Parallelizing the Fuzzy ARTMAP Algorithm on a Beowulf Cluster

Jimmy Secretan(*), José Castro(**), Michael Georgiopoulos(*),
Joe Tapia(*), Amit Chadha(*), Brian Huber(*), Georgios Anagnostopoulos(***), Samuel Richie(*)
(*) Dep. of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816
(**) Comp. Eng., Instituto Technologico de Costa Rica, Cartago, Costa Rica
(***) Dept of Electrical and Computer Engineering, Florida Institute of Technology, Melbourne, FL, 32901

***Abstract*— Fuzzy ARTMAP neural networks have been proven to be good classifiers on a variety of classification problems. However, the time that it takes Fuzzy ARTMAP to converge to a solution increases rapidly as the number of patterns used for training increases. In this paper we propose a coarse grain parallelization technique, based on a pipeline approach, to speed-up Fuzzy ARTMAP's training process. In particular, we first parallelized Fuzzy ARTMAP, without the match-tracking mechanism, and then we parallelized Fuzzy ARTMAP with the match-tracking mechanism. Results run on a Beowulf cluster with a well known large database (Forrest Covertype database from the UCI repository) show linear speedup with respect to the number of processors used in the pipeline.**

I. Introduction

Neural Networks have been used extensively and successfully to tackle a wide variety of problems. As computing capacity and electronic databases grow, there is an increasing need to process considerably larger databases. Neural network algorithms can have a prohibitively slow convergence to a solution, especially when they are trained on large databases. Even one of the fastest (in terms of training speed) neural network algorithms, the Fuzzy ARTMAP algorithm [3], and its faster variations ([6], [9]) tend to converge slowly to a solution as the size of the network increases.

One obvious way to address the problem of slow convergence to a solution is by the use of parallelization. This paper focuses on parallelization strategies for FAM on a Beowulf cluster. A Beowulf cluster is a collection of standard PC workstations formed into a single, cohesive supercomputer by a fast network and open source software. For many applications, especially those of interest to the data mining community, the Beowulf architecture offers unparalleled performance for the price.

Regarding the parallelization of ART neural networks it is worth mentioning the work by Manolakos [8] who has implemented the ART1 neural network [4] on a ring of processors, and the work of Malkani and Vassiliadis [7], who have implementated Fuzzy-ARTMAP on a hypercube architecture. In the latter paper, a hypercube topology is utilized for transferring data to all of the nodes involved in the computations. Each of the processors maintains a subset of the architecture's templates, and finds the template with the maximum match in its local collection. Finally, in its d-dimensional hypercube, it finds the global maximum through d different synchronization operations. This can limit the scalability of this approach.

The Fuzzy ARTMAP neural network has many desirable characteristics, such as the ability to solve any classification problem, the capability to learn from data in an on-line mode, the advantage of providing interpretations for the answers that it produces, the capacity to expand its size as the problem requires it and the ability to recognize novel inputs, among others. Due to all of its above properties it is worth investigating Fuzzy ARTMAP's parallelization in an effort to improve its convergence speed to a solution when it is trained with large datasets. In this paper, our focus is to improve the convergence speed of ART-like neural networks through a parallelization strategy applicable for a pipeline structure (Beowulf cluster). We chose to demonstrate the effectiveness of our proposed parallelization strategy on Fuzzy ARTMAP since, if we demonstrate its effectiveness for Fuzzy ARTMAP, its extension to other ART structures can be accomplished without much additional effort. This is due to the fact that the other ART structures share a lot of similarities with Fuzzy ARTMAP, and as a result, the proposed parallelization approach can be readily extended to other ART variants.

II. Fuzzy ARTMAP Algorithm

The Fuzzy ARTMAP neural network and its associated architecture was introduced by Carpenter and Grossberg in their seminal paper [3]. Kasuba [6] and Taghi, Baghmisheh, and Pavesic [9] presented a simplified version of the original Fuzzy ARTMAP architecture that was equivalent to Fuzzy ARTMAP for classification problems. In our paper, we have implemented the simplified Fuzzy ARTMAP version from Taghi, called SFAM2.0, that we refer to as FS-FAM or alternately as Fuzzy ARTMAP. We assume that the reader is familiar with the simplified Fuzzy ARTMAP architecture consisting of an input layer, a category representation layer (where compressed representations of the inputs are formed) and an output layer.

