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
jsecreta@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, Electrical and Computer Engineering Department, University of Central Florida, 4000 Central Florida Blvd. Engineering Building 1, Office 407, Orlando, Florida, 32816

Running title: Pipelining of FS-FAM without Matchtracking

Abstract

Fuzzy ARTMAP neural networks have been proven to be good classifiers on a variety of classification problems. However, the time that it takes Fuzzy ARTMAP to converge to a solution increases rapidly as the number of patterns used for training increases. In this paper we examine the time that it takes Fuzzy ARTMAP 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 in 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, et al., (Shafer, Agrawal, & Mehta, 1996)) and SLIQ (Mehta, et al., (Mehta, Agrawal, & Rissanen, 1996)) have been successfully adapted to handle very large data sets.

Neural network algorithms can have a prohibitively slow convergence to a solution, especially when they are trained on large databases. Even one of the fastest (in terms of training speed) neural network algorithms, the Fuzzy ARTMAP algorithm ((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 it is worth mentioning the work by Manolakos

(Manolakos, 1998) who has implemented 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 this 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 star architecture, as found in a Beowulf, this would likely come with a performance hit. Each of the processors maintains a subset of the architecture's templates, and finds the template with the maximum match in its local collection. Finally, in its d-dimensional hypercube, it finds the global maximum through d different synchronization operations. This can limit the scalability of this approach.

Mining of large databases is an issue that has been addressed by many researchers, such as Metha (Mehta et al., 1996), where SLIQ, a decision-tree based algorithm that combines techniques of tree-pruning and sorting to efficiently manage large datasets, is proposed. Furthermore, Shafer (Shafer et

al., 1996), proposed SPRINT, another decision-tree based algorithm, that removed memory restrictions imposed by SLIQ and is 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, the ability to recognize novel inputs, among others. Due to all of its above properties it is worth investigating Fuzzy ARTMAP's parallelization in an effort to improve its convergence speed to a solution when it is trained with large datasets.

In particular, in this paper our focus is to improve the convergence speed of ART-like neural networks through a parallelization strategy applicable for a pipeline structure (Beowulf cluster of workstations). In order to connect our work with previous work on Fuzzy ARTMAP it is worth emphasizing again the work by Kasuba (Kasuba, 1993), where a simplified Fuzzy ARTMAP structure (simplified Fuzzy ARTMAP) is introduced that is simpler and faster than the original Fuzzy ARTMAP structure, and functionally equivalent with Fuzzy ARTMAP for classification problems. Furthermore, Taghi, et al., in (Taghi et al., 2003), describe variants of simplified Fuzzy ARTMAP, called Fast Simplified Fuzzy ARTMAP, that reduce some of the redundancies of Simplified Fuzzy ARTMAP and speed up its convergence to a solution, even further. One of the Fuzzy ARTMAP fast algorithmic variants presented in (Taghi et al., 2003) is called, SFAM2.0 and it has the same functionality as Fuzzy ARTMAP (Carpenter et al., 1992) for classification problems. From now we will refer to this variant of Fuzzy ARTMAP as FS-FAM (Fast Simplified Fuzzy ARTMAP). The focus of our paper is FS-FAM. Note that FS-FAM is faster than Fuzzy ARTMAP because it eliminated some of the redundancies of the original Fuzzy ARTMAP that are not necessary when classification problems are considered. Since the functionality of Fuzzy ARTMAP (Carpenter et al.,

1992) and FS-FAM (Taghi et al., 2003) are the same for classification problems we will occasionally refer to FS-FAM as Fuzzy ARTMAP. We chose to demonstrate the effectiveness of our proposed parallelization strategy on FS-FAM since, if we demonstrate its effectiveness for Fuzzy ARTMAP, its 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 Fuzzy ARTMAP, 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 (Anagnostopoulos & Georgiopoulos, 2001), among others). It is also worth noting that this paper addresses the parallelization of a variant of Fuzzy ARTMAP, called no-match tracking Fuzzy ARTMAP, that was introduced by Anagnostopoulos, et al., (Anagnostopoulos, 2000). In (Anagnostopoulos, 2000) the match tracking mechanism of Fuzzy ARTMAP is disabled. This variant of Fuzzy ARTMAP was referred to as *null matchtracking* by Anagnostopoulos (Anagnostopoulos & Georgiopoulos, 2001), but in this paper we are referring to it as no-matchtracking 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. The extension of our work to Fuzzy ARTMAP is a topic of further research. Throughout the paper we will be referring to the Fuzzy ARTMAP network that we are focusing on as no-matchtracking FS-FAM, no-match tracking Fuzzy ARTMAP, and occasionally, simply FS-FAM or Fuzzy ARTMAP.

