PARALLEL COUNTER – PROPAGATION NETWORKS

by
Athanasios I. Margaris, Efthimios Kotsialos

Abstract: The objective of this research is to construct parallel models that simulate the behavior of artificial neural networks. The type of network that is simulated in this project is the counter – propagation network and the parallel platform used to simulate that network is the message passing interface (MPI). In the next sections the counter – propagation algorithm is presented in its serial as well as its parallel version. For the latter case, two approaches are presented, one that is based to the concept of the inter-communicator and one that uses remote access operations for the update of the weight tables and the estimation of the mean error for each training stage.
Keywords: Neural networks, counter – propagation, parallel programming, message passing interface, communicators, process groups, point to point and collective communication.

Introduction

As it is well known, one of the major drawbacks of the artificial neural networks is the time consumption and the high cost associated with their learning phase [1]. These disadvantages, combined with the natural parallelism that characterizes the operation of these structures, force the researchers to use the hardware parallelism technology to implement connectionist models that work in a parallel way [2]. In these models, the neural processing elements are distributed among independent processors and therefore, the inherent structure of the neural network is distributed over the workstation cluster architecture. Regarding the synapses between the neurons, they are realized by suitable connections between the processes of the parallel system [3].

A parallel neural network can be constructed using a variety of different methods [4-9], such as the parallel virtual machines (PVM) [10], the message passing interface (MPI) [11-13], the shared memory model and the implicit parallelization with parallel compiler directives [14]. Concerning the network types that have been paralellized by one of these methods, they cover a very broad range from the supervised back propagation network [15-17] to the unsupervised self-organizing maps [18-19]. In this research the counter – propagation network is parallelized by means of the message passing interface library [13].

The serial counter – propagation algorithm

Counter-propagation neural networks [20] were developed by Robert Hecht-Nielsen as a means to combine an unsupervised Kohonen layer with a teachable output layer known as Grossberg layer. The operation of this network type is very similar to that of the Learning Vector Quantization (LVQ) network in that the middle (Kohonen) layer acts as an adaptive look-up table.

The structure of this network type is shown in Figure 1. From this figure it is clear that the counter-propagation network is composed of three layers: an input layer that reads input patterns from the training set and forwards them to the network, a hidden layer that works in a competitive fashion and associates each input pattern with one of the hidden units, and the output layer which is trained via a teaching algorithm that tries to minimize the mean square error (MSE) between the actual network output and the desired output associated with the current input vector. In some cases a fourth layer is used to normalize the input vectors but this normalization can be easily performed by the application (i.e. the specific program implementation), before these vectors are sent to the Kohonen layer.

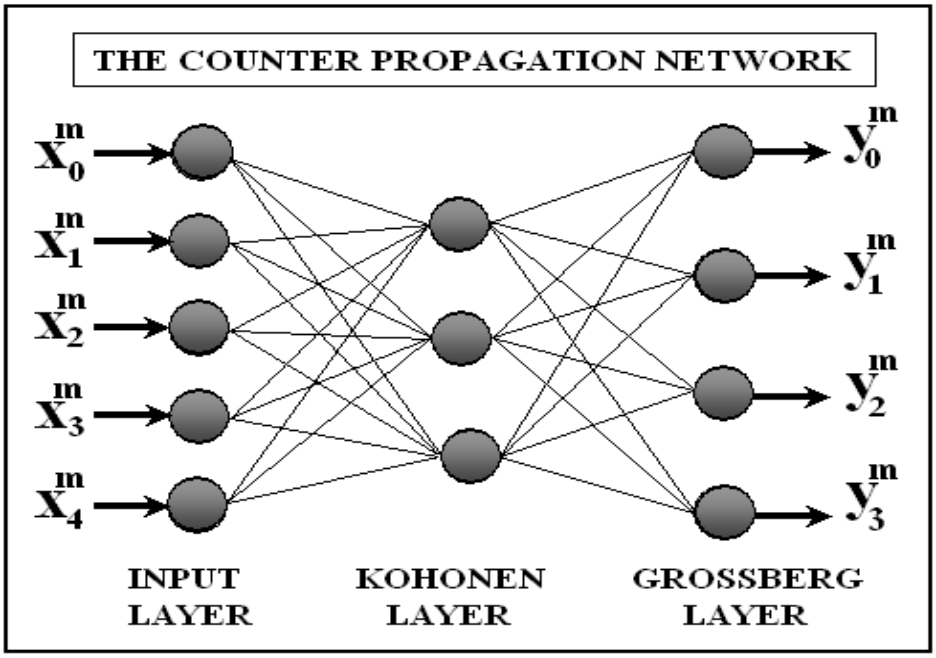

Figure 1: A typical counter – propagation network

Regarding the training process of the counter-propagation network, it can be described as a two-stage procedure: in the first stage the process updates the weights of the synapses between the input and the Kohonen layer, while in the second stage the weights of the synapses between the Kohonen and the Grossberg layer are updated. In a more detailed description, the training process of the counter – propagation network includes the following steps:

STAGE A ➔ Performs the training of the weights from the input to the hidden nodes

STEP 00: The synaptic weights of the network between the input and the Kohonen layer are set to small random values in the interval [0, 1].

STEP 01: A vector pair (x, y) of the training set, is selected in random

STEP 02: The input vector x of the selected training pattern is normalized

STEP 03: The normalized input vector is sent to the network

STEP 04: In the hidden competitive layer the distance between the weight vector and the current input vector is calculated for each hidden neuron j according to the equation

$$D_j = \sqrt{\sum_{i=1}^{K} (x_j - w_{ij})^2} \ ,$$

where K is the number of the hidden neurons and w_{ij} is the weight of the synapse that joins the i_{th} neuron of the input layer with the j_{th} neuron of the Kohonen layer.

