INFORMATICA, 2007, Vol. 18, No. 1, 79–102

Design and Implementation of Parallel Counterpropagation Networks Using MPI

Athanasios MARGARIS, Stavros SOURAVLAS,
Efthimios KOTSIALOS, Manos ROUMELIOTIS
University of Macedonia, Dept. of Applied Informatics
156 Engatia Str., GR 540-06,Thessaloniki, Greece
e-mail: amarg@uom.gr, sourstav@uom.gr, ekots@uom.gr, manos@uom.gr

Received: October 2004

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 counterpropagation network and the parallel platform used to simulate that network is the message passing interface (MPI). In the next sections the counterpropagation algorithm is presented in its serial as well as its parallel version. For the latter case, simulation results are given for the session parallelization as well as the training set parallelization approach. Regarding possible parallelization of the network structure, there are two different approaches that are presented; one that is based to the concept of the intercommunicator 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.

Key words: neural networks, counterpropagation, parallel programming, message passing interface, communicators, process groups, point to point communications, collective communication, RMA operations.

1. 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 (Haykin, 1994). 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 (Boniface *et al.*, 1999). 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 (Fuerle & Schikuta, 1997).

A parallel neural network can be constructed using a variety of different methods (Standish, 1999; Schikuta, 1997; Serbedzija, 1996; Schikuta *et al.*, 2000; Misra, 1992; Misra, 1997), such as the parallel virtual machines (PVM) (Quoy, 2000), the message passing interface (MPI) (Snir *et al.*, 1998; Cropp *et al.*, 1998; Pacheco, 1997), the

shared memory model and the implicit parallelization with parallel compiler directives (Boniface, 1999). 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 (Torresen *et al.*, 1994; Torresen and Tomita, 1998; Kumar, 1994) to the unsupervised self-organizing maps (Weigang *et al.*, 1999; Tomsich *et al.*, 2000). In this research the counterpropagation network is parallelized by means of the message passing interface library (Pacheco, 1997).

2. The Serial Counterpropagation Algorithm

Counterpropagation neural networks (Freeman, 1991) 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 characterized by the existence 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.

Regarding the training process of the counterpropagation 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 counterpropagation network includes the following steps:

(A) Training of the weights from the input to the hidden nodes: the training of the weights from the input to the hidden layer is performed as follows:

Step 0. 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 1. A vector pair (x, y) of the training set, is selected in random.

Step 2. The input vector x of the selected training pattern is normalized.

Step 3. The normalized input vector is sent to the network.

Step 4. 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 ith neuron of the input layer with the jth neuron of the Kohonen layer.

Step 5. The winner neuron W of the Kohonen layer is identified as the neuron with the minimum distance value D_j.

Step 6. 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 $\alpha(t) = \alpha_0[1-(t/T)]$ 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 7. 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 8. 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. The network also stops training where the specified number of iterations has been reached, but the error value has not converged to a specific value.

(B) Training of the weights from the hidden to the output nodes: the training of the weights from the hidden to the output layer is performed as follows:

Step 0. 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 1. A vector pair (x, y) of the training set, is selected in random.

Step 2. The input vector x of the selected training pattern is normalized.

Step 3. The normalized input vector is sent to the network.

Step 4. 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 ith neuron of the input layer with the jth neuron of the Kohonen layer.

Step 5. 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 6. 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 7. The above procedure is performed for each pattern of the training set currently used. 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 $E = \frac{1}{P}\sum_{j=1}^{N} D_j = \frac{1}{P}\sum_{j=1}^{P}\sqrt{\sum_{k=1}^{N}(y_k - w_{kj})^2}$.

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.

3. Parallel Approaches for the Counterpropagation Network

The parallelization of the counterpropagation network can be performed in many different ways. In this project three different parallelization modes are examined, namely the session parallelization, the training set parallelization and the network parallelization. In session parallelization, there are many instances of the neural network object running concurrently on different processors with different values for the training parameters. In training set parallelization the training set is divided into many fragments and a set of neural networks run in parallel and in different machines each one with its own training set fragment. After the termination of the learning phase, the synaptic weights are sent to a central process that merges them and estimates their final values. Finally, in the network parallelization, the structure of the neural network is distributed to the system processes with the information flow to be implemented by using message passing operations. In a more detailed description, these parallelization schemes, work as follows.

3.1. *Session and Training Set Parallelization*

The implementation of the session and the training set parallelization is based on the Neural Workbench simulator (Margaris *et al.*, 2003) that allows the construction and training of arbitrary neural network structures. In this application, a neural network is implemented as a single linked list of layers each one of them contains a single linked list of the neurons assigned to it. Regarding the fundamental neural processing elements they contain two additional linked lists of the synapses to which they participate as source or target neurons. This multilayered linked list architecture is used for the implementation of the training set too, as a linked list of training vector pairs each one of them contains two linked lists of the input values and the associated desired output values. Each neural network can be associated with a linked list of such training sets for training, while, each object has its own training parameters such as the learning rate, the momentum, and the slope of the sigmoidal function used for the calculation of the neuron output.