This paper is organized as follows: Section II presents the FS-FAM architecture. Section III continues with the pseudo-code of the FS-FAM algorithm that is the starting point of FS-FAM's parallelization. Sections II and III, in this paper, are background information on Fuzzy ARTMAP and can be omitted by

the reader who is familiar with the Fuzzy ARTMAP architecture and its training. Section IV focuses on the computational complexity of FS-FAM, and serves as a necessary motivation for the parallelization approach introduced in this paper. Section V is a very brief reference to the no-match tracking FS-FAM, introduced by Anagnostopoulos (see (Anagnostopoulos, 2000)), which is the specific FS-FAM variant that we have parallelized. Section VI proceeds with a discussion of the Beowulf cluster as our platform of choice. Section VII continues with the pseudocode of the parallel no-match tracking FS-FAM and associated discussion to understand the important aspects of this implementation. Section VIII focuses on theoretical results related to the proposed parallelization approach. In particular, we prove there that the parallel no-matchtracking FS-FAM is equivalent to the sequential no-matchtracking FS-FAM, and that the processors in the parallel implementation will be *reasonably* balanced by considering a worst case scenario. Furthermore, section IX proceeds with experiments and results comparing the performance and speedup of the parallel no-match tracking FS-FAM on three databases (one of them real and two artificial). The article concludes with section X, where a summarization of our experiences, from the conducted work, and future research are delineated.

II. THE FS-FAM NEURAL NETWORK ARCHITECTURE

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 (Anagnostopoulos & Georgiopoulos, 2001), Fuzzy Min-Max

(Simpson, 1992), LAPART2 (Caudell & Healy, 1999), and σ-FLNMAP (Petridis, Kaburlasos, Fragkou, & Kehagais, 2001), to mention only a few. For the purposes of the discussion that follows in this section it is worth mentioning again the work by Kasuba (Kasuba, 1993) and Taghi, Baghmisheh, and Pavesic (Taghi et al., 2003). In his paper, Kasuba introduces a simpler Fuzzy ARTMAP architecture, called *Simplified Fuzzy ARTMAP*. Kasuba's simpler Fuzzy ARTMAP architecture is valid only for classification problems. Taghi, et al., (Taghi et al., 2003) have eliminated some of the unnecessary computations involved in Kasuba's Simplified Fuzzy ARTMAP, and introduced two faster variants of Simplified Fuzzy ARTMAP, called SFAM2.0 and SFAM2.1. Kasuba's simpler Fuzzy ARTMAP variants were denoted as SFAM 1.0 and 1.1 in Taghi's paper. In order to connect the version of Fuzzy ARTMAP, implemented in this paper, with Carpenter's and Grossberg's Fuzzy ARTMAP, Kasuba's simplified Fuzzy ARTMAP (SFAM1.0) and Taghi's simplified Fuzzy ARTMAP versions, such as SFAM 1.1, SFAM2.0 and SFAM2.1, it is worth mentioning that in our paper we have implemented the Fuzzy ARTMAP version, called SFAM2.0 in Taghi's paper. As, we have mentioned in the introduction, we refer to this Fuzzy ARTMAP variant as FS-FAM. Once more, FS-FAM is algorithmically equivalent with Fuzzy ARTMAP for classification problems. Classification problems are the only focus in our paper.

The block diagram of FS-FAM 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), but very similar to the block diagram depicted in Kasuba's work (see (Kasuba, 1993)). 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 FS-FAM. This pre-processing operation (called complementary coding is described in more detail below).