FS-FAM can operate in two distinct phases: the *training phase* and the *performance phase*. In the training phase of FS-FAM a collection of input/output (such as $\{(\mathbf{I}^1, label(\mathbf{I}^1)), \ldots, (\mathbf{I}^r, label(\mathbf{I}^r)), \ldots, (\mathbf{I}^{PT}, label(\mathbf{I}^{PT}))\}$) are repeatedly presented to FS-FAM until FS-FAM learns the desired mappings from inputs to outputs (referred to as labels

of inputs). The training process in FS-FAM is succinctly described in Taghi's et al., paper [9]. We repeat it here to give the reader a good, well-explained overview of the operations involved in its training phase.

1) Find the nearest category in the category representation layer of FS-FAM that "resonates" with the input pattern.
2) If the labels of the chosen category and the input pattern match, update the chosen category to be closer to the input pattern.
3) Otherwise, reset the winner, temporarily increase the resonance threshold (called *vigilance parameter*), and try the next winner. This process is referred to as match-tracking.
4) If the winner is uncommitted, create a new category (assign the representative of the category to be equal to the input pattern, and designate the label of the new category to be equal to the label of the input pattern).

The nearest category to an input pattern $\mathbf{I}^r$ presented to FS-FAM is determined by finding the category that maximizes the function:

$$T_j(\mathbf{I}^r, \mathbf{w}_j, \alpha) = \frac{|\mathbf{I}^r \wedge \mathbf{w}_j|}{\alpha + |\mathbf{w}_j|} \quad (1)$$

The resonance of a category is determined by examining if the function, called *vigilance ratio*, and defined below

$$\rho(\mathbf{I}^r, \mathbf{w}_j) = \frac{|\mathbf{I}^r \wedge \mathbf{w}_j|}{|\mathbf{I}^r|} \quad (2)$$

satisfies the following condition:

$$\rho(\mathbf{I}^r, \mathbf{w}_j) \geq \rho \quad (3)$$

If the label of the input pattern ($\mathbf{I}^r$) is the same as the label of the resonating category, then the category's template ($\mathbf{w}_j$) is updated as follows:

$$\mathbf{w}_j = \mathbf{w}_j \wedge \mathbf{I}^r \quad (4)$$

If the category j is chosen as the winner and it resonates, but the label of this category $\mathbf{w}_j$ is different than the label of the input pattern $\mathbf{I}^r$, then this category is reset and the vigilance parameter ρ is increased to the level:

$$\frac{|\mathbf{I}^r \wedge \mathbf{w}_j|}{|\mathbf{I}^r|} + \epsilon \quad (5)$$

In the above equations the quantities α, ρ, and ϵ are FS-FAM network parameters; α usually takes small positive values, ρ is chosen as a value in the interval $[0, 1]$, and ϵ is a very small positive parameter.

In all of the above equations (equations (1)-(5)) there is a specific operator involved, called *fuzzy min operator*, and designated by the symbol $\wedge$. Actually, the fuzzy min operation of two vectors $\mathbf{x}$, and $\mathbf{y}$, designated as $\mathbf{x} \wedge \mathbf{y}$, is a vector whose components are equal to the minimum of components of $\mathbf{x}$ and $\mathbf{y}$. Another specific operator involved in these equations is designated by the symbol $|\cdot|$. In particular, $|\mathbf{x}|$ is the size of a vector $\mathbf{x}$ and is defined to be the sum of its components.

In the performance phase of FS-FAM, a test input is presented to FS-FAM and the category node in FS-FAM that has the maximum bottom-up input is chosen. The label of the chosen category is the label that FS-FAM predicts for this test input. By knowing the correct labels of test inputs, belonging to a test set, we can calculate the classification error of FS-FAM for this test set.

A simplification that can be applied to the FS-FAM algorithm is the elimination of the match-tracking process. This modification was originally proposed by Anagnostopoulos [1] and turned out to yield improved classification performance on some databases. Our interest in using this FS-FAM variant lies in the fact that it simplifies the FS-FAM algorithm and allows one to concentrate on the parallelization of the competition loop in Fuzzy ARTMAP.

