Implementation of Boolean Neural Networks on Parallel Computers

André Carlos Ponce de Leon Ferreira de Carvalho
Laboratório de Inteligência Computacional
Departamento de Ciências da Computação
Instituto de Ciências Matemáticas de São Carlos
Universidade de São Paulo
São Carlos, SP, Brasil

Michael C. Fairhurst
David L. Bisset
Electronic Engineering Laboratories
University of Kent at Canterbury
Canterbury, Kent, UK

Abstract

This paper analyses the parallel implementation using networks of transputers of a neural structure belonging to a particular class of neural architectures known as GSN neural networks. These architectures, belonging to the general class of RAM-based networks and composed of digitally specified processing nodes, have been implemented using different processing topologies, and performance in relation to both training and testing efficiency in a practical pattern recognition task has been evaluated.

1 Introduction

Neural network approaches to pattern recognition have become increasingly widespread in recent years, but have often been somewhat unsatisfactory for many practical applications because of the constraints imposed on processing speeds attainable, particularly in the implementation of appropriate training algorithms, and on their complex hardware implementation. As a result of the real-time requirements in many pattern recognition applications, a great deal of research effort has been devoted to the issue of implementing pattern recognition algorithms on parallel machines [1, 2, 3, 4, 5, 6, 7].

Boolean neural networks, due to their strong resemblance to Random Access Memory devices, offer an alternative approach to neural network design by being easier both to simulate on parallel computers and to implement in hardware. Goal seeking neural networks (GSN) are a family of Boolean neural networks which have been proposed to solve some problems faced by early Boolean models. This paper investigates the performance of GSN feedforward architectures (GSN^f) when implemented on several networks of transputers. Transputer technology [8] provides a very suitable environment for the implementation of parallel systems, since Transputer hardware makes it possible to connect any number of processors in several different topologies without the need for rigid synchronisation. The implementations considered here have used different number of processors and different topological structures.

2 Boolean neural networks

Boolean neural networks were developed as an engineering tool for pattern recognition applications, and thus they offer a different approach to neural network design. The effectiveness of Boolean neural networks for image processing applications has been demonstrated in a number of experiments [9, 10, 11, 12]. Boolean neural models usually receive and generate only binary values. Although it can be argued that this restricts their range of applications, there are a large number of neural network applications which involve only binary input and output values. Furthermore, their digital specification makes their hardware implementation potentially very simple and efficient. A further advantage of Boolean neural networks is that they can generally use fast (often single shot) learning algorithms, while the high functionality of individual Boolean neurons offers a large degree of processing flexibility [12].

A large number of Boolean neural network configurations have been designed, each configuration having its own individual characteristics. These architectures can be divided broadly in two groups. The first group comprises architectures whose constituent neurons are based on memory units or universal logic gates and can be easily implemented by Random Access Memory devices, which is why they are often called RAM-based models. A RAM-based neuron can be implemented in hardware by using Random Access Memory devices and uses the input values presented to its input terminals to access one of its 2^N memory contents, where N is the number of input terminals. The second group consists of architectures which restrict the functions represented by their units to a reduced number of Boolean functions [13, 14] or use binary weights [15, 16]. The first Boolean model of this type to be proposed was the N-tuple processor, developed by Bledsoe and Browning [17].

Many different algorithms have been proposed to train Boolean neural networks. The majority of these algorithms work through updating the logical function of the neurons. These updates occur either through one-shot learning, as is the case, for example, in GSN^f networks [18], through gradient descent, as with the ALN [13] or by using simulated annealing, which is the case with random Boolean networks [19]. There have also been attempts to train Boolean

