

# Pipelining of Fuzzy ARTMAP without Matchtracking: Correctness, Performance Bound, and Beowulf Evaluation

José Castro  
Dep. of Computer Engineering  
Technological Institute of Costa Rica  
Cartago, Costa Rica  
jcastro@itcr.ac.cr

Jimmy Secretan  
Dep. of Electrical and  
Computer Engineering  
University of Central Florida  
Orlando, FL 32816  
jsecretan@pegasus.cc.ucf.edu

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

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

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

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

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

*Running title: Pipelining of Fuzzy ARTMAP without Matchtracking*

## 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 supported in part by a National Science Foundation (NSF) grant CRCO 0203446, and the National Science Foundation grant DUE 05254209. Georgios C. Anagnostopoulos and Michael Georgiopoulos also acknowledge the partial support from the NSF grant CCLI 0341601. Jimmy Secretan's work was partially supported by an NSF Graduate Research Fellowship.

## Abstract

Fuzzy ARTMAP neural networks have been proven to be good classifiers on a variety of classification problems. However, the time that Fuzzy ARTMAP takes to converge to a solution increases rapidly as the number of patterns used for training is increased. In this paper we examine the time Fuzzy ARTMAP takes to converge to a solution and we propose a coarse grain parallelization technique, based on a pipeline approach, to speed-up the training process. In particular, we have parallelized Fuzzy ARTMAP, without the match-tracking mechanism. We provide a series of theorems and associated proofs that show the characteristics of Fuzzy ARTMAP's, without matchtracking, parallel implementation. Results run on a BEOWULF cluster with three large databases show linear speedup as a function of the number of processors used in the pipeline. The databases used for our experiments are the Forrest CoverType database from the UCI Machine Learning repository and two artificial databases, where the data generated were 16-dimensional Gaussianly distributed data belonging to two distinct classes, with different amounts of overlap (5 % and 15 %).

**keywords:** Fuzzy ARTMAP, Data Mining, BEOWULF cluster, Pipelining, Network Partitioning.

## 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. In this context, the algorithms of choice tend to be ad-hoc algorithms (Agrawal & Srikant, 1994) or tree based algorithms such as CART (King, Feng, & Shutherland, 1995) and C4.5 (Quinlan, 1993). Variations of these tree learning algorithms, such as SPRINT (Shafer, Agrawal, & Mehta, 1996) and SLIQ (Mehta, Agrawal, & Rissanen, 1996) have been successfully adapted to handle very large data sets.

Neural network algorithms, on the other hand, 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 ((Carpenter, Grossberg, & Reynolds, 1991) and (Carpenter, Grossberg, Markuzon, Reynolds, & Rosen, 1992)), and its faster variations ((Kasuba, 1993), (Taghi, Baghmisheh, & Pavesic, 2003)) 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. Extensive research has been done on the properties of parallelization of feed-forward multi-layer perceptrons (Mangasarian & Solodov, 1994) (Torresen & Tomita, 1998) (Torresen, Nakashima, Tomita, & Landsverk, 1995). This is probably due to the popularity of this neural network architecture, and also because the backpropagation algorithm (Rumelhart, Hinton, & Williams, 1986), 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.

Regarding the parallelization of ART neural networks we can find the work by Manolakos (Manolakos, 1998) who implements the ART1 neural network (Carpenter et al., 1991) on a ring of processors. To accomplish this Manolakos divides the communication in two bidirectional rings, one for the  $F_1$  layer of ART1 and another for the  $F_2$  layer of ART1. Learning examples are pipelined through the ring to optimize network utilization. Experimental results of Manolakos' work indicate close to linear speed-up as a function of the number of processors. This approach is efficient for ring networks and it is an open question of whether it can be extended for Fuzzy ARTMAP. Another parallelization approach that has been used with ART and other types of neural networks is the systems integration approach where the neural network is not implemented on a network of computers but on parallel hardware. Zhang (Zhang, 1998) shows how a fuzzy competitive neural network similar to ARTMAP can be implemented using a systolic array. Asanović (Asanović et al., 1998) uses a special purpose parallel vector processor SPERT-II to implement back-propagation and Kohonen neural networks. In (Malkani & Vassiliadis, 1995), a parallel implementation of the Fuzzy-ARTMAP algorithm, similar to the one investigated here, is presented. However, in his paper, a hypercube topology is utilized for transferring data to all of the nodes involved in the computations. While it is trivial to map the hypercube to the more flexible switched network typically found in a Beowulf, this would likely come with a performance hit. In this approach each one 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 the  $d$ -dimensional hypercube, all the processors cooperate to find the global maximum through  $d$  different synchronization operations. This can eventually limit the scalability of this approach, since the value  $d$  grows with the size of the hypercube, while the network bandwidth remains constant.

Mining of large databases is an issue that has been addressed by many researchers, Mehta (Mehta et al., 1996), developed SLIQ, a decision-tree based algorithm that combines techniques of tree-pruning and sorting to efficiently manage large datasets. Furthermore, Shafer (Shafer et al., 1996), proposed SPRINT, another decision-tree based algorithm, that removed memory restrictions imposed by SLIQ and is designed to be amenable to parallelization. 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 these virtues we investigate Fuzzy ARTMAP's parallelization in an effort to improve its convergence speed to a solution when it is trained with large datasets.

There are many variants within the Fuzzy ARTMAP family of neural networks. Kasuba (Kasuba, 1993), with only classification problems in mind, develops a simplified Fuzzy ARTMAP structure (simplified Fuzzy ARTMAP), while Taghi, et al., in (Taghi et al., 2003), describe variants of simplified Fuzzy ARTMAP, called Fast Simplified Fuzzy ARTMAP, variants. These Fuzzy ARTMAP variants are faster than the original Fuzzy ARTMAP algorithm, because they eliminated all the computations performed in the  $ART_b$  module of Fuzzy ARTMAP, and because they have simplified the computations performed in the  $ART_{ab}$  module of Fuzzy ARTMAP; the results produced by these simplified Fuzzy ARTMAP variants are the same as the results produced by the original Fuzzy ARTMAP, when the problem at hand is a classification problem.

One of the Fuzzy ARTMAP fast algorithmic variants presented in (Taghi et al., 2003) is called, SFAM2.0 and it this algorithmic Fuzzy ARTMAP variant (that is equivalent to Fuzzy ARTMAP for

classification problems) that is the focus of our paper. Furthermore, in this paper, we only concentrate on the no-match tracking version of SFAM2.0. No-match tracking was a concept introduced by Anagnostopoulos, in the framework of the ART networks (Anagnostopoulos and Georgiopoulos, 2003). No match-tracking is a specific ART network behavior, where whenever an input pattern is presented to the ART network and a category is chosen that maximizes the bottom-up input, passes the vigilance, but is mapped to the incorrect output, this category is deactivated and a new category (uncommitted category) is activated next that will encode the input pattern. As a reminder, in that case, the typical ART network behavior is to engage the match-tracking mechanism that deactivates the chosen category, increases the vigilance threshold and searches for another appropriate category that might be or might not be an uncommitted category. Anagnostopoulos has shown through experimentation in (Anagnostopoulos and Georgiopoulos, 2003) that no-match-tracking Fuzzy ARTMAP increases the number of categories created in the category representation layer compared to Fuzzy ARTMAP but it does so while providing improved generalization performance. No Match-tracking in Fuzzy ARTMAP should not be confused with the on-line operation in Fuzzy ARTMAP. On-line Fuzzy ARTMAP operation implies that an input-output pair is presented only once in Fuzzy ARTMAP's training phase, and it can be used in a match-tracking or a no-match tracking Fuzzy ARTMAP. The reason that we focus on the no match-tracking Fuzzy ARTMAP is because it gives us the opportunity to first parallelize the competitive aspect of Fuzzy ARTMAP, while ignoring the complications of the feedback mechanism that matchtracking introduces. Finally, we focus on the on-line version of this network, since a parallelization of the on-line version extends in a straightforward fashion to the off-line version of the network.

For simplicity, we refer to this Fuzzy ARTMAP variant (on-line, no-matchtracking SFAM2.0) simply

as *Fuzzy ARTMAP*, or *FAM*. If we demonstrate the effectiveness of our parallelization strategies for FAM, extension to other ART structures can be accomplished without a lot of effort. This is due to the fact that the other ART structures share a lot of similarities with FAM, and as a result, the advantages of the proposed parallelization approach can be readily extended to other ART variants (for instance Gaussian ARTMAP (Williamson, 1996), Ellipsoidal ARTMAP (G. C. Anagnostopoulos & Georgiopoulos, 2001), among others).

The remainder of this paper is organized as follows: Section II presents the Fuzzy ARTMAP neural network architecture and a few Fuzzy ARTMAP variants. Section III continues with the pseudo-code of the off-line, match-tracking Fuzzy ARTMAP, on-line match tracking Fuzzy ARTMAP, and on-line no-match tracking Fuzzy ARTMAP (referred to simply as FAM). Section IV focuses on the computational complexity of the on-line, match-tracking Fuzzy ARTMAP, and serves as a necessary motivation for the parallelization approach introduced in this paper. Section V presents a discussion of the Beowulf cluster as our platform of choice. Section VI continues with the pseudocode of the parallel Fuzzy ARTMAP, referred to as PFAM, and associated discussion to understand the important aspects of this implementation. Section VII focuses on theoretical results related to the proposed parallelization approach. In particular, we prove there that PFAM is equivalent to the FAM, and that the processors in the parallel implementation will be *reasonably* balanced by considering a worst case scenario. Furthermore, section VIII proceeds with experiments and results comparing the performance and PFAM and FAM on three databases, one of them real and two artificial. The article concludes with section IX, where a summarization of our experiences, from the conducted work, and future research are delineated.

## II. THE FUZZY ARTMAP NEURAL NETWORK ARCHITECTURE/FUZZY ARTMAP VARIATIONS

The Fuzzy ARTMAP neural network and its associated architecture was introduced by Carpenter and Grossberg in their seminal paper (Carpenter et al., 1992). Since its introduction, a number of Fuzzy ARTMAP variations and associated successful applications of this ART family of neural networks have appeared in the literature (for instance, ARTEMAP (Carpenter & Ross, 1995), ARTMAP-IC (Carpenter & Markuzon, 1998), Ellipsoid-ART/ARTMAP (G. C. Anagnostopoulos & Georgiopoulos, 2001), Fuzzy Min-Max (Simpson, 1992), LAPART2 (Caudell & Healy, 1999), and  $\sigma$ -FLNMAP (Petridis, Kaburlasos, Fragkou, & Kehagais, 2001), to mention only a few. For the purposes of the discussion that follows we rely on the work by Kasuba (Kasuba, 1993) and Taghi, Baghmisheh, and Pavesic (Taghi et al., 2003) (see section I). In this paper, we have implemented the Fuzzy ARTMAP version, called SFAM2.0 in Taghi's paper. As we have emphasized in the Introduction, SFAM2.0 is equivalent to the original Fuzzy ARTMAP algorithm (see (Carpenter et al., 1992)) for classification problems.

The block diagram of SFAM2.0 (also depicted in Kasuba (Kasuba, 1993)) is shown in figure 1. Notice that this block diagram is different than the block diagram of Fuzzy ARTMAP mentioned in (Carpenter et al., 1991), because it has eliminated the  $ART_b$  module, and inter-ART module of Fuzzy ARTMAP, and has replaced them with the a single layer of nodes, designated as  $F_2^b$  in Figure 1. The Fuzzy ARTMAP architecture of the block diagram of Figure 1 has three major layers. The *input layer* ( $F_1^a$ ) where the input patterns (designated by  $\mathbf{I}$ ) are presented, the *category representation layer* ( $F_2^a$ ), where compressed representations of these input patterns are formed (designated as  $\mathbf{w}_j^a$ , and called *templates*), and the *output layer* ( $F_2^b$ ) that holds the labels of the categories formed in the category representation layer. Another layer, shown in Figure 1 and designated by  $F_0^a$  is a pre-processing layer and its functionality



is to pre-process the input patterns, prior to their presentation to the Fuzzy ARTMAP architecture. This pre-processing operation (called complementary coding) is described in more detail later.

In this paper we focus on the on-line implementation of SFAM2.0. In the *on-line* implementation of SFAM2.0 the training data (input/output patterns) are presented to SFAM2.0 only once. This version of Fuzzy ARTMAP training is sometimes referred to as *one-epoch training* Fuzzy ARTMAP. Also, in this paper, we concentrate on a version of Fuzzy ARTMAP introduced by Anagnostopoulos (G. Anagnostopoulos, 2000), and referred to as *no-match tracking* Fuzzy ARTMAP. In the no-match tracking Fuzzy ARTMAP match-tracking is disengaged in the following sense. In the training phase of Fuzzy ARTMAP when a node in the category representation layer is chosen to represent an input pattern, but this node is mapped to the incorrect label, the node is deactivated and a search for another node in the category representation layer ensues (match-tracking mechanism). However, in no-match tracking Fuzzy ARTMAP when a node in the category representation layer is chosen to represent an input pattern, but this node is mapped to the incorrect label, a new (uncommitted node) is chosen to represent this pattern. In (G. Anagnostopoulos, 2000), Anagnostopoulos has shown that no-match tracking Fuzzy ARTMAP creates more nodes in the category representation layer of Fuzzy ARTMAP, but quite often improves the network's generalization performance. In our case, we chose to focus on the no-match tracking Fuzzy ARTMAP because it allows us to concentrate on implementing, in parallel, the competitive process in Fuzzy ARTMAP, without having to worry about the implementation of the match-tracking mechanism which has its own complications. Furthermore, the parallel implementation of the competitive process in Fuzzy ARTMAP has applicability to other neural networks in the literature that involve similar competitive phases in their design. From now on we will refer, for simplicity, to the *on-line, no-match tracking*,

SFAM2.0 network as Fuzzy ARTMAP.

In this paper, we will occasionally discuss the off-line, SFAM2.0 with match tracking, and the on-line SFAM2.0 with match tracking, and we will refer to these networks as *off-line Fuzzy ARTMAP with match-tracking*, and *on-line Fuzzy ARTMAP with match-tracking*. Once more, we reserve the simpler notation, *Fuzzy ARTMAP*, for the on-line, no-match tracking SFAM2.0 that is the focus in this paper.