STEP 05: The winner neuron W of the Kohonen layer is identified as the neuron with the minimum distance value D_j.

STEP 06: The synaptic weights between the winner neuron W and all M neurons of the input layer are adjusted according to the equation

$$W_{wi}(t+1) = W_{wi}(t) + \alpha(t)(x_i - W_{wi}(t)).$$

In the above equation the α coefficient is known as the Kohonen learning rate. The training process starts with an initial learning rate value α_0 that is gradually decreased during training according to the equation

$$\alpha(t) = \alpha_o(1 - \frac{t}{T}),$$

where T is the maximum iteration number of the stage A of the algorithm. A typical initial value for the Kohonen learning rate is a value of 0.7.

STEP 07: The steps 1 to 6 are repeated until all training patterns have been processed once. For each training pattern p the distance D_p of the winning neuron is stored for further processing. The storage of this distance is performed before the weight update operation.

STEP 08: At the end of each epoch the training set mean error is calculated according to the equation

$$E_i = \frac{1}{P}\sum_{k=1}^{P} D_k ,$$

where P is the number of pairs in the training set, D_k is the distance of the winning neuron for the pattern k and i is the current training epoch.

The network converges when the error measure falls below a user supplied tolerance value. The network also stops training in the case where the specified number of iterations has been performed, but the error value has not converged to a specific value.

STAGE B ➔ Performs the training of the weights from the hidden to the output nodes

STEP 00: The synaptic weights of the network between the Kohonen and the Grossberg layer are set to small random values in the interval [0, 1].

STEP 01: A vector pair (x, y) of the training set, is selected in random

STEP 02: The input vector x of the selected training pattern is normalized

STEP 03: The normalized input vector is sent to the network

STEP 04: In the hidden competitive layer the distance between the weight vector and the current input vector is calculated for each hidden neuron j according to the equation

$$D_j = \sqrt{\sum_{i=1}^{K} (x_j - w_{ij})^2} ,$$

where K is the number of the hidden neurons and w_{ij} is the weight of the synapse that joins the i_{th} neuron of the input layer with the j_{th} neuron of the Kohonen layer.

STEP 05: The winner neuron W of the Kohonen layer is identified as the neuron with the minimum distance value D_j. The output of this node is set to unity while the outputs of the other hidden nodes are assigned to zero values.

STEP 06: The connection weights between the winning neuron of the hidden layer and all N neurons of the output layer are adjusted according to the equation

$$W_{jw}(t+1) = W_{jw}(t) + \beta(y_j - W_{jw}(t))$$

In the above equation the β coefficient is known as the Grossberg learning rate

STEP 07: The above procedure is performed for each training pattern. In this case the error measure is computed as the mean Euclidean distance between the winner node's output weights and the desired output, that is

$$E = \frac{1}{P}\sum_{j=1}^{N} D_j = \frac{1}{P}\sum_{j=1}^{P}\sum_{k=1}^{N}\sqrt{(y_k - w_{kj})^2}$$

As in stage A, the network converges when the error measure falls below a user supplied tolerance value. The network also stops training after exhausting the prescribed number of iterations.

The parallel counter – propagation algorithm

The parallelization of the counter propagation algorithm presented in this paper is based on the message-passing interface (MPI) standard [13], which enables a set of processes to run concurrently on the same or different processors, and to exchange messages between each other. To simulate the counter propagation network, a separate process is used to model the behavior of each neuron [14]. This fact leads to a number of processes P equal to M+K+N where M is the number of the input neurons, K is the number of the Kohonen neurons and N is the number of the Grossberg neurons, respectively.

Since the number of the parameters M, K and N is generally known in advance, we can assign to each process a specific color. The processes with ranks in the interval [0, M-1] are associated with an "input" color; the processes with ranks in the interval [M, M+K-1] are associated with a "Kohonen" color, while the processes with ranks in the interval [M+K, M+K+N-1] are associated with a "Grossberg" color. Having assigned to each process one of these three color values, we can divide the process group of the default communicator MPI_COMM_WORLD into three disjoint process groups, by calling the function MPI_Comm_split with arguments (MPI_COMM_WORLD, color, rank, &intraComm). The result of this function is the creation of three process groups – the input group, the Kohonen group and the Grossberg group; each one of them simulates the corresponding layer of the counter propagation network. The size of each group is identical to the number of neurons of the corresponding layer, while the communication between the processes of each group is performed via the intracommunicator intraComm, created by the MPI_Comm_split function. The division of the initial process group in this arrangement is shown in Figure 2.

After the creation of the three process groups, we have to setup a mechanism for the communication between them. In the message-passing environment, this communication is performed via a special communicator type known as intercommunicator that allows the communication of process groups. In our case, we have to setup one intercommunicator for the message passing between the processes of the input group and the Kohonen group, and a second intercommunicator for the communication between the processes of the Kohonen group and the Grossberg group. The creation of these intercommunucators, identified by the names interComm1 and interComm2 respectively, is based on the MPI_Intercomm_create function and the result of the function invocation is shown in Figure 3.