The kernel of the Neural Workbench – which is a Windows application – was ported to Linux operating system and enhanced with message passing capabilities by using the appropriate functions of the MPI library. Based on these capabilities, the session and the training set parallelization, work as follows.

3.1.1. *Session Parallelization*

In session parallelization the parallel application is composed of N processes each one of them run in parallel a whole neural network with different training parameters. The synaptic weights are initialized by all processes in random values, while, the parameters of the training phase (such as the Kohonen learning rate α, the Grossberg learning rate β

and the tolerance τ) are initialized by one of the processes (for example, by the process with rank $R = 0$) and broadcasted to the system processes by using the MPI_Bcast function. This is an improvement over the serial approach where there is only one process running a loop of N training procedures with different conditions in each loop iteration. The session parallelization approach is shown in Fig. 1.

3.1.2. *Training Set Parallelization*

In training set parallelization the training set patterns are distributed over the system processes, each one of them runs a whole neural network with its own training set fragment. In this project the parallel application is composed by two processes and the concurrent training of the associated neural network is performed by using the even patterns in the first process and the odd patterns in the second process. After the termination of the training operation of the two networks, the one process sends its updated weights to the other one, that receives them, and updates its own weights by assign to them the mean value between their updated values and the values of the incoming weights. This approach is shown graphically in Fig. 2.

Since the structure of a neural network created by the Neural Workbench application is very complicated and the values of the synaptic weights are stored in non-contiguous memory locations, auxiliary utilities have been implemented for packing the weights in the sender and unpacking them, in the receiver. In other words, the source process packs its synaptic weights before sending, while, the target process unpacks the weights sent by the first process after receiving them, and then proceeds to the merge operation in the way described above. These packing and unpacking utilities have been implemented by using the MPI_Pack and the MPI_Unpack functions of the MPI library.

3.2. *Network Parallelization*

A typical parallelization scheme for the counterpropagation network is to use a separate process for modelling the behavior of each neuron of the neural network (Boniface,

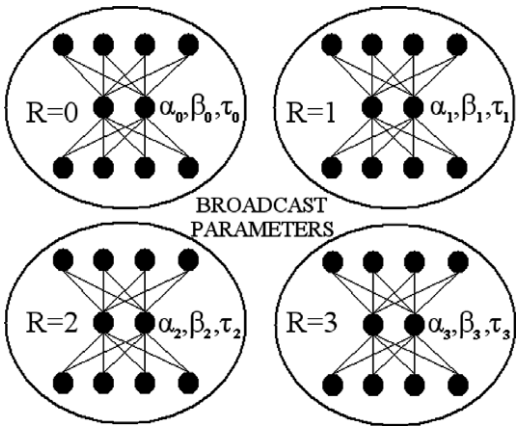

Fig. 1. Session parallelization in counterpropagation networks.

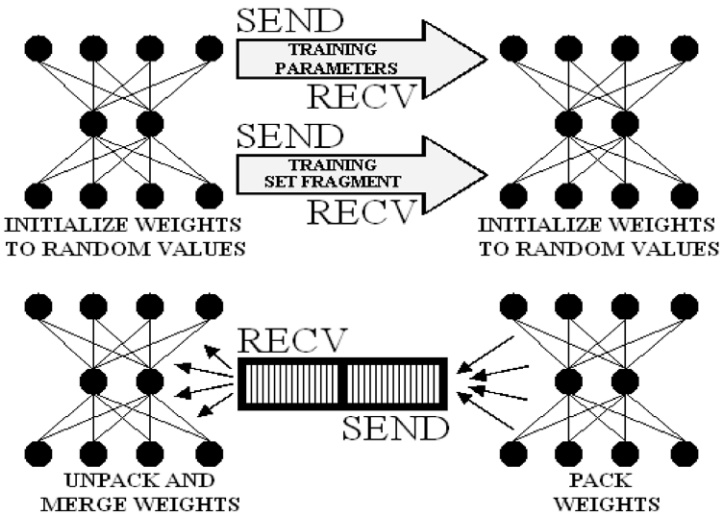

Fig. 2. Training set parallelization in counterpropagation networks.

1999). 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 counterpropagation 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.

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 Fig. 3.

At this point the system setup has been completed and the the training of the neural network 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 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 (such as MPI_File_read) and to the establishment of a different file type and file view for each input and output process.

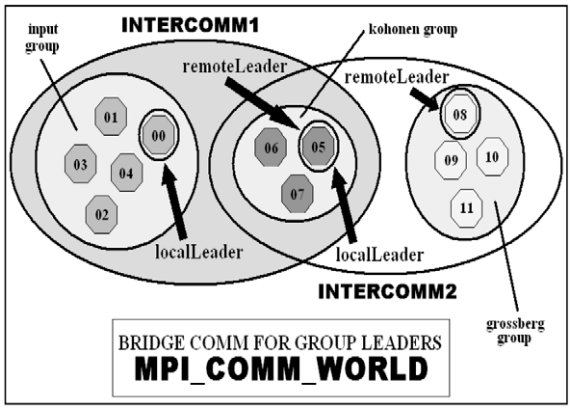

Fig. 3. The message passing between the three process groups is performed via the intercommunicators interComm1 and interComm2.

