JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 5, January 2008

Scalable Massively Parallel Artificial Neural Networks

Lyle N. Long* and Ankur Gupta†
The Pennsylvania State University, University Park, PA 16802

DOI: 10.2514/1.31026

Artificial Neural Networks (ANN) can be very effective for pattern recognition, function approximation, scientific classification, control, and the analysis of time series data; however they can require very large training times for large networks. Once the network is trained for a particular problem, however, it can produce results in a very short time. Traditional ANNs using back-propagation algorithm do not scale well as each neuron in one level is fully connected to each neuron in the previous level. In the present work only the neurons at the edges of the domains were involved in communication, in order to reduce the communication costs and maintain scalability. Ghost neurons were created at these processor boundaries for information communication. An object-oriented, massively-parallel ANN software package SPANN (Scalable Parallel Artificial Neural Network) has been developed and is described here. MPI was used to parallelize the C++ code. The back-propagation algorithm was used to train the network. In preliminary tests, the software was used to identify character sets consisting of 48 characters and with increasing resolutions. The code correctly identified all the characters when adequate training was used in the network. The training of a problem size with 2 billion neuron weights on an IBM BlueGene/L computer using 1000 dual PowerPC 440 processors required less than 30 minutes. Various comparisons in training time, forward propagation time, and error reduction were also made.

I. Introduction

ARTIFICIAL Neural Networks (ANN)[1–4] have been used for many complex tasks such as stock prediction and nonlinear function approximations. ANNs loosely mimic the human brain and consist of large networks of artificial neurons. These neurons have two or more input ports and one output port. Generally each input port is assigned a weight, and also a change of weight (delta-weight) to speed up the convergence. The output of a neuron is the weighted sum of the inputs. A transfer function is generally applied to the output depending on the desired behavior of the ANN. For example the sigmoid function is generally used when the output varies continuously but not linearly with input. The "learning" in ANNs occurs through iteratively modifying the input weights of each neuron, and often uses the back-propagation algorithm.[5–8] Training massive neural networks can be extremely time consuming however, since they do not scale well.

It is important to investigate scalable ANNs since supercomputers will soon have roughly the power of a human brain. The human brain has approximately 100 billion neurons[9] and each one has roughly 1,000 synapses. If we assume each synapse represents one byte, this amounts to roughly 10^{14} bytes of data (100 terabytes). The human

Received 14 March 2007; revision received 1 October 2007; accepted for publication 24 October 2007.

*AIAA Fellow, Distinguished Professor of Aerospace Engineering, Bioengineering, and Mathematics, The Pennsylvania State University, 229 Hammond Building, University Park, PA 16802. Email: lnl@psu.edu

†Graduate Research Assistant, Department of Computer Science and Engineering, The Pennsylvania State University, 234 Hammond Building, University Park, PA 16802. Email: azg139@psu.edu

brain is also capable of roughly 10^{14} to 10^{16} instructions per second.[9,10] This is far from being possible to simulate using traditional serial computers, but is not far from current massively parallel computers. There are many different types of ANNs (spiking,[11] multi-layer,[12] recurrent,[12] ART,[13] etc.), some are more biologically plausible than others. ANNs are important in attempts to model the brain, but they are also important for practical engineering applications. In modeling the human brain we should try to use biologically plausible networks[13–20] (e.g., spiking networks), but for engineering applications we simply want efficient software and algorithms.

Figure 1 shows some common devices and some living creatures in terms of memory and speed.[10] Currently, one of the largest parallel computers is the NASA SGI Columbia machine,[21] with 10,240 Itanium 2 processors (1.5 GHz) which has sustained $4 * 10^{13}$ floating point operations per second and has $2 * 10^{13}$ bytes of memory. This places it in a position as shown in the figure. IBM has developed two BlueGene computers that are even larger and faster than Columbia.[22] Current massively parallel computers do offer the hope of delivering significant "intelligence." In addition, Moore's law states that computer performance doubles roughly every 18 months, so computers could be 16 times larger and faster in 6 years, which means they might exceed the capacity of the human brain fairly soon. Even though computers are approaching the size and speed of mammal brains, there is the additional problem of machine learning. It would not be trivial to teach these machines to know what a human knows (but once this is accomplished it would be easy to transfer this knowledge to other computers!). Learning in humans is not necessarily fast either, since it typically takes about 18 (or more) years to train a human to function in modern society. In the next few paragraphs we discuss the implementation of parallel ANNs on computing machines and later provide some background on our parallel implementation of a back-propagation method on massively parallel machines.

The machines on which parallel Artificial Neural Networks (ANNs) have been implemented can be broadly divided into two categories: special purpose hardwired parallel processors and general purpose computers. Each of these implementations has their relative advantages and disadvantages. For example, hardwired parallel neural networks are low cost, high speed, and small size but they are not flexible once they are hard wired on silicon chips.