Any of the aforementioned Fuzzy ARTMAP variants can operate in two distinct phases: the *training phase* and the *performance phase*. During the training phase of a Fuzzy ARTMAP variant a set of  $PT$  inputs and associated labels pairs,  $\{(\mathbf{I}^1, label(\mathbf{I}^1)), \dots, (\mathbf{I}^r, label(\mathbf{I}^r)), \dots, (\mathbf{I}^{PT}, label(\mathbf{I}^{PT}))\}$ , is provided. Then, the training algorithm of this Fuzzy ARTMAP variant is engaged to learn the correct mapping from an input pattern to an associated label. The performance phase of any of the aforementioned Fuzzy ARTMAP variants works as follows: Given a set of  $PS$  input patterns, such as  $\tilde{\mathbf{I}}^1, \tilde{\mathbf{I}}^2, \dots, \tilde{\mathbf{I}}^{PS}$ , we want to find the Fuzzy ARTMAP output (label) produced when each one of the test patterns is presented at its  $F_1^a$  layer. In order to achieve this goal we present the test set to the trained Fuzzy ARTMAP architecture and we observe the network's output.

The training phase of the *off-line, match-tracking Fuzzy ARTMAP* is succinctly described in Taghi's et al., paper (Taghi et al., 2003). 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 the Fuzzy ARTMAP architecture 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, we reset the winner, temporarily increase the resonance threshold (called *vigilance parameter*), and try the next winner. This process is called *matchtracking*.
- 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 the Fuzzy ARTMAP architecture is determined by finding the category that maximizes the function:

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

This equation introduces two operands, one of them is the *fuzzy min operand*, and designated by the symbol  $\wedge$ . 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  $\mathbf{y}$ . The other operand introduced 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.

The above function is called the *bottom-up input* (or choice function) pertaining to the  $F_2^a$  node  $j$  with category representation (template) equal to the vector  $\mathbf{w}_j^a$ , due to the presentation of input pattern  $\mathbf{I}^r$ . This function obviously depends on a Fuzzy ARTMAP network parameter  $\alpha$ , called *choice parameter*, that assumes values in the interval  $(0, \infty)$ . In most simulations the useful range of  $\alpha$  is the interval  $(0, 10]$ . Larger values of  $\alpha$  create more category nodes in the category representation layer of the Fuzzy ARTMAP architecture.

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

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

satisfies the following condition:

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

If the above equation is satisfied we say that resonance is achieved. The parameter  $\rho_a$  is called the *vigilance parameter* and assumes values in the interval  $[0, 1]$ . As the vigilance parameter increases, more category nodes are created in the category representation layer ( $F_2^a$ ) of the Fuzzy ARTMAP architecture. 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 ( $\mathbf{w}_j^a$ ) is updated to incorporate the features of this new input pattern ( $\mathbf{I}^r$ ). The update of a category's template ( $\mathbf{w}_j^a$ ) is performed as depicted below:

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

If the category  $j$  is chosen as the winner and it resonates, but the label of this category  $\mathbf{w}_j^a$  is different than the label of the input pattern  $\mathbf{I}^r$ , then this category is reset and the vigilance parameter  $\rho_a$  is increased to the level (this is enforced in the match-tracking Fuzzy ARTMAP):

$$\rho(\mathbf{I}^r, \mathbf{w}_j^a) + \epsilon \quad (5)$$

In the above equation  $\epsilon$  takes very small values. Increasing the value of the vigilance barely above the level of vigilance ratio of the category that is reset guarantees that after this input/label-of-input pair is learned, immediate presentation of this input to the Fuzzy ARTMAP architecture will result in correct recognition of its label. It is difficult to correctly set the value of  $\epsilon$  so that you can guarantee that after category resets no legitimate categories are missed by Fuzzy ARTMAP. Nevertheless, in practice, typical values of the parameter  $\epsilon$  are taken from the interval  $[0.00001, 0.001]$ . After the reset of category  $j$  (if that's the case), other categories are searched that maximize the bottom-up input and they satisfy the vigilance (resonate). This process continues until a category is found that maximizes the bottom-up input, satisfies the vigilance and has the same label as the input pattern presented to the Fuzzy ARTMAP architecture. Once this happens, update of the category's template, as indicated by equation (4), ensues. If through this search process an uncommitted category (an uncommitted category is a category that has not encoded any input pattern before) is chosen, it will pass the vigilance, its label will be set to be equal to the label of the presented input pattern, and the update of the category's template will create a template that is equal to the presented input pattern.

All input patterns  $\mathbf{I}$  presented at the input layer ( $F_1^a$ ) of the Fuzzy ARTMAP architecture has the following form:

$$\mathbf{I} = (\mathbf{a}, \mathbf{a}^c) = (a_1, a_2, \dots, a_{M_a}, a_1^c, a_2^c, \dots, a_{M_a}^c) \quad (6)$$

where,

$$a_i^c = 1 - a_i; \forall i \in \{1, 2, \dots, M_a\} \quad (7)$$

The assumption here is that the input vector  $\mathbf{a}$  is such that each one of its components lies in the interval  $[0, 1]$ . Any input pattern can be, through appropriate normalization, represented by the input vector  $\mathbf{a}$ , where  $M_a$  stands for the dimensionality of this input pattern. The above operation that creates  $\mathbf{I}$  from  $\mathbf{a}$  is called *complementary coding* and it is required for the successful operation of Fuzzy ARTMAP. The number of nodes (templates) created in the  $F_2^a$  layer of the Fuzzy ARTMAP architecture (category representation layer) is designated by  $N_a$ , and it is not a parameter that needs to be defined by the user before training commences;  $N_a$ 's value is dictated by the needs of the problem at hand and the setting of the choice parameter ( $\alpha$ ) and baseline vigilance parameter  $\bar{\rho}_a$ . The *baseline vigilance parameter* is a parameter set by the user as a value in the interval  $[0, 1]$ . The vigilance parameter  $\rho_a$ , mentioned earlier (see equation (3)), is related with the baseline vigilance  $\bar{\rho}_a$  since at the beginning of training with a new (input pattern)/label pair, the vigilance parameter is set equal to the baseline vigilance parameter; during training with this (input pattern)/label pair the vigilance parameter could be raised above the baseline vigilance parameter (see equation (5)), only to be reset back to the baseline vigilance parameter value once a new (input pattern)/label pair appears. This raising of the vigilance parameter is accomplished according to equation (5). Prior to initiating the training phase of any Fuzzy ARTMAP variant the user has to set the values for the choice parameter  $\alpha$  (chosen as a value in the interval  $[0, 10]$ ), and the baseline vigilance parameter value  $\bar{\rho}_a$  (chosen as a value in the interval  $[0, 1]$ ).

Despite the fact that we focused above on describing the training phase of the off-line, match tracking Fuzzy ARTMAP, the equations presented are also pertinent for the on-line, match tracking Fuzzy ARTMAP, or the on-line no-match tracking Fuzzy ARTMAP (referred to for simplicity as Fuzzy ARTMAP); the only difference, emphasized many times by now, is that in the no-match tracking case we would never have to employ equation (5) that increases the value of the vigilance when the match-tracking mechanism is engaged.

In the performance phase of any Fuzzy ARTMAP variant, a test input is presented to the Fuzzy ARTMAP architecture and the category node in the  $F_2^a$  layer that has the maximum bottom-up input is chosen. The label of the chosen  $F_2^a$  category is the label that the Fuzzy ARTMAP architecture predicts for this test input. By knowing the correct labels of test inputs belonging to a test set allows us, in this manner, to calculate the classification error of the Fuzzy ARTMAP variant for this test set.

### III. THE FUZZY ARTMAP VARIANTS' PSEUDO-CODE

The off-line, match-tracking, Fuzzy ARTMAP algorithm is shown in figure 2. The on-line, match-tracking Fuzzy ARTMAP algorithm is shown in figure 3. Notice that in the off-line, match tracking Fuzzy ARTMAP training, the learning process (lines 4 through 30) of the algorithm are performed until no more network weight changes are made or until the number of iterations reached a maximum number (designated as *epochs*). In the on-line, match-tracking Fuzzy ARTMAP training, the learning process (lines 3-24) passes through the data once.

In this paper we are primarily concerned with the on-line training phase of Fuzzy ARTMAP. Notice though that by parallelizing the "on-line training" Fuzzy ARTMAP, in essence we are also parallelizing

the "off-line training" Fuzzy ARTMAP. This is because the "off-line training" Fuzzy ARTMAP, is an "on-line training" Fuzzy ARTMAP, where after an on-line training cycle is completed, another cycle starts with the same set of (input patterns)/label pairs; these on-line training Fuzzy ARTMAP cycles are repeated for as long as it is necessary for the Fuzzy ARTMAP network to learn the required mapping. In figures 2, 3 the match-tracking mechanism is employed, where if the label of the input pattern  $\mathbf{I}^r$  is different than the label of the template of the node  $j_{max}$  (i.e., template  $\mathbf{w}_{j_{max}}^a$ ), the the vigilance level is increased a search for a new template ensues.

In this paper we are only concerned with Fuzzy ARTMAP where training is on-line and the match-tracking mechanism is disengaged, and we refer to this Fuzzy ARTMAP version, for simplicity, as Fuzzy ARTMAP. The training phase of Fuzzy ARTMAP is shown in figure 5.

The performance phase of the algorithm is much simpler, and is common to all the above Fuzzy ARTMAP variants. In the performance phase we return the label associated with the template that wins the competition for the input pattern. It is common in this phase to set the parameter  $\bar{\rho}_a$  equal to 0 to assure that the network will produce a predicted label (classification) for every input pattern (albeit sometimes erroneous). The Fuzzy ARTMAP performance phase is shown in figure 4.

#### IV. COMPLEXITY ANALYSIS OF THE ON-LINE, MATCH TRACKING FUZZY ARTMAP

We concentrate on analyzing the time complexity of the on-line Fuzzy ARTMAP variants because this is the focus of the paper. Our approach requires making a few assumptions about the size of the networks created and the match-tracking cycles. This complexity analysis will motivate the pipelined implementation of Fuzzy ARTMAP.



We can see from the pseudocode (2, 3) that the on-line, match-tracking Fuzzy ARTMAP algorithm tests every input pattern  $\mathbf{I}$  in the training set against each template  $\mathbf{w}_j^a$  at least once. Let us call  $\Gamma$  the average number of times that the inner **repeat** loop (lines 5 to 19 of the online training phase algorithm of figure 3) is executed for each input pattern, and christen it the *matchtracking factor*. Then the number of times that a given input pattern  $\mathbf{I}$  passes through the code will be:

$$Time(I) = O(\Gamma \times templates) \quad (8)$$

It is then easy to see that the time complexity of the algorithm is:

$$Time(on-line, match-tracking, Fuzzy ARTMAP) = O(\Gamma \times PT \times templates) \quad (9)$$

In both of the above equations *templates* corresponds to the average number of templates created during the on-line, match-tracking Fuzzy ARTMAP training phase.

We have seen that with some databases the on-line, match-tracking Fuzzy ARTMAP algorithm achieves a certain *compression ratio* that is proportional to number of input patterns  $PT$  (*Case 1*). We have also seen that with other databases the algorithm creates templates which number saturates to a constant (*Case 2*). In either case, we denote the compression ratio (*Case 1*), or the constant (*Case 2*), by  $\kappa$ . Then the complexity of the algorithm ends up being equal to:

$$Time(on-line, match-tracking, Fuzzy ARTMAP) = O(\Gamma \times \kappa \times PT^2) \quad (10)$$

for Case 1, and

$$Time(Fuzzy ARTMAP) = O(\Gamma \times \kappa \times PT) \quad (11)$$

for Case 2.

Obviously for Case 1 implementing a parallel pipeline implementation of the algorithm makes sense. For Case 2 there are occasions where although the time complexity of the algorithm is linear in the number of patterns, the constant  $\kappa$  is large enough so that a parallel, pipeline implementation of the algorithm is still justified.

## V. THE BEOWULF PARALLEL PLATFORM

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 96 AMD nodes, each with dual AthlonMP 1500+ processors and 512MB of RAM. The nodes are connected through a *Fast Ethernet* network.

In general, the 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 of small number. Parallelization techniques in this platform are radically different from shared memory or vector machines. Also communication between nodes in the cluster is done by consent from all the parties involved; that is all communicating entities must agree to send/receive information in compatible formats. This has an impact on the design of the algorithm because receiving entities must know *before-hand* that they are going to receive information in order to be prepared to accept it. There is no central coordinating entity

and protocols must be based on listening/polling schemes and must dispense of any interrupt driven communication.

We have 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 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 master/slave approach tends to degrade communication performance as the size of the gather operation increases. Our design is based on fixed packet size communication through the network. No network bandwidth would be gained by using variable sized packets since packets are more efficient when they are large. Furthermore, to find out the size of a packet a receiving process would have to incur an extra (and expensive) communication.

To find an appropriate packet size for our experiments, we ran our system on 512,000 patterns of both the CoverType database and the Gaussian 5% database. Packet performance for the Gaussian 15% database was not evaluated, because classification overlap does not affect packet transmission time, and the 15% Gaussian database is on all other respects identical to the Gaussian 5% database. Figures 6 and 7 illustrate the results. For the CoverType database, any packet size 64 and above performed acceptably. For the Gaussian database, any packet size of 128 and above was sufficient. We translate this into bytes to give a guideline for the packet size of future database runs. To find the bytes transferred, we multiply

the number of templates in a packet, times the size of the template's vector (plus one for the label), times the number of bytes in each value of the vector ( 4 bytes for both 32-bit integers and floating point numbers).

For the CoverType database:

$$64 \times 55 \times 4 = (14080)\text{Bytes} \quad (12)$$

For the Gaussian 5% database:

$$128 \times 17 \times 4 = (8704)\text{Bytes} \quad (13)$$

These numbers will likely be dependent on characteristics of the Beowulf cluster, such as CPU power, network bandwidth and network latency. However, a good rule of thumb for similar clusters will be a packet size greater than or equal to 10Kbytes.

## VI. BEOWULF FUZZY ARTMAP IMPLEMENTATION

The parallel implementation of Fuzzy ARTMAP (on-line, no-match tracking Fuzzy ARTMAP algorithm) is discussed here. We call this implementation Parallel Fuzzy ARTMAP (PFAM). A depiction of the pipeline is shown in figure 8. The elimination of matchtracking makes the learning of a pattern a one-pass over the pipeline procedure and different patterns can be processed on the different pipeline steps to achieve optimum parallelization. For the understanding of PFAM we need the following definitions:

- $n$ : number of processors in the pipeline.

