Correctness and Performance Properties of Pipelined Parallel Competitive Neural Network Algorithms on Clusters of workstations

José Castro
Dept. of Computer Engineering
Computing Research Center
Costa Rican Institute of Technology
Cartago, Costa Rica
jcastro@itcr.ac.cr

Jimmy Secretan
Dept. of Electrical and
Computer Engineering
University of Central Florida
Orlando, FL 32816–2786
secretj@cfl.rr.com

Michael Georgiopoulos
Dept. of Electrical and
Computer Engineering
University of Central Florida
Orlando, FL 32816–2786
michaelg@mail.ucf.edu

Ronald DeMara
Dept. of Electrical and
Computer Engineering
University of Central Florida
Orlando, FL 32816–2786
demara@mail.ucf.edu

Georgios Anagnostopoulos
Dept. of Electrical and
Computer Engineering
Florida Institute of Technology
Melbourne, FL 32901
georgio@fit.edu

Avelino Gonzalez
Dept. of Electrical and
Computer Engineering
University of Central Florida
Orlando, FL 32816–2786
gonzalez@pegasus.cc.ucf.edu

Reprint requests to: Michael Georgiopoulos, Electrical and Computer Engineering Department, University of Central Florida, 4000 Central Florida Blvd. Engineering Building 1, Office 407, Orlando, Florida, 32816–2786

Correctness of Parallel Competitive Neural Network Algorithms

ACKNOWLEDGMENT

The authors would like to thank the Computer Research Center of the Technological Institute of Costa Rica, the Institute of Simulation and Training (IST) and the Link Foundation Fellowship program for partially funding this project. This work was also supported in part by the National Science Foundation under grants # CRCD:0203446 and # CCLI:0341601.

Abstract

Competitive neural network learning algorithms share a common structure where exemplars are used to represent categories. However, the time that it takes for them to converge to a solution increases rapidly as the number of patterns used for training increases. In this paper we rely on the common structure of competitive learning algorithms and propose a coarse grain parallelization technique, based on a pipeline approach, to speed up the training process. We prove, among other things, that the workload balance of the proposed parallelization does not exceed a bound which is small and independent of the pipeline length. An implementation of the algorithm is evaluated using a variant of the Fuzzy-ARTMAP neural network learning algorithm that confirms the formal properties in the proofs.

The Fuzzy ARTMAP algorithm is a popular neural network algorithm for solving classification problems.

keywords: Competitive NN Algorithm, BEOWULF cluster, Pipelining, Fuzzy–ARTMAP, Data Mining
I. INTRODUCTION

Competitive neural network learning algorithms have been used extensively and successfully to tackle a wide variety of problems. Examples of competitive neural network algorithms include, Kohonen neural networks (Kohonen, 1990), the Probabilistic neural network (Specht, 1990), ART neural networks (Carpenter, Grossberg, & Rosen, 1991; Carpenter, Grossberg, & Reynolds, 1991; Carpenter, Grossberg, Markuzon, Reynolds, & Rosen, 1992; Anagnostopoulos, 2000), and many others. These competitive neural network algorithms are quite often used in solving clustering or classification problems. There are also other examples of pattern recognition algorithms that are not neural-based, they solve clustering or classification problems, and they belong to the same class of competitive algorithms that we are focusing in this paper. For instance, the popular K-means clustering algorithm (Duda, Hart, & Stork, 2000), and the 1-nearest neighbor classification algorithm (Duda et al., 2000) are classic examples of competitive algorithms that could benefit from the parallelization strategies discussed in this paper.

As computing capacity and electronic databases grow, there is an increasing need to process considerably larger databases. Unfortunately, neural network algorithms, competitive neural network algorithms included, can have a prohibitively slow convergence to a solution, especially when they are trained on large databases. One obvious way to address the problem of slow convergence to a solution is by the use of parallelization. Extensive research has been done on the properties of parallelization of feed–forward multi–layer perceptrons (Torresen & Tomita, 1998). This is probably due to the popularity of this neural network architecture, and also because the backpropagation algorithm, used to train these type of networks, can be characterized mathematically by matrix and vector multiplications, mathematical structures that have been parallelized with extensive success.