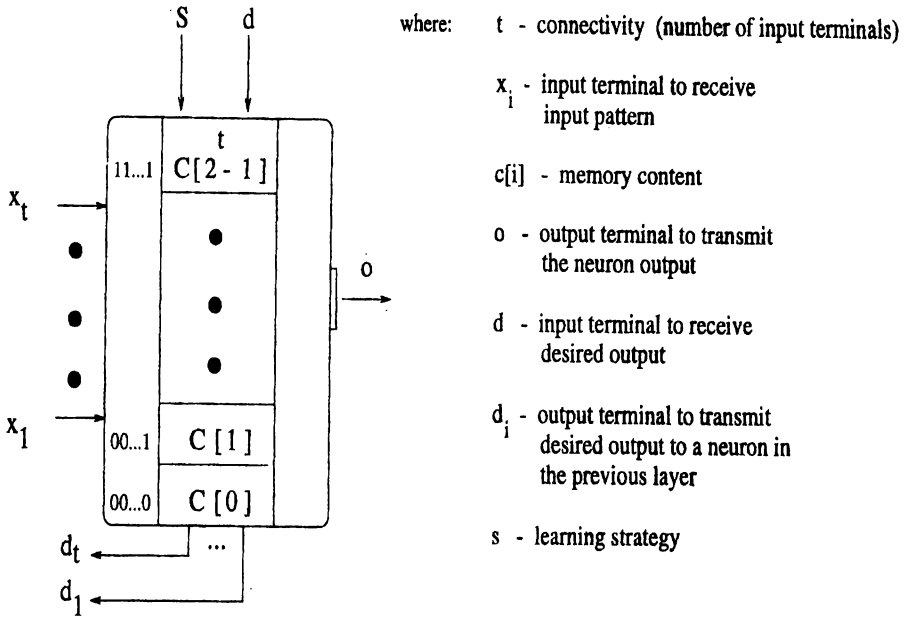

Figure 1: GSN neuron

neural networks by changing the connections instead of the values stored in the memory contents using simulated annealing [20, 21], with encouraging results.

3 GSN^f Architecture

A Goal Seeking Neuron, GSN, is a Boolean neuron which, unlike previous Boolean neurons, can input, store and generate values equal to 0, 1 or u (undefined). When there is at least one value u on its input terminals, then a *set* of memory contents will be addressed, the addresses of these memory locations being derived from the original address by changing the u values to 0 and 1. Before starting to train the network, all of its memory contents are filled with undefined values, signifying that the network does not initially store any specific knowledge of the external world. By outputting undefined values, GSN allows undefined values to be propagated in multi-layer architectures, providing very interesting generalisation capabilities.

As can be seen in Figure 1, GSN has a set of input terminals: $X = \{x_1, x_2, ..., x_t\}$, which are used to receive the input to the neuron, an output terminal, o, that represents the output of the neuron, a teach terminal, s, which indicates if the neuron is in the learning phase or in the recall phase, a desired output terminal, d, which is set to the desired output value when the neuron is in the learning phase and terminals of desired inputs, $d_1, ..., d_i$, which are used to determine the desired output of preceding GSN neurons, when more than one layer of neurons is used.

The input terminals of a Boolean neuron can address a set of 2^t different cells, where each cell can store one value from the set $\{0, 1, u\}$. During the learning phase, the teach terminal receives the value 1 and the value stored in the cell addressed by the input terminals is changed to the value of the desired output. In the recall phase, the teach terminal receives the value 0, and the output of the neuron is computed as a function of the values stored in its memory contents addressed by the input terminals.

GSN uses a fast and efficient learning algorithm. It is efficient because it maximizes the

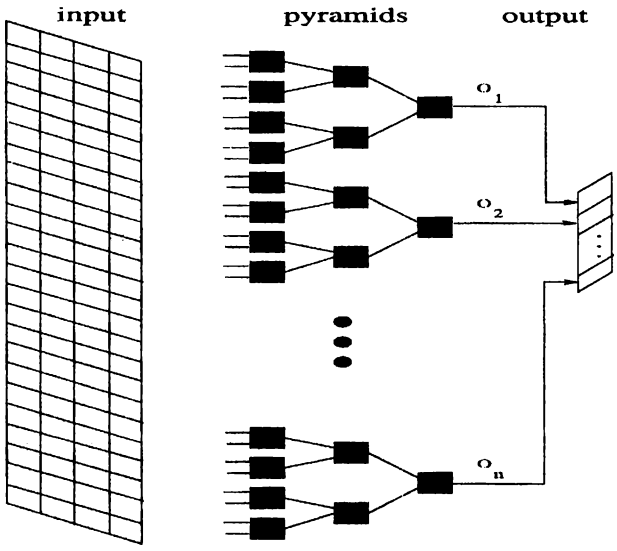

Figure 2: GSN^f Architecture

efficiency of the storage of values in its memory contents, avoiding the storage of new information whenever possible and storing new information without disruption of previously stored information. It is fast because it employs one-shot learning, which means that the training set is presented just once (i.e., the training happens in a single time epoch, rather than as an iterative process).

GSN-based neurons are usually grouped in pyramids. A pyramid is a multi-layer structure, where each layer has a fixed number of neurons with a fan-out equal to 1. Pyramids have a lower functionality than a single neuron covering the same input pixels, but their use reduces the size of memory needed for implementation and increases generalisation if compared to a single neuron covering the same set of pixels [22].