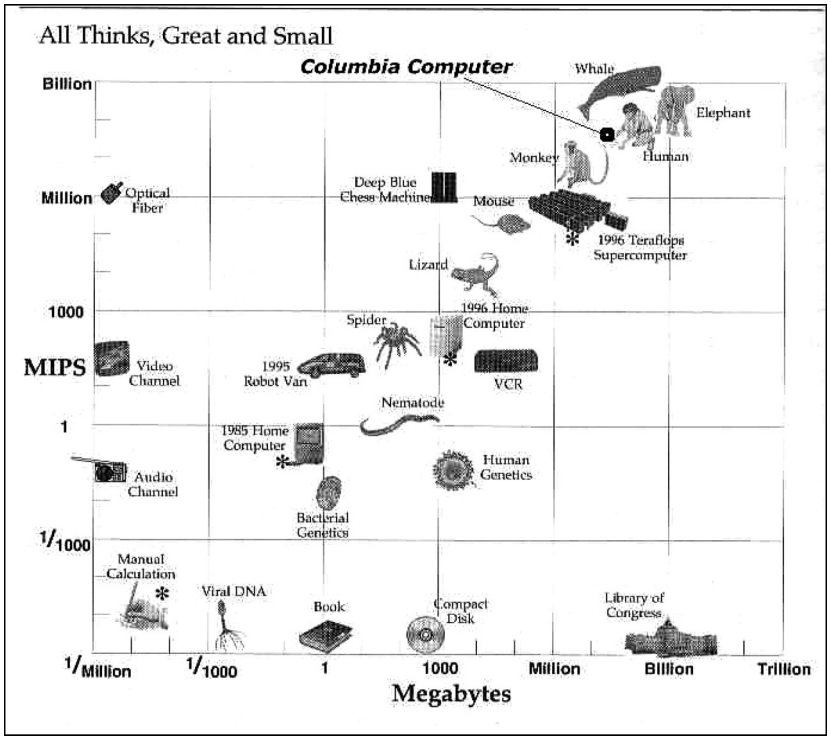

Fig. 1 Present status of technology (from Moravec[10]).

On the other hand, implementations on general purpose computers are relatively slow, but have the advantage of flexibility though often there is a compromise between speed and flexibility.

There have been many implementations of parallel training schemes for ANNs on massively parallel computers, with the Backpropagation (BP) scheme being the most extensively implemented.[23–26] It is also one of the most popular and computationally intensive training algorithms. Based on the way the parallel Back Propagation (BP) scheme is parallelized, it can be divided broadly into three categories: Network Partitioning, Pattern Partitioning, and Hybrid schemes. In network partitioning schemes, the neurons and weights of a Network are distributed among different processors. The neurons can be partitioned in two different ways: complete partitioning and vertical sectioning. In complete partitioning,[27 and 28] each processor gets one neuron whereas in vertical sectioning each processor gets some neurons from each layer.[27,29–34,35 and 36] Further, the weights can be partitioned in four different ways: complete partitioning, inset grouping, outset grouping, and checkerboarding. As suggested by the name, complete partitioning (also known as synapse level parallelism) assigns a single weight to each processor. Though this offers the finest level of parallelism, the communication costs are quite large. Inset grouping schemes group together the weights on the incoming edges of a specific neuron to form a set. This inset set of a neuron is assigned to the processor which possesses the neuron. Thus the need for communicating the values required to compute the activation during forward propagation is eliminated.[27,30,31 and 33] The outset grouping approach groups together the weights on the outgoing edges of a specific neuron in a set. This set is allocated to the processor possessing the neuron. Thus this grouping scheme eliminates the need for communication of error vectors which are needed during backward propagation.[37] Both inset and outset grouping schemes can be used together to improve the overall efficiency, but these schemes duplicate each weight on two processors, increasing the work during weight updates.[29 and 32] The checkerboard method partitions the weights by grouping the rows and columns of the weight matrix. It has been used in mesh architectures,[37] systolic arrays,[38] and hypercubes.[39]

In pattern partitioning, the whole network is replicated on each processor. Here the pattern set is divided equally among the processors. Each processor carries out its own training for the given set of local patterns and also stores the weight changes for updating weights. Finally the processors communicate to update the weights according to the communicated weight changes. This scheme is generally preferred when there is a large pattern set and on machines having efficient broadcast operation.[35,40–42] The computations in different layers of the network can also be pipelined,[27,33] that is, while one pattern is being processed for some layer, a different pattern can be processed for preceding layer.

Hybrid schemes can combine pattern partitioning and network partitioning. For example, pipelining with vertical sectioning,[27,33] vertical sectioning with simple pattern partitioning,[31] and pattern partitioning with checkerboarding.[39]

There have been several implementations of parallel ANNs in the past on specific parallel architectures. For example, on ring systolic arrays,[43] on SIMD arrays,[44–47] on MasPar MP-1,[48] on Torus,[49] and on Hypercube architectures.[39,50–52] These algorithms were systematically derived to map on to a specific parallel architecture, giving them the highest efficiency on that architecture. Most of these implementations date back to more than a decade ago, with few implementations on current supercomputers. Also, most of these previous efforts have concentrated more on implementing the traditional fully-connected ANN on a parallel architecture. In a fully connected ANN, all the neurons in a layer feed into all the neurons of the next layer. The problem with fully connected ANNs when implemented on parallel machines is their high communication costs.

In this paper we present a software implementation of the ANN on massively parallel machines, which uses the traditional BP algorithm for training. In the approach described here the neural network is constructed in a columnar manner, which loosely mimics the connections and layers in the human brain. Our software is scalable for both the forward propagation and the training phase. Moreover, it has several characteristics of the basic neural network computation model i.e. it has simple processing units (neurons), small local memory is required per neuron, and it is highly parallel. Our ANN is not fully connected except for the first layer, resulting in very low communication costs. In our network, all the inputs feed into all the neurons in the first layer. For the time being a very simple test case has been chosen to demonstrate the effectiveness of the parallel implementation. In the future we plan to train the software for more complex cases such as image recognition.

The work described here is *not* an attempt to accurately model the brain, it is simply an attempt to implement scalable ANNs on massively parallel computers. This may be useful for both a first step towards software models of networks in the human brain and also for developing useful engineering tools (e.g., for pattern recognition).

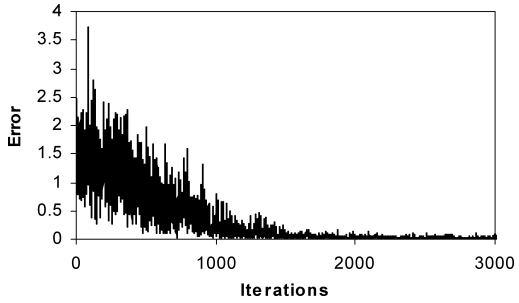

Fig. 2 Absolute error against number of training iterations.

The connections and flow of information in the brain are, of course, much more complex than a simple feed-forward network.