In this paper our focus is to improve the convergence speed of competitive neural networks through a parallelization strategy applicable for a pipeline structure in a BEOWULF cluster of workstations. To demonstrate the effectiveness of our proposed parallelization strategy we implemented a variant of the Fuzzy ARTMAP architecture developed by (Anagnostopoulos & Georgiopoulos, 2001) (called no-matchtracking Fuzzy ARTMAP) and tested the implementation on three databases, one real and two artificial. We chose to demonstrate the effectiveness of our proposed parallelization strategy on a member of the ART family of neural networks since, if we demonstrate its effectiveness for Fuzzy ARTMAP, its extension to other ART structures and other competitive algorithms can be accomplished without a lot of effort. This is due to the fact that other ART structures and other neural-based
or non-neural-network based competitive algorithms have a similarly functioning competitive loop as the one that Fuzzy ARTMAP possesses. For example, the advantages of the proposed parallelization approach, applied to Fuzzy ARTMAP, can be readily extended to other ART variants (for instance Gaussian ARTMAP (Williamson, 1996), Ellipsoidal ARTMAP (Anagnostopoulos & Georgiopoulos, 2001), among others).

This paper is organized as follows: Section II presents the competitive learning algorithm structure and examines the computational complexity of competitive learning, which serves as a necessary motivation for the parallelization approach introduced in this paper. Section III talks about parallelization considerations in a BEOWULF cluster of workstations, our platform of choice. Section IV continues with the pseudocode of the parallel competitive learning algorithm. Section V focuses on theoretical results related to the proposed parallelization approach. In particular, the theorems presented in this section show that the parallel competitive learning is equivalent to the sequential competitive learning, and that the processors in the parallel implementation will be reasonably balanced by considering a worst case scenario. Section VI proceeds with experiments and results comparing the performance and speedup achieved by the parallel no-match tracking Fuzzy ARTMAP learning algorithm using a real database (Forrest CoverType from the UCI repository). The article concludes with section VII, where the summarization of our experiences from the conducted work, are delineated.

II. THE COMPETITIVE NEURAL NETWORK ALGORITHM STRUCTURE

Consider a set of $PT$ patterns $\{I^1, I^2, \ldots, I^{PT}\}$ with associated class labels $\text{class}(I^i), \forall i = 1, 2, \ldots, PT$ which, for obvious reasons we will call the training set. Examples of patterns may be photographs, feature vectors, database tuples etc. Usually the $I^i$ patterns are normalized to belong to the unit hypercube $I^i \in [0,1]^M$ where $M$ is the dimensionality of the data. The problem of learning to classify this data concerns the problem of using the information in the training set to create a representation of the problem space, so that when a new pattern that is not present in the training set is given to the learning algorithm, the algorithm will respond with a (frequently) correct class label.

Most competitive algorithms create, through a learning process, exemplars $w$, which we will call templates. The nature or structure of the templates will not concern us, but they may be points in the $M$ dimensional space, hyper–rectangles, ellipsoids, or others. An exemplar (template) can, in most instances, be thought of as the compressed representation of a number of input patterns that chose this exemplar as their representative. For instance, a template
LEARN\(\{I^1, I^2, \ldots, I^{PT}\}, \Delta\)
1 \(\text{templates} \leftarrow \{\}\)
2 \textbf{for each} \(I^r\) \textbf{in} \(\{I^1, I^2, \ldots, I^{PT}\}\)
3 \textbf{do} \(D_r \leftarrow \text{MAXINT}\)
4 \textbf{for each} \(w_j\) \textbf{in} \(\text{templates}\)
5 \textbf{do if} \(D_r > D(w_j, I^r, \Delta)\)
6 \quad \textbf{then}
7 \quad \quad \quad \quad \quad \quad D_r \leftarrow D(w_j, I^r, \Delta)
8 \quad \quad \quad \quad \quad \quad j_{\text{max}} \leftarrow j
9 \quad \quad \textbf{if} (D_r < \text{MAXINT}) \textbf{ and } (\text{class}(I^r) = \text{class}(w_{j_{\text{max}}}))
10 \quad \quad \textbf{then} \(w_{j_{\text{max}}} \leftarrow U(w_{j_{\text{max}}}, I^r, \Delta)\)
11 \quad \quad \textbf{else} \(\text{templates} \leftarrow \text{templates} \cup \text{CREATE-TEMPLATE}(I^r, \Delta)\)
12 \textbf{return} \(\text{templates}\)

Fig. 1. Competitive "on-line training phase" algorithm

CLASSIFY\((I^r, \text{templates}, \Delta)\)
1 \(D_r \leftarrow \text{MAXINT}\)
2 \textbf{for each} \(w_j\) \textbf{in} \(\text{templates}\)
3 \textbf{do if} \(D_r > D(w_j, I^r, \Delta)\)
4 \quad \textbf{then}
5 \quad \quad \quad \quad \quad \quad D_r \leftarrow D(w_j, I^r, \Delta)
6 \quad \quad \quad \quad \quad \quad j_{\text{max}} \leftarrow j
7 \quad \quad \textbf{if} \(D_r < \text{MAXINT}\)
8 \quad \quad \textbf{then return} \(\text{class}(w_{j_{\text{max}}})\)
9 \quad \quad \textbf{else return} \text{None}