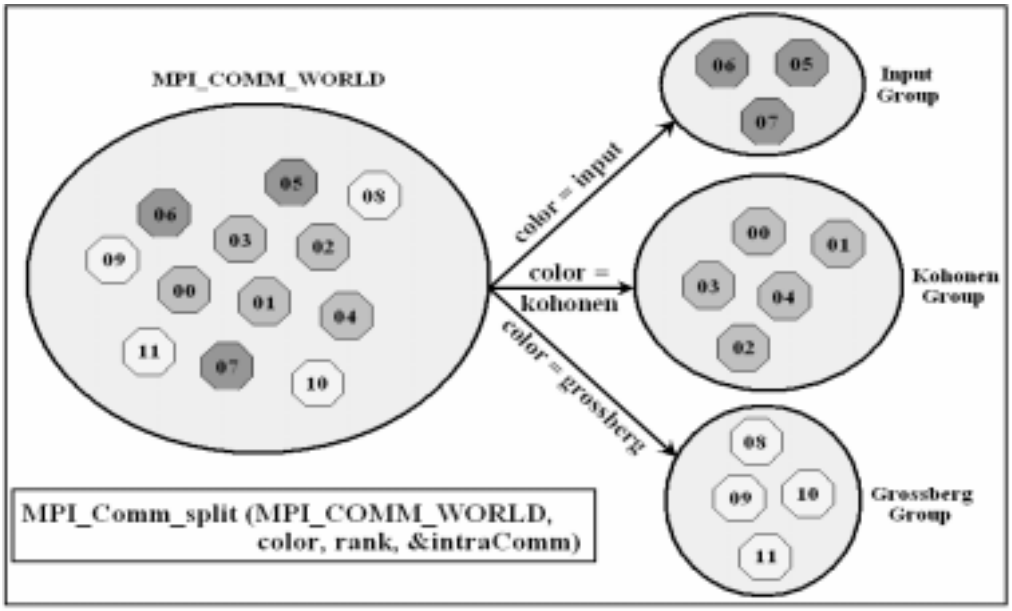

Figure 2: The division of the initial process group to the input, Kohonen and Grossberg sub-groups of processes

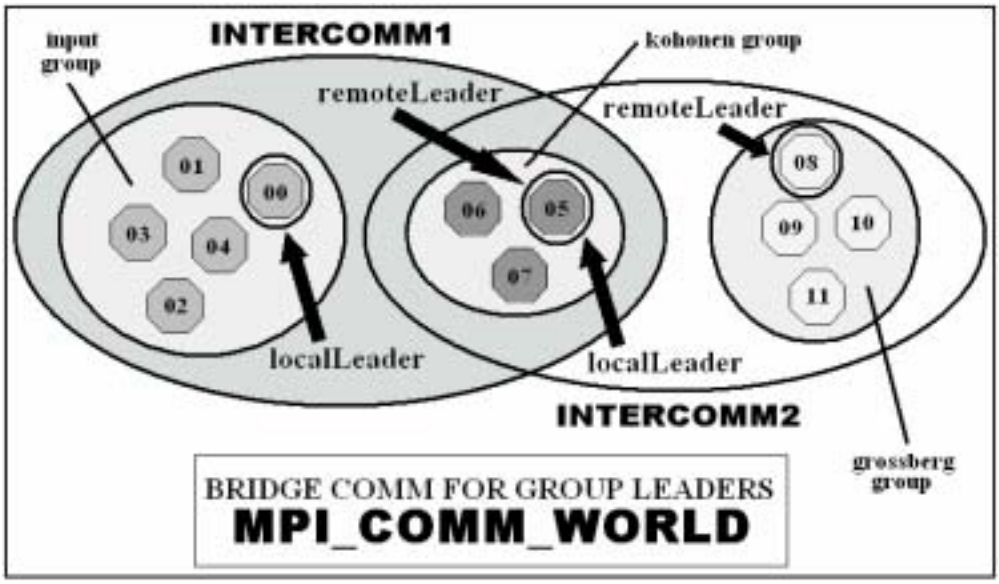

Figure 3: The message passing between the three process groups is performed via the intercommunicators interComm1 and interComm2

After the construction of the communication system, the training algorithm can be easily performed. In the first step the training set data are passed to the processes of the input and the output group according to the next figure, fig. 4. Since the number of input processes is equal to the size of the input vector, each process reads a "column" of the training set that contains the values of the training patterns with a position inside the input vectors equal to the rank of each input process. The distribution of the output vector values to the processes of the output group is performed in a similar way. The distribution of the pattern data to the system processes is based to the MPI I/O functions and to the establishment of a different file type and file view for each input and output process.