III. Parallel FS-FAM Architectures

The design of the parallel FS-FAM implementation used in this paper was somewhat inspired by the architecture in [8]. In this paper, we present two variations of the parallel FS-FAM, the parallel no match-tracking FS-FAM, and the parallel FS-FAM (which includes match-tracking). While these parallel implementations could be adapted to a physical pipeline/ring network topology, the run environment was emulated by nodes of the Beowulf cluster connected by a standard switch (star topology). The different nodes of the Beowulf were logically arranged in a pipeline, with the capability of bidirectional communication with their neighbors. The input patterns to be learned are read into the first node of the pipeline. Each node in the pipeline has its own collection of templates, against which the input patterns are compared. The input patterns are sent in batches, whose size is an adjustable parameter of the program. As a batch of inputs enters a node, it is processed sequentially. That is, the processing always starts with the first input pattern in the batch. The incoming patterns search through the node's collection of available templates. The maximum template that meets the vigilance is then coupled with the incoming input pattern. This coupling removes the pattern from the available pool of templates for the node. In all subsequent comparisons, if a better template is found, it is switched out with the one currently coupled to the input. The template comparisons proceed until the input pattern reaches the end of the pipeline. At this point, if the chosen template is mapped to the correct output, then learning of the pattern by the template ensues. If on the other hand the chosen template is mapped to the wrong output two different strategies are implemented. In the no match-tracking FS-FAM case a new template is added to the last node's template list which learns the designated input pattern. In the FS-FAM case the match-tracking mechanism is enforced and the input pattern is re-presented to the first node in the pipeline with an increased vigilance value and the search for the right template starts all over again. In addition to processing inputs the system is also performing load balancing. New templates are created only at

the end stage of the pipeline. To achieve load balancing the last node's pool of excess templates, must be re-distributed to other nodes, further upstream in the pipeline. At the time that each node sends its input patterns and winning templates forward through the pipeline, it also sends its excess templates backward through the pipeline. An illustration of the flow of input patterns and templates through the system can be seen in figure 1. In this figure, the focus is on processor k and the exchange of information (patterns and templates) between processor k and its neighboring processors (i.e., processors $k-1$ and $k+1$).

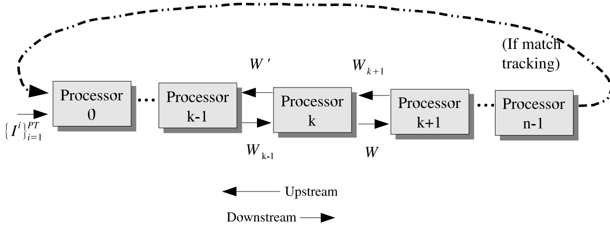

Fig. 1. Diagram to illustrate the exchange of input patterns and templates in the Beowulf processor pipeline.

A. No Match-Tracking Parallel FS-FAM

As part of our previous research, we presented a parallel pipelined version of the no match-tracking FS-FAM [5]. When inputs reach the end of the pipeline in the no match-tracking FS-FAM, they either update their associated templates or create new templates. This is the final point for all inputs. And as templates accumulate in the last processor in the pipeline, they are load balanced in the way explained earlier. Because of the lack of space in this paper, we refer the reader to our previous work for more a thorough explanation of the parallelization of the no match-tracking FS-FAM [5]. Note that in [5], we presented proofs of the parallel no match-tracking FS-FAM's equivalence to its serial counterpart, as well as description of some of its favorable load balancing properties. Throughout the rest of this paper, we concentrate on the generalization of the parallel no match-tracking FS-FAM design for the case of the match-tracking FS-FAM. However, we present scaling results for the no match-tracking FS-FAM for comparison purposes.