Fig. 2. Competitive classification phase algorithm

could be the average of all the input patterns that chose this template as their representative. Given a template and a set \(\Delta\) of parameters of the learning algorithm, an algorithm dependent distance function can be developed that measures the appropriateness of the template \(w\) for representing an arbitrary pattern \(I\)

\[D(w, I, \Delta) \in [0, \infty)\]

Also, an update function on the template \(w\) is usually applied, once the template is identified as the best representation of a pattern \(I\) so that

\[w \leftarrow U(w, I, \Delta)\]

Using these definitions we can express the general competitive learning algorithm by the pseudocode shown in figure 1. Once learning is completed, classification of a pattern \(I\) will proceed by finding the closest template \(w\) to the given pattern and reporting it’s class label. This algorithm is depicted in figure 2.
II.-A. Algorithm time complexity

We can see from the pseudocode in figure 1 that the algorithm tests every input pattern $I$ in the training set against each template $w_j$ at least once. This means that the time it takes the algorithm to execute the for loop for a given input pattern $I$ is: $Time(I) = O(|templates|)$. Also, under the unrealistic condition that the number of templates does not change during training and that the algorithm has a database specific compression ratio which we will call $\kappa$ it is easy to see that the time complexity of the algorithm is

$$O(Algorithm) = O(PT\kappa PT) = O(\kappa PT^2)$$  \hspace{1cm} (1)

quadratic with respect to the number of patterns $PT$ in the training set.

III. BEOWULF CLUSTER CONSIDERATIONS

The BEOWULF cluster of workstations is a network of computers where processes exchange information through the network’s communications hardware. In our case, it consisted of the OPCODE cluster of the Institute of Simulation and Training (IST) in Orlando, Florida. This cluster has 96 AMD nodes, each with dual Athlon-MP 1500+ processors and 512MB of RAM. The nodes are connected through a Fast Ethernet network. Our parallelization tests were made so that we ran no more than one process per processor.

In general, a BEOWULF cluster configuration is a parallel platform that has a high latency. This implies that to achieve optimum performance communication packets must be of large size and small in number. There is no central coordinating entity and protocols must be based on listening/polling schemes.

We had two choices for parallelization design. We can request from each node in the network to process a different input pattern. Or we can request that each node processes the same input patterns at the same time. If we want the parallel implementation to work equivalently to the sequential one the first design will lead to a pipelined approach where each node computes a stage in the pipeline. The second approach will lead to a star master/slave topology where all nodes communicate to a gathering master node. We chose to follow the pipelined approach because in this scenario we are only doing point to point communication, which is a constant time operation in a Fast Ethernet switched network. The star approach tends to degrade communication performance as the size of the gather operation increases.
IV. PARALLEL COMPETITIVE–ALGORITHM IMPLEMENTATION

The learning of a pattern in the competitive algorithm is a one–pass over the pipeline procedure, where different patterns can be processed on the different pipeline steps to achieve optimum parallelization. To understand the parallel framework of the competitive learning algorithm we need the following definitions:

- $n$: number of processors in the pipeline.
- $k$: index of the current processor in the pipeline, $k \in \{0, 1, \ldots, n - 1\}$.
- $p$: packet size, number of patterns sent downstream; $2p =$ number of templates sent upstream.
- $I^i$: input pattern $i$ of current packet in the pipeline. $i \in \{1, 2, \ldots, p\}$.
- $w^i$: current best candidate template for input pattern $I^i$.
- $D^i$: closest distance or best activation for input pattern $I^i$.
- $myTemplates$: set of templates that belong to the current processor.
- $nodes$: variable local to the current processor that holds the total number of templates the process is aware of (its own plus the templates of the other processors).
- $myShare$: amount of templates that the current processor should not exceed.
- $w^{i}_{k-1}$: template $i$ coming from the previous processor in the pipeline.
- $w^{i}_{k+1}$: template $i$ coming from the next processor in the pipeline.
- $w^{i}$: template $i$ going to the next processor in the pipeline.
- $w^{i}_{to(k-1)}$: template $i$ going to previous processor in the pipeline.
- $class(I)$: class label associated with a given input pattern.
- $class(w)$: class label associated with a given template.
- $index(w)$: sequential index assigned to a template.
- $newNodes_{k+1}$: number of new nodes (templates) that were created and that processor $k + 1$ communicates upstream in the pipeline.
- $newNodes_k$: number of new nodes (templates) that were created and that processor $k$ communicates upstream in the pipeline.