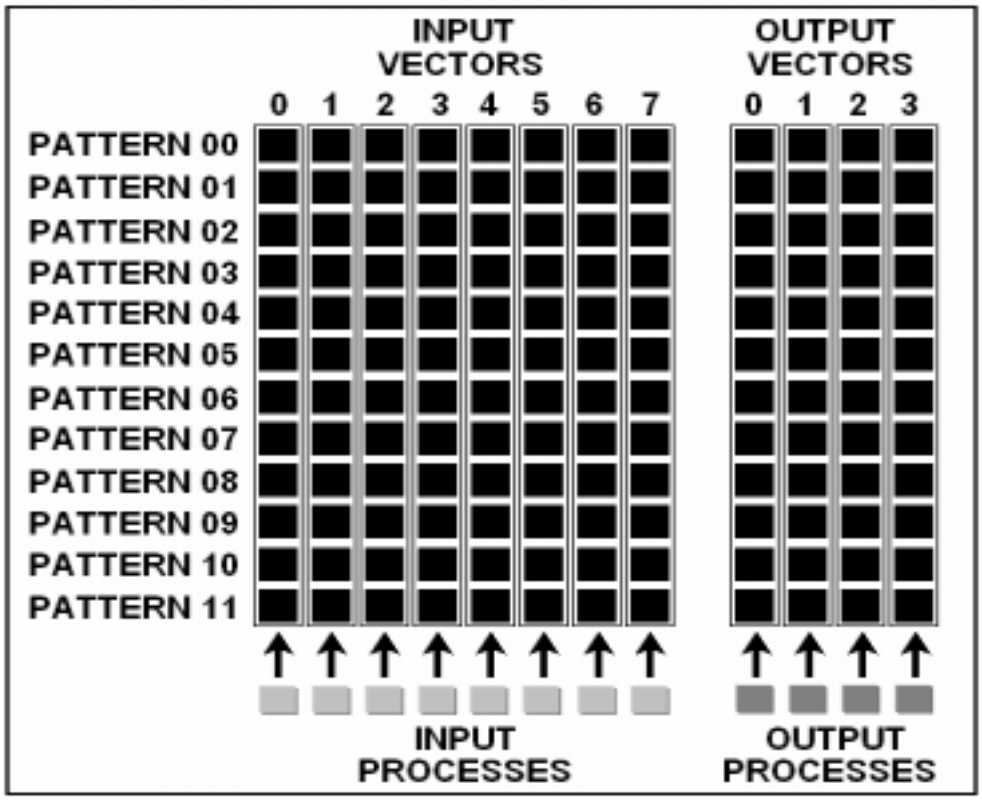

Figure 4: The distribution of the training set data to the input and the output processes for a training set of 12 training patterns with 8 inputs and 4 outputs

After the distribution of the training set data, the two stages of the counter propagation training can now be performed. The steps described in the following description are performed for each training cycle and for each pattern of the training set. In this description, the notation P_n is used to denote the process with a rank value equal to n.

STAGE A ➔ Performs the training of the weights from the input to the Kohonen processes

STEP 0: A two dimensional K x M matrix that contains the synaptic weights between the input and the Kohonen process group is initialized by process P_0 to small random values in the interval [0, 1] and is broadcasted by the same process to the processes of the default communicator MPI_COMM_WORLD. A similar initialization is done for a second matrix with dimensions M x N that contains the synaptic weight values between the Kohonen and the Grossberg process groups.

STEP 1: Process P_0 of the input group picks up a random pattern position that belongs in the interval [0, PAIRS-1] where PAIRS is the number of the training vector pair. Then, this value is broadcasted to all processes that belong to the input group. This broadcasting operation is performed by a function invocation of the form MPI_Bcast (&nextPattern, 1, MPI_DOUBLE, 0, intraComm). At this stage we may also perform a normalization of the data set.

STEP 2: Each function calls MPI_Bcast to read the next pattern position and then retrieves from its local memory the input value associated with the next pattern. Since the distribution of the training set data is based in a "column" fashion (see Figure 4), this input value is equal to the inputColumn[nextPattern] where the inputColumn vector contains the $(rank)_{th}$ input value of each training pattern. The steps 1 and 2 of the parallel counter propagation algorithm are shown in the next figure (Figure 5).

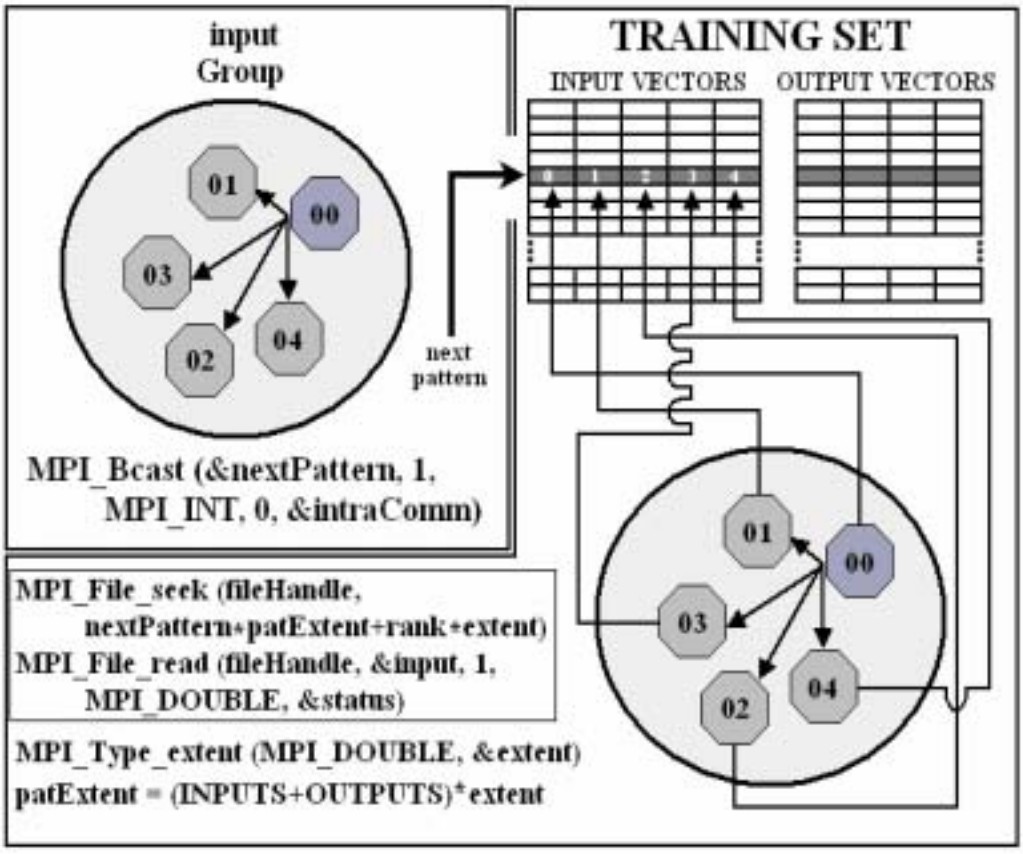

Figure 5: The retrieval of the training pattern input values from the processes of the input group