GSN neurons have been used in several different architectures, each suitable for a different range of applications. The architecture investigated in this paper, the GSN feedforward architecture (GSN^f), is formed by a fixed number of independent pyramids, where each pyramid covers a subset of the pixels of the input image. GSN^f is the most common GSN-based architecture [18]. It has been successfully used in character recognition applications and has been extensively investigated [23, 24, 12, 25, 26]. Figure 2 illustrates a typical GSN^f neural network.

The processing of GSN^f can be divided into three phases, each associated with a specific goal: a validating phase, a learning phase and a recall phase. The purpose of the validating phase is to produce a validating value for each pyramid. A validating value is the pyramid's output guess for a given input pattern. A pyramid's validating value defines which defined value(s) it can learn in the next learning phase, and a pyramid cannot be taught when its validating value is the opposite to its desired output value. The learning phase teaches the pyramids by changing the values stored in the memory locations of their constituent neurons. The recall phase produces the pyramids' output for an unknown input, seeking, for each neuron, to output the defined value with the greatest occurrence frequency in the addressable memory contents. The following subsections will explain in more detail each of the three states.

3.1 Validating state

The goal in this state is to determine the values that the neuron can learn in the next learn phase without disrupting previously stored information. In order to discover this, the input

to be learnt is fed into the input terminals. A function is applied to the values stored in the GSN addressed memory contents to determine its validating value. This validating value is propagated through the pyramid to its apex. The validating value produced by the neuron on the apex will be the pyramid validating value. If the output is equal to u (undefined) then it can learn any desired output. If, on the other hand, the output is a defined value (0 or 1), only this value can be taught. The output of a neuron n_i in the validating state is given by the Equation 1 below:

$$o_i = \begin{cases} 0 & \text{iff } \forall a_{i_m} \in A_i, C_i[a_{i_m}] = 0; \\ 1 & \text{iff } \forall a_{i_m} \in A_i, C_i[a_{i_m}] = 1; \\ u & \text{iff } \exists a_{i_m} \in A_i \mid C_i[a_{i_m}] = u \text{ or } \exists a_{i_m}, a_{i_l} \in A_i \mid C_i[a_{i_m}] \neq C_i[a_{i_l}]. \end{cases} \quad (1)$$

In this equation, A_i represents the set of addresses originated from the input, $C_i[a_{i_m}]$ and $C_i[a_{i_l}]$ are the contents of n_i that are accessed by addresses a_{i_m} and a_{i_l} respectively.

According to Equation 1, the output generated in the validating state can be equal to 0, 1 or u. When all the values stored in the addressable contents are equal to 0, the output will be equal to 0. If, on the other hand, all the values stored in the addressable contents are equal to 1, the output will be equal to 1. The output will be equal to u in any other case.

3.2 Learning state

The goal in the learning phase is to store the desired output in a location addressed by its input terminals. To use its cells more efficiently, GSN tries to store its desired output in an addressed memory content which already stores this value. When there is more than one option, a memory location holding an undefined value is selected. The choice of the content is illustrated by Equation 2.

$$a_{i_m} = \begin{cases} Ran(A_{i/_{d_i}}) & \text{iff } \|A_{i/_{d_i}}\| > 0; \\ Ran(A_{i/_u}) & \text{iff } \|A_{i/_{d_i}}\| = 0. \end{cases} \quad (2)$$

where $A_{i/_s} = \{a_{i_k} \in A_i \mid C_i[a_{i_k}] = s\}$ represents the addressable set of the neuron n_i that stores the value s, $Ran(A_{i/_s})$ is a element randomly chosen from $A_{i/_s}$ and $\|A_{i/_s}\|$ is the number of elements that belong to $A_{i/_s}$.

As can be seen in Equation 2, GSN searches for a cell in its addressable set which stores a value equal to its desired output. If there is no such cell, it chooses a cell storing an undefined value. If, for either of the two cases mentioned above, there is more than one possible choice, a cell from the addressed set is chosen at random. After choosing the cell, the address of the cell selected must then be sent back, through the desired input terminals, to be used as desired output by the neurons in the previous layer. The desired input d_{ij} that must be sent to the neuron n_j in the previous layer is the j^{th} bit of the address a_{i_m}. Thus each neuron n_j receives its desired output, learns and provides desired outputs to neurons in the previous layer.