B. Parallel FS-FAM with Match-Tracking

The parallel implementation of FS-FAM is shown in figure 2. The variables involved are as follows:

- 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.
- $(\mathbf{I}, \mathbf{w}, T, \rho)$: 4-tuple corresponding to the format of the elements that are packed downstream in the pipeline. $\mathbf{I}$ is the input pattern, $\mathbf{w}$ is the current best candidate template for input pattern $\mathbf{I}$. T is the activation of the pattern with the given template. And ρ is the current value of the vigilance parameter for this input pattern.
- *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.
- $\mathbf{W}_{k-1}$: *Ordered* set of 4-tuples coming from the previous processor in the pipeline.
- $\mathbf{W}_{k+1}$: Set of templates (*not* 4-tuples) coming from the next processor in the pipeline.
- $\mathbf{W}$: *Ordered* set of 4-tuples going to the next processor in the pipeline.
- $\mathbf{W}'$: Set of templates (*not* 4-tuples) going to previous processor in the pipeline.
- *class*$(\mathbf{I})$: class label associated with a given input pattern.
- *class*$(\mathbf{w})$: class label associated with a given template.
- *index*$(\mathbf{w})$: 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*: number of new nodes (templates) that were created and that processor k communicates upstream in the pipeline.

For the sake of brevity, we omit the pseudocode for the utility functions used here, but provide their high level descriptions. For a more thorough explanation of these functions, the reader is referred to [5]. To begin with, there are the network communications functions. They behave exactly as their names would indicate. The SENDNEXT functions, sends data to the next processor in the pipeline, and so fourth. The INIT function serves to initialize the variables and data structures for the algorithm. The function FINDWINNER is vital to the algorithm. This function searches through a set of templates $\mathcal{S}$ to find if there exists a template $\mathbf{w}^i$ that is a better choice for representing $\mathbf{I}$ than the current best representative $\mathbf{w}$. If it finds one it swaps it with $\mathbf{w}$, leaving $\mathbf{w}$ in $\mathcal{S}$ and extracting $\mathbf{w}^i$ from it.

Each processor in the pipeline will execute the algorithm of figure 2 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 2 through 43. This loop continues execution until there are no more input patterns to process. As before, the first activity of each process is to create a packet of excess templates to send back (line 3 to 5). Lines 6 to 9 correspond to the information exchange between contiguous nodes in the pipeline. The function RECVNEXT on line 7, does not do anything if the process is the last in the pipeline ($k = n-1$). The same is true for the function SENDPREV when the process is the first in the pipeline ($k = 0$). On the other hand, function SENDNEXT sends packets to the *first* processor in the pipeline, when a given input pattern is

forced to engage the match-tracking mechanism. In this case, the input pattern will increment its vigilance in its 4-tuple and start afresh at the beginning of the pipeline (process 0). The function RECVPREV does the normal procedure if it is not the first process in the pipeline. If it is the first process though, it will receive the pattern 4-tuples that come from the last process in the pipeline (the ones that are engaged in match-tracking), and read input patterns from the input stream if the amount of match-tracking packets is less than p. The fresh patterns from the input stream will be paired with a dummy template called the *uncommitted* node with index ∞ as their best representative so far. On all other cases these functions do the obvious information exchange between contiguous processes in the pipeline.

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 issues. 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 11 to 13 (communication with the next one in the pipeline) and lines 18-20 of PROCESS (communication with the previous process in the pipeline). We loop through each 4-tuple (lines 11–13) 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 $\mathbf{W}_{k+1}$. The net result of this is that $\mathbf{W}_{k+1}$ ends up containing the templates that lost the competition, and therefore the ones that process k should keep (line 13). The converse process is done on lines 18 to 20. Here we compare the pattern, template pairs 4-tuples that $k-1$ sent upstream in the pipeline with the templates in W' that process k sent downstream in the pipeline. On line 20 we set our current 4-tuple to the winners of this competition. The set W' is discarded since it contains the losing templates and therefore the templates that process $k-1$ keeps.

The primary competition loop is in lines 21 to 39. On line 22, we start by comparing the pertinent 4-tuples to the processor's main collection of templates with the FINDWINNER function as described earlier. If it is on the last processor ($n-1$), there are additional steps that must be taken. If the final associated template is uncommitted, we add a new template to the system (lines 25-27). We also remove the 4-tuple from $\mathbf{W}$ (line 28), as we are done processing it. If a regular (committed) template has been associated with the input in question, and it maps to the correct class, then we update the template as per the Fuzzy ARTMAP algorithm (lines 31 to 34). Again, as before, the 4-tuple in question is removed from $\mathbf{W}$ (line 34). If the labels of the input and template do not match, we must use our match tracking mechanism (lines 35-38). We first increase its associated vigilance (line 36). We then remove the template that was associated with the input back into the processor's general pool and reset the template in the 4-tuple to be the uncommitted node. Since this 4-tuple is not removed from $\mathbf{W}$ as in the other cases, it remains to be sent back to processor 0, and start its competition loop all over again.