STEP 3: After the retrieval of the appropriate input value of the current training pattern, each process of the input group sends its value to all processes of the Kohonen group. This operation simulates the full connection architecture of the actual neural network and it is performed via the MPI_Alltoall function that is invocated with arguments (&input, 1, MPI_DOUBLE, inputValues, 1, MPI_DOUBLE, interComm1). Since this operation requires the communication of processes that belong to different groups, the message passing function is performed via the intercommunicator interComm1, which is used as the last argument in the function MPI_Alltoall. An alternative (and apparently slower) way is to force input process P_0 to gather these values and to send them via the inter-communicator interComm1 to the group leader of the

Kohonen group, which, in turn, will pass them to the Kohonen group processes. However, this alternative approach is necessary, if the training vectors are not normalized. In this case, the normalization of the input and the output vectors has to be performed by the group leaders of the input and the Grossberg groups before their broadcasting to the appropriate processes.

STEP 4: The next step of the algorithm is performed by the units of the Kohonen layer. Each unit calculates the Euclidean distance between the received vector of the input values and the appropriate row of the weight table that simulates the corresponding weight vector. After the estimation of this distance, one of the Kohonen group processes is marked as the root process to identify the minimum input weight distance, and the process that corresponds to it. This operation simulates the winning neuron identification procedure of the counter propagation algorithm. This identification is performed by the MPI_Reduce collective operation, which is called with the value MPI_MINLOC as the opcode argument. The minimum distance for each training pattern is stored in a buffer, later to participate to the calculation of the mean winner distance of the current training epoch.

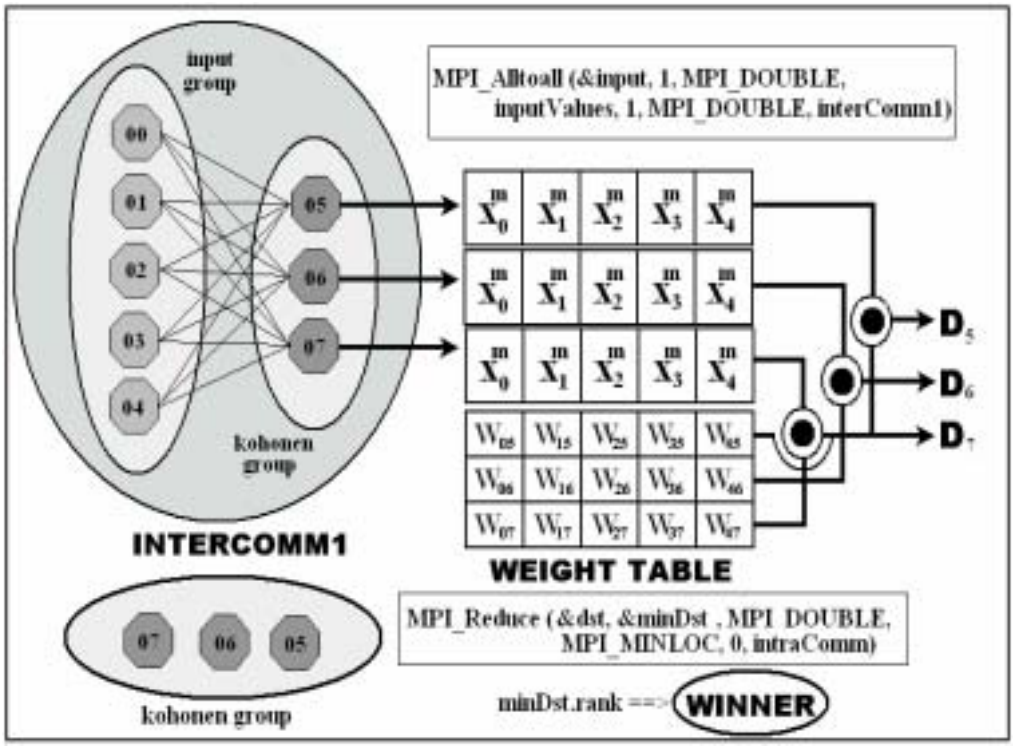

Figure 6: The identification of the winning process from the processes of the Kohonen group

STEP 5: The winning process updates the weights of its weight table row, according to the equation

$$W_{wi}(t+1) = W_{wi}(t) + \alpha(x_i - W_{wi}(t)),$$

which is used as in the case of the previous network implementation. In this step, the Kohonen learning rate α is known to all processes, but it is used only by the winning process of the Kohonen group to perform the weight update operation described above. This learning rate is gradually decreased at each iteration, as in the serial algorithm. Since each process uses its own local copy of the weight table, the table with the new updated values is broadcasted to all the processes of the Kohonen group. The weight update operation by the winning process is shown in the next figure, fig. 7.

The previously described steps are performed iteratively for each training pattern and training cycle. The algorithm will terminate when the mean winner distance falls below the predefined tolerance or when the number of iterations reaches the maximum iteration number.