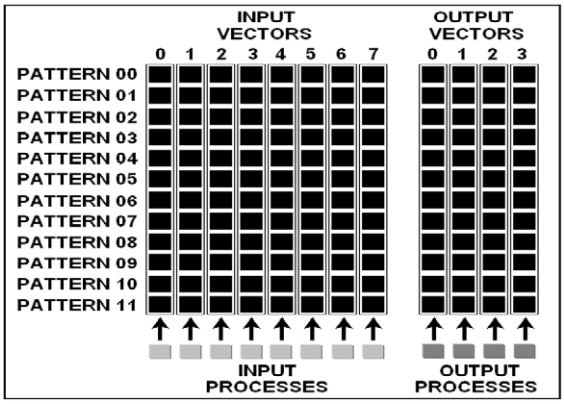

Fig. 4. The distribution of the training set data to the input and the output processes for a training set of 12 training patterns with 8 inputs and 4 outputs.

The parallel counterpropagation algorithm is a straightforward extension of its serial counterpart and it is composed of the following steps: (in the following description, the notation P_n is used to denote the process with a rank value equal to n).

(A) STAGE A: Performs the training of the weights from the input to the Kohonen processes.

Step 0. A two dimensional $K \times M$ 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 $M \times N$ 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 $[0, P - 1]$ where P is the number of the training vector pairs. 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 Fig. 4), this input value is equal to the inputColumn[nextPattern] where the inputColumn vector contains the $(\text{rank})_{\text{th}}$ input value of each training pattern. The Steps 1 and 2 of the parallel counter propagation algorithm are shown in the Fig. 5.

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,

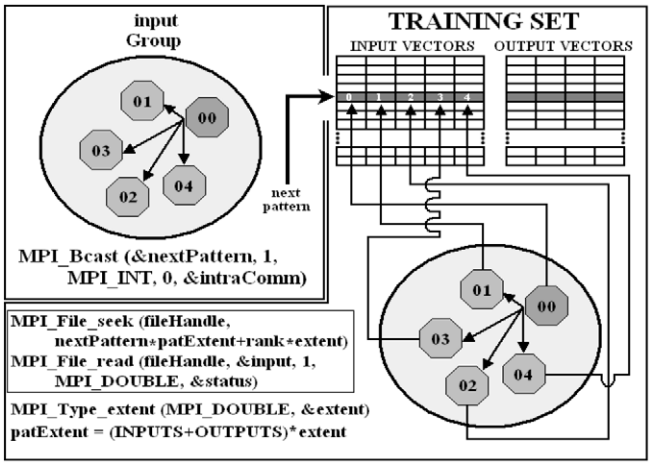

Fig. 5. The retrieval of the training pattern input values from the processes of the input group.

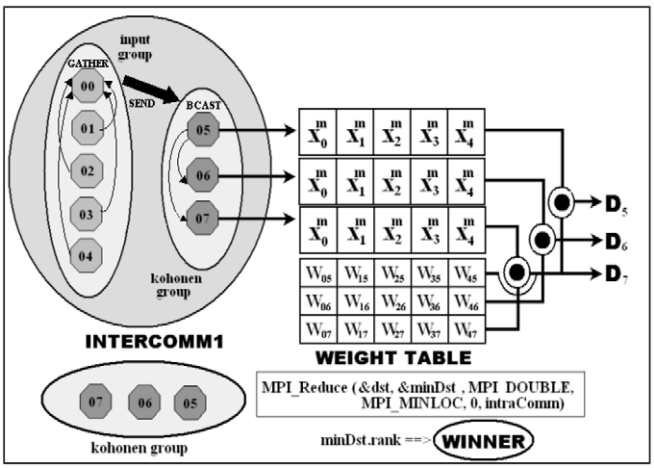

Fig. 6. The identification of the winning process from the processes of the Kohonen group.

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 intercommunicator 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 counterpropagation 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.

Step 5. The winning process updates the weights of its weight table row, according to the equation $W_{wi}(t+1) = W_{wi}(t) + \alpha(t)(x_i - W_{wi}(t))$, 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 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.

(B) 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 Rth 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 intercommunicator 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 – these 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 jth output process. These weights belong to the jth 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 $(O_0, O_1, O_2, \ldots, O_{N-1})$ and the desired output vector $(Y_0, Y_1, Y_2, \ldots, Y_{N-1})$. 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 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.

3.3. *The Recall Phase of the Parallel Simulator*

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 remain-

ing 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 $(\text{rank})_{\text{th}}$ value, in order to forward it to the processes of the Kohonen group.

3.4. *Delay Analysis of the Parallel Counterpropagation Algorithm*

In order to describe the communication delay of the proposed parallel algorithm, let us denote with S the message startup time, with T the transmission time per byte, and with L the message size in bytes. In this case, with no loss of generality we assume that the number of input and output processes M and N divide the number of hidden processes, K. If this does not hold, our analysis can be performed by adding a number of imaginary nodes. Whenever a node is imaginary, we simply ignore the corresponding communications.

