PARALLEL BACKPROPAGATION NEURAL NETWORKS FOR TASK ALLOCATION BY MEANS OF PVM

CRESPO, M., PICCOLI F., PRINTISTA M., GALLARD R.

Proyecto UNSL-338403[1]
Departamento de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950 - Local 106
5700 - San Luis
Argentina
E-mail: {mcrespo,mpiccoli,mprinti,rgallard}@unsl.edu.ar

Phone: +54 652 20823
Fax : +54 652 30224

ABSTRACT

Features such as fast response, storage efficiency, fault tolerance and graceful degradation in face of scarce or spurious inputs make neural networks appropriate tools for Intelligent Computer Systems.
A neural network is, by itself, an inherently parallel system where many, extremely simple, processing units work simultaneously in the same problem building up a computational device which possess adaptation (learning) and generalisation (recognition) abilities. Implementation of neural networks roughly involve at least three stages; design, training and testing. The second, being CPU intensive, is the one requiring most of the processing resources and depending on size and structure complexity the learning process can be extremely long. Thus, great effort has been done to develop parallel implementations intended for a reduction of learning time.
Pattern partitioning is an approach to parallelise neural networks where the whole net is replicated in different processors and the weight changes owing to diverse training patterns are parallelised. This approach is the most suitable for a distributed architecture such as the one considered here.

Incoming task allocation, as a previous step, is a fundamental service aiming for improving distributed system performance facilitating further dynamic load balancing.
A Neural Network Device inserted into the kernel of a distributed system as an intelligent tool, allows to achieve automatic allocation of execution requests under some predefined performance criteria based on resource availability and incoming process requirements.

This paper being, a twofold proposal, shows firstly, some design and implementation insights to build a system where decision support for load distribution is based on a neural network device and secondly a distributed implementation to provide parallel learning of neural networks using a pattern partitioning approach.
In the latter case, some performance results of the parallelised approach for learning of backpropagation neural networks, are shown. This include a comparison of recall and generalisation abilities and speed-up when using a socket interface or PVM.

KEYWORDS: Distributed systems workload, parallelised neural networks, backpropagation, partitioning schemes, pattern partitioning, system architecture.

[1] The Research Group is supported by the Universidad Nacional de San Luis and the ANPCYT (National Agency to Promote Science and Technology).

1. INTRODUCTION

Implementation of neural networks roughly involves at least three stages; design, training and testing. Training, being CPU intensive, is the one requiring most of the processing resources and depending on size and structure complexity the learning process can be extremely long. Thus, great effort has been done to develop parallel implementations intended for a reduction of learning time.
The backpropagation (BP) learning algorithm, due to its efficiency and wide range of applications, is one of the most popular learning algorithm.
BP can be parallelised through two partitioning schemes; either the network or the training pattern space is partitioned [9][14][15].

In *network partitioning*, the nodes and weights of the neural network are distributed among diverse processors and thus the computations for node activation, node errors and weight changes are parallelised.

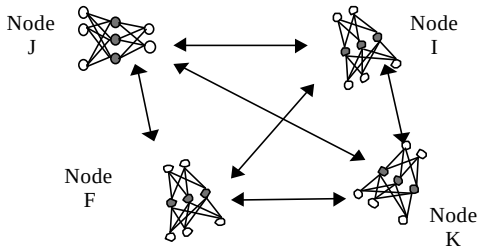
In *pattern partitioning* the whole neural net is replicated in different processors and the weight changes owing to diverse training patterns are parallelised.
This last scheme is suitable for problems with a large set of training patterns and fit properly to run on local memory architectures.
In this work we only concentrate on a pattern partitioning approach *with a new variant of the per-epoch-training regime*, to parallelise the learning process for transparent task allocation in a computer network. The variant called variable-epoch training regime consists in randomly assigning the number of epochs (epochs interval) locally performed before any exchange takes place. Higher speed-up was the motivation of this new approach previously envisioned for a socket-based interface [5]. In the following sections alternative parallel approaches, supporting architectures, the application and results concerning speedup, recall and generalisation capabilities when contrasted against the conventional sequential approach will be discussed.