3.3 Recall state

When the neuron reaches its recall state, it has as the goal to produce the value with the highest occurrence in the addressable set. This value is propagated through the pyramid layers

in a similar way as the validating value to produce the pyramid recall value. The method used to define the recall value is given by Equation 3.

$$o_i = \begin{cases} 0 & \text{iff } \|A_{i/0}\| > \|A_{i/1}\|; \\ 1 & \text{iff } \|A_{i/1}\| > \|A_{i/0}\|; \\ u & \text{iff } \|A_{i/0}\| = \|A_{i/1}\|. \end{cases} \tag{3}$$

In this equation, the neuron will generate as output a value 1 only if the number of such values in the addressable locations is greater than the number of 0's. If the inverse occurs (i.e., the number of 0's is greater than the number of 1's), the output will be equal to 0. It does not matter how many undefined values exist in the addressable locations. When the addressed memory contents store the same number of defined values 0 and 1, the output will be an undefined value. This rule has the effect of minimizing the propagation of undefined values.

Transputer technology opens the possibility to connect any number of processors in several different topologies without the need for rigid synchronisation, while the structure and processing characteristics of GSN-based neural networks make them very suitable for massive parallel implementation. The following sections of this paper describe the way in which GSN^f architectures can be implemented on networks of transputer processors, and evaluate the effectiveness of such a parallel implementation through the measurement of specific key performance parameters.

4 Parallel implementation

The ideal parallelization of a program is one where the program distributed on N processors runs N time faster than when implemented on a single processor. However, overheads associated with communication between processors and a perfectly equal division of tasks among them make the achievement of this desirable objective very difficult in practice. What must then be attempted is to divide the processing task approximatelly among the processors and to maintain an individual high working rate per processor for the maximum proportion of the time.

The ratio of the computational and communication times is a very important issue in the implementation of parallel programs in transputer networks [27]. This is a particularly important issue for transputer networks where communication overheads often constitute a bottleneck and so should be kept to a minimum. Another important point is that in a network of transputers, only one transputer has access to external input/output operations.

The independence with which the pyramids in a GSN^f neural network can be processed and the low connectivity of GSN neurons makes this architecture particularly suited to implementation in transputer networks. Pyramids can can be processed completely in parallel, with practically no interaction among them. At the same time, the low connectivity used by GSN^f neurons implies relatively low memory requirements.

T-800 transputer processors have been used in the experiments described in this paper. These processors have a clock rate of 20 MHz, 4 Mbyte memory and four bi-directional communication links which allow each transputer to be connected to four other transputers, making possible the use of different topologies. The implementation of GSN^f neural networks was written in Occam 2, which is a natural programming language for transputers.

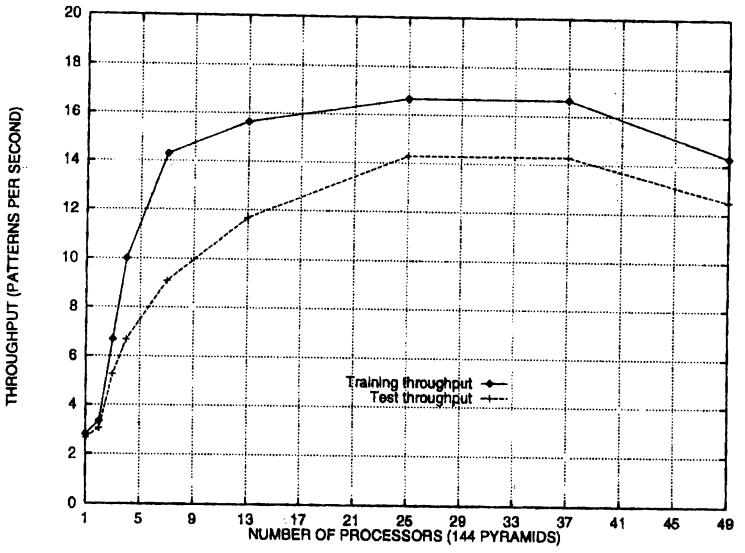

Figure 3: Data throughput

5 Experimental results

The major potential benefits of the parallel implementation of the GSN^f network architecture are in relation to the attainable processing speeds for training the network (both the training and the validating phases) and in classifying unknown patterns. These performance criteria have been investigated quantitatively in relation to the possible implementational topologies adopted. For such, different GSN^f architectures were trained and tested to recognize machine printed characters extracted from mail envelopes. Each character was represented by an array of 16x24 pixels. The first major issue to address is the question of the distribution of the processing cells of the neural network among the transputer elements. The recognition performance rates achieved by GSN^f networks when applied to character recognition are presented in [28].