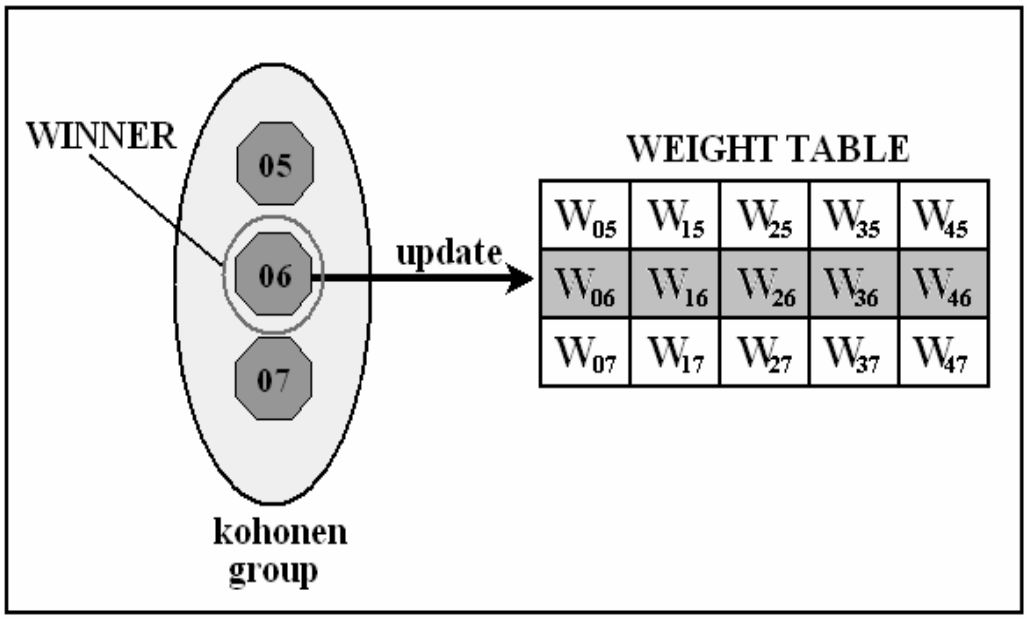

Figure 7: The update of the synaptic weights associated with the winning process

STAGE B ➔ Performs the training of the weights from the Kohonen to the Grossberg nodes

STEP 0: Process P_0 of the input group picks a random pattern position and broadcasts it to the processes of the input group.

STEP 1: Each process of the input group calls the MPI_Bcast function to read the next pattern position. Then it retrieves this position from the inputColumn local vector and by using the MPI_Alltoall function sends it to the set of processes that belong to the Kohonen group.

STEP 2: Each process of the Kohonen group calculates the distance between the current input vector and the associated weight vector – this vector

is the R_{th} row of the input – Kohonen weight matrix where R is the rank of the Kohonen process in the Kohonen group. Then one of the Kohonen processes is marked as the root process to identify the minimum distance and the process associated with it. The identification of this distance is based to the MPI_Reduce collective operation. The process with the minimum distance value is marked as the winner process. The output of this winner process is set to unity, while the outputs of the remaining processes is set to zero.

STEP 3: Each Kohonen process sends its output to the set of processes of the Grossberg group via the MPI_Alltoall inter-communicator function. Then, each output process calculates its own output according to the equation

$$O_j = \sum_{i=1}^{K} X_j W_{ij}$$

In this equation we use the notation X_j to denote the inputs of the Grossberg processes – this inputs are coming from the Kohonen processes and therefore their values are 1 for the winning process and 0 for the remaining processes, while W_{ij} are the weights associated with the j_{th} output process. These weights belong to the j_{th} row of the Kohonen-Grossberg weight matrix. After the calculation of the output of each Grossberg process we estimate the Euclidean distance between the real output vector (O_0, O_1, O_2, …, O_{N-1}) and the desired output vector (Y_0, Y_1, Y_2, …, Y_{N-1}). The stage B is completed when the mean error value for each training epoch falls below a user – supplied tolerance or when the number of iterations reaches the predefined maximum iteration number. Regarding the weigh update operation, this is applied only to the weights of the winning process of the Kohonen layer in the Kohonen – Grossberg weight matrix. The weight update operation is performed as it is shown in Figure 7 but for the Kohonen – Grossberg matrix and it is based to the equation

$$W_{jw}(t+1) = W_{jw}(t) + \beta(y_j - W_{jw}(t)),$$

which was used also in the case of the serial algorithm. The β constant in the above equation is known as the Grossberg learning rate – a typical value of this parameter is 0.1.

The recall phase

In the recall phase each input pattern is presented to the network. In the hidden layer the winning neuron is identified, its output is set to unity (while the outputs of the remaining neurons are set to zero), and, finally, the network output is calculated according to the algorithm described above. Then the real network output is estimated and the error between it and the desired output is identified. This procedure is applied to training patterns that belong to the training set and are presented to the network for testing purposes, while for unknown patterns, they are sent to the network, to calculate the corresponding output vector. This procedure can be easily modified to work with the parallel network, by adopting the methods described above for the process communication. It is supposed that the unknown patterns will be read from a pattern file with a similar organization as the training set file – in this case each input process can read its own $(rank)_{th}$ value, in order to forward it to the processes of the Kohonen group.

RMA Based Counter Propagation Algorithm

The main drawback of the parallel algorithm presented in the previous sections is the high traffic load associated with the weight table update for both training stages (i.e. stage A and stage B). Since each process maintains a local copy of the two weight tables (the input – Kohonen weight table and the Kohonen – Grossberg weight table), it has to broadcast these tables to all the processes of the Kohonen and Grossberg group in order to receive the new updated weight values. An improvement of this approach can be achieved by using an additional process that belongs to its own target group. This target process maintains a unique copy of the two weight tables and each process can read and update the weight values of these tables via remote memory access (RMA) operations. This new improved architecture of the counter propagation network is shown in Figure 8.