2. PARALLELISING BACK PROPAGATION

One significant point to think about when designing a parallel system is the parallelism granularity of the applications.
For parallelising the BP learning algorithm, we mentioned two schemes:

In *network partitioning* each processor processes its corresponding task and therefore, during the propagation and adaptation phases the processors need to establish communication with each other. Since this interaction and exchange of data is frequently done, this scheme demands a fine granularity of parallelism. This type of parallelism is advantageous on a Multiprocessor or Shared Memory Architecture[20].
On the other side, in *pattern partitioning*[1][10][12] a single program is replicated among processors and each computer will execute its personal copy of this program on different data elements (patterns). As shown in figure 1, pattern partitioning replicates the neural net structure (units, edges and associated weights) at each processor and then the partitioned training set is distributed among processors. Each processor performs the propagation and adaptation phases for the local set of patterns. Also, each processor accumulates the weight changes produced by the local patterns, which afterward are broadcast to other processors for updating weight values. This is done by using *Ts/P* patterns where *Ts* is the size of the training set and *P* is the number of processors committed to the learning process. Weight changes are performed in parallel and then the corresponding accumulated weight change vectors are exchanged between processors. This scheme is suitable for problems with a large set of training patterns, by

permitting a more coarse parallelism than the network partitioning scheme. This scheme fit properly to run on Message Passing Multiprocessors or Multicomputers [20].



Fig, 1. A Pattern Partitioning Scheme

Here we propose a distributed architecture supporting pattern partitioning for parallel training of neural networks. In the corresponding implementation the neural net is replicated in each system node where an individual learning process is running for the associated partition of the pattern space. Hence, weight changes are computed concurrently, exported, imported and adjusted accordingly until the whole parallel learning process is completed.

3. A NEURAL NETWORK DEVICE FOR ALLOCATION OF INCOMING TASKS

As an application of neural networks we used an intelligent facility to automatically allocate, in a computer network, a user incoming process to the most appropriate node in accordance to its computing requirements[2].
The model assumes that:

- The relevant performance feature to improve is the response time for user processes.
- Processes coming to be served in this network have different demands on system resources (CPU, Memory and I/O devices).
- The network is formed by a set of **N** nodes, such that each of them can contribute with different performance to a user process depending on its demands.
- Every user incoming process comes to the network through an entry node, before passing to the execution node (see Fig. 2). Process behavior and resource requirements can be determined by a program profile file or explicitly declared by the incoming process.
  - An *evaluator* module within the Operating System kernel evaluates process attributes, requirements and system state at the process arrival time.
  - Using the output of the evaluator, as input, a *decisor* module decides which node in the network can accomplish more efficiently the process execution and then process migration takes place.

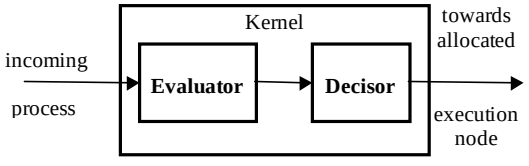


Fig. 2 The kernel portion of an entry node

As a simple example, let us assume the following scenario:

We have a system where N available nodes differ essentially in Current Memory Capacity (*CMC*) and MIPS provided. Due to system dynamics they also differ in Current Available Processing Power (*CAPP*). User processes are CPU intensive tasks and their main requirements are Memory Required (*MR*) and Desired Response Time (*DRT*).

CMC, *CAPP*, *MR* and *DRT*, are each divided into a number of levels (high, medium and low, or more levels). Other processes requirements on system resources, such as access to secondary storage, can be equally satisfied by any of the available nodes and there will not be network transfers (except for initial process migration, which we assume is equally costly for every node). Then the following simple allocation criterion can be applied:

- Having *MR* best fit satisfied, satisfy *DRT* by allocating the process to the best fitted node (the one with minimum *CAPP* fulfilling process requirement). In case of equal *CAPP* values for more than one node then node selection is random.
- If the strategy also considers the situation where idle nodes exists, then; if for two or more nodes *CAPP* is equal, and some of these nodes is in idle[2] state (*IS*) then the process is allocated to that idle node. This second decision attempts to balance the workload.