Large ANN's require a large amount of memory and computational time to train. Figure 2 shows the convergence of a serial ANN having 1000 neurons (using one hidden layer) for a period of 2,000 iterations. It uses the back-propagation algorithm[5,6,53] to adjust the weights. The time required to train this network (per iteration) was about 0.05 seconds using a 1.5 GHz Pentium 4 processor. The memory required for this network was roughly 1 MB (each weight was stored as a 4-byte floating point number). For comparison purposes, the time taken per iteration for a network having 30,000 neurons was about 50 seconds. Also, the amount of memory required by such a network was about 0.9 GB. Thus, serial ANNs have severe limitations in terms of memory and training time required. And it is well known that they do not scale well.[12]

II. Back-propagation Algorithm for Neural Networks

This section presents briefly the back-propagation algorithm for multi-layer neural networks. Figure 3 shows how a neuron calculates its activity using the inputs. A neuron in the output layer computes its activity using Eqs. (1) and (2). Here, $f(x)$ is the scaling function which is generally the sigmoid function. A unipolar sigmoid function varies from 0 to 1, and is calculated by Eq. (3). A bipolar sigmoid function varies from -1 to 1, and is given by Eq. (4). The gain used here is 1.0. After calculating the activity of the neurons, the network computes its error, given by Eq. (5).

$$x_j = \sum y_i W_{ij} \tag{1}$$

$$y_j = f(x_j) \tag{2}$$

$$f(x) = \frac{1}{(1 + e^{-x})} \tag{3}$$

$$f(x) = \frac{2}{1 + e^{-x_j}} - 1 \tag{4}$$

$$E = 0.5 \sum (y_i - d_i)^2 \tag{5}$$

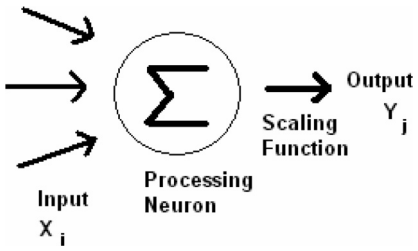

Fig. 3 A processing unit or neuron.

Back-propagation starts at the output layer with Eqs. (6) and (7). The learning rate, η, applies a scaling factor to the adjustment to the old weight. If the factor is set to a large value, then the neural network may learn quickly, provided that the variation in the training set is not large. Usually, it is better to set the factor to a small value initially and later increase it if the learning seems slow. We typically use a value of $\eta = 0.1$.

The momentum factor α basically allows a change to the weights to persist for a number of adjustment cycles. This can improve the learning in some situations, by helping to smooth out unusual conditions in the training set. We typically use a value of $\alpha = 0.5$.

$$w_{ij} = w'_{ij} + (1 - \alpha)\eta e_j x_i + \alpha(w'_{ij} - w''_{ij}) \tag{6}$$

$$e_j = y_j(1 - y_j)(d_j - y_j) \tag{7}$$

Once the error terms are computed and weights are adjusted for the output layer, the values are recorded and the previous layer is adjusted. The same weight adjustment process, determined by Eq. (6), is followed, but the error term is computed by Eq. (8).

$$e_j = y_j(1 - y_j) \sum e_k w'_{jk} \tag{8}$$

In contrast to Eq. (7), in this equation, the difference between the desired output and the actual output is replaced by the sum of the error terms for each neuron, k, in the layer immediately succeeding the layer being processed times the respective pre-adjustment weights. There is no exact criteria for how many weights are required for a neural network, but there are some rules of thumb. For example,[12] for a network with N_i input units, a lower bound for the number of hidden units would be $\mathrm{Log}_2(N_i)$, other sources[53] have suggested using the average of the number of inputs and number of outputs.

A single feed-forward and back-propagation operation requires $\mathrm{O}(N)$ operations,[12] where N is the number of adjustable weights. There is no exact formula for the amount of training required for a neural network, it depends on the problem and the network architecture. Barron[54] shows that the error in a feed-forward ANN can be $\mathrm{O}(1/N)$. There is also a rule of thumb[12] that states that the size of the training set, T, should satisfy the following relation:

$$T = \mathrm{O}(N/\varepsilon) \tag{9}$$

Where ε denotes the fraction of classification errors permitted. This means that for a case where 10% error rate is allowed, the number of training examples should be 10 times the number of weights. In the tests performed so far here, the networks appear to require far fewer iterations than the above might indicate. It is important to point out though that the feed-forward operation is $\mathrm{O}(N)$, so if we can use large parallel computers to train neural networks then we can copy the weights to a serial computer and put the network to use (assuming the serial computer uses the same network structure).

III. Software

We have developed ANN software, called SPANN (Scalable Parallel Artificial Neural Network), which runs on massively parallel computers and uses the back-propagation training algorithm. An object oriented (C++)[55,56] approach is used to model the neural network. The software implements several C++ objects, such as Neuron, Layer, and Network. The software can have any number of layers and each layer can have any number of neurons. Each neuron stores information such as weights, delta-weight for momentum, error and its output. Figure 4 shows a UML type diagram of the data and methods associated with each of the classes (objects). As one would expect, the Neuron object has output, error, and an array of weights. The Layer objects have arrays of Neurons, and the Network objects have arrays of layers. The Message Passing Interface (MPI) library[57,58] is used for parallelization. Neurons in each layer are distributed roughly equally to all the processors, however all the inputs are fed to each processor. Each layer in a processor has two ghost neurons at the boundaries. Error, outputs, or weights of the boundary neurons are communicated and are stored in the ghost neurons in the neighboring processors,