As described in previous sections, the first stage of the parallel counterpropagation algorithm requires interprocessor communication between the nodes of the input layer to store the values to the memory of the leader of the input group, that will pass these values to the group leader of the Kohonen group. In its turn, the leader process of the Kohonen group will broadcast these values to the appropriate process. The same communication pattern also incurs when performing the training of the weights from the Kohonen to the Grossberg nodes. We symbolize the two phases by $R(M, K)$ and $R(K, N)$. In the following, we will perform the cost analysis for the communications performed for $R(M, K)$; the cost of $R(K, N)$ is computed similarly.

The communication grid for $R(M, K)$ can be represented by a two-dimensional table T_{dp} that stores the indices of the processors where the messages will move to. Row and column indexing of T_{dp} begins from zero. For example, consider $R(6, 3)$. Table 1 shows that there are five messages for each Kohonen layer node (the first row of the table indicates that there are five messages for node K_0 of the Kohonen layer, the second row indicates that there are five messages for node K_1 etc). Note that T_{dp} is divided into a number of sub-matrices of size $K \times K$, in our example 3×3.

To gather messages for the Kohonen nodes to the leader process (we assume that it is executed by processor M_0 of the input layer), we perform the following steps.

Step 1. For each sub-matrix, we circularly shift each column by λ times, where λ is the indexing value of each column, that is, $\lambda = 0$ for column 0, $\lambda = 1$ for the first column etc. This step describes internal reading operations in the memory of each node. Table 2 shows the result of internal memory reading operations for $R(3, 6)$.

Step 2. For each sub-matrix, we circularly shift each row leftwards by μ, where μ is the indexing value of each row. This step represents interprocessor communication

Table 1
T_{dp} with its triangular sub-matrices for $R(6,3)$

M_0	M_1	M_2	M_3	M_4	M_5
K_0	K_0	K_0	K_0	K_0	K_0
K_1	K_1	K_1	K_1	K_1	K_1
K_2	K_2	K_2	K_2	K_2	K_2

Table 2
T_{dp} after Step 1 for $R(6,3)$

M_0	M_1	M_2	M_3	M_4	M_5
K_0	K_2	K_1	K_0	K_2	K_1
K_1	K_0	K_2	K_1	K_0	K_2
K_2	K_1	K_0	K_2	K_1	K_0

Table 3
T_{dp} after Step 2 for $R(6,3)$

M_0	M_1	M_2	M_3	M_4	M_5
K_0	K_2	K_1	K_0	K_2	K_1
K_0	K_2	K_1	K_0	K_2	K_1
K_0	K_2	K_1	K_0	K_2	K_1

between members of a layer. The result of these communications is that every node of the input layer stores in its memory data destined for exclusively one node of the Kohonen layer. As seen in Table 3, nodes M_1, M_4 will transfer to M_0 all necessary data to update node K_2 of the Kohonen layer, while nodes M_2, M_5 will transfer to M_0 all necessary data to update node K_1. Finally, M_3 will transfer to M_0 the data required for updating K_0. These transfer will incur in Step 3.

Step 3. After Step 2, there are groups of (M/K) columns containing the same index value. We simply perform communication from all nodes to M_0. This will transfer all the necessary data from the input layer nodes to the leader process being executed by processor M_0.

Theorem 1 analyzes the complexity of these steps.

Theorem 1. *The number of communications required to perform the two stages of the parallel counterpropagation algorithm is at most* $(M + 2K + N - 4)$.

Proof. Consider the message broadcasts for $R(M, K)$ and assume that K divides M. In Step 2 (we consider the cost of reading operations of Step 1 to be minimal) there are at most $K - 1$ circular shifts in each sub-matrix that execute in parallel. In Step 2, there are $M - 1$ transmissions to P_0. Thus $R(M, K)$ needs at most $K - 1 + M - 1 = K + M - 2$ communication steps (if K does not divide M then the steps are reduced due to the existence of imaginary nodes. Similarly, K, N requires at most $N+K-2$ communication steps. Thus the maximum number of communication steps is $M + 2K + N - 4$.

For the serial version of the algorithm, KM and KN communication steps are needed to perform $R(M, K)$ and $R(K, N)$ respectively, for a total of $KM + KN$ steps. From Theorem 1, we assume that the total delay of the parallel counterpropagation algorithm is $(\alpha + L\beta)(M + 2K + N - 4)$.

4. Experimental Results

The proposed parallel architectures of the counter propagation network were tested by using three different training examples with increasing network and training set size. For each case the execution time of the serial as well as the parallel implementation was measured in order to calculate the speedup and the efficiency of the parallel system. For the serial case, the neural network was trained for a single run as well as for many runs (up to three) and the execution time of all these runs was recorded. Since the objective of the research was not to configure an appropriate network structure and to tune the parameter values to get a converging system but only to measure the speedup and the efficiency of the parallel architecture, very small tolerance values were used, such that the epoch numbers of both stages to be exhausted. Furthermore, for sake of simplicity, the number of epochs of stages A and B (M and N respectively) was the same.