In this approach the additional target process creates and maintains the weight tables of the neural network while each process of the Kohonen and the Grossberg group reads the appropriate weights with the function MPI_Get and updates their values (by applying the equations described above). This can be done using the function MPI_PUT. An optional third window can be used to store the minimum input weight distance for each training pattern and for each epoch. In this case one of the processes of the Kohonen group can use the

MPI_Accumulate function (with the MPI_SUM opCode) to add the current minimum distance to the window contents. In this way, at the end of each epoch this window will have the sum of these distances that is used for the calculation of the mean error for stage A; a similar approach can be used for the stage B. The synchronization of the system processes can be performed either by the function MPI_Win_Fence or by the set of four functions MPI_Win_Post, MPI_Win_start, MPI_Win_complete and MPI_Win_wait, which are used to indicate the beginning and the termination of the access and the exposure epochs of the remote process target windows.

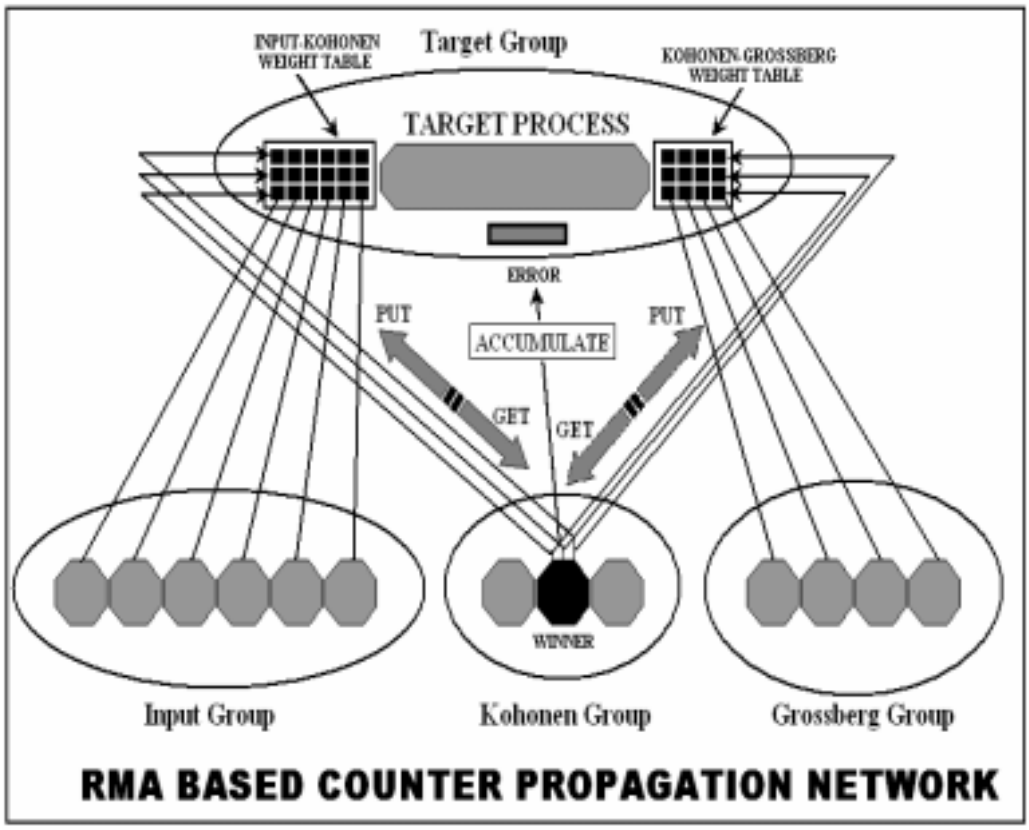

Figure 8: RMA based counter propagation network

Conclusions and Future Work

The objective of this research was the parallelization of the counter – propagation network by means of the message-passing interface (MPI). The development of the application was based to the MPICH2 implementation of the MPI of Argonne National Laboratory that supports advanced features of the interface, such as parallel I/O and remote memory access functions. This parallelization was applied on two different aspects: (a) the training set patterns were distributed to the processes of the input group in such a way that each process retrieves the $(rank)_{th}$ column of the set with P values, where (rank) is the rank of the process in the input group. This distribution is applied for the input vectors as well as for the output vectors that are distributed to the processes of the Grossberg group. (b) the two – dimensional weight tables were distributed to the processes of the Kohonen group with each table row to be associated with its corresponding Kohonen process.

The next step of this research is the presentation of comparative results between the serial and the parallel versions. This could not be affected at the moment, since the development took place on a single—processor machine, where the performance of the two approaches was almost the same. In order to measure the speedup of the parallel algorithm, a multiprocessing system – such as a dual processor PC, a computational grid or a computer network – is necessary, but this system was not available at the time of the paper writing. Therefore we point out the main guidelines that one has to follow to build a parallel neural network – a more practical approach on this subject is one of the immediate future work subjects.

There are many topics that are open in the design and implementation of parallel neural networks. By restricting ourselves to the development of such structures via MPI, it is of interest to investigate the improvement achieved if non-blocking communications are used – in this research the data communication was based on the blocking functions MPI_Send and MPI_Recv. Another very interesting topic is associated with the application of the models described above for the simulation of arbitrary neural network architectures. As it is well known, the counter – propagation network is a very simple one, since is has (in the most cases) only three layers. However, in general, a neural network may have as many as layers the user wants. In this case we have to find ways to generate process groups with the correct structure. Furthermore, in our design, each processes simulated only one neuron; an investigation of the mechanism that affects the performance of the network

when we assign to each process more than one neurons, is a challenging prospect.