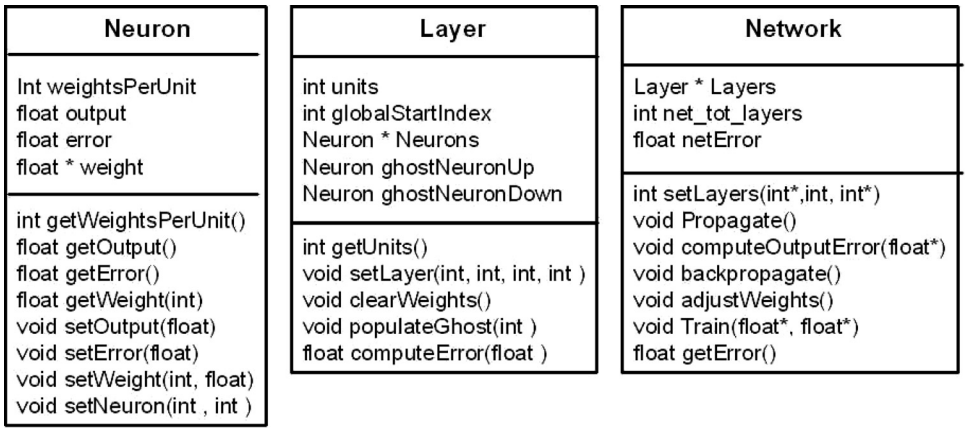

Fig. 4 A UML type diagram of the important classes, data, and methods used in the code.

whenever required. Section IV explains in detail the implementation of ghost neurons. The main MPI routines used are:

- MPI_Bcast to broadcast the input variables to other processors
- MPI_Allreduce to get the errors at all processors when required
- MPI_Gather, MPI_Gatherv to get the final combined output onto the master processor
- MPI_SendRecv to send and receive data in ghost nodes
- MPI_Barrier to synchronize processors

An important issue in the parallel implementation of ANNs is the communication load due to the nature of the feed-forward and back-propagation algorithms traditionally used. Conventional feed-forward algorithms have every neuron connected to all the neurons in the previous layer. This approach results in huge communication costs which would affect the scalability, and it is not biologically plausible either.

The human neocortex has six layers and the layers are not fully connected. In our approach, only values at the boundaries of the processor domains are communicated thus reducing the communication cost significantly. Figure 5 shows the communication involved in a layer for a 3-processor (P0, P1, and P2) simulation, with 12 inputs, 2 hidden layers with 9 hidden units each and 6 outputs. It also shows an example of how the ghost neurons are used at the edges of the processor domains. The figure shows all the inputs connected to all the neurons in the first hidden layer of processor P0, similar connections feed into all the other processors. Each of these connections stores the value of the weight and delta-weight for momentum. The neurons are distributed among the processors such that each processor has the same load. The network is trained using the back-propagation algorithm.

In order to better understand the parallel ANN, it might be useful to discuss how the number of weights varies with number of neurons. First, for a traditional ANN with 1 hidden layer, the total number of weights in the network can be computed from:

$$\text{Total Weights}_{\text{serial}} = (N_{\text{in}} + N_{\text{out}})N_{\text{hidden}}$$

where N_{in}, N_{out}, and N_{hidden} are the number of neurons in the input, output, and hidden layer, respectively. The CPU time for one forward/backward propagation (for the serial case) varies linearly with the total number of weights. For the parallel ANN used here, the total number of weights varies according to:

$$\text{Total Weights}_{\text{parallel}} = N_{\text{in}}N_{\text{hidden}} + ((L-3)N_{\text{hidden}} + N_{\text{out}})\frac{N_{\text{hidden}}}{N_{\text{proc}}} + 4(L-1)N_{\text{proc}}$$

Where L is the number of layers (including input and output) and N_{proc} is the number of processors being used. All of the inputs (N_{in}) are fed into every processor, but each processor has $N_{\text{hidden}}/N_{\text{proc}}$ hidden neurons/layer and

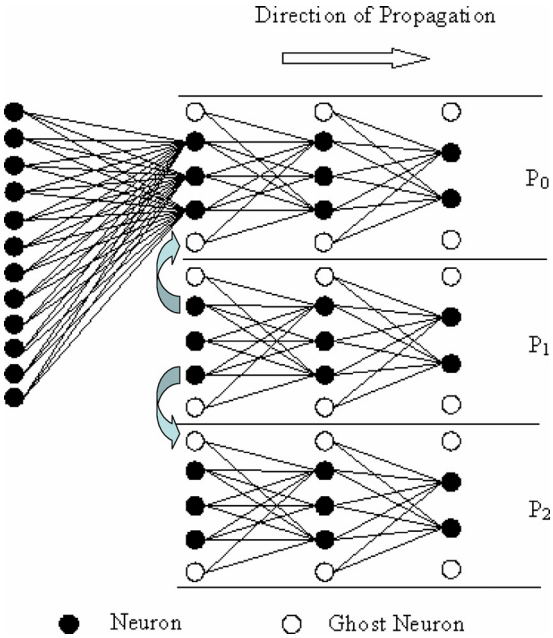

Fig. 5 Neuron connections and communication in a three processor system.

$N_{\text{out}}/N_{\text{proc}}$ output neurons. The number of weights per processor is given by:

$$\text{Weights/Processor}_{\text{parallel}} = \frac{N_{\text{in}} N_{\text{hidden}}}{N_{\text{proc}}} + ((L-3)N_{\text{hidden}} + N_{\text{out}})\frac{N_{\text{hidden}}}{N_{\text{proc}}^2} + 4(L-1)$$

For example, for the serial ANN, if $N_{\text{in}} = 200$, $N_{\text{out}} = 48$, and $N_{\text{hidden}} = 125$ then the total number of weights is 31,000. As an example for the parallel case, if $N_{in} = 200$, $N_{\text{out}} = 48$, $N_{\text{hidden}} = 120$, $L = 6$, and $N_{\text{proc}} = 8$ then the total number of weights would be 30,280 and each processor would have 3785 weights. This also shows how the two different networks could be configured to have roughly the same number of total weights. The CPU time on the parallel computer will vary linearly with the weights/processor, plus there will be inter-processor communication costs.