For this allocation criterion, with N system nodes, 4+5N binary inputs will suffice to depict process requirements (4 bits) and system state (5 bits per system node), while the size of total pattern space is given by:

$$T = [\,3^{2(N+1)}2^{N}\,] - [\,3^{N+1}(2^{2N} + 2^{N})]$$

Because only legal inputs conform a training set for the neural net, the second term of *T* excludes the cases in which *MR* is greater than *CMC* available.

4. PARALLEL LEARNING IMPLEMENTATION

4.1 THE ALGORITHM

The basic steps of a parallel backpropagation learning algorithm using variable-epoch regime is depicted below. We recall that, under these approaches, during a number (one ore more) of epochs, the submission of all patterns in the partition, the corresponding computations and the accumulation of weight changes must be performed before weights update takes place, then the next epoch interval begin.
The Parallel Training Algorithm

Repeat
1. For each pattern
   1.1 Compute the output of units in the hidden layer.
   1.2 Compute the output of units in the output layer
   1.3 Compute error terms for the units in the output layer.
   1.4 Compute error terms for the units in the hidden layer.
   1.5 Compute weight changes in the output layer.
   1.6 Compute weight changes in the hidden layer.
2. Exchange of accumulated weight vectors
   2.1 If epoch interval was reached then
      send local hidden weight vectors and output weight vectors.
   2.2 Receive remote hidden weight vectors and output weight vectors.

[2]A node is defined as being in an idle state when no user process is running.

3. Update weights changes in the output layer.
4. Update weights changes in the hidden layer.

Until (current error < max. accept. err.) or (number of iterations = maximum number of iterations)

Here a parent process spawns several BP processes with the corresponding parameters.
Each process during the propagation phase, if epoch interval was reached, the accumulated weight change vector is broadcast to others processes and remote accumulated weight change vectors are received from other processes. The reception is not blocking, since if nothing has arrived, the children go ahead. Finally, each BP process performs the adaptation phase and completes one epoch.
Each children finish when the current error of the neural network is less than the maximum accepted error or the number of iterations is greater than the admissible number of iterations.

4.2 ALTERNATIVE SUPPORTS

In previous works[5][6] a real implementation was built on the processors distributed in a LAN of workstations (multicomputers). Each process ran in a workstation. The routines used a socket interface as an abstraction of IPC (Interprocess Communication) mechanism [3][4][16][17].

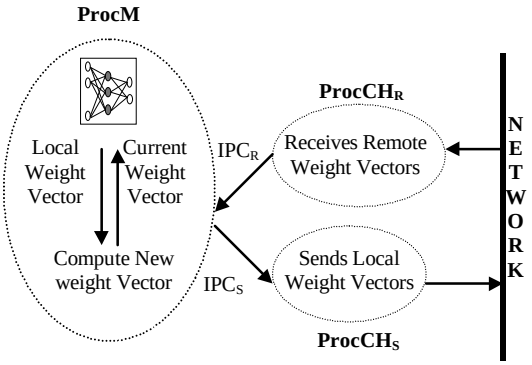


Fig. 3. A node in a System Architecture for real processors.

A UDP protocol was chosen because we were working in a reliable LAN and even, if a package missing happens the learning process is not be sensitively affected (each process will update the weights with the packages received). Figure 3 depicts the underlying system architecture and procedures supporting the parallel learning process for this approach.

The current work with PVM is discussed now. PVM is created to link computing resources and provide to the user with a parallel platform for running their computer applications, independent of the number of processors[18][19]. PVM supports a very efficient message-passing model.

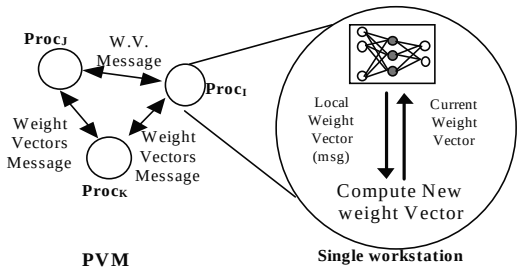


Fig. 4. System Architecture for PVM

Figure 4 shows an alternative support to implement our particular application on Parallel Virtual Machine.