Fuzzy ARTMAP can operate in two distinct phases: the *training phase* and the *performance phase*. The training phase of Fuzzy ARTMAP can be described as follows: Given a set of *PT* inputs and associated labels pairs, $\{(\mathbf{I}^1, label(\mathbf{I}^1)), \ldots, (\mathbf{I}^r, label(\mathbf{I}^r)), \ldots, (\mathbf{I}^{PT}, label(\mathbf{I}^{PT}))\}$, we want to train Fuzzy ARTMAP to map every input pattern of the training set to its corresponding label. To achieve the aforementioned goal we present the training set to the Fuzzy ARTMAP architecture repeatedly. That is, we present $\mathbf{I}^1$ to F_1^a, $label(\mathbf{I}^1)$ to F_2^b, $\mathbf{I}^2$ to F_1^a, $label(\mathbf{I}^2)$ to F_2^b, and finally $\mathbf{I}^{PT}$ to F_1^a, and $label(\mathbf{I}^{PT})$ to F_2^b. We present the training set to Fuzzy ARTMAP as many times as it is necessary for Fuzzy ARTMAP to correctly classify all these input patterns. The task is considered accomplished (i.e., the learning is complete) when the weights do not change during a training set presentation. The aforementioned training scenario is called *off-line learning*. There is another training scenario, the one considered in this paper, that is called *o*n-line training, where each one of the input/label pairs are presented to Fuzzy ARTMAP only once. The performance phase of Fuzzy ARTMAP works as follows: Given a set of *PS* input patterns, such as $\tilde{\mathbf{I}}^1, \tilde{\mathbf{I}}^2, \ldots, \tilde{\mathbf{I}}^{PS}$, we want to find the Fuzzy ARTMAP output (label) produced when each one of the aforementioned test patterns is presented at its F_1^a layer. In order to achieve the aforementioned goal we present the test set to the trained Fuzzy ARTMAP architecture and we observe the network's output.

The training process in FS-FAM 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 Fuzzy ARTMAP that "resonates" with the input pattern.

2) If the labels of the chosen category and the input pattern match, update the chosen category to be closer to the input pattern.

3) Otherwise, reset the winner, temporarily increase the resonance threshold (called *vigilance parameter*), and try the next winner.

4) If the winner is uncommitted, create a new category (assign the representative of the category to be equal to the input pattern, and designate the label of the new category to be equal to the label of the input pattern).

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

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

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 an FS-FAM network parameter α, called *choice parameter*, that assumes values in the interval $(0, \infty)$. In most simulations of Fuzzy ARTMAP the useful range of α is the interval $(0, 10]$. Larger values of α create more category nodes in the category representation layer of FS-FAM.

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

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

satisfies the following condition:

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

If the above equation is satisfied we say that resonance is achieved. The parameter ρ_a appearing in the above equation is called *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 Fuzzy ARTMAP. 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:

$$\mathbf{w}_j^a = \mathbf{w}_j^a \wedge \mathbf{I}^r \tag{4}$$

The update of templates, illustrated by the above equation, has been called *fast-learning* in Fuzzy ARTMAP. Our paper is concerned only with the fast learning Fuzzy ARTMAP.

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 ρ_a is increased

to the level:

$$\frac{|\mathbf{I}^r \wedge \mathbf{w}_j^a|}{|\mathbf{I}^r|} + \epsilon \tag{5}$$

In the above equation ϵ takes small values. The parameter ϵ assumes very small values. Increasing the value of 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 by FS-FAM, immediate presentation of this input to FS-FAM will result in correct recognition of its label by Fuzzy ARTMAP. It is difficult to correctly set the value of ϵ so that you can guarantee that after category resets no legitimate categories are missed by FS-FAM. Nevertheless, in practice, typical values of the parameter ϵ are taken from the interval $[0.00001, 0.001]$. In our experiments we took $\epsilon = 0.0001$. After the reset of category j, 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 FS-FAM. 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.

In all of the above equations (equations (1)-(5)) there is specific operand involved, called *fuzzy min operand*, and designated by the symbol $\wedge$. Actually, the fuzzy min operation of two vectors x, and y, designated as $x \wedge y$, is a vector whose components are equal to the minimum of components of x and

y. Another specific operand involved in these equations is designated by the symbol $|\cdot|$. In particular, $|x|$ is the size of a vector x and is defined to be the sum of its components.

It is worth mentioning that an input pattern $\mathbf{I}$ presented at the input layer (F_1^a) of FS-FAM has the following form:

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

where,

$$a_i^c = 1 - a_i; \ \forall i \in \{1, 2, \ldots, M_a\} \tag{7}$$

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, be 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 FS-FAM (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 is a parameter, whose value is dictated by the needs of the problem that FS-FAM is trained with and the setting of the choice parameter (α) 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 ρ_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/label pattern pair, the vigilance parameter is set equal to the

baseline vigilance parameter; during training with this input/label pattern 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/label pattern pair appears.

Prior to initiating the training phase of FS-FAM the user has to set the values for the choice parameter α (chosen as a value in the interval $[0, 10]$), baseline vigilance parameter value $\bar{\rho}_a$ (chosen as a value in the interval $[0, 1]$).