In order to test the software, a character set, shown in Fig. 6, consisting of 48 characters was used.[59] Each character is represented by (at a minimum) an array of 3×5 pixels. Each pixel has a value of 1 or -1 according to whether it

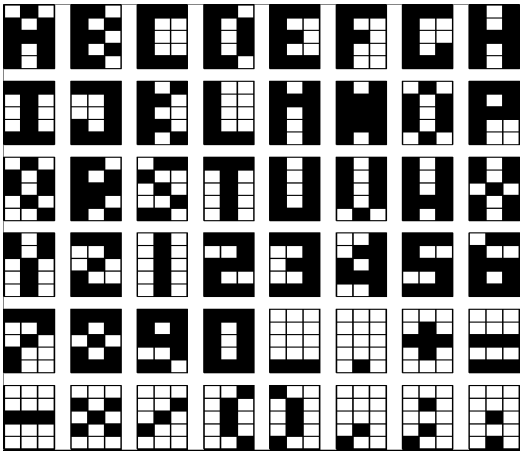

Fig. 6 Character set used for testing.

is on or off. Thus, input for each character is represented by a vector of (at a minimum) 15 real numbers. The output for each character is a vector of length 48, with all but one value being zeros. The size of the input could be increased by increasing the resolution of the characters. Thus, for a resolution of R, each pixel in the original character set was replaced by R $\times$ R identical pixels. For the serial case, the output remained the same for any resolution. As the parallel code required at least two neurons per processor in a single layer, the output was padded with zeros whenever required. Though the character set is a simple classification problem, it allows for increasing the problem size by increasing the resolution.

IV. Implementation of Ghost Neurons

This section discusses implementation of ghost neurons at the boundaries of the domains in detail. As shown in Fig. 5, ghost neurons are used to communicate information from the "real" neuron of a layer at a processor's boundary to the ghost neuron at the neighboring processor's boundary so that this information can be processed by the "real neuron" of the next layer. There are three types of information that needs to be passed on: neuronal output, error, and weights and "delta weight" for implementing momentum calculations. The timings of these boundary communications are important. They need to be passed on only when needed to avoid unnecessary computations. In this implementation particular attention was paid so that no unnecessary computation is done at any step. This resulted in very good scalability even when the problem size was increased linearly with processors up to 1000 processors. We believe that the code will be scalable for even larger problem sizes or more number of processors. The next paragraph discusses the issue of what information needs to be passes at which stage.

For the forward propagation step only the neuron outputs and weights in the ghost neuron need to be updated when a layer finishes its computations. One pass in the learning or training stage involves four steps in the flowing order: First a forward propagation pass as described above updates the outputs, weights and delta weights. Second the errors in the output layer are computed using the target values. Next the errors are propagated back starting from the last layer to the first. This step involves communication of the errors to/from the ghost neurons. Finally, the weights are adjusted according to the output, errors, weights, and delta weights. The ghost neurons have already communicated the required values in the final step during the first three steps. One additional point needs to be mentioned. So the ghost neurons can be thought of as agents which pass information from say neurons present in a particular processor to the neighboring processors. So having more layers in the network helps information to be propagated to all the processors. In the present case having five to six layers helps this cause.

V. Results

Several serial runs were made for comparison with the parallel code. The serial runs were performed on an IBM P640 RS6000 Server.[60] Table 1 shows the training time required for the serial case as the resolution of the characters is increased. Most of the characters were recognized correctly. Figure 7 shows the percentage of the characters correctly recognized compared to the training iterations for different values of total weights in the network. It is observed that there seems to be an optimal value of weights for the network, which gives the most correct character recognitions for lesser number of iterations. This is due to the fact that the network is unable to "learn" due to underfitting when there are less than the required numbers of weights. Also, when the number of weights is much larger than the required, the network tends to remember rather than "learn", due to over fitting. It should be noted that what seems to be true for this problem might not hold for other neural nets or other applications, and generalization can only be made after proper verification. The parallel code has been tested using roughly the same number of weights as in the serial case.

Table 1 Training time required for the serial ANN

Resolution	Total weights	Iterations	Training time (sec)
1	1890	37,824	5.4
2	3240	64,800	13.1
4	8640	172,800	76
5	12,690	253,776	157
8	30,240	604,800	842
9	37,890	757,776	1313

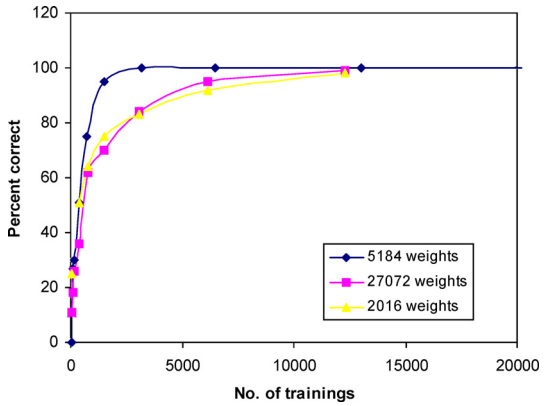

Fig. 7 Percentage of correct characters recognized against number of training iterations for different weights in a serial ANN.

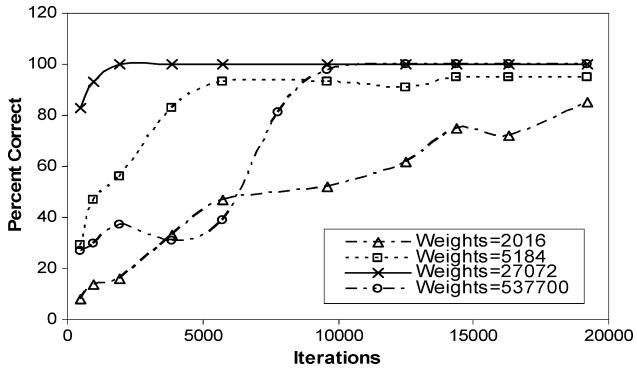

Fig. 8 Percentage of correct characters recognized codes against number of training iterations for different weights in parallel ANN. Number of processors was 4.