The exchange of packets between processors is pictorially illustrated in figure IV. In this figure, the focus is on processor $k$ and the exchange of packets between processor $k$ and its neighboring processors (i.e., processors $k - 1$
Fig. 3. Pipelined parallel implementation of competitive learning algorithm

```plaintext
PROCESS\(\(k, n, p, \Delta\)\)
1: INIT\(\(\Delta\)\)
2: myTemplates \(\leftarrow \{\}\)
3: while continue
4: do
5: while |myTemplates| > myShare
6: do
7: EXTRACT-TEMPLATE \(\left\{myTemplates, \{w^i_{t_0(k-1)}\}\right\}\)
8: SEND-NEXT \((k, n, \{\langle w^i, \Gamma^i, T^i \rangle : i = 1, 2, \ldots, p \}\})\)
9: RECEV-NEXT \((k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p\} . newNodes_{k+1})\)
10: SEND-PREV \((k, \{w^i_{t_0(k-1)} : i = 1, 2, \ldots, 2p\} . newNodes)\)
11: RECEV-PREV \((k, \{w^i_{k+1}, \Gamma^i_{k+1}, T^i_{k+1} : i = 1, 2, \ldots, p\})\)
12: newNodes \(\leftarrow newNodes_{k+1}\)
13: \(S \leftarrow \{w^i_{k+1}\}\)
14: for each \(i \in \{1, 2, \ldots, p\}\)
15: do FIND-WINNER\(\(w^i, \Gamma^i, D^i, S, \Delta\)\)
16: myTemplates \(\leftarrow myTemplates \cup S\)
17: if \(\Gamma^i_{k+1} = \text{none}\)
18: then continue \(\leftarrow \text{FALSE}\)
19: else \(S \leftarrow \{w^i_{t_0(k-1)}\}\)
20: for each \(i \in \{1, 2, \ldots, p\}\)
21: do FIND-WINNER\(\(w^i_{k-1}, \Gamma^i_{k-1}, D^i_{k-1}, S, \Delta\)\)
22: \((\Gamma^i, w^i) \leftarrow (\Gamma^i_{k-1}, w^i_{k-1}, D^i_{k-1})\)
23: for each \(i \in \{1, 2, \ldots, p\}\)
24: do FIND-WINNER\(\(w^i, \Gamma^i, D^i, myTemplates, \Delta\)\)
25: if \(k = n - 1\)
26: then if \(\text{class}(\Gamma^i) = \text{class}(w^i)\)
27: then
28: myTemplates \(\leftarrow myTemplates \cup \{\Gamma^i \wedge w^i\}\)
29: else newTemplate \(\leftarrow \Gamma^i\)
30: index(newTemplate) \(\leftarrow newNodes + \text{nodes}\)
31: myTemplates \(\leftarrow myTemplates \cup \{\Gamma^i, w^i\}\)
32: newNodes \(\leftarrow newNodes + 1\)
33: if newNodes > 0
34: then
35: nodes \(\leftarrow \text{nodes} + \text{newNodes}\)
36: myShare \(\leftarrow \left[\frac{\text{nodes}}{n}\right]\)
37: SEND-NEXT \((k, n, \{\langle \text{none, none, 0} \rangle \})\)
38: RECEV-NEXT \((k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p\} . newNodes_{k+1})\)
39: myTemplates \(\leftarrow myTemplates \cup \{w^i_{k+1} : i = 1, 2, \ldots, 2p\}\)
```

and \(k+1\). The parallel implementation of competitive learning is shown in figure 3. The initialization procedure is algorithm dependent and not shown. Each element of the pipeline will execute the algorithm of figure 3 for as long as there are input patterns to be processed. The input parameter \(k\) tells the process which stage of the pipeline it is, where the value \(k\) varies from 0 to \(n - 1\). After initializing most of the values as empty we enter the loop of lines 3 through 36. This loop continues execution until there are no more input patterns to process. The first activity of
Fig. 4. Exchange of packets between processors. Note, packets are listed for processor “k” only.

each process is to create a packet of excess templates to send back (line 5 to 7). Lines 8 to 11 correspond to the
information exchange between contiguous nodes in the pipeline. The functions SEND-NEXT and RECV-NEXT on
lines 8 and 9, respectively, don’t do anything if the process is the last in the pipeline \( (k = n - 1) \). The same is true
for the function SEND-PREV when the process is the first in the pipeline \( (k = 0) \). On the other hand, the function
RECV-PREV reads input patterns from the input stream if the process is the first in the pipeline. These fresh patterns
will be paired with a dummy template called the uncommitted node with index \( \infty \) as their best representative so
far. On all other cases these functions do the obvious information exchange between contiguous processes in the
pipeline. We assume that all communication happens at the same time and is synchronized. We can achieve this
in an MPI environment by doing non–blocking sends and using an MPI–Waitall to synchronize the receive of
information.