- $k$ : index of the current processor in the pipeline,  $k \in \{0, 1, \dots, n-1\}$ .
- $p$ : packet size, number of patterns sent downstream;  $2p = \text{maximum number of templates sent upstream}$ .
- $\mathbf{I}^i$ : input pattern  $i$  of current packet in the pipeline.  $i \in \{1, 2, \dots, p\}$ .
- $\mathbf{w}^i$ : current best candidate template for input pattern  $\mathbf{I}^i$ .
- $T^i$ : current maximum activation for input pattern  $\mathbf{I}^i$ .
- *myTemplates*: variable local to the current processor, set of templates that belong to the current processor.
- *nodes*: variable local to the current processor that holds the total number of templates the processor is aware of (its own plus the templates of the other processors).
- *myShare*: maximum amount of templates that the current processor is allowed to have.
- $\mathbf{w}_{k-1}^i$ : template  $i$  coming from the previous processor  $k-1$  upstream in the pipeline.
- $\mathbf{w}_{k+1}^i$ : template  $i$  coming from the next processor  $k+1$  downstream in the pipeline.
- $\mathbf{w}^i$ : template  $i$  going to the next processor  $k+1$  downstream in the pipeline.
- $\mathbf{w}_{to(k-1)}^i$ : template  $i$  going to previous processor  $k-1$  upstream in the pipeline.
- *I.class*: class label associated with a given input pattern.
- *w.class*: class label associated with a given template.
- *index(w)*: sequential index assigned to a template.
- *newNodes<sub>k+1</sub>*: integer that holds the number of new created nodes that processor  $k+1$  communicates upstream in the pipeline.
- *newNodes<sub>k</sub>*: integer that holds the number of new created nodes that processor  $k$  communicates

upstream in the pipeline.

The exchange of packets between processors is pictorially illustrated in figure 9. In this figure, the focus is on processor  $k$  and the exchange of packets between processor  $k$  and its neighboring upstream and downstream processors (i.e., processors  $k - 1$  and  $k + 1$  respectively).

The PFAM implementation is in the procedure PFAM shown in Figure 11 and the initialization procedure INIT is shown in figure 10. The pseudocode of PFAM is the main heart of the parallel algorithm. In the theorems that follow, there is a one to one correspondence between PFAM instances and computing processors, and we will therefore loosely talk about PFAM instances and processors as names referring to the same entity (the meaning will be clear from the context).

Each element of the pipeline will execute PFAM for as long as there are input patterns to be processed. The input parameter  $k$  tells the routine PFAM 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 (figure 10) we enter the loop of lines 2 through 35 (Figure 11). This loop continues execution until there are no more input patterns to process. The first activity of each processor is to create a packet of excess templates to send upstream (lines 4 to 6 of PFAM). Lines 7 to 10 correspond to the information exchange between contiguous nodes in the pipeline. The functions SEND-NEXT and RECV-NEXT on lines 7 and 8, respectively, don't do anything if the processor is the last in the pipeline ( $k = n - 1$ ). The same is true for the function SEND-PREV when the processor is the first in the pipeline ( $k = 0$ ). On the other hand, the function RECV-PREV reads input patterns from the input stream it's the first in the pipeline. These fresh patterns will be paired with an uncommitted node  $(1, 1, \dots, 1)$  with index  $\infty$  as their best representative so far. On all other cases these functions do the obvious information exchange between contiguous processors 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.

The function `FIND-WINNER` (see figure 12) is also important. 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 (using FAM criteria) 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. By sending the input patterns downstream in the pipeline coupled with their current best representative template we guarantee that the templates are not duplicated amongst different processors and that we do not have multiple-instance consistency issues.

Also when exchanging templates between processors 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 12 to 15 (communication with the processor downstream in the pipeline) and lines 18 to 21 of PFAM (see Figure 11). On line 12 we set  $\mathcal{S}$  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  $(\mathbf{I}, \mathbf{w})$  (lines 13 to 15) 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  $\mathcal{S}$ . The net result of this is that  $\mathcal{S}$  ends up containing the templates that lost the competition, and therefore the ones that processor  $k$  should keep (line 15). The converse computation is performed on lines 18 to 21. On line 18 we set  $\mathcal{S}$  to represent the set of templates that were sent upstream to the previous node  $k - 1$  in the pipeline. On lines 19 to 20 we compare the pattern, template pairs  $(\mathbf{I}_{k-1}^i, \mathbf{w}_{k-1}^i)$  that  $k - 1$  sent downstream in the pipeline with the templates in  $\mathcal{S}$  that processor  $k$  sent upstream. On line 21 of PFAM

we set our current pattern, template pairs to the winners of this competition. The set  $\mathcal{S}$  is discarded since it contains the losing templates and therefore the templates that process  $k - 1$  keeps. Another way of looking at this is that what happens in lines 12 to 15 in  $\text{PFAM}(k)$  is duplicated exactly in lines 18 to 21 in  $\text{PFAM}(k + 1)$ , this ensures that both processors compare and select exactly the same templates to go upstream or downstream in the pipeline from the packets that had just been exchanged.

Finally, on line 30 of figure 11 we add both the input pattern  $\mathbf{I}^i$  and the template  $\mathbf{w}^i$  to the set of templates. This does the obvious *myTemplates* update except when the template  $\mathbf{w}^i$  happens to be the uncommitted node in which the addition is ignored.

The main loop of PFAM starts with line 2 and ends with line 35. 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  $(\mathbf{w}^i, \mathbf{I}^i, T^i) = (\text{none}, \text{none}, 0)$  to the next processor (see line 36 of figure 11). Lines 37 and 38 of PFAM 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.

## VII. PROPERTIES OF THE PFAM ALGORITHM

We present and prove a series of fourteen (14) theorems. These theorems are distinguished in two groups. The group of theorems associated with the correctness of the PFAM, and the group of theorems associated with the performance of the PFAM. 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.



Theorem	Name
7.1	Non-duplication
7.5	Bundle size sufficiency
7.11	Overflow impossibility
7.13	Partial evaluation correctness

TABLE I  
PFAM CORRECTNESS THEOREMS

The major purpose of these theorems is to prove that PFAM (a) is equivalent to the sequential version of FAM, (b) it does not suffer from any inconsistencies, and (c) it exhibits good performance. Examples of inconsistencies would be: a template residing in more than one place in the pipeline (not possible as theorem 8.1 (non-duplication) proves), or that the first processor in the pipeline sends templates upstream (not possible as theorem 8.11 (overflow impossibility) proves). It is worth mentioning that theorems 8.2 through 8.9 facilitate the demonstration of the overflow impossibility theorem. The equivalence of the parallel and sequential version of the algorithm is demonstrated through the partial evaluation correctness theorem (theorem 8.13). Good performance is dependent on the distribution of templates amongst 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 8.12) and it is equal to  $p + 1$ , where  $p$  is the packet size. 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  $p + 1 = 65$ ).

*Definition 7.1:* A template  $w_j^a$  is *in transit* if the template has been received by the current processor  $i$  from the processor upstream  $i - 1$  in the pipeline, and the current processor has not made the decision

Theorem	Name
7.2	Template awareness delay
7.3	Weak upstream migration precondition
7.4	Upstream packet size sufficiency
7.6	Strong upstream migration precondition
7.7	Strong upstream migration postcondition
7.8	Template ownership delay
7.9	Network size lower bound
7.10	Template ownership bound
7.12	Pipeline depth invariance
7.14	Workload balance variance bound

TABLE II  
PFAM PERFORMANCE THEOREMS

yet of whether to send this template upstream, downstream, or keep it. Templates in transit are stored in the  $\mathbf{w}^i$ 's array of processor  $i$ .

*Definition 7.2:* A template  $\mathbf{w}_j^a$  is *owned* by a processor  $i$  in the pipeline if it is stored in the *myTemplates* array of processor  $i$ .

*Theorem 7.1:* Non-duplication

A template  $\mathbf{w}$  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).

*Proof:* First let us note that templates start their existence in processor  $n - 1$  on line 30 of PFAM. Here they are immediately added to the templates of process  $n - 1$ , so they start belonging to a single processor.

Templates only change location when

- 1) They are compared with a given input pattern  $I^r$  and selected to represent it, in which case they are deleted from the template list owned by the processor and added to the templates in transit.

- 2) They are in transit and lose competition to another template, in which case they are removed from the templates in transit and added to the templates owned by the processor.
- 3) They are sent upstream or sent downstream as in-transit templates.

The only possible situation where the templates may be in two places at once is in situation 3 when they are exchanged between processors in the pipeline. This is the only scenario where two processors hold a copy of the same template.

So the only possible problem will arise when two consecutive processors exchange templates. Now a template that is sent downstream on line 7 of PFAM by a process  $k - 1$  is received by process  $k$  on line 10 of PFAM. Every template  $w$  that is sent downstream is tagged along with an input pattern  $I$ . Processor  $k$  will keep the template in transit if it is the best candidate for input pattern  $I$ . To verify this, processor  $k$  will compare template  $w$  against the templates that he himself sent upstream. If a template  $w'$  that was sent upstream is a better candidate than  $w$  for the input pattern  $I$  (lines 19–21) then process  $k$  will discard template  $w$  and keep template  $w'$ .

Concurrently, processor  $k - 1$  will check the pair of template  $w$  and input pattern  $I$  it sent to processor  $k$  and compare them against the templates that it receives from processor  $k$ . If a template  $w'$  that was received from processor  $k$  is a better candidate than  $w$  for input pattern  $I$  (lines 12–15) then processor  $k - 1$  will keep template  $w$  and discard template  $w'$ .

As we can see, these concurrent operations guarantee that a template that was sent downstream or upstream will not reside in two places at the same time. Furthermore, it is guaranteed that this template will be compared against all the input patterns that flow through the pipeline. ■

*Theorem 7.2:* Template awareness delay

The total number of templates that a processor  $k = 0, 1, \dots, n-1$  in the pipeline is aware of is equal to the number of templates that existed in the system  $n - k - 1$  iterations ago.

*Proof:* Consider processor  $n-1$ , the last in the pipeline. This processor knows immediately when a template is created, and as a result it knows how many templates exist  $n-1-k = n-1-(n-1) = 0$  iterations ago.

The number of templates created per iteration is sent upstream to the previous processor in the variable *newNodes*. This variable is received by processor  $n-2$  one iteration after the templates have been created, by processor  $n-3$  two iterations after the templates have been created, and in general, by processor  $i$ ,  $n-1-i$  iterations after the templates have been created. This means that a processor  $k$  always receives on the current iteration the value of the variable *newNodes* that was created  $n-k-1$  iterations ago, and this implies that processor  $k$  is aware of the amount of templates that existed  $n-k-1$  iterations ago. ■

*Theorem 7.3:* Weak upstream migration precondition

A processor  $k$  in the pipeline sends templates upstream only if and only if on the current iteration:

$$|myTemplates| > myShare \quad (14)$$

*Proof:* It will suffice to say that PFAM creates the packet of templates to be sent upstream in lines 4 through 6. Looking at line 4 of the PFAM pseudocode we can see that templates are packed to be sent upstream only when condition 14 is met. ■

*Theorem 7.4:* Upstream packet size sufficiency

No processor in the pipeline, except the first one, can have, at any point in time, an excess of templates greater than  $2p$ .

*Proof:* By an excess of templates we mean the number of templates over its self imposed fair share indicated in the variable *myShare*. What we need to prove that equation 15 never holds, or that it is impossible for a processor in the pipeline to reach a situation where the number of templates it has is greater than its fair share plus the amount it can send upstream.

$$|myTemplates| > myShare + 2p \quad (15)$$

Let us notice that at the beginning of execution there are no templates in transit and that all the processes have their fair share of templates. In other words they comply with the condition 16

$$|myTemplates| \leq myShare \quad (16)$$

Now let's consider the processor  $n - 1$ , the last in the pipeline. If this processor complies with the equation 16 and receives  $p$  templates from processor  $n - 2$ , it would have a total of at most  $p + myShare$  templates. In the worst case scenario all of the  $p$  templates that have been sent are not of the correct category and will force the creation of another  $p$  templates giving a maximum total of  $2p + myShare$  of templates, where  $2p$  are in transit. At the beginning of the next iteration, processor  $n - 1$  will pack  $2p$  templates to be sent upstream to the previous process in the pipeline (assuming its variable *myShare* does not increase, which would ease the pressure on the processor) and will receive  $p$  templates from the previous processor upstream. Notice that the  $p$  templates extra that it ended up with are not part of its fair share because they are templates in transit. Consequently, processors'  $n - 1$  number of templates  $|myTemplates|$  did not exceed *myShare*.

Now consider any other processor that is not the last or the first in the pipeline and assume (as it does when it starts) that it complies with equation 16. This processor can receive in the worst case scenario a total of  $p$  templates from its upstream neighbor and  $2p$  templates from its downstream neighboring processor. Now the  $p$  templates that arrived from upstream will continue their journey downstream (maybe not the same ones but *at least* that quantity), so they will not increase the total number of templates that the processor owns. The excess of the  $2p$  templates coming from downstream over *myShare* will, on a worst case scenario, be packed and sent upstream to the previous processor (some of them could be kept if *myShare* increases) ■

*Theorem 7.5:* Bundle size sufficiency

The excess templates for a processor  $k \neq 0$ , at any given time, always fits in the packet of size  $2p$  to be sent upstream.

*Proof:* See theorem 7.4. ■

*Theorem 7.6:* Strong upstream migration precondition

If a processor  $k \in \{0, 1, \dots, n-1\}$  in the pipeline sends templates upstream, then it is true that:

- 1 iteration ago processor  $k+1$  complied with condition 14 and sent templates upstream.
- 2 iterations ago processor  $k+2$  complied with condition 14 and sent templates upstream.
- ⋮
- $n-1-k$  iterations ago processor  $n-1$  complied with condition 14 and sent templates upstream.

*Proof:* If processor  $k$  sends templates upstream then by theorem 7.3 it complies with condition 14.