These results were obtained from a 160-processor Beowulf computing cluster.[61] The learning rate and momentum factors were set to 0.5 for both the serial and the parallel cases. Figure 8 shows the percentage of the characters correctly recognized against the training iterations for different values of total weights in the network.

More results, comparing the convergence of the serial and parallel codes, are shown in Fig. 9. For this case, both networks had roughly 12,000 weights and the number of inputs was 135. The serial case used a 135-70-48 network, with a learning rate of 0.1 and momentum factor of 0.5. The parallel case used 8 processors and a 135-76-76-76-48

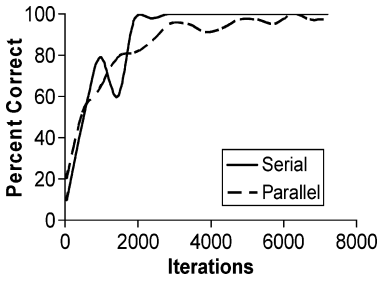

Fig. 9 Convergence of serial and parallel.

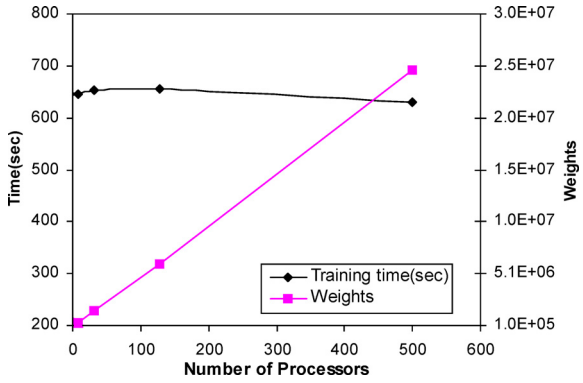

Fig. 10 Training time taken as the problem size was increased linearly with the number of processors.

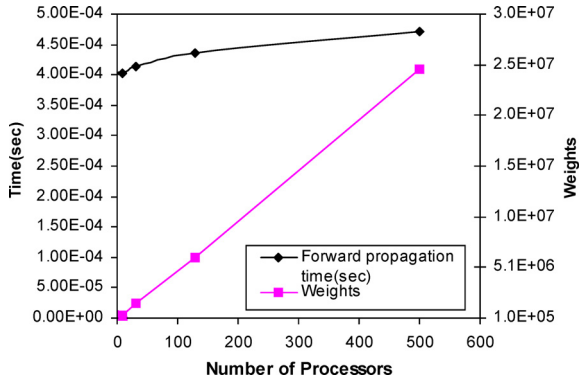

Fig. 11 Time taken for one forward propagation as the problem size increases linearly with the number of processors.

network, with a learning rate of 0.7 and momentum factor of 0.7. This figure shows that even though the network structures of the serial and parallel networks are quite different, it is possible to achieve similar convergence results.

Figures 10, 11, and 12 show the performance results of SPANN on massively parallel computers. In order to maintain parallel efficiency, the number of neurons per layer was scaled linearly with the number of processors. The runs for Figs. 10 and 11 were performed on the NASA SGI Columbia computer.[21]

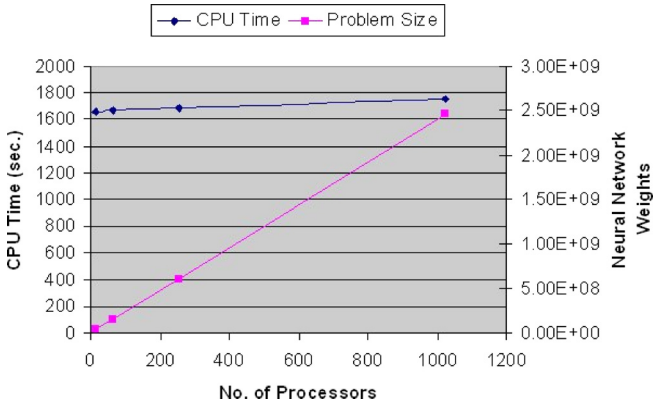

Fig. 12 Training time taken as the problem size was increased linearly with the number of processors.

Table 2 SPANN results for scaling number of inputs of neural network with number of processors.

Processors	Inputs	Neurons	Neurons Per Hidden Layer	Weights	Percent Correct	Memory Used (GB)	CPU Time(sec.)
16	37,500	1584	256	9,613,376	100 %	0.08	246
64	150,000	6272	1024	153,652,480	100 %	1.20	2489
500	600,000	25,000	4000	2,400,106,384	89 %	19.0	6238

The runs for Fig. 12 were performed on an IBM Bluegene.[22] Figures 10 and 12 show that the training time is essentially constant when the number of neurons is scaled linearly with the number of processors. All of these runs used the same number of training intervals. In practice, the larger networks may require more training for the same level of accuracy. Figure 11 shows that the time taken for a single forward propagation step also remains essentially constant as the number of neurons are scaled linearly with processors. For 64,000 neurons per layer (using 6 layers) on 500 processors, the total memory required was about 0.2GB/processor (each weight was stored as a 4-byte floating point number). The memory required by a single neuron for this case was about 1KB. The largest case (on 500 processors) used more than 24 million neuron weights.

In order to show the scalability of the code in another manner, Table 2 shows results for cases where the number of inputs was scaled with the number of processors. All of these cases used six hidden layers, and were run on NASA's Columbia computer (1.5 GHz Itanium2 processors) using the Intel C++ compiler (vers. 7.1) with the options: -ftz -ipo -IPF_fltacc -IPF_fma –O3. Note that Columbia has 1.9GB memory/processor, so these cases were using a small fraction of the total memory. A single feed-forward operation on the largest case required only 0.25 seconds.