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

III. THE FS-FAM PSEUDO-CODE

The FS-FAM algorithm (off-line training phase) is shown in figure 2. The FS-FAM algorithm (on-line training phase) is shown in figure 3. Notice that in the off-line training phase of the network 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 training phase of the network 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 FS-FAM. Notice though that by parallelizing the "on-line training" FS-FAM, in essence we are also parallelizing its "off-line training" FS-FAM. This is because the "off-line training FS-FAM", is an "on-line training FS-FAM", where after an on-line training cycle is completed, another cycle starts with the same set of training input patterns/label

pairs; these on-line training FS-FAM cycles are repeated for as long as it is necessary for the FS-FAM network to learn the required mapping.

It is worth noting that in figures 2, 3 we enter the match-tracking operation 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 $\mathbf{w}^a_{j_{max}}$). In this paper we are concerned only with FS-FAM where the match-tracking mechanism is disengaged (no-match tracking FS-FAM).

The performance phase of the algorithm is much simpler. 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 FS-FAM performance phase is shown in figure 4.

IV. FS-FAM COMPLEXITY ANALYSIS

To analyze the time complexity of the FS-FAM algorithm we will only concentrate on the online version of the algorithm, since this is our major focus in this paper. Our approach requires making a few assumptions about FS-FAM's size and match-tracking cycles. The time-complexity analysis of FS-FAM will motivate the pipelined implementation of FS-FAM.

We can see from the pseudocode (2, 3) that the FS-FAM algorithm tests every input pattern $\mathbf{I}$ in the training set against each template $\mathbf{w}^a_j$ at least once. Let us call Γ 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

I passes through the code will be:

$$Time(I) = O(\Gamma \times |templates|) \tag{8}$$

Also, under the unrealistic condition that the number of templates does not change during training it is easy to see that the time complexity of the algorithm is:

$$Time(\textit{FS–FAM}) = O(\Gamma \times \textit{PT} \times |templates|) \tag{9}$$

Usually for a specific database the FS-FAM algorithm achieves a certain *compression ratio*. This means that the number of templates created is actually a fraction of the number of patterns PT in the training set. Let us call this compression ratio κ so that:

$$|templates| = \kappa \textit{PT} \tag{10}$$

and

$$O(\textit{FS–FAM}) = O\left(\Gamma \textit{PT}\kappa \textit{PT}\right) = O(\kappa\Gamma \textit{PT}^2) \tag{11}$$

Thus, the on-line complexity of FS-FAM is proportional to the square of the number of input patterns in the training set. Or, viewed differently, it is proportional to the product of the number of input patterns patterns in the training set and the number of templates created during the training phase of FS-FAM.

V. "No Matchtracking" FS-FAM

A simplification that can be applied to the FS-FAM algorithm is the elimination of the matchtracking process. This modification was originally proposed by Anagnostopoulos (Anagnostopoulos, 2003) and it was found there that it actually improves the classification performance of FS-FAM on some databases. Our interest in using this FS-FAM variant lies in the fact that it simplifies the FS-FAM algorithm and allows one to concentrate on the parallelization of the competition loop in Fuzzy ARTMAP. The pseudo-code of Anagnostopoulos' no-matchtracking, on-line training FS-FAM phase is shown in figure 5.

VI. 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 star master/slave topology where all nodes communicate to a gathering master node. We chose to follow the pipelined approach because in this scenario we are only doing point to point communication, which is a constant time operation in a Fast Ethernet switched network. The star approach tends to degrade communication performance as the size of the gather operation increases. 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.

For the Covertype database:

$$64 \times 55 \times 4 = (14080)\texttt{Bytes} \tag{12}$$

For the Gaussian 5% database:

$$128 \times 17 \times 4 = (8704)\texttt{Bytes} \tag{13}$$

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.

VII. PARALLEL, NO MATCHTRACKING, FS-FAM IMPLEMENTATION

Anagnostopoulos' FS-FAM variant is particularly amenable to a production–line style pipeline parallel implementation since patterns can be evenly distributed amongst the nodes in the pipeline. 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 the parallel implementation of the no-matchtracking FS-FAM we need the following definitions:

- n: number of processors in the pipeline.

- k: index of the current processor in the pipeline, $k \in \{0, 1, \ldots, n-1\}$.

- p: packet size, number of patterns sent downstream; $2p =$ number of templates sent upstream.