On line 31 we add 2 templates to the template set myTemplates. This is because a new template was created and
the current candidate winner \( w \) is not of the correct category and has to be inserted back into the pool of templates.

The function FIND-WINNER although not presented is also important. This function searches through a set
of templates \( S \) to find if there exists a template \( w^i \) that is a better choice for representing \( I \) than the current best
representative \( w \). If it finds one it swaps it with \( w \), leaving \( w \) in \( S \) and extracting \( w^i \) from it. By sending the
input patterns downstream in the pipeline coupled with their current best representative template we guarantee that
the templates are not duplicated among different processors and that we do not have multiple–instance consistency
Also when exchanging templates between nodes in the pipeline we have to be careful that patterns that are sent downstream do not miss the comparison with templates that are being sent upstream. This is the purpose of lines 13 to 16 (communication with the next one in the pipeline) and lines 19-22 of PROCESS. On line 13 we set $S$ to represent the set of templates that have been sent upstream to node $k$ by node $k+1$. We loop through each pattern, template pair $(I, w)$ (lines 14–16) to see if one of the templates, sent upstream, has a higher activation (bottom-up input) than the ones that were sent downstream; if this is true then the template will be extracted from $S$. The net result of this is that $S$ ends up containing the templates that lost the competition, and therefore the ones that process $k$ should keep (line 16). The converse process is done on lines 19 to 22. On line 19 we set $S$ to represent the set of templates that are sent back to the previous node $k-1$ in the pipeline. On lines 20 to 21 we compare the pattern, template pairs $(I_{k-1}^i, w_{k-1}^i)$ that $k-1$ sent upstream in the pipeline with the templates in $S$ that process $k$ sent downstream in the pipeline. On line 22 we set our current pattern, template pairs to the winners of this competition. The set $S$ is discarded since it contains the losing templates and therefore the templates that process $k-1$ keeps.

Finally, on line 31 of figure 3 we add both the input pattern $I^i$ and the template $w^i$ to the set of templates. This does the obvious myTemplates update except when the template $w^i$ happens to be the uncommitted node in which the addition is ignored.

Once more, we reiterate that the main loop of the process starts with line 3 and ends with line 36. The main loop is executed for as long as there are input patterns to process. The first processor that becomes aware that there are no more input patterns to process is processor 0 (first processor in the pipeline). It communicates this information to the other processors by sending a $(w^i, I^i, D^i) = (\text{none, none, 0})$ to the next processor (see line 37 of figure 3). Lines 38 and 39 of process make sure that the templates that are sent upstream in the pipeline are not lost after the pool of training input patterns that are processed is exhausted.

V. PROPERTIES OF THE PARALLEL COMPETITIVE ALGORITHM

In the sequel we present a series of twelve (12) theorems. These theorems are distinguished in two groups. The group of theorems associated with the correctness of the parallel competitive learning algorithm, and the group of theorems associated with the performance of the competitive learning algorithm. For ease of reference Table I
lists the theorems and their names dealing with the correctness of the algorithm, while Table II lists the theorems dealing with the performance of the algorithm.

The major purpose of these theorems is to prove that the parallel competitive learning algorithm (a) is equivalent to the sequential version of competitive learning, (b) it does not suffer from any inconsistencies, and (c) it exhibits good performance. Examples of inconsistencies are: a template exists in more than one places in the pipeline (not possible as theorem 5.1 (non-duplication) proves), or the first processor in the pipeline is required to send templates upstream (not possible as theorem 5.10 (overflow impossibility) proves). It is worth mentioning that theorems 5.2 through 5.7 facilitate the demonstration of the overflow impossibility theorem. Good performance is dependent on the distribution of templates among the processors in the pipeline (workload balance). An upper bound on the difference between the number of templates that two processors in the pipeline could own has been established through the pipeline length invariance theorem (theorem 5.11) and it is equal to $p + 1$, where $p$ is the packet size. Furthermore, this upper bound is independent of the pipeline depth $n$. For instance, if 100,000 templates are present in the pipeline and $p = 64$, the templates that any two processors possess cannot differ by more than 65 (where $p + 1 = 65$). Due to space limitations, only a few of the theorems are proven here.
Definition 5.1: A template $w^j_k$ is in transit if the template has been received by the current processor from the previous processor in the pipeline, and the current processor has not made the decision yet of whether to send this template to the next processor, previous processor, or keep it. Templates in transit are stored in the $w^j$’s array.