Any GSN neuron in one of the intermediate layers of any given pyramid is constantly exchanging information with either the neurons in the previous layer whose outputs are connected to its input terminals or with the neuron in the next layer which has an input terminal connected to its output terminal. This makes it inadvisable to allocate neurons from the same pyramid to different processors. The independence with which the pyramids operate makes the allocation of a set of pyramids to each transputer the best policy and this is how GSN^f has been implemented. Since the ideal situation is one where the processing task is equally distributed among the transputers, whenever more than one transputer is used in a network, one transputer is exclusively reserved for input, output and final classification operations and an equal number of pyramids are allocated to each transputer.

Communication overheads are reduced in the GSN^f implementation by sending, whenever possible, sets of data that will not necessarily be used immediately but are already available in a large block. This strategy was used, for example, to send the whole training set to each transputer before the initiation of the learning phase. In order to evaluate the effectiveness of parallel implementation, a number of experiments were carried out with a network of 144 pyramids, each with 31 neurons of connectivity 2, when implemented in networks using different numbers of transputers.

Some initial performance figures are presented in Figure 2, which first shows network

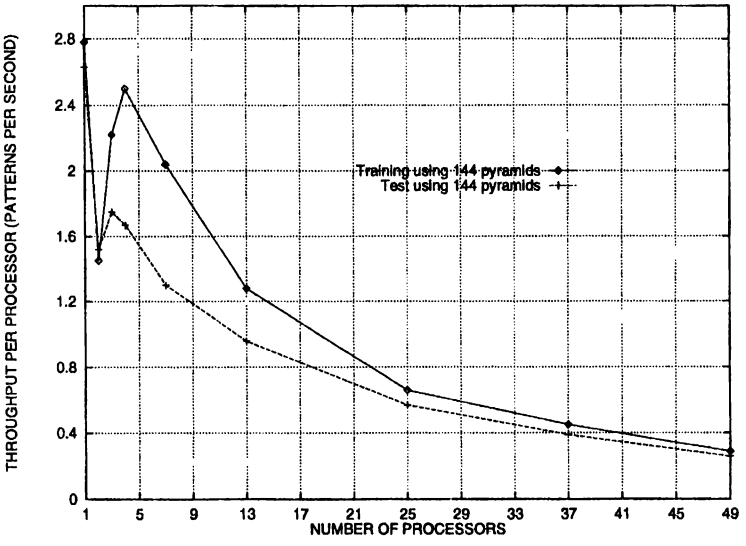

Figure 4: Data throughput per processor

throughput during the training and test phases when different numbers of transputers are used by the network. Apart from the case where only one transputer has been used, one of the transputers of each network was exclusively dedicated to input, output and final classification, with GSN^f pyramids being distributed among the remaining transputer processors.

According to Figure 2, the training throughput increases almost linearly with an array size from 2 to 4 processors. Above this the increase in throughput starts to decline until the number of transputer processors used is equal to 25. The training throughput stabilizes when the number of transputer processors is larger then 37, showing a decrease when around 49 pyramids are used. The test throughput does not increase as much as the training throughput, stabilizing when the network comprises 25 transputer processors.

The smaller increase in the test throughput reflects the fact that the validating and the learning operations present in the learning phase require much more computational effort than those operations found in the recall phase, which makes the test computing/communication ratio become smaller. Figure 3 illustrate how the increase in the number of processors affects the processing carried out by each processor by showing the training and test throughputs per processor when different numbers of processors are used.

Because of the importance of the proportion of overall processing time devoted to communication operations in determining the performance achieved by transputer networks, the topology of the network becomes a very important issue when more than one transputer processor is used. A suitable choice of topology may play a significant part in improving the training and test throughput attainable. In order to investigate this issue, GSN^f was implemented in two different topologies, the chain topology and the tree topology, as illustrated in Figures 4 and 5 respectively.