- $\mathbf{I}^i$: input pattern i of current packet in the pipeline. $i \in \{1, 2, \ldots, p\}$.

- $\mathbf{w}^i$: current best candidate template for input pattern $\mathbf{I}^i$.

- T^i: current maximum activation for input pattern $\mathbf{I}^i$.

- *myTemplates*: 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*: amount of templates that the current processor should have.

- $\mathbf{w}^i_{k-1}$: template i coming from the previous processor in the pipeline.

- $\mathbf{w}^i_{k+1}$: template i coming from the next processor in the pipeline.

- $\mathbf{w}^i$: template i going to the next processor in the pipeline.

- $\mathbf{w}^i_{to(k-1)}$: template i going to previous processor in the pipeline.

- $\mathbf{I}.class$: class label associated with a given input pattern.

- $\mathbf{w}.class$: class label associated with a given template.

- *index*$(\mathbf{w})$: sequential index assigned to a template.

- *newNodes*$_{k+1}$: number of new nodes that were created that processor $k+1$ communicates upstream

  in the pipeline.

- *newNodes*$_k$: number of new nodes that were created 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 processors

(i.e., processors $k-1$ and $k+1$). The parallel implementation of no-match tracking FS-FAM is shown

in Figure 11 and the initialization procedure is shown in figure 10. The pseudocode of PROCESS is the

main heart of the parallel algorithm, shown in Figure 11. Each element of the pipeline will execute this procedure for as long as there are input patterns to be processed. The input parameter k tells the process which stage of the pipeline it is, where the value k varies from 0 to $n-1$. After initializing most of the values as empty (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 process is to create a packet of excess templates to send back (line 12 to 14). 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 process is the last in the pipeline ($k = n-1$). The same is true for the function SEND-PREV when the process is the first in the pipeline ($k = 0$). On the other hand, the function RECV-PREV reads input patterns from the input stream if the process is the first in the pipeline. These fresh patterns will be paired with an uncommitted node $(1, 1, \cdots, 1)$ with index ∞ as their best representative so far. On all other cases these functions do the obvious information exchange between contiguous processes in the pipeline. We assume that all communication happens at the same time and is synchronized. We can achieve this in an MPI environment by doing non–blocking sends and using an MPI-Waitall to synchronize the receive of information.

On line 30 of Figure 11 we add 2 templates to the template set $myTemplates$. This is because a new template was created and the current candidate winner $\mathbf{w}$ is not of the correct category and has to be inserted back into the pool of templates.

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 FS-FAM's 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 nodes in the pipeline we have to be careful that patterns that are sent downstream do not miss the comparison with templates that are being sent upstream. This is the purpose of lines 12 to 15 (communication with the next one in the pipeline) and lines 18-21 of PROCESS (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 13–15) 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 process k should keep (line 15). The converse process is done on lines 18 to 21. On line 18 we set $\mathcal{S}$ to represent the set of templates that are sent *back* to the previous node $k - 1$ in the pipeline. On lines 19 to 20 we compare the pattern, template pairs $(\mathbf{I}^i_{k-1}, \mathbf{w}^i_{k-1})$ that $k - 1$ sent upstream in the pipeline with the templates in $\mathcal{S}$ that process k sent downstream in the pipeline. On line 21 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.

Finally, on line 30 of figure11 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.

Once more, we reiterate that the main loop of the process 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 $(\mathbf{w}^i, \mathbf{I}^i, T^i) = (\texttt{none}, \texttt{none}, 0)$ to the next processor (see line 36 of figure 11). Lines 37 and 38 of process make sure that the templates that are sent upstream in the pipeline are not lost after the pool of training input patterns that are processed is exhausted.

VIII. Properties of the Parallel, no matchtracking, FS-FAM Algorithm

In the sequel 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 parallel no-matchtracking FS-FAM, and the group of theorems associated with the performance of the no-match tracking FS-FAM. For ease of reference Table I lists the theorems and their names dealing with the correctness of the algorithm, while Table II lists the theorems dealing with the performance of the algorithm.