Definition 5.2: A template $w^j_k$ is owned by a processor $i$ in the pipeline if it is stored in the $myTemplates$ array of processor $i$.

Theorem 5.1: Non–duplication – A template $w$ will either be owned by a single processor, or it will be in transit on a single processor (i.e. only one copy of the template exists in the system).

Theorem 5.2: Template awareness delay – The total number of templates that a process $k = 0, 1, \ldots, n – 1$ in the pipeline is aware of is equal to the number of templates that existed in the system $n – k – 1$ iterations ago.

Theorem 5.3: Weak upstream migration precondition – A process $k$ in the pipeline sends templates upstream only if on the current iteration:

$$|myTemplates| > myShare$$

(2)

Theorem 5.4: Upstream bundle size sufficiency – No process in the pipeline, except the first one, can have, at any point in time, an excess of templates greater than $2p$.

Theorem 5.5: Strong upstream migration precondition – If a process $k \in \{0, 1, \ldots, n – 1\}$ in the pipeline sends templates back, then it is true for all $i \in \{1, 2, \ldots, n – k – 1\}$, that $i$ iterations ago process $k + i$ complied with condition 2 and sent templates back.

Theorem 5.6: Strong upstream migration postcondition – If a process $k \in \{0, 1, \ldots, n – 1\}$ in the pipeline sends templates back, then

1) It is true for all $i \in \{0, 1, \ldots, n – k – 1\}$ that $i$ iterations ago process $k$ kept $myShare$ templates

2) All of the values of $myShare$ are the same for all the processes.

3) The templates that each processor keeps are distinct.

Theorem 5.7: Template ownership delay – The templates that on the current iteration a process $k$ owns or has in transit were created at least $n – k – 1$ iterations ago.

Theorem 5.8: Network size lower bound – If a process $k$ sends templates back on a given iteration, then the number of templates $N$ that existed in the system $n – 1 – k$ iterations ago complies with the condition:

$$N > (n – k)myShare$$

(3)
Theorem 5.9: Template ownership bound – A process $k$ in the pipeline cannot have more than myShare templates and cannot own less than $\max(0, myShare - p(2(n - 1 - k) - 1))$

Theorem 5.10: Overflow impossibility – The first process in the pipeline will always be able to absorb the templates that have been sent to it from the next process downstream.

Theorem 5.11: Pipeline depth invariance – The difference in the number of myShare that two arbitrary processes in the pipeline have cannot exceed $p + 1$ where $p$ is the packet size. Note that the difference in number of templates is independent of the pipeline size $n$.

Proof: First, by theorem 5.2 we know that a process $k$ is aware of the number of templates that existed $n - 1 - k$ iterations ago. Also, the largest difference in the number of templates that two process are aware of is found in the difference between process 0 and process $n - 1$. Now, let us assume that process 0 is aware of $\text{nodes}_0$ templates. Since this amount of templates existed $n - 1$ iterations ago and we can create a maximum of $p$ templates per iteration then the maximum number of templates that process $n - 1$ can be aware of is $\text{nodes}_0 + (n - 1)p$. This means that the value of myShare for process 0 is

$$myShare_0 = \left\lfloor \frac{\text{nodes}_0}{n} \right\rfloor \geq \frac{\text{nodes}_0}{n} \tag{4}$$

and the value of myShare for process $n - 1$ is at the most

$$myShare_{n-1} = \left\lfloor \frac{\text{nodes}_0 + (n - 1)p}{n} \right\rfloor \leq \frac{\text{nodes}_0 + (n - 1)p}{n} + 1 \tag{5}$$

We also know that the number of templates that each processor $k$ owns is less than or equal to myShare$_k$. Hence, the maximum amount of difference in templates between 2 processors in the pipeline is less than or equal to

$$myShare_{n-1} - myShare_0 = \left\lfloor \frac{\text{nodes}_0 + (n - 1)p}{n} \right\rfloor - \left\lfloor \frac{\text{nodes}_0}{n} \right\rfloor \leq \frac{\text{nodes}_0 + (n - 1)p}{n} + 1 - \frac{\text{nodes}_0}{n} = \frac{(n - 1)p}{n} + 1 \leq p + 1$$