VI. Conclusions

Traditional ANN's do not scale well. For massively parallel ANNs it is not practical to have the neurons at each level connected to all the neurons on the previous level, since this would require an enormous amount of inter-processor communication. In the approach described here the neural network is constructed in a columnar manner, which loosely mimics the connections and layers in the human brain. In the future we could have layers of neurons in a two-dimensional grid with communication taking place between arrays of neurons. We could also have layers of neurons in a three-dimensional block with communication taking place between the surface neurons of the blocks. The software developed here is scalable and permits the use of billions of weights or synapses. The approach used here also allows an ANN to be trained on a massively parallel computer and then used on smaller serial computers. In the future we plan to train the software for more complex cases such as image recognition.

Acknowledgments

The authors would like to thank the NASA Advanced Supercomputing Division (NAS) for the use of the Columbia supercomputer,[21] and Argonne National Labs for the use of the IBM Bluegene/L supercomputer.[22] We'd also like to thank the Office of Naval Research for funding this work (Grant No. N00014-05-1-0844).

References

[1]Seiffert, U., "Artificial Neural Networks on Massively Parallel Computer Hardware," *European Symposium on Artificial Neural Networks,* Bruges (Belgium), 24–26 April 2002.

[2]Takefuji, Y., *Neural Network Parallel Computing*, Kluwer Academic Publishers, Norwell, MA, 1999.

[3]Hassoun, M. H., *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, 1995.

[4]White, D. A., and Sofge, D. A., *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, Van Nostrand Reinhold, 1 June, 1992.

[5]Werbos, P., *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, John Wiley & Sons, New York, 1994.

[6]Werbos, P. J., *Backpropagation*: *Basics and New Development*: *The Handbook of Brain Theory and Neural Networks*, 1st ed., MIT Press, 1995.

[7]Mitchell, T. M., *Machine Learning*, McGraw-Hill, NY, 1997.

[8]Rumelhart, D. E., and McClelland, J. L., *Parallel Distributed Processing*, MIT Press, 1986.

[9]Kurzweil, R., *The Age of Spiritual Machines*, Penguin, NY, 1999.

[10]Moravec, H., *Robot: Mere Machine to Transcendent Mind*, Oxford University Press, November 1998.

[11]Gerstner, W., and Kistler, W. M., *Spiking Neuron Models*, Cambridge University Press, Cambridge, 2002.

[12]Haykin, S., *Neural Networks: A Comprehensive Foundation*, 2nd ed., Prentice-Hall, 1999.

[13]Carpenter, G. A., and Grossberg, S., "Adaptive Resonance Theory," *The Handbook of Brain Theory and Neural Networks*," 2nd ed., Cambridge, MA, MIT Press, 2003, pp. 87–90,

[14]LeDoux, J., "*Synaptic Self*," Penguin, New York, 2002.

[15]Hawkins, J., and Blakeslee, S., *On Intelligence*, Times Books, New York, 2004.

[16]Taylor, J., "Neural Networks of the Brain: Their Analysis and Relation to Brain Images," *Proceedings of the IEEE International Joint Conference on Neural Networks*, Montreal, 2005.

[17]Dean, T., "A Computational Model of the Cerebral Cortex", *American Association of Artificial Intelligence Conference*, Pittsburgh, 2005

[18]Mountcastle, V. B., "Introduction to the Special Issue on Computation in Cortical Columns," *Cerebral Cortex*, Vol. 13, No. 1, 2003.

[19]Mumford, D., "On the Computational Architecture of the Neocortex I: The Role of the Thalamo-cortical Loop," *Biological Cybernetics*, Vol. 65, 1991.
doi: 10.1007/BF00202389

[20]Mumford, D., "On the Computational Architecture of the Neocortex II: The Role of Cortico-cortical Loops", *Biological Cybernetics*, Vol. 66, 1992.
doi: 10.1007/BF00198477

[21]http://www.nas.nasa.gov/Resources/Systems/columbia.html

[22]http://www.research.ibm.com/bluegene/

[23]Sejnowski, T. J., and Rosenburg, C. R., "Nettalk: A Parallel Network that Learns to Read Aloud," Tech. Rep. JHU/EECS-86/01, Department of Electrical Engineering and Computer Science, Johns Hopkins University, Baltimore, MD, USA, 1986.

[24]Shekhar, S., and Amin, M. B., "Generalization Performance of Feed-forward Neural Networks," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, April 1992, pp. 177–185.
doi: 10.1109/69.134256

[25]Dutta, S., and Shekhar, S., "Bond-rating: A Non-conservative Application of Neural Networks," *IEEE International Conference on Neural Networks*, 1988, pp. 11-443–11-450.

[26]White, H., "Economic Prediction Using Neural Networks: The Case of IBM Daily Stock Returns," *IEEE International Conference on Neural Networks*, 1988, pp. 451–458.

[27]Allen, W., and Saha, A., "Parallel Neural Network Simulation Using Backpropagation for the Es-kit Environment," *Proceedings of the1989 Conference Hypercubes: Concurrent Computers and Applications*, 1989, pp. 1097–1102.

[28]Blelloch, G., and Rosenberg, C. R., "Network Learning on the Connection Machine," Tech. Rep., MIT, Cambridge, MA, USA, Nov. 1986.

[29]Yoon, H., and Nang, J. H., "Multilayer Neural Networks on Distributed Memory Multiprocessors," *IEEE International Conference on Neural Networks*, 1991, pp. 669–672.

[30]Zhang, X., "An Efficient Implementation of the Backpropagation Algorithm on the Connection Machine CM-2," Tech. Rep. RL89-1, Thinking Machines Corp., Aug. 1989.

[31]Zhang, X., and McKenna, M., "The Backpropagation Algorithm on Grid and Hypercube Architectures," Tech. Rep. RL90-9, Thinking Machines Corp., 1990.

[32]Baiardi, F., Mussard, R., Serr, R., and Valastro, G., "Feed-forward Neural Networks on Message Passing Parallel Computers," *Proceedings of the Second Italian Workshop on Parallel Architectures and Neural Networks*, edited by E. R. Caianielo, Singapore, 1990.