The training examples and the structure of the neural network used in each case are presented below:

1) The Iris database (Fisher, 1936): this is a famous and widely used training set in pattern recognition consisting of 150 labelled four dimensional feature vectors describing three types of Iris flowers, namely, Iris Setosa, Iris Versicolour, and Iris Virginica. Each flower type is described by 50 feature vectors in the training set. The input vector is composed by four real values describing the sepal width, the petal width, the sepal length, and the petal length respectively, while the output vector is composed of three binary values that identify the three flower types (more specifically, type I is modelled as [1 0 0], type II is modelled as [0 1 0] and type III is modelled as [0 0 1]).

The training set used in this example was composed of 75 feature vectors (one half for each flower type) while the remaining vectors were used in the recall phase. The neural network structure was characterized by 4 neurons in the input layer, 3 neurons in the hidden layer and 3 neurons in the output layer while the parameter values was different for different runs.

2) The logistic map (Margaris *et al.*, 2001): the logistic map is a well known one dimensional chaotic attractor described by the equation $y_n = x_{n+1} = \lambda x_n(1 - x_n)$

with the λ parameter to get values in the interval $[1, 4]$. In this example the training set was composed by 1000 pairs in the form (x_i, y_i), where the inputs x_i were uniformly distributed in the interval [0, 1], while the outputs, y_i, were calculated by the equation $y_i = \lambda x_i(1 - x_i)$. Regarding the structure of the neural network used, it was a three-layered feed-forward network with one input neuron, three hidden neurons and one output neuron.

3) Speech frames database (Margaris, 2005): this example is associated with a neural-based application that recognizes speech frames emerged from a set of recorded audio files containing pronounces of a set of specific words. The training set is composed of 184 vector pairs each one of them contains 10 LPC coefficients as the input values and the corresponding 10 Cepstral coefficients as the desired output values. The structure of the neural network used in this case was characterized by the existence of three layers with 10 input neurons, 15 hidden neurons and 10 output neurons respectively.

4.1. *Simulation Results for the Serial Case*

In this simulation there is only one non-MPI process running on a single CPU. The process runs the training counterpropagation algorithm N times ($N = 1, 2, 3$) by using a loop in the form *for* ($i = 0;\ i < N;\ i++$) *RunCounterPropagation* (*...*);. The results of this simulation for the three training examples, are presented in Table 4 and contain the execution times (in seconds) for each training examples for different number of runs and for different values for the iteration numbers M and N.

4.2. *Simulation Results for the Session Parallelization*

In the session parallelization simulation the maximum number of neural networks run concurrently was equal to three, since the experimental cluster used for the simulation was composed by three computing nodes each one of them had a CPU running on 800 MHz and a physical memory of 128 MBytes. To verify the system's implementation, all the possible combinations were used, namely, one process runs in one host, two processes run in one and two hosts, and three processes run in one, two, and three hosts. The simulation

Table 4
Experimental results for the serial case for all training examples

	LOGISTIC MAP			IRIS DATABASE			SPEECH FRAMES		
M, N	1 run	2 runs	3 runs	1 run	2 runs	3 runs	1 run	2 runs	3 runs
000100	0002	0004	0006	0001	0001	0001	00019	00039	00056
001000	0022	0042	0064	0006	0012	0018	00191	00374	00555
005000	0106	0211	0317	0030	0058	0089	00957	01928	02811
010000	0212	0424	0636	0059	0117	0177	01804	03997	05888
020000	0424	0847	1271	0117	0234	0352	04014	07603	11521
050000	1059	2117	3175	0292	0589	0888	09713	19571	28915

results for the session parallelization and for the three training examples are shown in Tables 5, 6, and 7.

To provide performance estimates of the parallel system, we present the simulation results for the three examples (see Tables 8, 9, 10 respectively) with respect to the speedup and efficiency. More specifically, we measure the speedup $S(3)$ as the ratio of the total execution time $T(1)$ of three processes running on a sequential computer to the corresponding execution time $T(3)$ for the same processes running on 3 nodes. The efficiency is then computed as the ratio of $S(3)$ to the number of nodes (3 in our case). The results show that in most cases we achieve ideal parallel efficiency of 1.0, that is three nodes run three times faster than one for the same problem.

4.3. *Simulation Results for the Training Set Parallelization*

In this simulation a training set of $2N$ training patterns is divided into two training sets of N patterns that contain the even and the odd patterns. The parallel application is composed by exactly 2 processes each one of them runs the whole neural network (as in session parallelization) but with its own odd or even training pattern. After the termination of the simulation, the process $R = 1$ sends its synaptic weights to the process $R = 0$ that receives them and estimates the final weights as the mean value of its own weights and the corresponding received weights. The send and the receive operations are performed by the blocking functions MPI_Send and MPI_Recv. The simulation results in training set parallelization for all the training examples are shown in Table 11 (the shown execution times are measured in seconds).