The major purpose of these theorems is to prove that the parallel no-match tracking FS-FAM (a) is equivalent to the sequential version of the no-matchtraking FAM, (b) it does not suffer from any inconsistencies, and (c) it exhibits good performance. Examples of inconsistencies are: a template exists in more than one places in the pipeline (not possible as theorem 8.1 (non-duplication) proves), or the first processor in the pipeline is required to send 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

Theorem	Name
8.1	Non–duplication
8.5	Bundle size sufficiency
8.11	Overflow impossibility
8.13	Partial evaluation correctness

TABLE I

NO–MATCHTRACKING FS-FAM CORRECTNESS THEOREMS

Theorem	Name
8.2	Template awareness delay
8.3	Weak upstream migration precondition
8.4	Upstream packet size sufficiency
8.6	Strong upstream migration precondition
8.7	Strong upstream migration postcondition
8.8	Template ownership delay
8.9	Network size lower bound
8.10	Template ownership bound
8.12	Pipeline depth invariance
8.14	Workload balance variance bound

TABLE II

NO–MATCHTRACKING FS-FAM PERFORMANCE THEOREMS

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. Furthermore, this upper bound is independent of the pipeline

depth n. For instance, if 100,000 templates are present in the pipeline and $p = 64$, the templates that

any two processors possess cannot differ by more than 65 (where $p + 1 = 65$).

Definition 8.1: A template $\mathbf{w}_j^a$ is *in transit* if the template has been received by the current processor from the previous processor in the pipeline, and the current processor has not made the decision yet of whether to send this template to the next processor, previous processor, or keep it. Templates in transit are stored in the $\mathbf{w}^i$'s array.

Definition 8.2: 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 8.1: 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 process $n-1$ on line 30 of PROCESS. Here they are immediately added to the templates of process $n-1$, so they start belonging to a single processor.

Also, 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 2 processors hold a copy of the same template.

So the only possible problem will arise when 2 consecutive processors exchange templates. Now a template that is sent downstream on line 7 of PROCESS by a process $k - 1$ is received by process k on line 10 of PROCESS. Every template $\mathbf{w}$ that is sent downstream is tagged along with an input pattern $\mathbf{I}$. Process k will keep the template in transit if it is the best candidate for input pattern $\mathbf{I}$. To verify this, process k will compare template $\mathbf{w}$ against the templates that process k sent upstream. If a template $\mathbf{w}'$ that was sent upstream is a better candidate than $\mathbf{w}$ for the input pattern $\mathbf{I}$ (lines 19–21) then process k will discard template $\mathbf{w}$ and keep template $\mathbf{w}'$.

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

As we can see, these concurrent operations guarantee that a template that was sent downstream or upstream will not reside in 2 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 8.2: Template awareness delay

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

Proof: Consider the last process in the pipeline $(n - 1)$. This process 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 process in the variable

newNodes. This variable is received by process $n - 2$ one iteration after the templates have been created, by process $n - 3$ two iterations after the templates have been created, and in general, by process i, $n - 1 - i$ iterations after the templates have been created. This means that a process 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 process k is aware of the amount of templates that existed $n - k - 1$ iterations ago. ∎

Theorem 8.3: Weak upstream migration precondition

A process k in the pipeline sends templates upstream only if on the current iteration:

$$|myTemplates| > myShare \tag{14}$$

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

Theorem 8.4: Upstream packet size sufficiency

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

Proof: By an excess of templates we mean the number of templates over its known *fair share*. What we need to prove then, is that it is impossible for a processor in the pipeline to reach a situation where

$$|myTemplates| > myShare + 2p \tag{15}$$

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

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

Now lets consider the process $n-1$, the last in the pipeline. If this process complies with the equation 16 and receives p templates from the previous process, 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, the process will pack $2p$ templates to be sent upstream to the previous process in the pipeline (assuming its variable $myShare$ does not increase) and will receive p templates from the previous process. Notice that the p templates extra that it ended up with are not part of it's fair share because they are templates in transit. Consequently, processor's $n-1$ number of templates $|myTemplates|$ did not exceed $myShare$.

Now consider any other process 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 process can receive in the worst case scenario a total of p templates from the previous process in the pipeline and $2p$ templates from the next process in the pipeline. Now the p templates brought from the previous process in the pipeline will continue their journey to the next process (maybe not the same ones but *at least* that quantity), so they will not increase the total number of templates that the process owns. The excess of the $2p$ templates coming from the next process over $myShare$ will be packed and sent to the previous process. ∎

Theorem 8.5: Bundle size sufficiency

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

sent upstream.

Proof: See theorem 8.4. ∎

Theorem 8.6: Strong upstream migration precondition

If a process $k \in \{0, 1, \ldots, n-1\}$ in the pipeline sends templates back, then it is true that:

- 1 iteration ago process $k + 1$ complied with condition 14 and sent templates back.