For all these different situations, one has to measure the execution time and the speedup of the system in order to draw conclusions for the simulation of neural networks by parallel architectures. Finally, another point of interest is the comparison of the MPI – based parallel neural models with those that are based on other approaches, such as parallel virtual machines (PVM).

References:

[01] Simon Haykin: "Neural Networks – A Comprehensive Foundation", Prentice Hall, 1994, ISBN 0-132-73350-1.

[02] Yann Boniface, Frédéric Alexandre, Stéphane Vialle: "A Bridge between two Paradigms for Parallelism: Neural Networks and General Purpose MIMD Computers", in Proceedings of International Joint Conference on Neural Networks, (IJCNN'99) 1999, Washington, D.C.

[03] Thomas Fuerle and Erich Schikuta: "PAANS – A Parallelized Artificial Neural Network Simulator", in Proceedings of 4th International Conference on Neural Information Processing (ICONIP'97), Dunedin, New Zeland, Springer Verlag, November 1997.

[04] Russell K. Standish: "Complex Systems Research on Parallel Computers", Web Text, http://parallel.hpc.unsw.edu.au/rks/docs/parcomplex

[05] Erich Schikuta: "Structural Data Parallel Neural Network Simulation", in Proceedings of 11th Annual International Symposium on High Performance Computing Systems (HPCS'97), Winnipeg, Canada, July 1997.

[06] N. B. Serbedzija: "Simulating Artificial Neural Networks on Parallel Architectures", Computer 29 (3), pages 56-63, May 1996, ISSN 0018-9162.

[07] Eric Schikuta, Thomas Fuerle and Helmut Wanek: "Structural Data Parallel Simulation of Neural Networks", Journal of System Research and Information Science, Volume 9, pages 149 – 172, Gordon and Breach Publishing Group, 2000.

[08] Manavendra Misra: "Implementation of Neural Networks on Parallel Architectures", PHD Thesis, University of Southern California, December 1992.

[09] Manavendra Misra: "Parallel Environment for Implementing Neural Networks", Neural Computing Surveys, 1, pages 48 – 60.

[10] Mathias Quoy, Sorin Moga, Philippe Gaussier and Arnaud Revel: "Parallelization of Neural Networks Using PVM", in J. Dongarra, P. Kacsuk and N. Podhorszki (Eds.): "Recent Advances in Parallel Virtual Machines and Message Passing Interface", pages 289 – 296, Berlin, 2000, Lecture Notes on Computer Science 1908, Springer Verlag.

[11] Mark Snir et al: "MPI – The Complete Reference, Volume 1: The MPI Core", 2nd Edition, Scientific and Engineering Computational Series, The MIT Press, Massachusetts, 1998, ISBN 0-262-69215-5.

[12] William Cropp et al: "MPI – The Complete Reference, Volume 2: The MPI Extensions", Scientific and Engineering Computational Series, The MIT Press, Massachusetts, 1998, ISBN 0-262-57123-4.

[13] Peter Pacheco: "Parallel Programming with MPI", Morgan Kaufmann Publishers Inc, San Francisco, California, 1997, ISBN 1-55860-339-5.

[14] Yann Boniface, Frédéric Alexandre, Stéphane Vialle: "A Library to Implement Neural Networks on MIMD Machines", in Proceedings of 6th European Conference on Parallel Processing (EUROPAR'99), Toulouse, France, pages 935 – 938.

[15] Jim Torresen et al: "Parallel Back Propagation Training Algorithm for MIMD Computer with 2D – torus Network", in Proceedings of 3rd Parallel Computing Workshop (PCW'94), 1994, Kawasaki, Japan.

[16] Jim Torresen, Shinji Tomita: "A Review of Parallel Implementation of Back Propagation Neural Networks", Chapter 2 in the book of N. Sundararajan and P. Saratchandram (Eds.): "Parallel Architectures of Artificial Neural Networks", IEEE CS Press, 1998, ISBN 0-8186-8399-6.

[17] V. Kumar, S. Shekhar, M. B. Amin: "A Scalable Parallel Formulation of the Back Propagation Algorithm for Hypercubes and Related Architectures", IEEE Transactions on Parallel and Distributed Systems, Volume 5, Issue 10, pages 1073 – 1090, 1994, ISSN 1045-9219.

[18] Li Weigang, Nilton Correia da Silva: "A Study of Parallel Neural Networks, in Proceedings of International Joint Conference on Neural Networks", Volume 2, pages 1113-1116, 1999, Washington, D.C.

[19] Philipp Tomsich, Andreas Rauber and Dieter Merkl: "Optimizing the parSOM Neural Network Implementation for Data Mining with Distributed Memory Systems and Cluster Computing", in Proceedings of 11th International Workshop on Databases and Expert Systems Applications, September 2000, Greenwich, London UK, pages 661 – 666.

[20] James A. Freeman, David M. Skapura: "Neural Networks: Algorithms, Applications, and Programming Techniques", Addison-Wesley Publishing Company, 1991, ISBN 0-201-51376-5.

Authors:

Athanasios I. Margaris - University of Macedonia, Thessaloniki, Greece, E-mail address: amarg@uom.gr
Efthimios Kotsialos - University of Macedonia, Thessaloniki, Greece, E-mail address: ekots@uom.gr