But by the reasoning in theorem 7.4 all excess templates fit in the packet size so they are sent upstream

on the next iteration that they are received. This means that the excess templates were received from processor  $k + 1$  one iteration ago. Similarly, if processor  $k + 1$  sent templates upstream one iteration ago then by theorem 7.3 processor  $k + 1$  must have complied with condition 14 two iterations ago, and this can only happen if 2 iterations ago processor  $k + 2$  sent templates upstream. By induction on  $i$  and repeating this argument we can state that, in general, processor  $k + i$  complied with condition 14 and sent templates upstream  $i$  iterations ago. ■

*Theorem 7.7:* Strong upstream migration postcondition

If a processor  $k \in \{0, 1, \dots, n - 1\}$  in the pipeline sends templates upstream, then it is true that:

- 1)
  - at this iteration processor  $k$  keeps *myShare* templates.
  - 1 iteration ago processor  $k + 1$  kept *myShare* templates.
  - 2 iterations ago processor  $k + 2$  kept *myShare* templates.
  - ⋮
  - $n - 1 - k$  iterations ago processor  $n - 1$  kept *myShare* templates.
- 2) All of the values of *myShare* were the same for all the processors.
- 3) The templates that each processor keeps are distinct.

*Proof:* First let us notice that by theorem 7.2

- on the current iteration processor  $k$  is aware of the templates that existed in the system  $n - k - 1$  iterations ago.
- 1 iteration ago processor  $k + 1$  was aware of the templates that existed in the system  $n - k - 1$  iterations ago.

- 2 iterations ago processor  $k + 2$  was aware of the templates that existed in the system  $n - k - 1$  iterations ago.

$\vdots$

- $n - k - 1$  iterations ago processor  $n - 1$  was aware of the templates that existed in the system  $n - k - 1$  iterations ago.

This means that all the processors were aware of the same amount of templates and therefore their values for *myShare* were all the same. It is evident by looking at lines 12 to 14 of PFAM that the processor keeps *myShare* templates when it sends upstream. We also know by theorem 7.6 that they all sent templates upstream on the corresponding iterations. Now for any pair of processors  $k + i$  and  $k + j$  where  $i < j$ , the templates that processor  $k + i$  kept  $i$  iterations ago cannot be the ones that processor  $k + j$  kept  $j$  iterations ago. This is true because it takes at least  $(j - i)$  iterations to transmit templates from  $j$  to  $i$  and processor  $k + j$  kept them  $j$  iterations ago, and consequently, they cannot reach processor  $k + i$  by  $j - (j - i) = i$  iterations ago. ■

*Theorem 7.8: Template ownership delay*

The templates that a processor  $k$  has, at the current iteration, were created at least  $n - k - 1$  iterations ago

*Proof:* Since templates are created in process  $n - 1$  on line 30 of the code of PFAM. These templates maybe sent upstream one step of the pipeline per iteration. The distance from  $k$  to processor  $n - 1$  is equal to  $n - k - 1$ , so the templates that  $k$  has must have been created at least  $n - k - 1$  iterations ago. ■



*Theorem 7.9:* Network size lower bound

If a processor  $k$  sends templates upstream on a given iteration, then the number of templates  $N$  that existed in the system  $n - 1 - k$  iterations ago complies with the condition:

$$N > (n - k)myShare \quad (17)$$

*Proof:* Notice that if processor  $k$  sends templates upstream then it complies with condition 14 and by Theorem 8.7 all processors from  $k$  on-wards kept  $myShare$  templates and these templates are all distinct. Also by theorem 7.8 all these templates where created at least  $n - k - 1$  iterations ago. So the number of templates that existed in the system  $n - k - 1$  iterations ago is at least:

$$N \geq |myTemplates| + (n - 1 - k)myShare$$

$$> myShare + (n - 1 - k)myShare = (n - k)myShare \quad (18)$$

■

*Theorem 7.10:* Template ownership bound

A processor  $k$  in the pipeline cannot have more than  $myShare$  templates, and it cannot own less than  $\max(0, myShare - p(2(n - 1 - k) - 1))$  templates.

*Proof:* The fact that a processor  $k$  can not exceed  $myShare$  of templates has already been shown by theorem 7.4. Furthermore, the fact that it cannot own less than 0 templates is obvious. What needs to be proven then is that if  $myShare > p(2(n - k - 1) - 1)$  Then the number of templates will never be less than  $myShare - p(2(n - k - 1) - 1)$  templates.

To prove this let us assume a steady state in the pipeline where node  $k$  has  $myShare$  templates, and the

worst case possible scenario. In this scenario process  $k$  would receive from process  $k - 1$  packets of  $p$  pattern/template  $(\mathbf{I}^i, \mathbf{w}^i)$  pairs where the  $\mathbf{w}^i$  could be the uncommitted node, and would send downstream packets of  $p$  pattern/template pairs where the  $\mathbf{w}^i$  no longer is the uncommitted node. This means that on each iteration process  $k$  would be losing  $p$  patterns to the neighboring processors in the pipeline.

Patterns lost to the neighboring processors in the pipeline will travel, in a worst case scenario, all the way downstream to the last processor in the pipeline and afterwards find their way back to processor  $k$ . If this is the situation then processor  $k$  will have to wait  $n - 1 - k$  units of time, for the patterns sent, to reach processor  $n - 1$  and then wait another  $n - 1 - k$  iterations for the patterns to come back upstream. This is a total of  $2(n - 1 - k)$  iterations before a packet of  $p$  templates sent downstream by processor  $k$  is seen again by processor  $k$ . If during these  $2(n - 1 - k) - 1$  iterations processor  $k$  has the bad luck of sending  $p$  templates of it's own templates downstream at each iteration, then during that time processor  $k$  would have lost  $p(2(n - 1 - k) - 1)$  templates and would possess a total of  $myShare - p(2(n - 1 - k) - 1)$  templates. ■

*Theorem 7.11: Overflow impossibility*

The first processor in the pipeline will always be able to absorb the templates that have been sent to it from the next processor downstream.

*Proof:* Let us assume the contradiction that it cannot absorb the templates it has received from the next processor downstream. This means that processor 0 complies with condition 14 and that it has to send templates back. By theorem 7.9 the number of templates  $N$  that existed in the system  $n - 1$  iterations ago complies with equation 17. But by line 35 of PFAM we have:

$$N > n \times myShare = n \left\lceil \frac{nodes}{n} \right\rceil \geq n \left( \frac{nodes}{n} \right) = nodes \quad (19)$$

This means that the number  $N$  of templates that existed in the system  $n - 1$  iterations ago is greater than  $nodes$ , the number of templates that processor 0 is aware of, and this is a contradiction of theorem 7.2. ■

*Theorem 7.12: Pipeline depth invariance*

The difference in the number of  $myShare$  that two arbitrary processors 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 7.2 we know that a processor  $k$  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 processors are aware of is found in the difference between processor 0 and processor  $n - 1$ . Now, let us assume that processor 0 is aware of  $nodes_0$  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 processor  $n - 1$  can be aware of is  $nodes_0 + (n - 1)p$ . This means that the value of  $myShare$  for processor 0 is

$$myShare_0 = \left\lceil \frac{nodes_0}{n} \right\rceil \geq \frac{nodes_0}{n} \quad (20)$$

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

$$myShare_{n-1} = \left\lceil \frac{nodes_0 + (n - 1)p}{n} \right\rceil \leq \frac{nodes_0 + (n - 1)p}{n} + 1 \quad (21)$$

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

$$myShare_{n-1} - myShare_0 = \left\lceil \frac{nodes_0 + (n-1)p}{n} \right\rceil - \left\lceil \frac{nodes_0}{n} \right\rceil \leq \frac{nodes_0 + (n-1)p}{n} + 1 - \frac{nodes_0}{n} = \frac{(n-1)p}{n} + 1 \leq p + 1 \quad (22)$$

■

*Theorem 7.13:* Partial evaluation correctness

If we make the packet size  $p$  of PFAM equal to the size of the training set and set the number of processors to  $n = 1$ , then the parallel algorithm presented here is equivalent to Fuzzy ARTMAP (FAM).

*Proof:* Let us start by noting that if the number of the processor is  $n = 1$  then the functions RECV-NEXT and SEND-PREV do not perform any computation, and can be omitted. This implies that the variables exchanged in these processors also do not hold any information and can be eliminated too. These variables are the set of templates  $\{\mathbf{w}_{k+1}^i\}$  coming from the next process in the pipeline and the set of variables  $\{\mathbf{w}_{to(k-1)}^i\}$  going to the processor upstream in the pipeline. By eliminating these lines of code and doing partial evaluation and eliminating unnecessary variables we end up with the code of figure 13

Notice that the only differences with Fuzzy ARTMAP are that

- 1) the set of patterns doesn't come as a parameter.
- 2) We are using the function FIND-WINNER to find the winner node and
- 3) Templates are being extracted and reinserted in the template set.

To guarantee that the first templates created receive priority over newer templates we number the templates when created with a sequential index and use this index to determine who wins competition in case of a tie between templates. ■

*Theorem 7.14:* 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 exceed

$$\frac{p}{2\sqrt{3}} \quad (23)$$

*Proof:* Given that in the PFAM algorithm there are many templates in transit we cannot know exactly how many templates each processor 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 processor performs on each iteration will be proportional to the number of templates that it is allowed to possess or  $O(myShare)$ . 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

$$\begin{aligned} Avg(myShare) &= \frac{\sum_{k=0}^{n-1} \frac{nodes_0 + kp}{n}}{n} = \\ &= \frac{nodes_0 + \frac{p}{n} \sum_{k=0}^{n-1} k}{n} = \end{aligned}$$

$$\frac{nodes_0 + \frac{p}{2}(n-1)}{n} =$$

$$\frac{nodes_0}{n} + \frac{p}{2n}(n-1)$$

and the variance will be

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

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

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

and finally this gives us a standard deviation of

$$\sqrt{p^2 \frac{n^2 - 1}{12n^2}} = \frac{p}{2\sqrt{3}} \sqrt{1 - n^{-2}} < \frac{p}{2\sqrt{3}} \quad (24)$$

■

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

$$\frac{64}{2\sqrt{3}} = \frac{32}{\sqrt{3}} = 18.4752$$

*regardless* of the pipeline size  $n$ .

## VIII. EXPERIMENTS

Experiments were conducted on three databases: one real-world database and two artificially-generated databases (Gaussian distributed data). Training set sizes of  $1000 \times 2^i, i \in \{5, 6, \dots, 9\}$ , that is 32,000 to 512,000 patterns were used for the training of PFAM and FAM. 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 sizes used were 64 and 128 for the CoverType and the Gaussian databases, respectively.

To avoid additional computational complexities in the experiments (beyond the one that the size of the training set brings along) the values of the ART network parameters  $\bar{\rho}_a$ , and  $\alpha$  were fixed (i.e., the values chosen were ones that gave reasonable results for the database of focus). For each database and for every combination of  $(p, PT) = (\text{partition}, \text{training set size})$  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 values of the network parameters  $\bar{\rho}_a$ , and  $\alpha$ , as well as the order of pattern presentation within the training set.

All the tests were conducted on the OPCODE Beowulf cluster of workstations of the Institute for Simulation and Training. This cluster consists of 96 nodes, with dual Athlon 1500+ processors and 512MB of RAM. The runs were done in such a 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:

- 1) Classification performance of PFAM (Higher classification performance is better).
- 2) Size of the trained, PFAM.

### 3) Speedup of PFAM compared to FAM.

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

#### *VIII.-A. Forest CoverType Database*

The first database used for testing was the Forest CoverType database provided by Blackard (Blackard, 1999), and donated to the UCI Machine Learning Repository (University of California, Irvine, 2003). 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. 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. A visualization of the first 3 dimensions of the Forest CoverType database can be seen in figure 14. Different tones correspond to different classes. As it can be seen from the figure the class boundaries are quite complex. Classification performance of different machine learning algorithms for this database has been reported in the range of 75%.

#### *VIII.-B. Gaussian Databases*

The Gaussian data was artificially generated using the polar form of the Box-Muller transform with the R250 random number generator by Kirkpatrick and Scholl (Kirkpatrick & Stoll, 1981). We generated 2-class, 16 dimensional data. All the dimensions are identically distributed with the same mean  $\mu$  and



variance  $\sigma^2$  except one. The discriminating dimension has offset means so that the overlap between the Gaussian curves is set at 5% for one database and at 15% for the other. 532,000 patterns were generated for each Gaussian database. 512,000 patterns were used for training; the remaining 20,000 patterns were used for testing.

The speed-up performance of the CoverType, and the Gaussian 5% overlap, and the Gaussian 15% overlap are reported in Figures 15, 16 and 17, respectively. One important conclusion from these results is that the speed-up achieved using PFAM 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.

Tables IV and V exhibit the generalization performance and the size of the architectures created by FAM. For the Gaussian 5% overlap database the best generalization performance observed is around 93%, while the observed compression ratio (i.e., ratio of number of patterns used in training versus number of templates) is equal to 5. For the 15% Gaussian dataset these numbers are 80% (maximum generalization performance) and 3 (compression ratio). Note that the best generalization performance expected with the 5% Gaussian and the 15% Gaussian databases are 5% and 15%, respectively. Also, note that the compression achieved with FAM is relatively small, but the objective of this paper was not to produce a high compression for the training data but to demonstrate the correct, sensible pipelined implementation

of the main loop in FAM (that can be easily extended to other ART architectures).

## IX. SUMMARY - CONCLUSIONS

We have produced a pipelined implementation of Fuzzy ARTMAP. This implementation can be extended to other ART neural network architectures that have similar competitive structure as Fuzzy ARTMAP. It can also be extended to other neural networks that are designated as "competitive" neural networks, such as PNN, RBFs, as well as other "competitive" classifiers. We have introduced and proven a number of theorems pertaining to our pipeline implementation. The major purpose of these theorems was to show that PFAM (a) is equivalent with the sequential version of Fuzzy ARTMAP, (b) it does not suffer from inconsistencies, and (c) it exhibits good performance. In particular, the good performance of PFAM was exhibited by observing the linear speed-up achieved as the number of processors increased from 1 to 32. In the process, we produced other performance results related to the generalization performance and the size of the architectures that Fuzzy ARTMAP created. We believe that our objective of appropriately implementing Fuzzy ARTMAP on a Beowulf cluster has been accomplished and a clear evidence of this assertion are the speed-up results exhibited by PFAM and illustrated in Figures 15-17. Extension of our implementation approach to other "competitive" classifiers is possible. Extension of our implementation to the match-tracking Fuzzy ARTMAP algorithm is more involved and it is the topic of our current research.