- 2 iterations ago process $k + 2$ complied with condition 14 and sent templates back.

$$\vdots$$

- $n - 1 - k$ iterations ago process $n - 1$ complied with condition 14 and sent templates back.

Proof: If process k sends back templates then by theorem 8.3 it complies with condition 14. But by the reasoning in theorem 8.4 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 process $k + 1$ one iteration ago. Similarly, if process $k + 1$ sent templates back one iteration ago then by theorem 8.3 process $k + 1$ must have complied with condition 14 two iterations ago, and this can only happen if 2 iterations ago process $k + 2$ sent templates back. By repeating this argument we can state that, in general, process $k + i$ complied with condition 14 and sent templates back i iterations ago. ∎

Theorem 8.7: Strong upstream migration postcondition

If a process $k \in \{0, 1, \ldots, n-1\}$ in the pipeline sends templates back, then it is true that:

1) - at this iteration process k keeps $myShare$ templates.

   - 1 iteration ago process $k + 1$ kept $myShare$ templates.

- 2 iterations ago process $k+2$ kept $myShare$ templates.

$$\vdots$$

- $n-1-k$ iterations ago process $n-1$ kept $myShare$ templates.

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

3) The templates that each processor keeps are distinct.

Proof: First let us notice that by theorem 8.2

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

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

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

$$\vdots$$

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

This means that all the processes were aware of the same amount of templates and therefore their values for *myShare* are all the same. It is evident by looking at lines 12 to 14 of PROCESS that the process keeps *myShare* templates when it sends templates back. We also know by theorem 8.6 that they all sent templates back on the corresponding iterations. Now for any pair of processes $k+i$ and $k+j$

where $i < j$, the templates that process $k + i$ kept i iterations ago cannot be the ones that process $k + j$ kept j iterations ago. This is true because it takes at least $(j - i)$ iterations to transmit templates from j to i and process $k + j$ kept them j iterations ago, and consequently, they cannot reach process $k + i$ by $j - (j - i) = i$ iterations ago. ∎

Theorem 8.8: Template ownership delay

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

Proof: This is obvious since templates are created in process $n - 1$ on line 30 of the code of PROCESS. These templates maybe sent back in the pipeline one step of the pipeline per iteration. The distance from k to process $n - 1$ is equal to $n - k - 1$, so the templates that k has must have been created at least $n - k - 1$ iterations ago. ∎

Theorem 8.9: Network size lower bound

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

$$N > (n - k)myShare \tag{17}$$

Proof: Notice that if process k sends templates back then it complies with condition 14 and by Theorem 8.7 all processes from k on-wards kept $myShare$ templates and these templates are all distinct. Also by theorem 8.8 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:

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

$$> myShare + (n - 1 - k)myShare = (n - k)myShare \tag{18}$$

■

Theorem 8.10: Template ownership bound

A process 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 process k can not exceed $myShare$ of templates has already been shown by theorem 8.4. Furthermore, the fact that it owns less that 0 templates is also 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 to the next process 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 processes in the pipeline.

Patterns lost to the neighboring processes in the pipeline will travel, in a worst case scenario, all the way to the last process in the pipeline and afterwords find their way back to process k. If this is the situation then process k will have to wait $n - 1 - k$ units of time, for the patterns sent, to reach process $n - 1$ and then wait another $n - 1 - k$ iterations for the patterns to come back. This is a total of $2(n - 1 - k)$ iterations before a packet of p templates sent downstream by process k is seen again by

process k. If during these $2(n-1-k)-1$ iterations process k has the bad luck of sending p templates of it's own templates downstream at each iteration, then during that time process 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 8.11: Overflow impossibility

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

Proof: Let us assume by contradiction that it cannot absorb the templates it has received from the next process downstream. This means that process 0 complies with condition 14 and that is has to send templates back. By theorem 8.9 the number of templates N that existed in the system $n-1$ iterations ago complies with equation 17. But by line 35 of PROCESS we have:

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

This means that the number N of templates that existed in the system $n-1$ iterations ago is greater than $nodes$, which is a contradiction of theorem 8.2 ∎

Theorem 8.12: Pipeline depth invariance

The difference in the number of myShare that 2 arbitrary processes in the pipeline have cannot exceed $p+1$ where p is the packet size. Note that the difference in number of templates is independent of the pipeline size n.