5. RMA Based Counterpropagation 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 Fig. 7.

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

Table 5
Session parallelization results for the logistic map

	1 Host		2 Hosts		3 Hosts	
	Epochs	Seconds				
1 Process	00100	0002.155				
	01000	0021.515				
	05000	0107.560				
	10000	0215.103				
	20000	0430.148				
	50000	1075.467				
	Epochs	Seconds	Epochs	Seconds		
2 Processes	00100	0003.940	00100	0002.156		
		0004.201		0002.155		
	01000	0042.710	01000	0021.555		
		0042.901		0021.546		
	05000	0215.056	05000	0107.710		
		0215.216		0107.712		
	10000	0430.655	10000	0216.028		
		0430.665		0216.042		
	20000	0861.414	20000	0431.762		
		0861.703		0430.773		
	50000	2165.654	50000	1077.049		
		2169.179		1076.840		
	Epochs	Seconds	Epochs	Seconds	Epochs	Seconds
3 Processes	00100	0006.293	00100	0004.129	00100	0002.144
		0006.267		0002.146		0002.146
		0006.020		0004.084		0002.146
	01000	0064.150	01000	0042.724	01000	0021.427
		0064.196		0021.444		0021.458
		0063.981		0042.731		0021.456
	05000	0321.783	05000	0214.111	05000	0107.090
		0321.978		0107.110		0107.186
		0321.652		0214.209		0107.257
	10000	0643.318	10000	0428.381	10000	0214.338
		0643.604		0214.454		0214.403
		0643.391		0428.493		0214.508
	20000	1287.674	20000	0857.291	20000	0428.476
		1287.731		0428.745		0428.794
		1287.383		0857.402		0429.001
	50000	3218.685	50000	2143.201	50000	1071.292
		3220.368		1072.007		1071.762
		3219.566		2143.731		1071.431

Table 6
Session parallelization results for the IRIS database

	1 Host		2 Hosts		3 Hosts	
	Epochs	Seconds				
1 Process	00100	000.601				
	01000	005.885				
	05000	029.587				
	10000	059.835				
	20000	119.680				
	50000	297.572				
	Epochs	Seconds	Epochs	Seconds		
2 Processes	00100	000.848	00100	000.591		
		000.805		000.599		
	01000	011.546	01000	005.994		
		011.584		005.883		
	05000	059.462	05000	029.941		
		059.072		029.936		
	10000	118.774	10000	059.898		
		118.839		058.844		
	20000	236.473	20000	119.151		
		236.486		119.160		
	50000	587.694	50000	297.909		
		587.463		297.846		
	Epochs	Seconds	Epochs	Seconds	Epochs	Seconds
3 Processes	00100	001.535	00100	000.943	00100	000.595
		001.576		000.588		000.597
		001.357		000.942		000.592
	01000	017.876	01000	011.568	01000	005.879
		017.903		005.879		005.989
		017.752		011.593		005.945
	05000	092.413	05000	059.254	05000	029.891
		092.422		029.931		029.587
		091.605		059.193		029.867
	10000	183.652	10000	119.252	10000	058.721
		183.295		059.529		059.116
		183.146		119.199		058.703
	20000	367.138	20000	234.633	20000	119.032
		366.886		119.661		119.050
		368.174		234.480		118.931
	50000	891.302	50000	594.232	50000	297.318
		890.662		299.198		295.606
		890.974		593.805		293.486

Table 7
Session parallelization results for the speech frames database

	1 Host		2 Hosts		3 Hosts	
	Epochs	Seconds				
1 Process	00100	00020.095				
	01000	00197.748				
	05000	00925.377				
	10000	01793.960				
	20000	03828.180				
	50000	09441.553				
	Epochs	Seconds	Epochs	Seconds		
2 Processes	00100	00037.152	00100	00019.575		
		00037.236		00019.001		
	01000	00401.272	01000	00180.191		
		00402.954		00182.440		
	05000	01793.795	05000	00947.687		
		01802.148		00902.133		
	10000	03925.213	10000	01851.708		
		03942.807		01874.798		
	20000	07175.727	20000	03744.900		
		07208.764		03952.044		
	50000	22147.286	50000	09110.900		
		22232.592		09588.634		
	Epochs	Seconds	Epochs	Seconds	Epochs	Seconds
3 Processes	00100	00057.490	00100	00037.757	00100	00019.962
		00057.445		00019.770		00020.080
		00057.268		00038.770		00018.778
	01000	00549.270	01000	00399.225	01000	00187.767
		00549.734		00191.478		00182.394
		00549.503		00399.231		00194.594
	05000	03025.275	05000	01828.238	05000	00965.519
		03025.724		00911.945		00979.553
		03025.471		01829.578		00988.850
	10000	05646.237	10000	04026.084	10000	01915.560
		05648.781		01929.454		01823.962
		05702.921		04027.124		01826.842
	20000	12107.604	20000	07312.064	20000	03685.665
		12112.123		03860.654		04020.912
		12110.864		07317.681		03591.262
	50000	30641.980	50000	18802.907	50000	10121.582
		30653.922		09971.625		09495.003
		30652.644		18809.625		09534.072