## REFERENCES

- Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, & C. Zaniolo (Eds.), *Proceedings of the Twentieth International Conference*

- on Very Large Databases* (pp. 487–499). Santiago, Chile: Morgan Kaufmann.
- Anagnostopoulos, G. (2000). *Novel approaches in Adaptive Resonance Theory for machine learning*. Unpublished doctoral dissertation, Computer Engineering, UCF.
- Anagnostopoulos, G. C., & Georgiopoulos, M. (2001). Ellipsoid ART and ARTMAP for incremental unsupervised and supervised learning. In *Proceedings of the IEEE-INNS-ENNS* (Vol. 2, pp. 1221–1226). Washington DC: IEEE-INNS-ENNS.
- Asanović, K., Beck, J., Kingsbury, B., Morgan, N., Johnson, D., & Wawrzynek, J. (1998). Parallel architectures for artificial neural networks: Paradigms and implementations. In P. S. Editors N. Sundararajan (Ed.), (chap. Training Neural Networks with SPERT-II). IEEE Computer Society Press and John Wiley & Sons.
- Blackard, J. A. (1999). *Comparison of neural networks and discriminant analysis in predicting forest cover types*. Unpublished doctoral dissertation, Department of Forest Sciences, Colorado State University.
- Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., & Rosen, D. B. (1992, September). Fuzzy ARTMAP: A neural network architecture for incremental learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3(5), 698–713.
- Carpenter, G. A., Grossberg, S., & Reynolds, J. H. (1991). Fuzzy ART: An adaptive resonance algorithm for rapid, stable classification of analog patterns. In *International Joint Conference on Neural Networks, IJCNN'91* (Vol. II, pp. 411–416). Seattle, Washington: IEEE-INNS-ENNS.
- Carpenter, G. A., & Markuzon, N. (1998). ARTMAP-IC and medical diagnosis: Instance counting and inconsistent cases. *Neural Networks*, 11, 793–813.

- Carpenter, G. A., & Ross, W. D. (1995). ART-EMAP: A neural network architecture for object recognition by evidence accumulation. *IEEE Transactions on Neural Networks*, 6(5), 805–818.
- Caudell, T. P., & Healy, M. J. (1999). Studies of generalization for the LAPART-2 architecture. In *International Joint Conference on Neural Networks* (Vol. 3, pp. 1979–1982). Washington D.C.: IEEE-INNS-ENNS.
- Kasuba, T. (1993, November). Simplified Fuzzy ARTMAP. *AI Expert*, 18–25.
- King, R., Feng, C., & Shutherland, A. (1995, May/June). STATLOG: Comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3), 259–287.
- Kirkpatrick, S., & Stoll, E. (1981). A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40, 517–526.
- Malkani, A., & Vassiliadis, C. A. (1995). Parallel implementation of the Fuzzy ARTMAP neural network paradigm on a hypercube. *Expert Systems*, 12(1), 39–53.
- Mangasarian, O., & Solodov, M. (1994). Serial and parallel backpropagation convergence via nonmonotone perturbed minimization. *Optimization Methods and Software*, 4(2), 103–116.
- Manolakos, E. S. (1998). Parallel architectures for neural networks: Paradigms and implementations. In N. Sundararajan & P. Saratchandran (Eds.), (chap. Parallel Implementation of ART1 Neural Networks on Processor Ring Architectures). IEEE Computer Society Press and John Wiley & Sons.
- Mehta, M., Agrawal, R., & Rissanen, J. (1996). SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology* (p. 18–32). Avignon, France: Springer.
- Petridis, V., Kaburlasos, V. G., Frangkou, V. G., & Kehagais, A. (2001). Text classification using the

- $\sigma$ -FLNMAP neural network. In *Proceedings of the International Joint Conference on Neural Networks* (Vol. 2, pp. 1362–1367). Washington D.C.: IEEE-INNS-ENNS.
- Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. San Mateo, California: Morgan Kaufmann.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, 318–362.
- Shafer, J. C., Agrawal, R., & Mehta, M. (1996, September). SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, & N. L. Sarda (Eds.), *Proc. 22nd Int. Conf. Very Large Databases, VLDB* (pp. 544–555). Bombay, India: Morgan Kaufmann.
- Simpson, P. K. (1992, September). Fuzzy Min-Max neural networks—Part 1: Classification. *IEEE Transactions on Neural Networks*, 3(5), 776–786.
- Taghi, M., Baghmisheh, V., & Pavesic, N. (2003). A fast simplified Fuzzy ARTMAP network. *Neural Processing Letters*, 17, 273–316.
- Torresen, J., Nakashima, H., Tomita, S., & Landsverk, O. (1995, November 27<sup>th</sup> – December 1<sup>st</sup>). General mapping of feed-forward neural networks onto an MIMD computer. In *Proceedings of the IEEE International Conference on Neural Networks*. Perth, Western Australia.
- Torresen, J., & Tomita, S. (1998, November). Parallel architectures for artificial neural networks: Paradigms and implementations. In N. Sundararajan & P. Saratchandran (Eds.), (pp. 41–118). IEEE Computer Society Press and John Wiley & Sons.
- University of California, Irvine. (2003). *UCI Machine Learning Repository*.  
<http://www.icsf.uci.edu/mllearn/MLRepository.html>.

Williamson, J. R. (1996). Gaussian ARTMAP: A neural network for fast incremental learning of noisy multidimensional maps. *Neural Networks*, 9(5), 881–897.

Zhang, D. (1998). *Parallel VLSI neural systems design*. Springer.



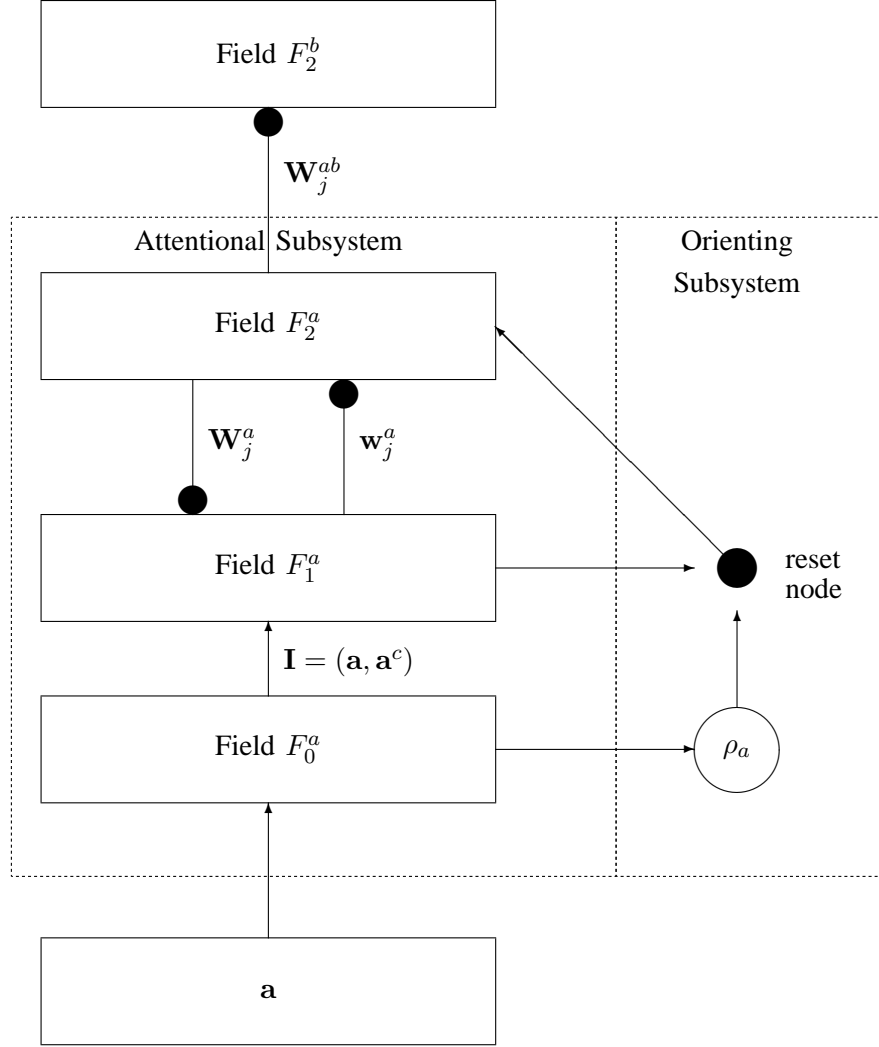


Fig. 1. Simplified Block Diagram of the Fuzzy ARTMAP Architecture. It consists of an input layer ( $F_1^a$ ), where the input patterns are applied, a category representation layer ( $F_2^a$ ) where compressed representations of these input patterns are formed, and an output layer ( $F_2^b$ ) where labels of the input patterns are produced. Layer  $F_0^a$  is a pre-processing layer that complementary encodes the input patterns  $\mathbf{a}$  to form input patterns  $\mathbf{I}$ .



OFF-LINE, MATCH TRACKING, FUZZY ARTMAP TRAINING PHASE( $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}, \bar{\rho}_a, \alpha, epochs, \varepsilon$ )

```

1   $\mathbf{w}_0 \leftarrow \underbrace{(1, 1, \dots, 1)}_{2M_a}$ 
2   $templates \leftarrow \{\mathbf{w}_0\}$ 
3   $iterations \leftarrow 0$ 
4  repeat
5       $modified \leftarrow \text{FALSE}$ 
6      for each  $\mathbf{I}^r$  in  $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}$ 
7          do  $\rho \leftarrow \bar{\rho}_a$ 
8              repeat
9                   $T_{max} \leftarrow 0$ 
10                  $status \leftarrow \text{none}$ 
11                 for each  $\mathbf{w}_j^a$  in  $templates$ 
12                     do if  $\rho(\mathbf{I}^r, \mathbf{w}_j^a) \geq \rho$  and  $T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha) > T_{max}$ 
13                         then
14                              $T_{max} \leftarrow T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha)$ 
15                              $j_{max} \leftarrow j$ 
16
17                 if  $\mathbf{w}_{j_{max}}^a \neq \mathbf{w}_0$ 
18                     then if  $label(\mathbf{I}^r) = label(\mathbf{w}_{j_{max}}^a)$ 
19                         then  $status \leftarrow \text{Allocated}$ 
20                         else  $status \leftarrow \text{Matchtracking}$ 
21                              $\rho \leftarrow \rho(\mathbf{I}^r, \mathbf{w}_{j_{max}}^a) + \varepsilon$ 
22                 until  $status \neq \text{Matchtracking}$ 
23                 if  $status = \text{Allocated}$ 
24                     then if  $\mathbf{w}_{j_{max}}^a \neq (\mathbf{w}_{j_{max}}^a \wedge \mathbf{I}^r)$ 
25                         then  $\mathbf{w}_{j_{max}}^a \leftarrow \mathbf{w}_{j_{max}}^a \wedge \mathbf{I}^r$ 
26                          $modified \leftarrow \text{TRUE}$ 
27                     else  $templates \leftarrow templates \cup \{\mathbf{I}^r\}$ 
28                      $modified \leftarrow \text{TRUE}$ 
29                  $iterations \leftarrow iterations + 1$ 
30             until  $(iterations = epochs)$  or  $(modified = \text{FALSE})$ 
31 return  $templates$ 

```

Fig. 2. Off-Line, Match-Tracking Fuzzy ARTMAP's Training phase. It involves finding the category that best matches the input pattern, and passes the vigilance, and is mapped to the correct label. Input patterns are presented repeatedly to the Fuzzy ARTMAP architecture until a certain number of iterations, referred to as epochs, is reached. The match-tracking mechanism is enforced.

ON-LINE, MATCH-TRACKING FUZZY ARTMAP TRAINING PHASE( $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}, \bar{\rho}_a, \alpha, \varepsilon$ )

```

1   $\mathbf{w}_0 \leftarrow \underbrace{(1, 1, \dots, 1)}_{2M_a}$ 
2   $templates \leftarrow \{\mathbf{w}_0\}$ 
3  for each  $\mathbf{I}^r$  in  $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}$ 
4  do  $\rho \leftarrow \bar{\rho}_a$ 
5    repeat
6       $T_{max} \leftarrow 0$ 
7       $status \leftarrow \text{NoneFound}$ 
8      for each  $\mathbf{w}_j^a$  in  $templates$ 
9      do if  $[\rho(\mathbf{I}^r, \mathbf{w}_j^a) \geq \rho]$  and  $[T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha) > T_{max}]$ 
10         then
11            $T_{max} \leftarrow T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha)$ 
12            $j_{max} \leftarrow j$ 
13
14         if  $\mathbf{w}_{j_{max}}^a \neq \text{uncommitted}$ 
15           then if  $\text{label}(\mathbf{I}^r) = \text{label}(\mathbf{w}_{j_{max}}^a)$ 
16             then  $status \leftarrow \text{Allocated}$ 
17             else  $status \leftarrow \text{Matchtracking}$ 
18              $\rho \leftarrow \rho(\mathbf{I}^r, \mathbf{w}_{j_{max}}^a) + \varepsilon$ 
19         until  $status \neq \text{Matchtracking}$ 
20     if  $status = \text{Allocated}$ 
21       then
22          $\mathbf{w}_{j_{max}}^a \leftarrow \mathbf{w}_{j_{max}}^a \wedge \mathbf{I}$ 
23     else
24        $templates \leftarrow templates \cup \{\mathbf{I}^r\}$ 
25 return  $templates$ 

```

Fig. 3. On-Line, Match-Tracking Fuzzy ARTMAP's Training phase. It involves finding the category that best matches the input pattern, and passes the vigilance, and is mapped to the correct label. Input patterns are presented to the Fuzzy ARTMAP architecture only once, in contrast to the off-line version of match-tracking Fuzzy ARTMAP. The match-tracking mechanism is enforced.