Theorem 5.12: Workload balance variance bound – In a pipeline with an arbitrary number of processors and a downstream packet size $p$, the standard deviation of the number of templates that each processor owns cannot
Proof: Given that in the parallel competitive learning algorithm there are many templates in transit we cannot know exactly how many templates each process possesses. We can though, approximate a worst case workload balance scenario if we assume, as will usually be the case, that the number of comparisons that a given process performs on each iteration will be proportional to the number of templates that it is allowed to possess or myShare. In a worst case scenario, on every iteration the network will be creating $p$ new templates so process $k$ will have a value of

$$nodes_k = nodes_0 + kp$$

The expected value of myShare for this worst case scenario will be

$$\text{Avg}(\text{myShare}) = \sum_{k=0}^{n-1} \frac{nodes_0 + kp}{n} = \frac{nodes_0}{n} + \frac{p}{2n}(n-1)$$

and the variance will be

$$\frac{1}{n} \sum_{k=0}^{n-1} \left( \frac{nodes_0 + kp}{n} - \frac{nodes_0}{n} - \frac{p}{2n}(n-1) \right)^2 = \frac{p^2 n^2 - 1}{12n^2}$$

after some algebraic calculations we can show that the variance is equal to

$$p^2 n^2 - 1 \geq \frac{p^2}{12\sqrt{3}}$$

this result gives us a standard deviation of

$$\sqrt{\frac{p^2 n^2 - 1}{12n^2}} = \frac{p}{2\sqrt{3}} \sqrt{1 - n^{-2}} < \frac{p}{2\sqrt{3}}$$

If, for example, we use a packet size of 64 patterns, then the worst possible standard deviation in the value of myShare would not exceed

$$\frac{64}{2\sqrt{3}} = 32 \div \sqrt{3} = 18.4752$$
regardless of the pipeline size \( n \).

VI. EXPERIMENTS

As we have mentioned earlier we chose to demonstrate the effectiveness of our parallelization strategy on one representative of the class of neural-based competitive algorithms, the no-match tracking Fuzzy ARTMAP, which is a variant of the well-known Fuzzy ARTMAP algorithm. The competitive loop in the no-match tracking Fuzzy ARTMAP shares a lot of similarities with the competitive loop of other neural network competitive algorithms, as well as competitive algorithms that are not neural-based. Hence, by demonstrating the effectiveness of our approach on the no-match tracking Fuzzy ARTMAP we are, in essence, assuring its effectiveness on many other competitive algorithms.

Experiments were conducted on 3 databases: 1 real–world database and 2 artificially–generated databases (Gaussian distributed data), but we are reporting here only on the results for the real database. The real database used was the Forest CoverType database provided by Blackard (Blackard, 1999), and donated to the UCI Machine Learning Repository (Blake & Merz, 1998). The database consists of a total of 581,012 patterns each one associated
with 1 of 7 different forest tree cover types. The number of attributes of each pattern is 54, but this number is misleading since attributes 11 to 14 are actually a binary tabulation of the attribute *Wilderness-Area*, and attributes 15 to 54 (40 of them) are a binary tabulation of the attribute *Soil-Type*. The original database values are not normalized to fit in the unit hypercube. Thus, we transformed the data to achieve this (required by the Fuzzy ARTMAP neural network). There are no omitted values in the data. Patterns 1 through 512,000 were used for training. The test set for all trials were patterns 561,001 to 581,000. Classification performance of different machine learning algorithms for this database has been reported in the range of 75% (Blackard, 1999).

Training set sizes of $1000 \times 2^i, i \in \{5, 6, \ldots, 9\}$, that is 32,000 to 512,000 patterns were used for the training of no-matchtracking Fuzzy ARTMAP and pipelined no matchtracking Fuzzy ARTMAP. The test set size was fixed at 20,000 patterns. The number of processors in the pipeline varied from $p = 1$ to $p = 32$. Pipeline sizes were also increased in powers of 2. The packet size used was equal to 64.

To avoid additional computational complexities in the the experiments (beyond the one that the size of the training set brings along) the values of the ART network parameters i.e., the values chosen were the ones that gave reasonable results for the database of focus. For every combination of $(p, PT) = (partition, training set size)$ values we conducted 12 independent experiments (training and performance phases), corresponding to different orders of pattern presentations within the training set. As a reminder Fuzzy ARTMAP performance depends on the order of pattern presentation within the training set.