A processor continues until it receives nothing in $\mathbf{W}_{k-1}$. Finally, lines 45 and 46 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.

IV. EXPERIMENTS

The database used for the testing the performance of both the parallel no match-tracking FS-FAM and parallel FS-FAM was the Forest CoverType database provided by Blackard and donated to the UCI Machine Learning Repository. The database consists of a total of 581,012 patterns, with 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. There are no omitted values in the data. Patterns 1 through 512,000 were used for training. The test set consisted of patterns 561,001 to 581,000. Although lack of space does not allow a comprehensive comparison of how different classification algorithms performed on this database, Blackard cites performance of 70% for backpropagation neural networks and 58% for Linear Discriminant Analysis [2]. Training set sizes of $1000 \cdot 2^i$, where $i \in \{5,6,7,8,9\}$ were used, that is 32,000 to 512,000 patterns were used for the training of parallel no match-tracking FS-FAM and FAM. The test set size, as mentioned above, was fixed at 20,000 patterns. The number of processors in the pipeline varied from $p=1$ to $p=32$, in powers of 2 (obviously the case of $p=1$ corresponds to the sequential no match-tracking FS-FAM, and FS-FAM). To avoid additional computational complexities in the experiments the values of the ART network parameters ρ and α were fixed (i.e., the values chosen were ones that gave reasonable results). For every combination of (p, PT)= (pipeline size, training set size) values, we conducted 12 independent experiments (training and performance phases), corresponding to different orders of pattern presentations within the training set. All results reported are averages over the 12 runs. All the tests where conducted on the OPCODE Beowulf cluster at the Institute for Simulation and Training, an institute affiliated with the University of Central Florida. This cluster consists of 96 nodes, with dual Athlon 1500+ processors and 512MB of RAM. The implementation of the algorithm was done in C++ with the MPI (Message Passing Interface) libraries. MPI provides a simple interface for communication among processes running on either one node or several different nodes. The runs were done in such as way as to utilize half as many nodes as p. Thus, there were two MPI processes per node, one per processor.

The metrics used to measure the performance of the pipelined approach were:

Fig. 2: Pseudocode for the parallel FS-FAM.

```
Procedure: Process(k, n, ρ̄_a, α, ε, p)
1  Init(p);
2  while continue do
3      W' = {};
4      while |myTemplates| > myShare do
5          ExtractTemplate(myTemplates, W');
6      SendNext(k, n, W);
7      RecvNext(k, n, W_{k+1}, newNodes_{k+1});
8      SendPrev(k, W', newNodes);
9      RecvPrev(k, n, p, ρ̄_a, α, W_{k-1});
10     newNodes= newNodes_{k+1};
11     foreach (I, w, T, ρ) ∈ W do
12         FindWinner(I, w, T, ρ, α, W_{k+1});
13     myTemplates= myTemplates ∪ W_{k+1};
14     W = {};
15     if |W_{k-1}| == 0 then
16         continue= FALSE;
17     else
18         foreach (I, w, T, ρ) ∈ W_{k-1} do
19             FindWinner(I, w, T, ρ, α, W');
20             W = W ∪ { I, w, T, ρ };
21         foreach (I, w, T, ρ) ∈ W do
22             FindWinner(I, w, T, ρ, α, myTemplates);
23             if k == n − 1 then
24                 if w == uncommitted then
25                     newTemplate= I;
26                     index(newTemplate) = newNodes+
                       nodes;
27                     myTemplates= myTemplates ∪
                       {newTemplate};
28                     W = W - { I, w, T, ρ };
29                     newNodes= newNodes+ 1;
30                 else
31                     if class(I) == class(w) then
32                         w = I ∧ w;
33                         myTemplates= myTemplates ∪
                           {w};
34                         W = W - { I, w, T, ρ };
35                     else
36                         ρ = ρ(I,w) + ε;
37                         myTemplates= myTemplates ∪
                           {w};
38                         w = uncommitted;
39
40         if newNodes> 0 then
41             nodes= nodes+newNodes;
42             myShare= ⌈ nodes/n ⌉;
43
44 SendNext( k, n, W);
45 RecvNext( k, n, W_{k+1}, newNode_{k+1});
46 myTemplates= myTemplates ∪ W_{k+1};
```

1) Classification performance of pipelined no match-tracking FS-FAM and pipelined FS-FAM.
2) Size of the trained, pipelined, no match-tracking FS-FAM and pipelined FS-FAM.
3) Speedup of pipelined no match-tracking FS-FAM and FS-FAM compared to their sequential counterparts.