5. EXPERIMENTS DESCRIPTION

Experiments covered here refer solely to the variable-epoch training regime. The variable number of epochs locally performed before any exchange took place was chosen as a random number r between 1 and 15. Better results were observed for greater r values, but reported results corresponds to average values.
Let be *S* the total pattern space. The processors training sets (*ts*) were subsets of *S*.
For sequential training, the training set *Ts* was built by uniform selection of X% of the pattern space *S.*
For parallel training the pattern space was divided into *n* subsets and each subset was assigned to one processor (virtual or not) in parallel execution. The training subsets ts_i were built by uniform selection of *(X/n)%* of the pattern space *S.*

Values for *X* was chosen as 30 and 60.

In what follows the experiment identifiers indicate:
<Training type>-<size(%) of Training Set>/<number of subsets for parallel execution>

To compare results, the neural net was trained sequentially (i), in parallel using socket (ii) and in parallel using PVM (iii):

- (i) Experiments **SBP-X/1:** SBP-30/1 and SBP-60/1. *Size(Ts)* = 30% and 60% of *S* respectively.

- (ii) Experiments **PBP-X/n:** PBP-30/3 and PBP-60/3. Three disjoint subsets of *(X/n)%* of *S* were selected and each subset was assigned to one processor in different workstations. Size(ts_i) = 10% and 20% of *S* respectively.

- (iii) Experiments **PVM-X/n:** PVM-30/3, PVM-30/6, PVM-60/3 and PVM-60/6**.** The software allows any numbers of processors to be created without any relationship to the number of real processors. In this state three and six disjoint subsets of *(X/n)%* of *S* were selected respectively, and each subset was assigned to one process. The number *n* of parallel processes was set to 3 and 6. *Size(ts_i)* = 10%, 5%, 20% and 10% of *S* respectively.

As we were working in two stages (learning and testing), the following *parameters* were used in each case:

- For Sequential Processing, *Ts* (the training set of the unique neural network) was used on the learning stage.
- For Parallel Processing, on the learning stages, ts_i was the local training subset submitted to the BP_i, with the accumulated weight changes vectors received from other BP_j networks (with training subset ts_j, $j \neq i$). For that reason, $Ts = ts_1 \cup ts_2 \cup ts_3 \cup ... \cup .ts_n$ was the training set for all BP_i networks at the learning stages.

In both cases *Ts* and *S* (the whole sample space) were used in the testing stage (for Recall and Generalisation respectively).

Figure 5 shows an example of parallel partitioning scheme for experiments SBP-X/1, PBP-X/3 and PVM-X/3:

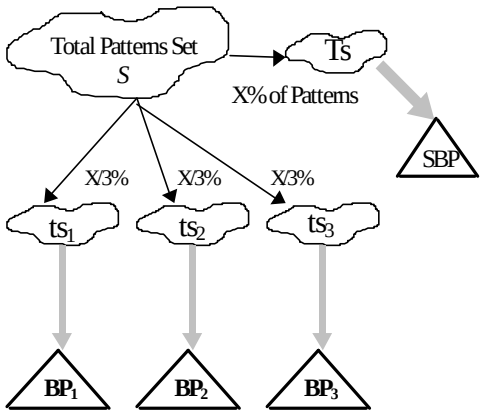


Fig. 5 - The Partitioning Approach for three processes in parallel.

During these processes, the following relevant *performance variables* were examined:

Training process:

L_T: Learning time, is the running time of the learning algorithm.

N_{iter}: Number of iterations needed to reach an acceptable error value while training.

Testing process:

R = rcg/Size(Ts). Is the recall ability of the neural net. Where *rcg* is the total number of patterns recognized when only patterns belonging to the Training Set (Ts) are presented, after learning, to the network. The objective is to analyse if each net can assimilate (can acquire) the learning of other networks that were running in parallel with it.

G = gnl/Size(*S*). Is the combined recall and generalization ability. Where *Size(S)* is the size of the Total Pattern Space and *gnl* is the total number of patterns recognized when all possible patterns are presented, after learning, to the network.

Sp = $L_{Tapproach1} / L_{Tapproach2}$ is the ratio between the learning times under different approaches (sequential or parallel).

$Rec_{B/C}$ = $Sp/(R_{seq} - R_{par})$ is the benefit-cost ratio for recall. It indicates the benefit of speeding up the learning process, which is paid by the cost of (possibly) loosing recall ability.