All the tests where conducted on the OPCODE BEOWULF cluster of workstations of the Institute for Simulation and Training in Orlando, Florida. This cluster consists of 96 nodes, with dual Athlon 1500+ processors and 512MB of RAM. The runs were done in such as way as to utilize half as many nodes as $p$ (packet size; chosen equal to 64 for the Forrest CoverType database). Thus, there were two MPI processes per node, one per processor.

<table>
<thead>
<tr>
<th>Examples (Thousands)</th>
<th>Classification Performance</th>
<th>Average Templates Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>70.29</td>
<td>5148.83</td>
</tr>
<tr>
<td>64</td>
<td>74.62</td>
<td>11096.66</td>
</tr>
<tr>
<td>128</td>
<td>75.05</td>
<td>22831.0</td>
</tr>
<tr>
<td>256</td>
<td>77.28</td>
<td>49359.33</td>
</tr>
<tr>
<td>512</td>
<td>79.28</td>
<td>100720.75</td>
</tr>
</tbody>
</table>

**TABLE III**

*COverType Run Statistics*
The metrics used to measure the performance of the pipelined approach were (1) Classification performance of pipelined no matchtracking Fuzzy ARTMAP (Higher classification performance is better), (2) Size of the trained, pipelined, no matchtracking Fuzzy ARTMAP (smaller size is better), and (3) Speedup of pipelined no-matchtracking Fuzzy ARTMAP compared to the no-matchtracking Fuzzy ARTMAP.

To calculate the speedup, we simply measured the CPU time for each run. The classification performance of the pipelined no-match tracking Fuzzy ARTMAP ranged from 70% (when the training set consisted of 32,000 patterns) to 79% (when the training set consisted of 512,000 patterns). This compares very favorably with the best performance results reported in the literature (performance of 75%). The number of templates created with the pipelined no-matchtracking Fuzzy ARTMAP was one fifth (approximately) of the number of patterns used in its training phase (e.g., with 256,000 training patterns approximately 50,000 templates were created). The complete performance results (in terms of classification performance and size of the neural network architecture created are shown in Table III; note that the size of the neural network architecture is proportional to the number of templates created). Note, that the no-match tracking Fuzzy ARTMAP is not (due to its design) producing the smallest possible size ART architecture that someone can obtain within the class of ART architectures. Nevertheless, the no-match tracking Fuzzy ARTMAP algorithm is a perfect example of an algorithm to demonstrate the effectiveness (primarily in terms of speed-up observed) of our approach in terms of handling competitive-based algorithms. The obvious benefits (in terms of speed-up) of our parallelization strategy are illustrated in figure 5. One important conclusion from these results is that the speed-up achieved using the pipelined no-matchtracking Fuzzy ARTMAP grows linearly with the number of processors used in the pipeline. Also, we notice that the slope of increase varies depending on the number of patterns used in the training phase of Fuzzy ARTMAP. Furthermore, for 32,000 training patterns and 64,000 training patterns the speed-up curve exhibits a knee (saturation phenomenon). This is likely due to the fact that for the smaller training sets, the overhead for pattern transfer becomes more pronounced. This saturation is more obvious for the 32,000 training patterns than for the 64,000 patterns. This phenomenon is not observed for training patterns 128,000, 256,000 or 512,000.

VII. SUMMARY - CONCLUSIONS

We have produced a pipelined implementation of the no-matchtracking Fuzzy ARTMAP algorithm. This implementation can be extended to other neural network architectures that have similar competitive structure as
Fuzzy ARTMAP, as well as to other competitive algorithms that are not neural-network based. For instance, it can be extended to “competitive” neural networks, such as PNN, Kohonen neural networks, as well as to other “competitive” classifiers or clusterers (such as K-means, 1 and k nearest neighbor).

We have also introduced and proven a number of theorems pertaining to our pipelined implementation. The major purpose of these theorems was to show that the parallel competitive algorithm (a) is equivalent with the sequential version of the competitive algorithm, (b) it does not suffer from inconsistencies, and (c) it exhibits good performance. In particular, the good performance of the parallel no-match tracking Fuzzy ARTMAP was exhibited by observing the linear speed-up achieved as the number of processors in the pipeline increased from 1 to 32.

We believe that our objective of providing a strategy for implementing competitive algorithms on a BEOWULF cluster has been accomplished and a clear evidence of this assertion are the speed-up results exhibited by the pipelined no-match tracking Fuzzy ARTMAP and illustrated in figure 5.

REFERENCES


This work has been submitted to the the 2005 International Conference of Parallel Processing for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.