FUZZY ARTMAP'S PERFORMANCE PHASE( $\mathbf{I}^r, templates, \bar{\rho}_a, \beta$ )

```

1   $T_{max} \leftarrow 0$ 
2   $j_{max} \leftarrow \text{NIL}$ 
3  for each  $\mathbf{w}_j^a$  in templates
4  do
5      if  $\rho(\mathbf{I}^r, \mathbf{w}_j^a) \geq \bar{\rho}_a$  and  $T(\mathbf{I}^r, \mathbf{w}_j^a, \beta) > T_{max}$ 
6      then
7           $T_{max} \leftarrow T(\mathbf{I}^r, \mathbf{w}_j^a, \beta)$ 
8           $j_{max} \leftarrow j$ 
9
10 if  $\mathbf{w}_{j_{max}}^a \neq \mathbf{w}_0$ 
11 then return  $label(\mathbf{w}_{j_{max}}^a)$ 
12 else return NIL

```

Fig. 4. Fuzzy ARTMAP's performance phase. In this phase a test input pattern is presented to Fuzzy ARTMAP and the category that best matches this input pattern and passes the vigilance is chosen. The predicted label of this test input pattern is the label that this chosen category is mapped to. Quite often, in the performance phase of Fuzzy ARTMAP the baseline vigilance parameter value is chosen equal to zero.

```

ON-LINE, NO MATCH-TRACKING FUZZY ARTMAP TRAINING PHASE( $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}, \rho, \alpha$ )
1   $\mathbf{w}_0 \leftarrow \underbrace{(1, 1, \dots, 1)}_{2M_a}$ 
2   $templates \leftarrow \{\mathbf{w}_0\}$ 
3  for each  $\mathbf{I}^r$  in  $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^{PT}\}$ 
4  do
5      repeat
6           $T_{max} \leftarrow 0$ 
7           $status \leftarrow \text{NoneFound}$ 
8          for each  $\mathbf{w}_j^a$  in  $templates$ 
9              do if  $\left[\rho(\mathbf{I}^r, \mathbf{w}_j^a) \geq \rho\right]$  and  $\left[T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha) > T_{max}\right]$ 
10                 then
11                      $T_{max} \leftarrow T(\mathbf{I}^r, \mathbf{w}_j^a, \alpha)$ 
12                      $j_{max} \leftarrow j$ 
13
14                 if  $\mathbf{w}_{j_{max}}^a \neq \mathbf{w}_0$  and  $label(\mathbf{I}^r) = label(\mathbf{w}_{j_{max}}^a)$ 
15                     then  $\mathbf{w}_{j_{max}}^a \leftarrow \mathbf{w}_{j_{max}}^a \wedge \mathbf{I}^r$ 
16                     else  $templates \leftarrow templates \cup \{\mathbf{I}^r\}$ 
17             until
18 return  $templates$ 

```

Fig. 5. On-Line, No-Match-Tracking Fuzzy ARTMAP's Training phase. This is the version of Fuzzy ARTMAP that we implemented in this paper, and for simplicity it is referred to as Fuzzy ARTMAP. It involves finding the category that best matches the input pattern, and passes the vigilance, and is mapped to the correct label. If the first chosen node is not mapped to the correct label an uncommitted node is chosen next to represent this input pattern (i.e., the match-tracking mechanism is disengaged).

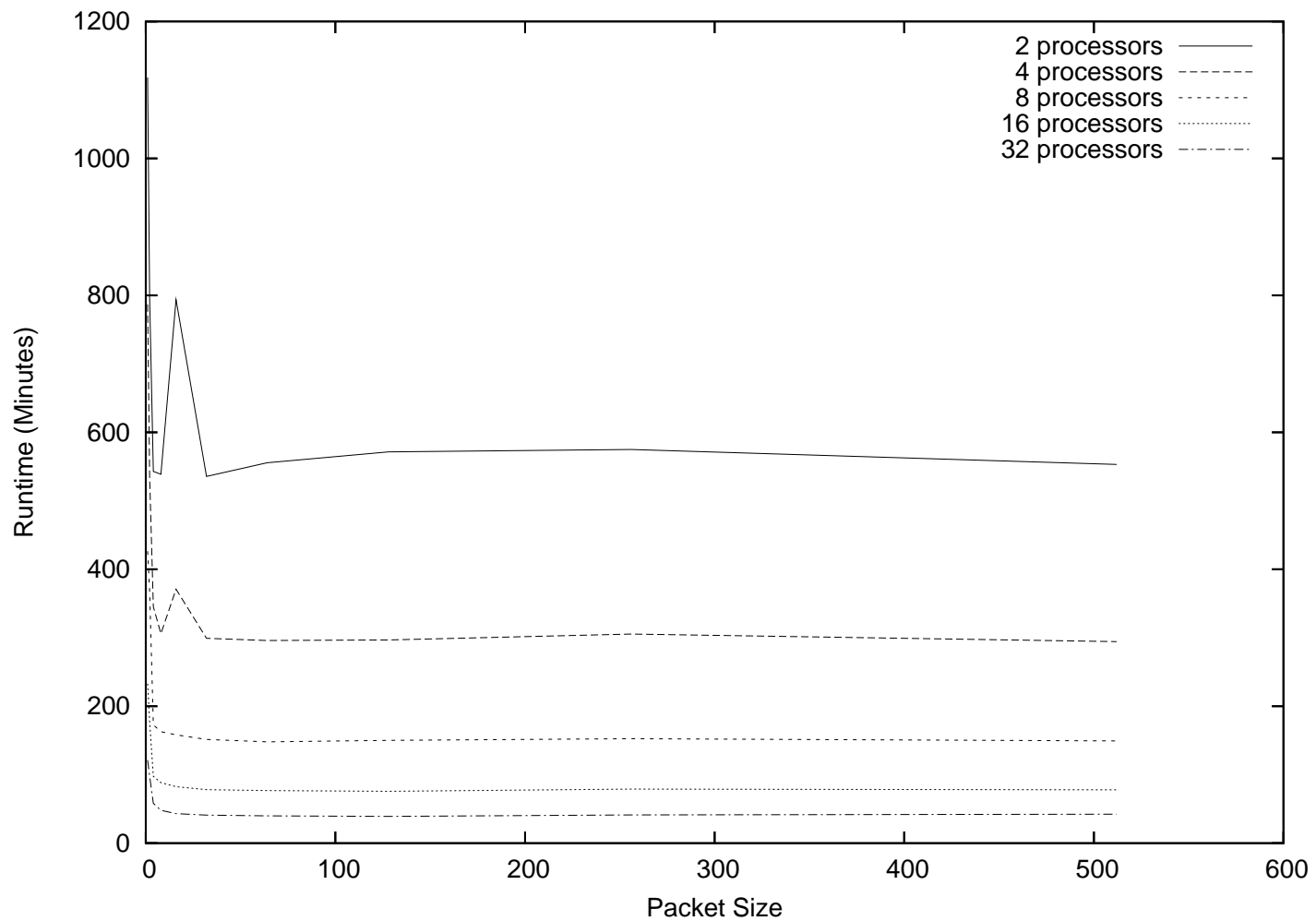


Fig. 6. Running time of training the Covertypes database using different packet sizes between the processors. Packet sizes are in bytes. Since the transmission time is much slower than the processing time it is beneficial to reduce the number of transmissions relative to processing, and this behavior is reflected in the graph. Also, given that for every transmission there is a constant transmission set up time, regardless of the packet size, we would expect the running time to reach a saturation level, as it does.

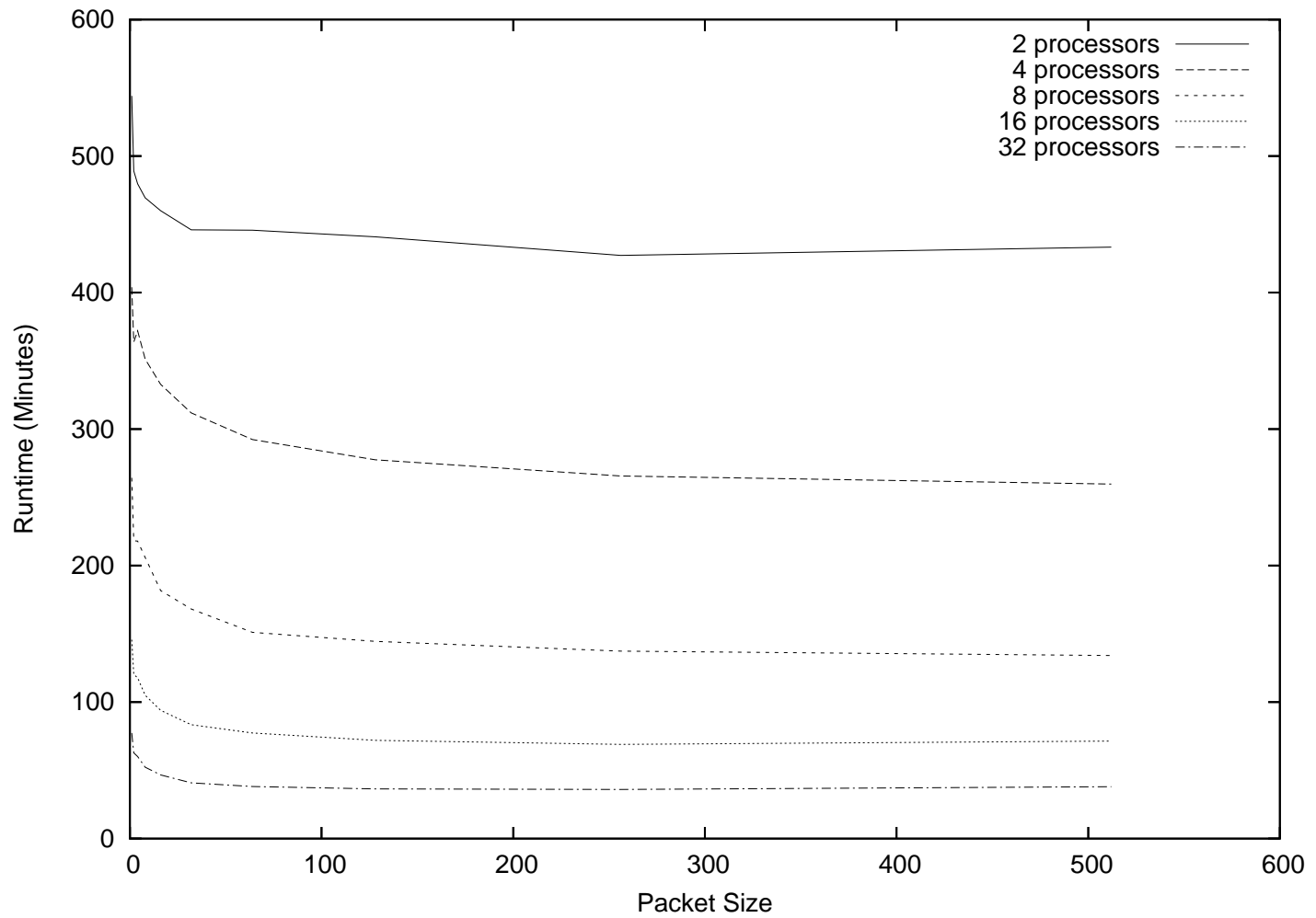


Fig. 7. Running time of training the Gaussian 5% overlap database using different packet sizes between the processors. Packet sizes are in bytes. Since the transmission time is much slower than the processing time it is beneficial to reduce the number of transmissions relative to processing, and this behavior is reflected in the graph. Also, given that for every transmission there is a constant transmission set up time, regardless of the packet size, we would expect the running time to reach a saturation level, as it does.

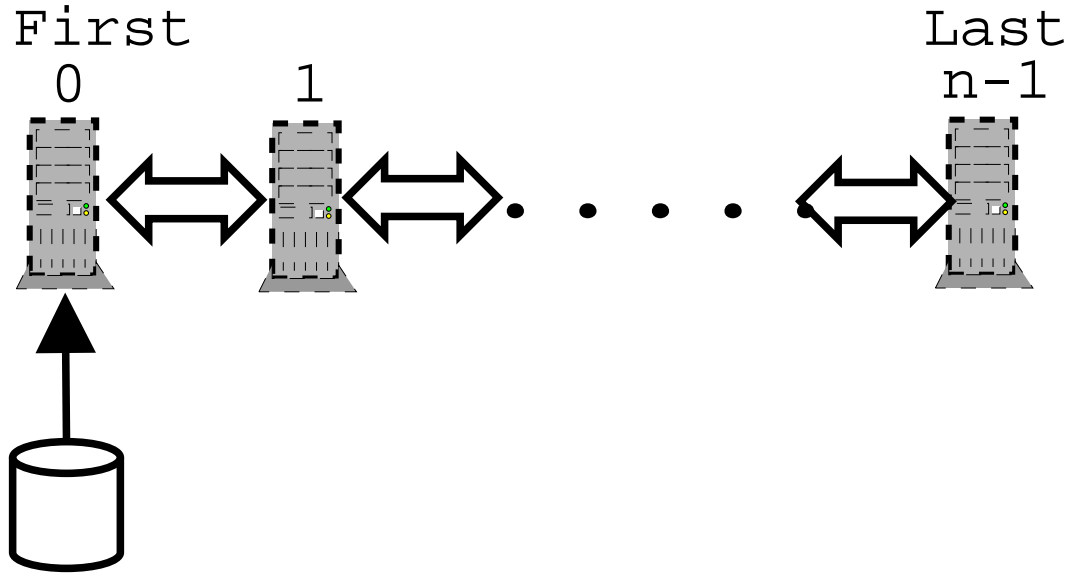


Fig. 8. Pipeline Structure.