$Gen_{B/C}$ = $Sp/(G_{seq} - G_{par})$ is the benefit-cost ratio for generalisation. It indicates the benefit of speeding up the learning process, which is paid by the cost of (possibly) loosing generalisation ability.

6. RESULTS

As we previously said, the neural net was trained sequentially (SBP), in parallel using socket (PBP) and in parallel using PVM (PVM). The corresponding mean values of the performance variables are shown in the following figures and tables.

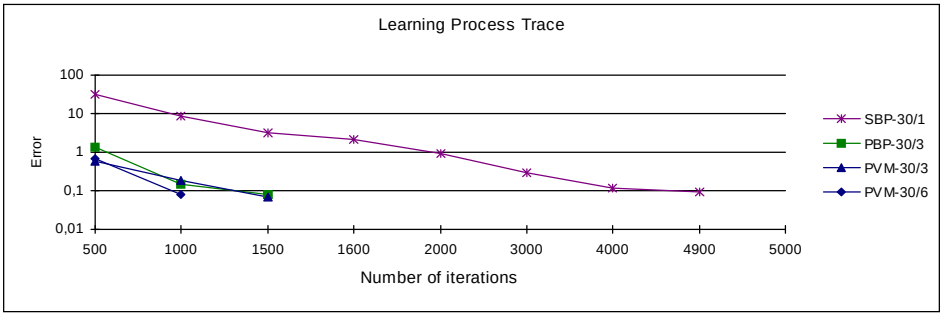


Fig. 6 - Number of iteration needed under sequential and parallel processing (best case)

As we can observe in figure 6 a reduction, greater than one third in the number of iteration needed to achieve permissible error values, was achieved. Results for the partitioning scheme of 30% are shown but using either parallel partitioning approach attains similar results.

Table 1, is a summary of the experiments performed and their results:

Experiment	Learning Time	Recall %	Generalisation %
SBP-30/1	1989	100	98
PBP-30/3	426.91	97-85	96.84
PVM-30/3	123.33	97.84	95.71
PVM-30/6	53.05	92.8	92
SBP-60/1	5996	100	99.5
PBP-60/3	1137.66	99.33	98.83
PVM-60/3	275.66	99.91	99.73
PVM-60/6	87.22	98.97	.97.64

Table 1 – Summary of L_T, R y G results. L_T expresses in seconds while R y G are expressed in percentile values.

Figures 7 and 8 show the associated loss in recall and generalisation of the neural network for different sizes of the training set.

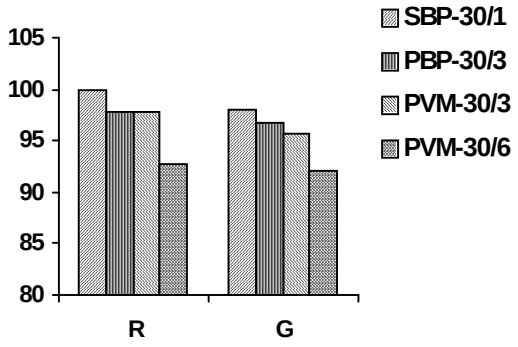


Fig. 7- Values of Recall and Generalisation for Ts = 30 % of *S*

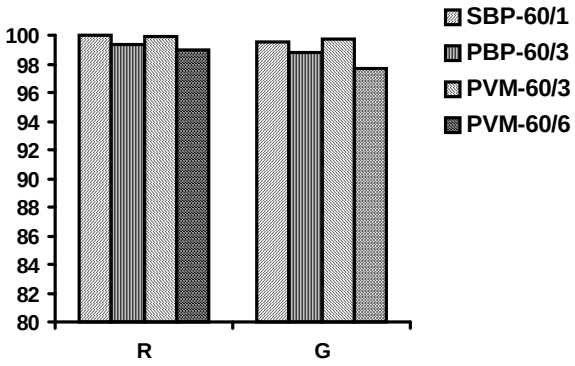


Fig. 8 - Values of Recall and Generalisation for Ts = 60% of *S*

In general, the detriment of recall and generalisation capabilities decreases as the training set size is incremented. Their values range from 0.09% (PVM-60/3) to 7.2% (PVM-30/6) for recall. In the case of generalisation the values range from -0.23% (PVM-60/3) to 6% (PVM-30/6). Opposite to the expected, in this case, an improvement was also detected: PVM-60/3 achieved a generalisation capability better than the sequential BP.