Table 8
Speedup of the parallel system – The logistic map

Epochs	Execution Time (1 node)	Execution Time (3 nodes)	Speedup $S(3) = \frac{T(1)}{T(3)}$	Efficiency $E(3) = \frac{S(3)}{3}$
00100	0006.293	0002.144	2.935160	0,97838
05000	0321.783	0107.090	3.004000	1,00130
20000	1287.674	0428.476	3.005240	1,00170
50000	3218.685	1071.292	3.004448	1,00140

Table 9
Speedup of the parallel system – The IRIS database

Epochs	Execution Time (1 node)	Execution Time (3 nodes)	Speedup $S(3) = \frac{T(1)}{T(3)}$	Efficiency $E(3) = \frac{S(3)}{3}$
00100	001.535	000.595	2.570	0,850
05000	092.413	029.891	3.091	1,030
20000	367.138	119.032	3.084	1,028
50000	891.302	297.318	2.997	0,998

Table 10
Speedup of the parallel system – The speech frames database

Epochs	Execution Time (1 node)	Execution Time (3 nodes)	Speedup $S(3) = \frac{T(1)}{T(3)}$	Efficiency $E(3) = \frac{S(3)}{3}$
00100	00057.490	00019.962	2.879	0,959
05000	03025.275	00965.519	3.130	1,040
20000	12107.604	03685.665	3.285	1,095
50000	30641.980	10121.582	3.027	1,009

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.

Table 11
Simulation results for the training set parallelization

	LOGISTIC MAP		IRIS DATABASE		SPEECH FRAMES	
M, N	R=0	R=1	R=0	R=1	R=0	R=1
000100	0000.857	0000.853	0000.291	0000.283	0009.620	0009.791
001000	0008.529	0008.536	0002.924	0002.843	0092.410	0097.858
005000	0042.621	0042.626	0014.543	0014.028	0467.074	0453.241
010000	0085.328	0085.233	0029.224	0028.004	0985.505	0916.098c
020000	0170.502	0170.420	0058.450	0056.836	1881.286	1942.054
050000	0426.230	0430.659	0146.929	0139.954	4485.041	4451.929

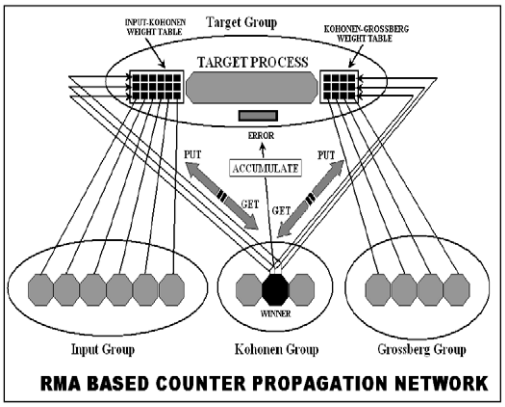

Fig. 7. RMA based counter propagation network.

6. Conclusions and Future Work

The objective of this research was the parallelization of the counterpropagation 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. In this research, two parallelization aspects were tested with respect to the session and the training set of the neural network. Regarding the network parallelization approach two schemes were proposed: (a) the training set patterns were distributed to the processes of the input group in such a way that each process retrieves the $(\text{rank})_{\text{th}}$ 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.

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 (the collective operations are by default blocking operations). 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 counterpropagation 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

Boniface, Y., F. Alexandre, S. Vialle (1999). 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)*. Washington, D.C.

Boniface, Y., F. Alexandre, S. Vialle (1999). A library to implement neural networks on mimd machines. In *Proceedings of 6th European Conference on Parallel Processing (EUROPAR'99)*. Toulouse, France. pp. 935–938.

Cropp, W. *et al.* (1998). *MPI – The Complete Reference*. Vol. 2. The MPI Extensions. Scientific and Engineering Computational Series, The MIT Press, Massachusetts.

Fisher, R.A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**(1), 179–188.

Freeman, J., D. Skapura (1991). *Neural Networks: Algorithms, Applications, and Programming Techniques*. Addison–Wesley Publishing Company.

Fuerle, T., E. Schikuta (1997). PAANS – a parallelized artificial neural network simulator. In *Proceedings of 4th International Conference on Neural Information Processing (ICONIP'97)*. Dunedin, New Zeland, Springer Verlag.

Haykin, S. (1994). *Neural Networks – A Comprehensive Foundation*. Prentice Hall.

Kumar, V., S. Shekhar, M. Amin (1994). A scalable parallel formulation of the back propagation algorithm for hypercubes and related architectures. *IEEE Transactions on Parallel and Distributed Systems*, **5**(10). pp. 1073–1090.

Margaris, A. *et al.* Development of neural models for the logistic equation, and study of the neural based trajectories in the convergence, periodic, and chaotic regions. *Neural, Parallel & Scientific Computations*, **9**, 221–230.

Margaris, A. *et al.* (2003). Neural workbench: an object oriented neural network simulator. In *Proceedings of International Conference on Theory and Applications of Mathematics and Informatics (ICTAMI2003)*. Acta Universitatis Apulensis. Alba Ioulia, Romania. pp. 309–326.

Margaris, A. *et al.* (2005). Speech frames extraction using neural networks and message passing techniques, 1. In *Proceedings of International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2005). Lecture Series on Computers and Computational Sciences*. Brill Academic Publishers. Vol. 4, pp. 384–387.