Each different topology was implemented on a network of 13 transputers. This particular number of transputers was chosen because it allows the use of balanced trees with a reasonable number of transputers at a short distance from the root. In these implementations, 144 pyramids were equally distributed among 12 "worker" processors. The 13^{th} processor, the "master", was used to support the network, being responsible for operations such as input, output, final classification decision and definition of network configuration. Table 1 shows the training and

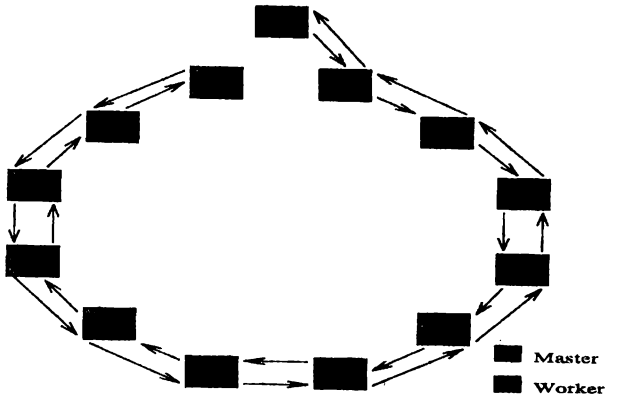

Figure 5: Chain topology

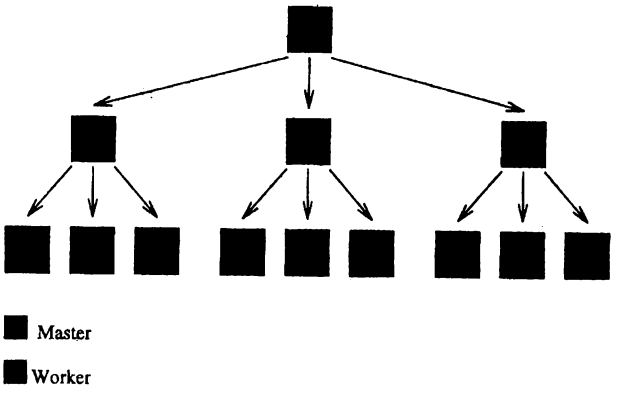

Figure 6: Tree topology

test times for each of these topologies.

GSN^J processing time (pattern/second)		
Topology	*Phase*	
	Training	*Test*
chain	15.63	11.68
tree	15.63	12.17

Table 1: Times for different topologies

These results show that the two different topologies require the same training time, which is to be expected due to the higher proportion of computational operations compared with communication operations. Considering the test times, a reduction is achieved when the tree topology is used, indicating the relatively lower number of communication operations required with this topology. While with a chain topology data transmission between a server processor and the master processor uses an average of 6.5 communication links, with a tree topology the average number of communication links is 1.75.

The reason for the gain in training and test throughput when networks of more than one transputer are used to be lower than the ideal is the low computing/communication ratio. In

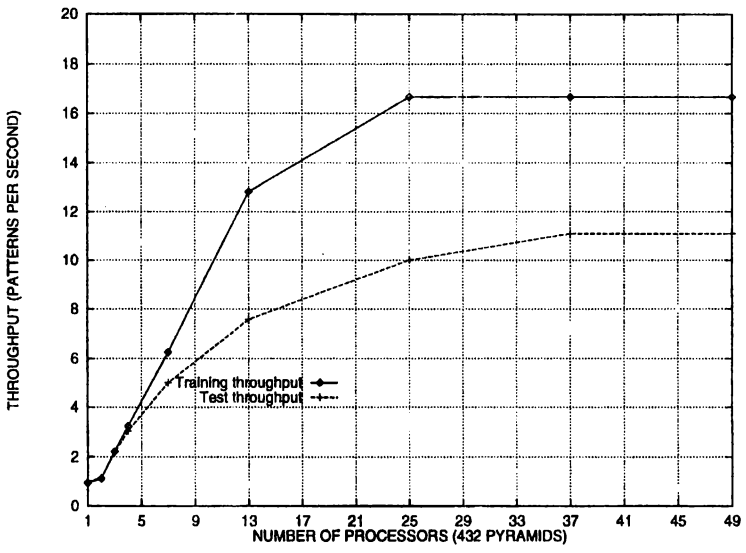

Figure 7: Data throughput

order to evaluate how a further reduction in the processing time can be achieved by increasing the proportion of computational operations, the same experiments were carried out with the number of pyramids increased by a factor of three, to 432. Figure 6 shows the training and test throughput achieved when 432 pyramids were distributed in networks with different numbers of transputers using a chain topology.

Figure 6 shows that the use of 432 pyramids allows larger gains in the training and test throughputs by increasing the number of transputer processors used. By triplicating the number of pyramids, the throughput was also expected to triplicate. This happens until around 4 transputer processors are used, after which the throughput of the 432-pyramid architecture gets successively closer to that achieved by the 144 pyramid architecture. Using 432 pyramids, the increase in the training throughput is equal to the increase in the number of processors until the number of transputer processors used is around 13. The training throughput becomes stable when more than 25 transputers are used. The training and test throughputs per processor when 432 processors are used is shown in Figure 7.

