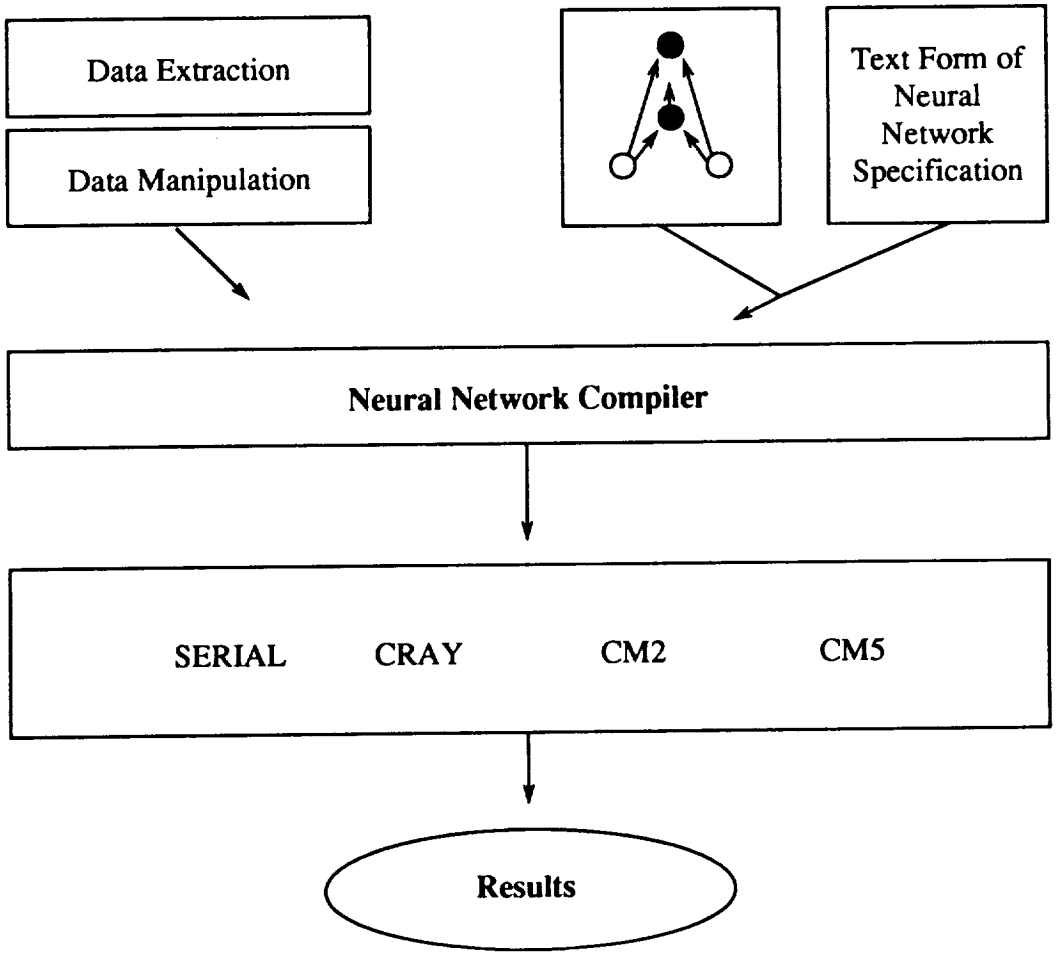


Fig 9: Block Diagram of Neural Network System



References

A. S. Lapedes, R. M. Farber, "*How Neural Networks Work*" Proceedings of the IEEE, Denver Conference on Neural Networks, 1987.

A. S. Lapedes, R. M. Farber, *"Non-linear Signal Processing using Neural Networks, Prediction and System Modeling "*, LANL technical report LA-UR-87-2662

A. S. Lapedes, C. Barnes, C. Burks, R. M. Farber, K. Sirotkin, "*Application of Neural Networks and Other Machine Learning Algorithms to DNA Sequence Analysis*" published in Computers and DNA, SFI Studies in the Sciences of Complexity, vol. VII (1989).

R. M. Farber, A. S. Lapedes, K. Sirotkin, *"Determination of Eukaryotic Protein Coding Regions Using Neural Networks and Information Theory"*, J. Mol. Biol. (1992) **226**, 471-479.

F. J. Pineda, *"Generalization of Backpropagation to Recurrent and Higher Order Neural Networks"*, Neural Information Processing Systems, Dana Z. Anderson editor, pg. 602-611.

W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *"Numerical Recipes in C"*, Cambridge University Press, 1988.

D. Rummelhart, J. McLelland, *"Parallel Distributed Processing"*, Vol. 1, M.I.T. Press, Cambridge, MA. (1986)

Singer A., *"Implementations of Artificial Neural Networks on the Connection Machine"*, Parallel Computing, **14**:305-315, 1990