To calculate the speedup, we simply measured the CPU time for each run.

V. Results

The Forrest Covertype results are depicted in Tables I and II (parallel no match-tracking FS-FAM and parallel FS-FAM average classification performance and average number of templates created), and in Figures 3 and 4 (speed-up of the parallel FS-FAM versions compared to their sequential counterparts). As seen in [1], the no match-tracking version of the algorithm can yield increased classification performance at the expense of creating more templates in the system. The speed-up curves for the parallel no match-tracking FS-FAM and, to a lesser degree, the ones for the FS-FAM exhibit a linear behavior with respect to the number of processors in the pipeline. The speed-up curves though level off after we reach a certain number of processors in the pipeline for lower values of training patterns in our training collection; this is due to the fact that for smaller number of training patterns the number of templates created is not large enough to justify the usage of processors beyond a certain number. It is also worth emphasizing that the speed-ups achieved by the no match-tracking parallel FS-FAM are more impressive, compared to the corresponding speed-ups attained by the parallel FS-FAM. This result is also expected, since the number of templates that the no match-tracking parallel FS-FAM creates is significantly larger than the corresponding number of templates that the parallel FS-FAM creates (at times 10 times as many). In essence, the benefits of the proposed parallelization strategy are more pronounced for larger datasets and/or larger ART architectures.

The speedup curve for the FS-FAM has a spike toward the beginning of the curve. The spike occurs as the scaling goes up relatively well for p = 2 and then drops down to a more moderate scaling trend for $p \geq 4$. This is most likely related to the cluster design itself. As mentioned earlier, the cluster consists of dual processor nodes, with each processor running a copy of the program. In the case with p=2, this means both copies of the program were communicating on the same motherboard, which is much faster than the fast Ethernet network. This suggests that the parallel FS-FAM architecture's performance will improve by utilizing faster networking technologies. For 32k inputs, the graph is superlinear for the 2 processor case. It is very likely that this is because of caching. For the dimensionality of the templates (108, 4-bytes floats), and the average number of templates generated in this case (1263.09), split up between two processors is equal to about 256k bytes. This means that in this case, the templates fit much better into the cache than they did in the single processor case.

Examples (1000s)	Avg. Classification	Avg. Templates
32	70.29	5148.83
64	74.62	11096.66
128	75.05	22831
256	77.28	49359.33
512	79.28	100720.75

TABLE I

PARALLEL NO MATCH-TRACKING FS-FAM AVERAGE CLASSIFICATION PERFORMANCE AND AVERAGE NUMBER OF TEMPLATES CREATED WITH THE FORREST COVERTYPE DATA

Examples (1000s)	Avg. Classification	Avg. Templates
32	69.84	1263.09
64	73.26	2147.36
128	73.30	3346.64
256	73.93	5178.27
512	75.13	8013.82

TABLE II

PARALLEL FS-FAM AVERAGE CLASSIFICATION PERFORMANCE AND AVERAGE NUMBER OF TEMPLATES CREATED WITH FORREST COVERTYPE DATA.

VI. SUMMARY/CONCLUSIONS

We have produced a pipelined implementation of the no match-tracking FS-FAM and FS-FAM algorithms. This implementation can be extended to other ART architectures that have similar competitive structure as FS-FAM. It can also be extended to other neural network architectures that are designated as "competitive" neural networks, such as PNNs, RBFs, as well as other "competitive" classifiers. We have introduced and proven a number of theorems (see [5]) that show that the pipeline implementations of FS-FAM are efficient and correct. These theorems were omitted due to lack of space, but the interested reader can consult [5] for more details. We believe that our objective of appropriately implementing FS-FAM on the Beowulf cluster has been accomplished, as evidenced by Figures 3 and 4.