It should be noticed, from Figures 2 and 6, that the training throughput after 25 transputers is the same when 144 and 432 pyramids are used, apart from the case where 144 pyramids are distributed among 49 transputers. For the test times, the increase in the test throughput is smaller than the increase in the training throughput. The test throughput is also smaller than that achieved when 144 pyramids were used and stabilizes when the network has more than 37 transputer processors. Further gains can be achieved by increasing the neurons connectivity, because this will further increase the computing/communication ratio.

Table 2 illustrates the training and test throughputs achieved when a GSN^f architecture based on 432 pyramids was implemented in a network of 13 transputers using chain and tree topologies.

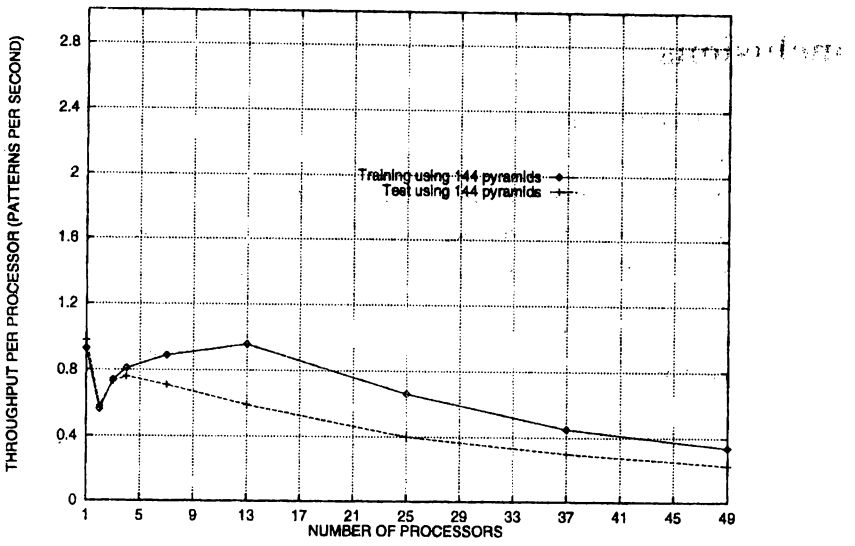

Figure 8: Data throughput per processor

GSN^f processing time (pattern/second)		
Topology	*Phase*	
	Training	*Test*
chain	12.82	7.58
tree	12.82	7.79

Table 2: Times for different topologies

The throughput figures presented by Table 2 indicate that even when substantially increasing the number of pyramids, the reduction of communication overheads leads to similar training throughput achieved for the two topologies used. The test throughput, however has different values for the two topologies, being lower when a chain topology is used. The smaller diferencein the test throughput figures achieved by these topologies when compared to Table 1 reflects the reduction in communication overhead.

These results confirm that an increase in the throughput in both training and test time may be achieved through the parallelization of GSN^f in a network of transputers. They also show that the chain topologies is slightly less effective than the tree topology. Such performance figures demonstrate a capacity for overall processing times attainable with a parallel implementation which makes its use viable in a wide range of practical applications.

The question of parallel implementation has been investigated, and GSN^f architectures have been successfully implemented on a network of transputers using different topologies, leading to a considerable improvement in processing speeds with respect to equivalent implementation on a single processor. The results presented in Figures 2 and 6 provide an important benchmark against which to define the number of transputer processors to be used in a practical task. Although it is difficult to define beforehand how precisely the increase in the number of transputer processors will affect the throughput achieved, these results indicate that the higher the computational/communication ratio, the higher the benefits achieved by the increase in the number of transputer processors.

6 Conclusions

This paper has described the implementation of GSN feedforward architectures on networks of transputers. In relation to the implementation of the network and its implications in relation to processing speed, it has been shown that the GSN architecture maps conveniently to a parallel infrastructure and that training and classification times can be reduced by an order of magnitude for a typical network when implemented on even a modest array of transputer processors.

Acknowledgement: A. de Carvalho would like to acknowledge the support of the Brazilian Research Agency (CNPq).

References

[1] P L Springer and S Gulati. Parallelizing the cascade-correlation algorithm using time warp. *Neural Networks*, 8(4):571–578, 1995.

[2] I Mahadevan and L M Patnaik. Performance evaluation of bidirectional associative memory on a transputer-based parallel system. *Parallel Computing*, 18:401–413, 1992.