[33]Marchesi, M., Orlandi, G., Piazza, F., and Uncini, A., "Linear Array Architecture Implementing the Backpropagation Neural Network," *Proceedings of the Second Italian Workshop on Parallel Architectures and Neural Networks*, edited by E. R. Caianielo, Singapore, 1990.

[34]Newhall, D. S., and Horvath, J. C., "Analysis of Text Using a Neural Network: A Hypercube Implementation," in *Proceedings of the 1989 Conference Hypercubes, Concurrent Computers and Applications*, 1989, pp. 1119–1122.

[35]Joe, K., Mori, Y., and Miyake, S., "Simulation of a Large-scale Neural Network on a Parallel Computer," *Proceedings of the 1989 Conference Hypercubes, Concurrent Computers and Applications*, 1989, pp. 1111–1118.

[36]Pomerleau, D. A., Gusciora, G. L., Kung, H. T., and Touretzky, D. S., "Neural Network Simulation at Warp Speed How We Got 17 Million Connections per Second," *IEEE Second International Conference on Neural Networks*, 1988, pp. 143–150.

[37]Petrowski, A., Personnaz, L., Dreyfus, G., and Girault, C., "Parallel Implementations of Neural Network Simulations," *Proceedings of the First European Workshop on Hypercube and Distributed Computing*, Amsterdam, Netherlands, 1989, pp. 205–218.

[38]Kung, S. Y., and Hwang, J. N., "A Unified Systolic Architecture for Artificial Neural Networks," *Journal of Parallel and Distributed Computing*, Vol. 6, 1989, pp. 358–387.
doi: 10.1016/0743-7315(89)90065-8

[39]Kumar, V., Shekhar, S., Amin, M. B., "A Scalable Parallel Formulation of the Backpropagation Algorithm for Hypercubes and Related Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 10, Oct. 1994.

[40]Witbrock, M., and Zagha, M., "An Implementation of Back-propagation Learning on GFII: A Large SIMD Parallel Computer," Tech. Rep. CMU-CS-89-208, Carnegie Mellon University, Pittsburgh, PA, USA, Dec. 1989.

[41]Bourrley, J., "Parallelization of a Neural Learning Algorithm," *Proceedings of the First European Workshop on Hypercube and Distributed Computers*, edited by F. Andre, Amsterdam, Netherlands, 1989.

[42]Mak, B. K., and Egecioglu, O., "Communication Parameter Tests and Parallel Backpropagation Algorithms on ZPSCR Hypercube Multiprocessor," *IEEE Frontier*, 1990, pp. 1353–1364.

[43]Hwang, J., and Kung, S., "Parallel Algorithms/Architectures for Neural Networks," *Journal of VLSI Signal Processing*, Vol. 1, No. 3, 1989.

[44]Tomboulian, S., "Introduction to a System for Implementing Neural Net Connections on SIMD Architectures," *Proceedings of the Neural Information Processing Systems*, 1987, pp. 804–813.

[45]Lin, W., Prasanna, V. K., and Przytula, K. W., "Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines," *IEEE Transactions on Computers*, Vol. 40, No. 12, Dec. 1991, pp. 1390–1401.

[46]Shams, S., and Gaudiot, J. L., "Implementing Regularly Structure Neural Networks on the DREAM Machine," *IEEE Transactions on Neural Networks*, Vol. 6, No. 2, 1995, pp. 407–421.

[47]Kim, Y., Noh, M. J., Han, T. D., and Kim, S. D., "Mapping of Neural Networks onto Memory-Processor Integrated Architecture," *Neural Networks*, Vol. 11, 1998, pp. 1083–1098.
doi: 10.1016/S0893-6080(98)00092-6

[48]Ayoubi, R. A., and Bayoumi, M. A., "An Efficient Implementation of Multilayer Perceptron on Mesh Architecture," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Vol. 2, pp. 109–112, 2002.

[49]Ayoubi, R. A., and Bayoumi, M. A., "Efficient Mapping Algorithm of Multilayer Neural Network on Torus Architecture," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 9, Sept. 2003.

[50]Kim, K., and Kumar, V. K. P., "Efficient Implementation of Neural Networks on Hypercube SIMD Arrays," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, 1989, pp. 614–617.

[51]Malluhi, Q. M., Bayoumi, M. A., and Rao, T. R., "An Efficient Mapping of Multilayer Perceptron with Backpropagation ANNs on Hypercubes," *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 368–375.

[52]Zhang, X., Mckenna, M., Mesirov, P., and Waltz, D. L., "The Backpropagation Algorithm on Grid and Hypercube Architectures," *Parallel Computing*, Vol. 14, 1990, pp. 317–327.
doi: 10.1016/0167-8191(90)90084-M

[53]http://www.statsoft.com/textbook/stneunet.html

[54]Barron, A. R., "Universal Approximation Bounds for Superpositions of a Sigmoid Function," *IEEE Transactions on Information Theory*, Vol. 39, 1993, pp. 930–945.
doi: 10.1109/18.256500

[55]Yang, D., *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*, Springer, New York, 2000.

[56]Shtern, V., *Core C++: A Software Engineering Approach*, Prentice-Hall, NJ, 2000.

[57]Kumar, V., Grama, A., Gupta, A., and Karypis, G., *Introduction to Parallel Computing*, San Francisco: Benjamin Cummings/Addison Wesley, 2002.

[58]The MPI Forum On-line, http://www.mpi-forum.org.

[59]Al-Alaoui, M. A., Mouci, R., Mansour, M. M., Ferzli, R., "A Cloning Approach to Classifier Training," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, Vol. 32, No. 6, Nov. 2002.

[60]http://gears.aset.psu.edu/hpc/systems/rs6k/

[61]http://www.cse.psu.edu/mufasa/

Fernando Figueroa
Associate Editor