Table 2 indicates the speed-up in learning time attained through parallel processing when different sizes of the portions of the total pattern space *S* are selected for training the neural network. Table 2(a) shows the ratio between the sequential learning and the parallel learning times ($S_P = L_{T(S)} / L_{T(P)}$).

SBP-30/1 vs.			SBP-60/1 vs.		
PBP-30/3	PVM-30/3	PVM-30/6	PBP-60/3	PVM-60/3	PVM-60/6
4.65	16.12	37.49	5.27	21.75	68.74

Table 2(a) - Speed-up values achieved through parallel processing vs. sequential processing

In general, as the size of *Ts* increases (from 30% to 60%) then an increment of the speed-up can be observed under variable-epoch training.

This effect shows a substantial improvement over the per-epoch approach used in earlier implementations. Moreover, increment of the speed-up can be observed among different parallel implementations. Table 2(b) shows the ratio between both parallel learning times ($S_{PVM} = L_{T(PBP)} / L_{T(PVM)}$).

PBP-30/3 vs.		PBP-60/3 vs.	
PVM-30/3	PVM-30/6	PVM-60/3	PVM-60/6
3.46	8.04	4.12	13.04

Table 2(b) - Speed-up values achieved through parallel processing with PVM vs. parallel processing with Socket

Both parallel implementations showed comparable capability, but PVM-X achieved a substantial increment in speed-up with values ranging from 3.46 to 13.04 times faster than PBP-X.

It is interesting to observe in table 3 the Benefit-Cost Ratio, which gives an indication of a speed-up *Sp* obtained at the cost of a detriment in recall or generalisation. Table 3(a) shows for PVM-X, the benefit-cost ratio for recall ability

$$\mathbf{Rec_{B/C}} = Sp/(R_{seq} - R_{parPVM})$$

and table 3(b) shows the benefit-cost ratio for generalisation ability

$$\mathbf{Gen_{B/C}} = Sp/(G_{seq} - G_{parPVM}).$$

SBP-30/1 vs. PVM- 30/3	SBP-30/1 vs. PVM-30/6	SBP-60/1 vs. PVM- 60/3	SBP-60/1 vs. PVM-60/6
13.96	30.29	21.66	67.71

Table 3(a) - Benefit-Cost Ratio for Recall.

SBP-30/1 vs. PVM- 30/3	SBP-30/1 vs. PVM-30/6	SBP-60/1 vs. PVM- 60/3	SBP-60/1 vs. PVM-60/6
7.03	6.24	--	37.35

Table 3(b) - Benefit-Cost Ratio for Generalisation

In all cases, it can be observed good ratios between benefits and costs. In the particular case of PVM-60/3 an increment of speed-up was simultaneously detected with an increment in generalisation capability, hence the benefit cost ratio is not registered. This performance variable is of great help to inspect the goodness of a parallel design for training neural nets.

7. CONCLUSIONS

A preliminary set of experiments in our investigation of parallel training of neural networks, using a pattern-partitioning approach, revealed that the beneficial effects of parallel processing can be achieved with minor capability loss.

Using this technique, in this work we presented feasible alternative architectures for a system supporting parallel learning of backpropagation neural networks implementing a neural network device for automatic task allocation in computer systems.

We discussed and showed results of an improved implementation using a Parallel Virtual Machine approach and a new variant called variable-per-epoch approach.
Most recent results were contrasted against sequential and parallel approaches previously implemented. In the parallel case the variable-per-epoch and the per-epoch approaches were compared showing better performance for the new variant.

Furthermore, we need to remark that PVM provided us a unified framework within which our parallel application was developed in an efficient and clear manner. That resulted in a straightforward program structure and very simple implementation. PVM transparently manipulated all message routing, synchronization aspects, data conversion, message packing and unpacking, process group manipulation and all aspect regarding heterogeneity. All these factors contributed to reduced development and debugging time. It worth remarking that a more effective implementation of parallel backpropagation neural network was completed.