[3] K Obermayer, H Ritter, and K Schulten. Large-scale simulations of self-organising neural networks on parallel computers: application to biological modelling. *Parallel Computing*, 14:381–404, 1990.

[4] J M J Murre. Transputers and neural networks: An analysis on implementation constraints and performance. *IEEE Transactions on Neural Networks*, 4(2):284–292, March 1993.

[5] T E Lange. Simulation of heterogeneous neural networks on serial and parallel machines. *Parallel Computing*, 14:287–303, 1990.

[6] H Yoon, J H Nang, and S R Maeng. Parallel simulation of multilayered neural networks on distributed-memory multiprocessors. *Microprocessing And Microprogramming*, 29:185–195, 1990.

[7] M C Fairhurst and P Brittan. Matching structural and implementational modules in the specification of image classifiers. *Pattern Recognition*, 24:555–566, 1991.

[8] Inmos, editor. *Transputer Reference Manual.* Prentice Hall, 1988.

[9] S Ramanan, R S Petersen, T G Clarkson, and J Taylor. pRAM nets for detection of small targets in sequences of infrared images. *Neural Networks*, 8(7/8):1227–1237, 1995.

[10] N M Allinson and M J Johnson. Application of self-organising digital neural networks to attentive vision systems. In *Proceedings of the 4th International Conference on Image Processing and its Applications*, pages 193–196, Maastricht, The Netherlands, April 1992. IEE.

[11] E C Mertzanis, D A Dukic, I D Benest, and J Austin. A neural network based position invariant pattern recognition system with a constrained hypermedia style user-interface. In *Proceedings of the 4th International Conference on Image Processing and its Applications*, pages 217–220, Maastricht, The Netherlands, April 1992. IEE.

[12] A de Carvalho, M C Fairhurst, and D L Bisset. Classifying images using goal seeking neural network architectures. *IEE Proceedings-I Communications, Speech and Vision*, 40(1):12–18, February 1993.

[13] W W Armstrong and J Gecsei. Adaptation algorithms for binary tree networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:276–285, 1979.

[14] A Garzotto. The NAND-Net. In *Proceedings of the ICANN 92 Conference*, pages 1419–1422, Brighton, UK, September 1992. Elsevier.

[15] D J Willshaw, O P Buneman, and H C Longuet-Higgins. Non-holographic associative memory. *Nature*, 222:960–962, 1969.

[16] R P Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4:4–22, April 1987.

[17] W W Bledsoe and I Browning. Pattern recognition and reading by machine. In *Proceedings of The Eastern Joint Computer Conference*, pages 225–232, 1959.

[18] E Filho, M C Fairhurst, and D L Bisset. Adaptive pattern recognition using goal-seeking neurons. *Pattern Recognition Letters*, 12:131–138, March 1991.

[19] A Wuensche. Basis of attraction in disordered networks. In *Proceedings of the ICANN 92 Conference*, Brighton, UK, September 1992. Elsevier.

[20] S Patarnello and P Carnevali. Learning networks of neurons with Boolean logic. *Europhysics Letters*, 4(4):503–508, 1987.

[21] J R Doyle. Supervised learning in n-tuple neural networks. *Int. J. Man-Machine Studies*, 33:21–40, 1990.

[22] W K Kan and I Aleksander. A probabilistic logic neuron network for associative learning. In *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 541–548, San Diego, California, June 1987. IEEE.

[23] E Filho, D L Bisset, and M C Fairhurst. Networks of Boolean goal seeking neurons for pattern recognition. In *International Symposium on Neural Networks for Sensory and Motor Systems*, Neuss, F.R.G., March 1990.

[24] R G Bowmaker and G G Coghill. Improved recognition capabilities for goal seeking neuron. *Electronics Letters*, 28:220–221, 1992.

[25] J D Mason, E L Hines, and J W Gardner. Analysis of electronic nose data using weightless neural networks and genetic algorithms. In *Weightless Neural Network Workshop'93, Computing with Logical Neurons*, pages 93–97, University of York, York, UK, April 1993.

[26] W Martins and N M Allinson. Two improvements for GSN neural networks. In *Weightless Neural Network Workshop'93, Computing with Logical Neurons*, pages 58–63, University of York, York, UK, April 1993.

[27] K C Bowler, R D Kenway, G S Pawley, D Roweth, and G V Wilson. *An introduction to OCCAM programming*. Chartwell-Bratt, 1989.

[28] E Filho. Signature recognition using Boolean neural networks. Master's thesis, Universidade Federal de Pernambuco, Brazil, 1987. Original in Portuguese.