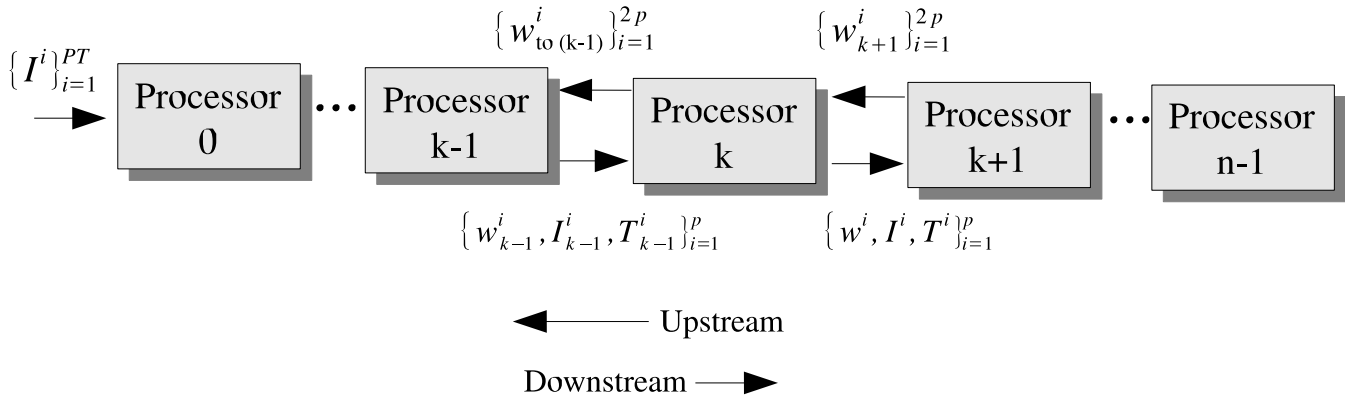


Fig. 9. Exchange of packets between processors. Note, packets are listed for processor  $k$  only. Optimum packet sizes were estimated using the graphs of figure 6 and 7. Packets sent upstream are depicted on top of the PROCESSOR boxes and consist of only the templates that have to be reassigned to a specific Processor template pool. Packets sent downstream are depicted below of the PROCESSOR boxes and consists of triads of template  $w^i$ , input pattern  $I^i$ , and activation  $T^i$ .

```

INIT( $p$ )
1   $nodes \leftarrow 0$ 
2   $myTemplates \leftarrow \{\}$ 
3   $\forall_{i=1}^{2p} \left( \mathbf{w}_{to(k-1)}^i \leftarrow \text{none} \right)$ 
4   $\forall_{i=1}^p \left( \mathbf{w}^i \leftarrow \text{none} \right)$ 
5   $\forall_{i=1}^p \mathbf{I}^i \leftarrow \text{none}$ 
6   $myShare \leftarrow 0$ 
7   $newNodes \leftarrow 0$ 
8   $newNodes_{k+1} \leftarrow 0$ 
9   $continue \leftarrow \text{TRUE}$ 

```

Fig. 10. Initialization procedure for the PFAM implementation. Since no processing has begun, all counters are set to 0. These correspond to the *nodes*, *newNodes* and *myShare* variables. The corresponding sets and packets are initialized to empty (since packets are of fixed size, their elements are initialized none).



```

PFAM( $k, n, \rho_a, \alpha, p$ )
1  INIT( $p$ )
2  while continue
3  do
4    while  $|myTemplates| > myShare$ 
5    do
6      EXTRACT-TEMPLATE ( $myTemplates, \{\mathbf{w}_{to(k-1)}^i\}$ )
7      SEND-NEXT ( $k, n, \{(\mathbf{w}^i, \mathbf{I}^i, T^i) : i = 1, 2, \dots, p\}$ )
8      RECV-NEXT ( $k, n, \{\mathbf{w}_{k+1}^i : i = 1, 2, \dots, 2p\}, newNodes_{k+1}$ )
9      SEND-PREV ( $k, \{\mathbf{w}_{to(k-1)}^i : i = 1, 2, \dots, 2p\}, newNodes$ )
10     RECV-PREV ( $k, \{(\mathbf{w}_{k-1}^i, \mathbf{I}_{k-1}^i, T_{k-1}^i) : i = 1, 2, \dots, p\}$ )
11      $newNodes \leftarrow newNodes_{k+1}$ 
12      $\mathcal{S} \leftarrow \{\mathbf{w}_{k+1}^i\}$ 
13     for each  $i$  in  $\{1, 2, \dots, p\}$ 
14     do FIND-WINNER( $\mathbf{I}^i, \mathbf{w}^i, T^i, \rho_a, \alpha, \mathcal{S}$ )
15      $myTemplates \leftarrow myTemplates \cup \mathcal{S}$ 
16     if  $\mathbf{I}_{k-1}^i = \text{EOF}$ 
17     then continue  $\leftarrow \text{FALSE}$ 
18     else  $\mathcal{S} \leftarrow \{\mathbf{w}_{to(k-1)}^i\}$ 
19     for each  $i$  in  $\{1, 2, \dots, p\}$ 
20     do FIND-WINNER( $\mathbf{I}_{k-1}^i, \mathbf{w}_{k-1}^i, T_{k-1}^i, \rho_a, \alpha, \mathcal{S}$ )
21      $(\mathbf{I}^i, \mathbf{w}^i, T^i) \leftarrow (\mathbf{I}_{k-1}^i, \mathbf{w}_{k-1}^i, T_{k-1}^i)$ 
22     for each  $i$  in  $\{1, 2, \dots, p\}$ 
23     do FIND-WINNER( $\mathbf{I}^i, \mathbf{w}^i, T^i, \rho_a, \alpha, myTemplates$ )
24     if  $k = n - 1$ 
25     then if  $class(\mathbf{I}^i) = class(\mathbf{w}^i)$ 
26     then
27        $myTemplates \leftarrow myTemplates \cup \{\mathbf{I}^i \wedge \mathbf{w}^i\}$ 
28     else  $newTemplate \leftarrow \mathbf{I}^i$ 
29        $index(newTemplate) \leftarrow newNodes + nodes$ 
30        $myTemplates \leftarrow myTemplates \cup \{\mathbf{I}^i, \mathbf{w}^i\}$ 
31        $newNodes \leftarrow newNodes + 1$ 
32     if  $newNodes > 0$ 
33     then
34        $nodes \leftarrow nodes + newNodes$ 
35        $myShare \leftarrow \lceil \frac{nodes}{n} \rceil$ 
36     SEND-NEXT ( $k, n, \{(\text{none}, \text{none}, 0)\}$ )
37     RECV-NEXT ( $k, n, \{\mathbf{w}_{k+1}^i : i = 1, 2, \dots, 2p\}, newNode_{k+1}$ )
38      $myTemplates \leftarrow myTemplates \cup \{\mathbf{w}_{k+1}^i : i = 1, 2, \dots, 2p\}$ 

```

Fig. 11. Parallel Fuzzy ARTMAP (PFAM) implementation. Exchange of templates between processors is done in lines 4–10. Templates received from the processor downstream are compared with the input pattern/template pairs sent downstream in lines 12–15. Templates sent to the processor upstream are compared with the incoming templates from this processor in lines 18–21. The input pattern/template pairs going downstream are compared with the local pool of templates in lines 22–23. The rest of the loop on lines 24–35 do necessary bookkeeping updates if the current processor is the last in the pipeline.

```

FIND-WINNER( $\mathbf{I}, \mathbf{w}, T, \rho_a, \alpha, \mathcal{S} = \{\mathbf{w}^i\}$ )
1   $idx \leftarrow -1$ 
2  for each  $\mathbf{w}^i$  in  $\mathcal{S}$ 
3  do if  $[\rho(\mathbf{I}, \mathbf{w}^i) \geq \rho_a]$ 
4      then
5          if  $[T(\mathbf{I}, \mathbf{w}^i, \alpha) > T]$ 
6              then
7                   $T \leftarrow T(\mathbf{I}, \mathbf{w}^i, \alpha)$ 
8                   $idx \leftarrow i$ 
9              else if  $[T(\mathbf{I}, \mathbf{w}^i, \alpha) = T]$  and  $index(\mathbf{w}^i) < index(\mathbf{w})$ 
10                 then  $T \leftarrow T(\mathbf{I}, \mathbf{w}^i, \alpha)$ 
11                      $idx \leftarrow i$ 
12 if  $idx \neq -1$ 
13     then
14         EXTRACT( $\mathbf{w}^{idx}, \mathcal{S}$ )
15         ADD( $\mathbf{w}, \mathcal{S}$ )
16          $\mathbf{w} \leftarrow \mathbf{w}^{idx}$ 
17     return TRUE
18 else
19     return FALSE

```

Fig. 12. Utility function to find the best candidate template in a template list. Needed by the PFAM implementation. Given a current input pattern  $\mathbf{I}$  with its current template  $\mathbf{w}$  and activation  $T$ , this procedure compares the input with all the members of a given template pool  $\mathcal{S}$ . If the set  $\mathcal{S}$  holds a template that is better representative of  $\mathbf{I}$  than  $\mathbf{w}$ , then these templates are swapped (lines 14–15) and the winner is left in  $\mathbf{w}$ .

```

PROCESS( $\rho_a, \alpha$ )
1   $myTemplates \leftarrow \{\}$ 
2   $\forall_{i=1}^p \mathbf{I}^i \leftarrow \text{none}$ 
3   $newNodes \leftarrow 0$ 
4   $continue \leftarrow \text{TRUE}$ 
5  RECV-PREV ( $k, \{(\mathbf{w}_{k-1}^i, \mathbf{I}_{k-1}^i, T_{k-1}^i) : i = 1, 2, \dots, p\}$ )
6   $newNodes \leftarrow 0$ 
7  for each  $i$  in  $\{1, 2, \dots, p\}$ 
8  do FIND-WINNER( $\mathbf{I}^i, \mathbf{w}^i, T^i, \rho_a, \alpha, myTemplates$ )
9      if  $class(\mathbf{I}^i) = class(\mathbf{w}^i)$ 
10         then
11              $myTemplates.ADD(\{\mathbf{I}^i \wedge \mathbf{w}^i\})$ 
12
13         else
14              $newTemplate \leftarrow \mathbf{I}^i$ 
15              $index(newTemplate) \leftarrow newNodes$ 
16              $myTemplates.ADD(\{\mathbf{I}^i, \mathbf{w}^i\})$ 
17              $newNodes \leftarrow newNodes + 1$ 
18

```

Fig. 13. Partial evaluation of PFAM using number of processors  $p = 1$ . Having only one processor implies that the packet size is the size of the training set. Also the while loop and transmissions can be eliminated since these events happen between processors (we have only one). The only transmission procedure that survives is the RECV-PREV which reads from the input stream. Comparison with pool of templates is sequential and is taken care of by the FIND-WINNER procedure.

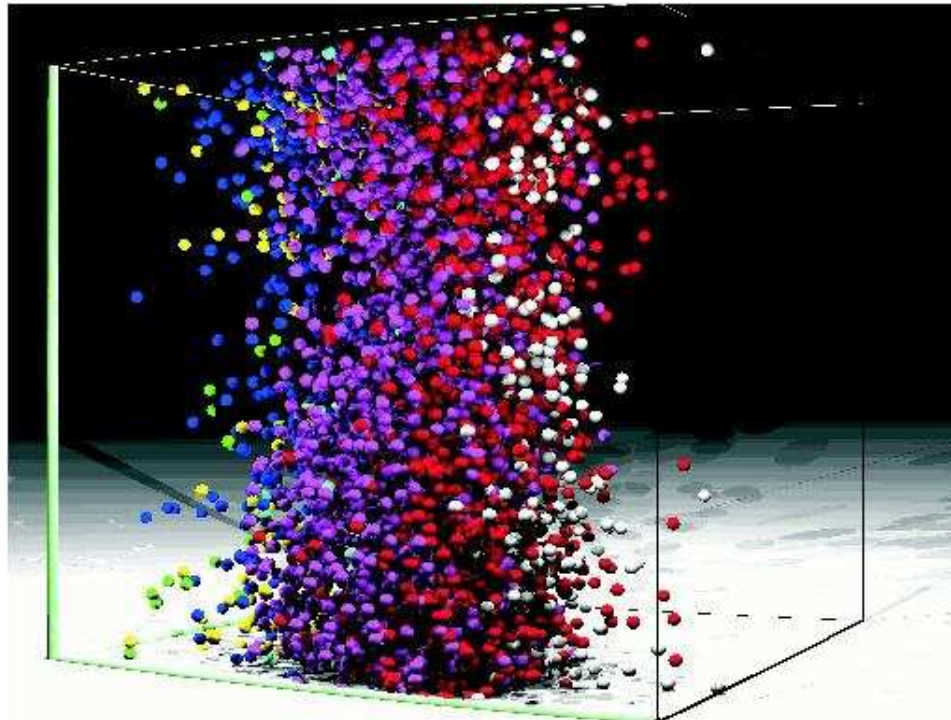


Fig. 14. A random sample of 5,000 Forest Covertype data-points out of the available 581,012 data-points is shown. The data-points are projected to the first 3 dimensions of the database. Different colors for the data-points represent different class labels.