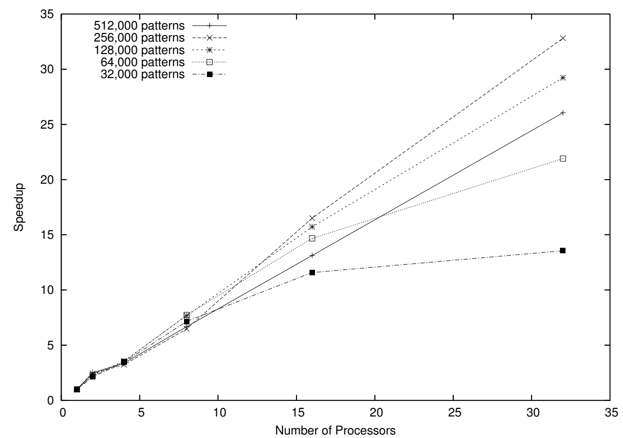

Fig. 3. Speedup of the parallel NMT-FS-FAM algorithm for differing numbers of processors and input patterns.

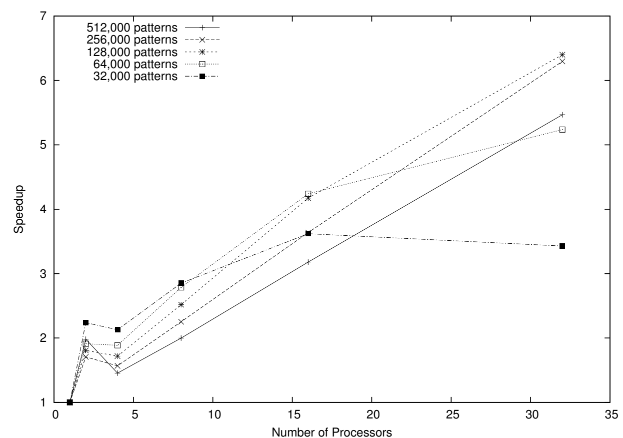

Fig. 4. Speedup of the parallel version of the FS-FAM algorithm for differing numbers of processors and input patterns.

ACKNOWLEDGMENT

José Castro 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.

REFERENCES

[1] G. C. Anagnostopoulos, "Putting the utility of match tracking in fuzzy ARTMAP to the test," in *Proceedings of the Seventh International Conference on Knowledge–Based Intelligent Information Engineering*, vol. 2, University of Oxford, UK. KES'03, 2003, pp. 1–6.

[2] J. A. Blackard, "Comparison of neural networks and discriminant analysis in predicting forest cover types," Ph.D. dissertation, Department of Forest Sciences, Colorado State University, 1999.

[3] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds, and D. B. Rosen, "Fuzzy ARTMAP: A neural network architecture for incremental learning of analog multidimensional maps," *IEEE Transactions on Neural Networks*, vol. 3, no. 5, pp. 698–713, September 1992.

[4] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, "Fuzzy ART: An adaptive resonance algorithm for rapid, stable classification of analog patterns," in *International Joint Conference on Neural Networks, IJCNN'91*, vol. II, IEEE/INNS Inc. Seattle, Washington: IEEE–INNS–ENNS, 1991, pp. 411–416.

[5] J. Castro, J. Secretan, M. Georgiopoulos, R. Demara, G. Anagnostopoulos, and A. Gonzalez, "Pipelining of fuzzy ARTMAP without matchtracking: Correctness, performance bound, and Beowulf evaluation," *Neural Networks (under review)*.

[6] T. Kasuba, "Simplified Fuzzy ARTMAP," *AI Expert*, pp. 18–25, November 1993.

[7] A. Malkani and C. A. Vassiliadis, "Parallel implementation of the Fuzzy ARTMAP neural network paradigm on a hypercube," *Expert Systems*, vol. 12, no. 1, pp. 39–53, 1995.

[8] E. S. Manolakos, *Parallel Architectures for Neural Networks: Paradigms and Implementations*. IEEE Computer Society Press and John Wiley & Sons, 1998, ch. Parallel Implementation of ART1 Neural Networks on Processor Ring Architectures.

[9] M. Taghi, V. Baghmisheh, and N. Pavesic, "A fast simplified Fuzzy ARTMAP network," *Neural Processing Letters*, vol. 17, pp. 273–316, 2003.