Misra, M. (1992). Implementation of neural networks on parallel architectures. *PHD Thesis*, University of Southern California.

Misra, M. (1997). Parallel environment for implementing neural networks. *Neural Computing Surveys*, **1**, 48–60.

Pacheco, P. (1997). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc, San Francisco, California.

Quoy, M., S. Moga, P. Gaussier, A. Revel (2000). 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*. Berlin. pp. 289–296. *Lecture Notes on Computer Science*, **1908**.

Scikuta, E. (1997). Structural data parallel neural network simulation. In *Proceedings of 11th Annual International Symposium on High Performance Computing Systems (HPCS'97)*. Winnipeg, Canada.

Serbedzija, N. (1996). Simulating artificial neural networks on parallel architectures. *Computer*, **29**(3), 56–63.

Schikuta, E., T. Fuerle, H. Wanek (2000). Structural data parallel simulation of neural networks. *Journal of System Research and Information Science*, **9**, 149–172.

Snir, M. *et al.* (1998). *MPI – The Complete Reference*. Vol. 1. The MPI Core, 2nd edition. Scientific and Engineering Computational Series, The MIT Press, Massachusetts.

Standish, R. (1999). *Complex Systems Research on Parallel Computers*. `http://parallel.hpc.unsw.edu.au/rks/docs/parcomplex`.

Tomsich, P., A. Rauber, D. Merkl (2000). 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*. Greenwich, London UK. pp. 661–666.

Torresen, J. *et al.* (1994). Parallel back propagation training algorithm for MIMD computer with 2D-torus network. In *Proceedings of 3rd Parallel Computing Workshop (PCW'94)*. Kawasaki, Japan.

Torresen, J., S. Tomita (1998). A review of parallel implementation of back propagation neural n-etworks. In N. Sundararajan and P. Saratchandram (Eds.) *Parallel Architectures of Artificial Neural Networks*. IEEE CS Press.

Weigang, L., N. Correia da Silva (1999). A study of parallel neural networks. In *Proceedings of International Joint Conference on Neural Networks*, Vol. 2. Washington, D.C. pp. 1113–1116.

A. Margaris was awarded the bachelor in physics from Aristotle University of Thessaloniki in 1992, the master of science degree from Sheffield University (Computer Science Department) in 1995 and the doctor of philosophy from University of Macedonia, Thessaloniki (Department of Applied Informatics) in 2003. Currently he teaches informatics at Technological Educational Institute of Thessaloniki. His research interests include the applications of neural networks to a category of different problems, as well as the use of MPI architecture for parallel simulations.

S. Souravlas was awarded the degree in applied informatics from the University of Macedonia in 1998 and the doctor of philosophy from the same department in 2004. Currently, he works as adjunct lecturer at the Department of Computer & Communication Engineering, University of Western Macedonia, Kozani where he teaches digital design, and also at the Department of Marketing and Operations Management, University of Macedonia, where he teaches computing.

E. Kotsialos is a research associate in the University of Macedonia, Thessaloniki, Applied Informatics Dept. His research is in 3-D nonlinear dynamical systems, their bifurcation properties and their modeling, using a variety of tools and methodologies, both in the theoretical as well as in the numerical simulation level.

M. Roumeliotis is an associate professor in the University of Macedonia, Thessaloniki, Applied Informatics Dept. He obtained his PhD from Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA. His field of interest is in the architecture of computer systems and the development of tools for computer systems simulations.

Priešingo sklidimo lygiagrečiojo tinklo modelis ir realizacija naudojant MPI

Athanasios MARGARIS, Stavros SOURAVLAS, Efthimios KOTSIALOS, Manos ROUMELIOTIS

Šio tyrimo tikslas yra sukurti lygiagrečius modelius, kurie imituotų dirbtinių neuroninių tinklų elgseną. Šiame straipsnyje modeliuojamas priešingo sklidimo neuroninis tinklas. Lygiagrečiajai realizacijai naudojamas pranešimų perdavimų funkcijų standartas MPI. Straipsnyje pateikiami priešingo sklidimo neuroninio tinklo nuoseklusis ir lygiagretusis algoritmai. Yra pateikti kelių lygiagretinimo būdų (seansų ir mokymo aibės) rezultatai. Seansų lygiagrečioji sistema sudaryta iš kelių lygiagrečiai veikiančių procesų, kiekvienas jų dirba su visu neuroniniu tinklu tik su skirtingais mokymo parametrais. Mokymo aibės lygiagrečiojoje sistemoje mokymo aibės elementai yra paskirstomi visiems procesams, kiekvienas jų dirba su visu tinklu su savo mokymo aibės fragmentu. Atsižvelgiant į galimą neuroninių struktūrų lygiagretinimo tipą, yra pateikiami du skirtingi būdai: vienas paremtas tarpinio komunikatoriaus idėja, kitame – svorių lentelių pakeitimui ir vidutinės paklaidos kiekviename mokymo etape vertinimui naudojamos nuotolinės prieigos operacijos.