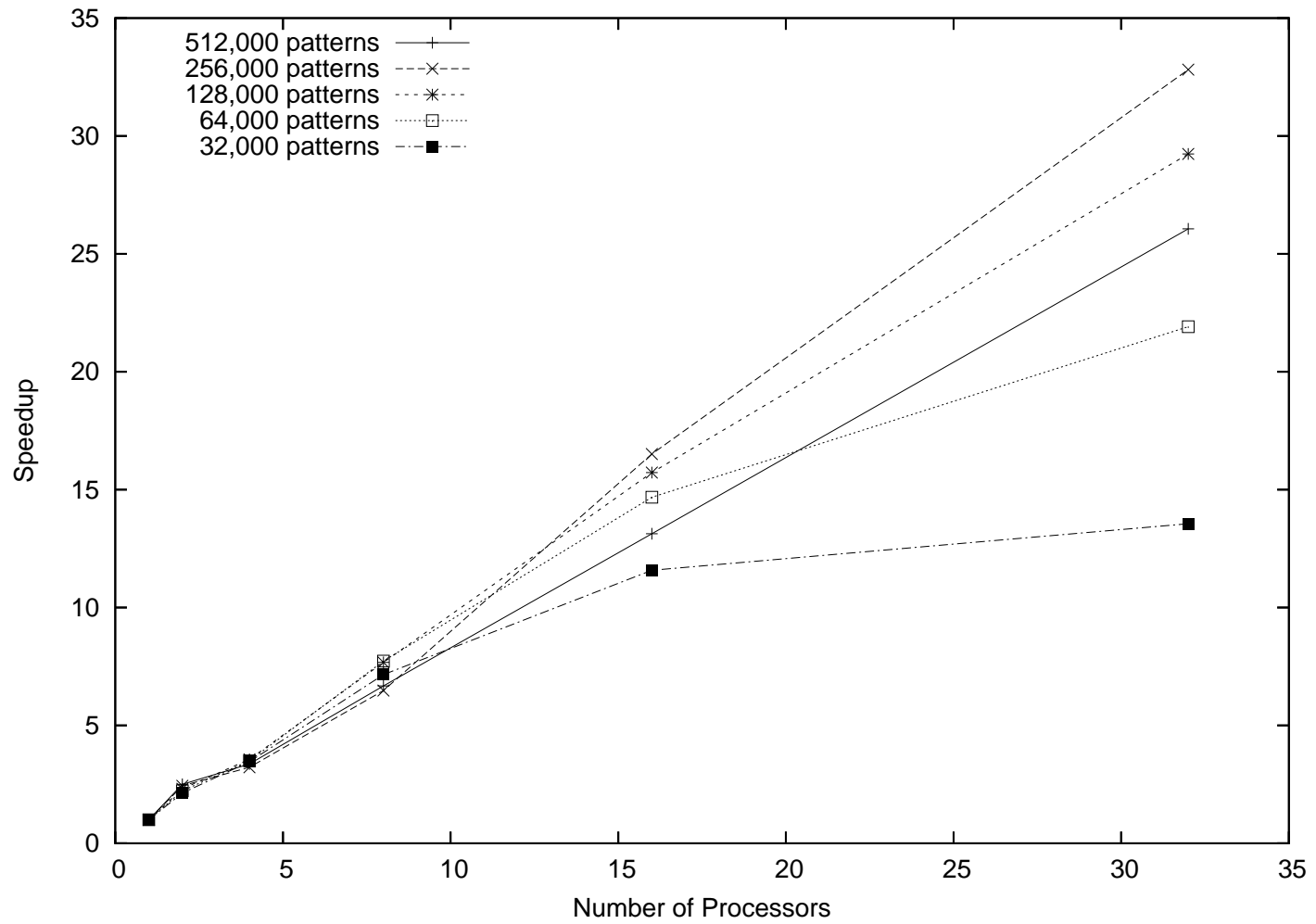


Fig. 15. Parallel speedup versus number of processors for Covertypes database. Speedup is measured as the ratio of the time  $T_p$  it takes  $p$  processors to process the input patterns over the time  $T_1$  it takes one processor to do the same amount of work, or as  $\frac{T_p}{T_1}$ . Speedup for the smallest database sizes 32,000 and 64,000 input patterns levels out faster after 16 processors. Speedup for the largest database sizes is close to linear, which is the theoretical optimum.

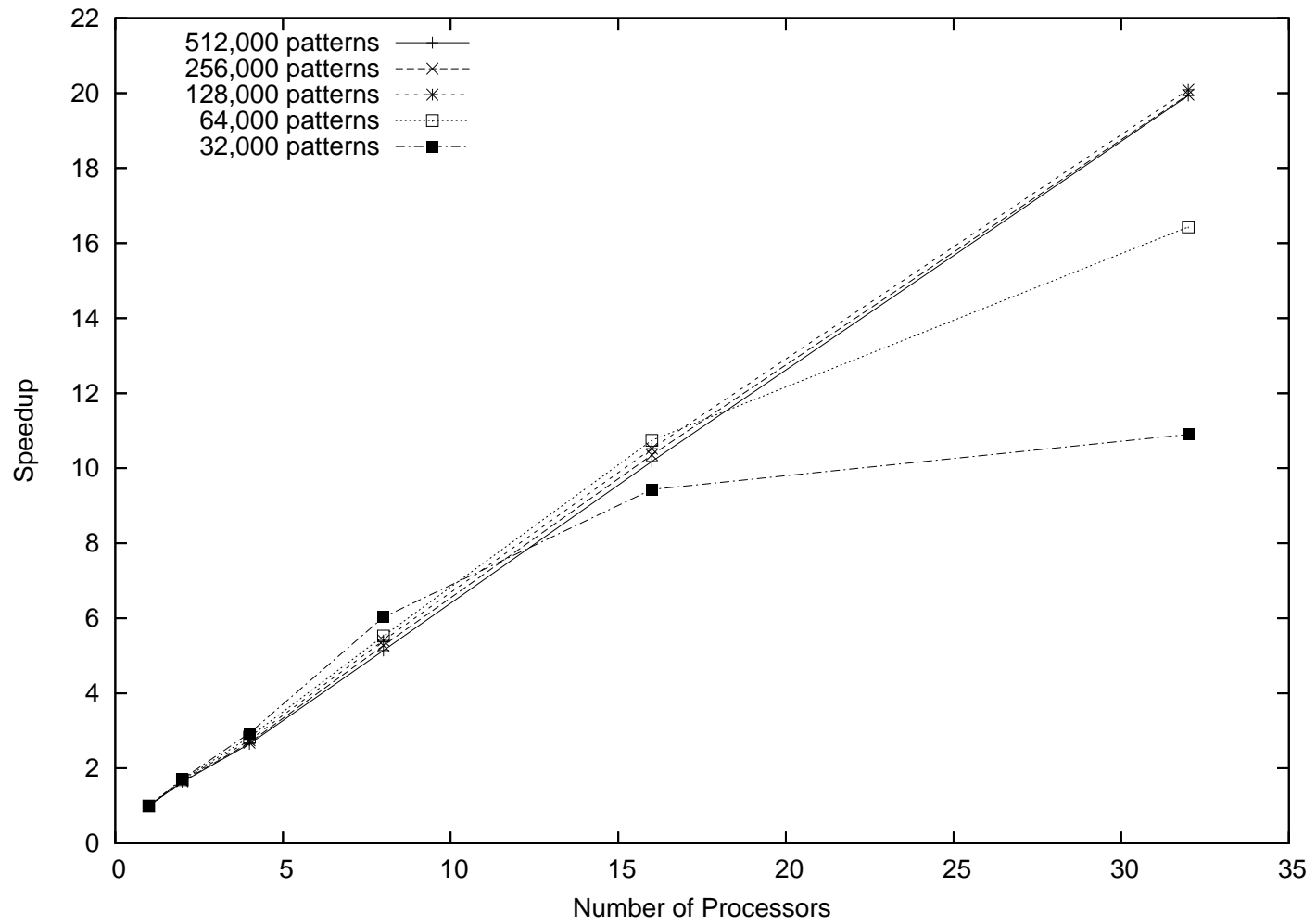


Fig. 16. Parallel speedup versus number of processors for the Gaussian 5% overlap database. Speedup is measured as the ratio of the time  $T_p$  it takes  $p$  processors to process the input patterns over the time  $T_1$  it takes one processor to do the same amount of work, or as  $\frac{T_p}{T_1}$ . Speedup for the smallest database sizes 32,000 and 64,000 input patterns levels out faster after 16 processors. Speedup for the largest database sizes is close to linear, which is the theoretical optimum.

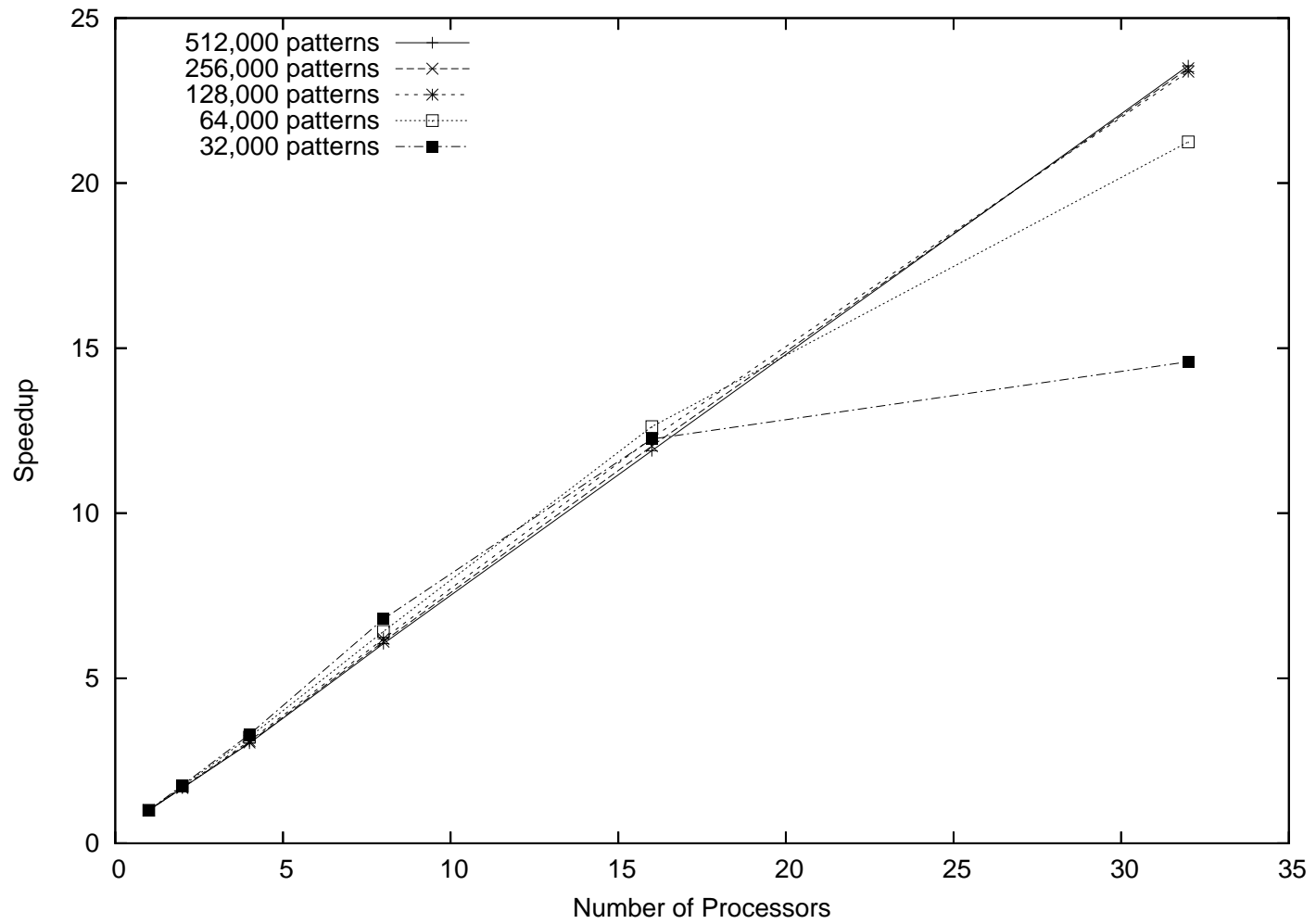


Fig. 17. Parallel speedup versus number of processors for the Gaussian 15% overlap database. Speedup is measured as the ratio of the time  $T_p$  it takes  $p$  processors to process the input patterns over the time  $T_1$  it takes one processor to do the same amount of work, or as  $\frac{T_p}{T_1}$ . Speedup for the smallest database sizes 32,000 and 64,000 input patterns levels out faster after 16 processors. Speedup for the largest database sizes is close to linear, which is the theoretical optimum.

Examples (Thousands)	Classification Performance	Average Templates Created
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 III

PERFORMANCE RESULTS ON THE COVERTYPE DATABASE. IN THIS TABLE, THE AVERAGE CLASSIFICATION PERFORMANCE AND THE AVERAGE NUMBER OF TEMPLATES CREATED BY FAM OR PFAM ARE DEPICTED, AS THE NUMBER OF POINTS IN THE TRAINING SET INCREASES FROM 32K TO 512K. AVERAGE PERFORMANCES ARE REPORTED FOR 12 DIFFERENT ORDERS OF TRAINING PATTERN PRESENTATIONS IN THE TRAINING PHASE OF FUZZY ARTMAP. CLASSIFICATION RESULTS ARE REPORTED FOR A TEST SET OF 20K INPUT PATTERNS THAT IS DIFFERENT FROM THE PATTERNS USED IN THE TRAINING SET.

Examples (Thousands)	Classification Performance	Average Templates Created
32	92.50	7032.83
64	92.74	13513.41
128	92.91	25740.5
256	93.11	48854.5
512	93.21	92365.66

TABLE IV

PERFORMANCE RESULTS ON A 2-CLASS, 16-DIMENSIONAL GAUSSIAN DATASET WITH 5% OVERLAP. THE OVERLAP OF 5% IMPLIES THAT THE BEST POSSIBLE CLASSIFIER FOR THIS DATASET (I.E., THE BAYES CLASSIFIER) CAN ACHIEVE AN OPTIMUM ERROR RATE OF 5%. IN THIS TABLE, THE AVERAGE CLASSIFICATION PERFORMANCE AND THE AVERAGE NUMBER OF TEMPLATES CREATED BY FAM OR PFAM ARE DEPICTED, AS THE NUMBER OF POINTS IN THE TRAINING SET INCREASES FROM 32K TO 512K. AVERAGE PERFORMANCES ARE REPORTED FOR 12 DIFFERENT ORDERS OF TRAINING PATTERN PRESENTATIONS IN THE TRAINING PHASE OF FUZZY ARTMAP. CLASSIFICATION RESULTS ARE REPORTED FOR A TEST SET OF 20K INPUT PATTERNS THAT IS DIFFERENT FROM THE PATTERNS USED IN THE TRAINING SET.



Examples (Thousands)	Classification Performance	Average Templates Created
32	79.25	10608.83
64	79.82	20695.83
128	80.10	40319
256	80.32	78540.58
512	80.54	152827.91

TABLE V

PERFORMANCE RESULTS ON A 2-CLASS, 16-DIMENSIONAL GAUSSIAN DATASET WITH 15% OVERLAP. THE OVERLAP OF 15% IMPLIES THAT THE BEST POSSIBLE CLASSIFIER FOR THIS DATASET (I.E., THE BAYES CLASSIFIER) CAN ACHIEVE AN OPTIMUM ERROR RATE OF 15%. IN THIS TABLE, THE AVERAGE CLASSIFICATION PERFORMANCE AND THE AVERAGE NUMBER OF TEMPLATES CREATED BY FAM OR PFAM ARE DEPICTED, AS THE NUMBER OF POINTS IN THE TRAINING SET INCREASES FROM 32K TO 512K. AVERAGE PERFORMANCES ARE REPORTED FOR 12 DIFFERENT ORDERS OF TRAINING PATTERN PRESENTATIONS IN THE TRAINING PHASE OF FUZZY ARTMAP. CLASSIFICATION RESULTS ARE REPORTED FOR A TEST SET OF 20K INPUT PATTERNS THAT IS DIFFERENT FROM THE PATTERNS USED IN THE TRAINING SET.