Finally, at the light of the effectiveness showed by the distributed approach for the parallel learning process by means of PVM, at the present time, testing with larger number of processors and different training set sizes are being performed for different neural networks.

8. ACKNOWLEDGEMENTS

We acknowledge the cooperation of the project group for providing new ideas and constructive criticisms. Also to the Universidad Nacional de San Luis the CONICET and the ANPCYT from which we receive continuous support.

9. REFERENCES

[1] Berman F., Snyder L. - *On mapping parallel algorithms into parallel architectures*- Parallel and Distributed Computing, pp 439-458, 1987.

[2] Cena M., Crespo M. L., Gallard R. *Transparent Remote Execution in LAHNOS by Means of a Neural Network Device*. Operating System Reviews, Vol. 29, Nro 1, ACM Press, 1995.

[3] Colouris G., Dollimore J., Kindberg T. -*Distributed Systems: Concept and Design* - Addison-Wesley, 1994.

[4] Comer, D. E., Stevens, D. L. - *Internetworking with TCP/IP* - Vol. III - Prentice Hall.

[5] Crespo M., Píccoli F., Printista M., Gallard R.- *A Parallel Approach for Backpropagation Learning of Neural Networks*- Proceedings of 3er Congreso Argentino de Ciencias de la Computación, Universidad Nacional de La Plata, Vol. 1., pp 145 – 156,, September 1997.

[6] Crespo M., Píccoli F., Printista M., Gallard R.- *Parallel Shaping of Backpropagation Neural Networks in Workstations-Based Distributed Systems*- Proceedings of International ICSC Syposium on Engineering of Intelligent Systems – EIS' 98, University of La Laguna, Vol. 2., pp 334 – 340, Tenerife, Spain, February 1998.

[7] Foster, Ian T. : *Designing and Building Parallel Programs* - Addison Wesley, 1995.

[8] Freeman, J., Skapura, D. *Neural Networks. Algorithms, Applications and Programming Techniques*. Addison-Wesley, Reading, MA, 1991.

[9] Girau, B. - *Mapping Neural Network Back-Propagation onto Parallel Computers with Computation/Communication Overlapping*. Proceedings of Euro Par'95.

[10] Kumar, V., Shekhar, S., Amin, M.- A *Scalable Parallel Formulation of the*

Backpropagation Algorithm for Hypercubes and Related Architectures. IEEE transactions on Parallel and Distributed Systems, Vol. 5. Nro.10, pp 1073 - 1090, October 1994.

[11] McEntire, P. L., O'Reilly, J. G., Larson, R. E. (Editors) : *Distributed Computing: Concepts and Implementations* - Addison Wesley, 1984 .

[12] Petrowski, A., Dreyfus, G., Girault, C.- *Performance Analysis of a Pipelined Backpropagation Parallel Algorithm*. IEEE. Trasactions Networks, Vol. 4, pp 970 - 981, November 1993.

[13] Plaut, D., Nowlan, S., Hinton, G. *Experiments on Learning by Backpropagation*. Tech. Report, CMU-CS-86-126, Carniege Mellon University, Pittsburg, PA, 1986.

[14] Rumelhart, D., Hinton, G., Willams, R. *Learning Internal Representations by Error Propagation.* MIT Press, Cambridge, Massachusetts, 1986.

[15] Rumelhart, D., McClelland, J. *Parallel Distributed Processing, vol. 1 y 2*. MIT Press, Cambridge, MA, 1986.

[16] Stevens, R.W.- *Advanced Programming in the UNIX Envionment*- AdDison-Weslwy Publishing Company. 1992.

[17] Stevens, R.W.- *UNIX Network Programing.* Prentice Hall-Englewood Cliff. 1990.

[18] Sunderam, V., Manchek, R., Jiang,, W., Dongara, J., Bengeuelin, A., Geist, A. – *PVM3 – User' s Guide and Reference Manual* – Oak Ridge National Laboratory: Tenesse, 1994.

[19] Sunderam, V., Manchek, R., Jiang,, W., Dongara, J., Bengeuelin, A., Geist, A. – *PVM: Parallel Virtual Machine* – The MIT Press, Cambridge, Massachusetts, 1994.

[20] Tanembaun, A.. *Modern Operating Systems*-Prentice Hall 1992-