Proof: First, by theorem 8.2 we know that a process k is aware of the number of templates that existed $n-1-k$ iterations ago. Also, the largest difference in the number of templates that two process

are aware of is found in the difference between process 0 and process $n - 1$. Now, let us assume that process 0 is aware of *nodes*$_0$ templates. Since this amount of templates existed $n - 1$ iterations ago and we can create a maximum of p templates per iteration then the *maximum* number of templates that process $n - 1$ can be aware of is *nodes*$_0 + (n - 1)p$. This means that the value of *myShare* for process 0 is

$$myShare_0 = \left\lceil \frac{nodes_0}{n} \right\rceil \geq \frac{nodes_0}{n} \tag{20}$$

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

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

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

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

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

■

Theorem 8.13: Partial evaluation correctness

If we make the packet size p of PROCESS equal to the size of the training set and set the number of processes to $n = 1$, then the parallel algorithm presented here is equivalent to the no Matchtracking FS-FAM.

Proof: Let us start by noting that if the number of process 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 processes 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 previous process 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 the no matchtracking FS-FAM 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 8.14: 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

$$\frac{p}{2\sqrt{3}} \tag{22}$$

Proof: Given that in the parallel FS-FAM algorithm there are many templates in transit we cannot know exactly how many templates each process possesses. We can though, approximate a worst case workload balance scenario if we assume, as will usually be the case, that the number of comparisons

that a given process performs on each iteration will be proportional to the number of templates that it is allowed to possess or *myShare*. In a worst case scenario, on every iteration the network will be creating p new templates so process k will have a value of

$$nodes_k = nodes_0 + kp$$

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

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

$$\frac{nodes_0 + \frac{p}{n}\sum_{k=0}^{n-1} k}{n} =$$

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

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

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

$$\sqrt{p^2 \frac{n^2 - 1}{12n^2}} = \frac{p}{2\sqrt{3}} \sqrt{1 - n^{-2}} < \frac{p}{2\sqrt{3}} \tag{23}$$

∎

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.

IX. EXPERIMENTS

Experiments were conducted on 3 databases: 1 real–world database and 2 artificially–generated databases (Gaussian distributed data). Training set sizes of $1000 \times 2^i, i \in \{5, 6, \ldots, 9\}$, that is 32,000 to 512,000 patterns were used for the training of no-matchtracking FS-FAM and pipelined no matchtracking FS-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 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 α 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 $(p, PT) =$ (partition, training set size) values we conducted 12 independent

experiments (training and performance phases), corresponding to different orders of pattern presentations within the training set. As a reminder FS-FAM performance depends on the values of the network parameters $\bar{\rho}_a$, and α, as well as the order of pattern presentation within the training set.

All the tests where conducted on the `OPCODE` Beowulf cluster of workstations of the Institute for Simulation and Training. This cluster consists of 96 nodes, with dual Athlon 1500+ processors and 512MB of RAM. The runs were done in such as way as to utilize half as many nodes as p. 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 pipelined no matchtracking FS-FAM (Higher classification performance is better).

2) Size of the trained, pipelined, no matchtracking FS-FAM.

3) Speedup of pipelined no-matchtracking FS-FAM compared to the no-matchtracking FS-FAM.

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

IX.-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%.

IX.-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 μ and variance σ^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 where 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 the pipeline no-matchtracking FS-FAM 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 the no-match tracking FS-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 the no-match tracking FS-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 competitive loop in FS-FAM (that can be easily extended to other ART architectures).

X. Summary - Conclusions

We have produced a pipelined implementation of the no-matchtracking FS-FAM algorithm. This implementation can be extended to other ART neural network architectures that have similar competitive structure as FS-FAM. 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 the parallel no-match tracking FS-FAM (a) is equivalent with the sequential version of the no-match-tracking FS-FAM , (b) it does not suffer from inconsistencies, and (c) it exhibits good performance. In particular, the good performance of the parallel no-match tracking FS-FAM 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 no-match tracking FS-FAM created. We believe that our objective of appropriately implementing no-match tracking FS-FAM on a Beowulf cluster has been accomplished and a clear evidence of this assertion are the speed-up results exhibited by the pipelined no-match tracking FS-FAM and illustrated in Figures 15-17. Extension of our implementation approach to other "competitive" classifiers is obvious. Extension of our implementation to the match-tracking FS-FAM algorithm is more involved and it is the topic of our future research.

<div align="center">REFERENCES</div>

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. (2003). Putting the utility of match tracking in fuzzy ARTMAP to the test. In *Proceedings of the Seventh International Conference on Knowledge–Based Intelligent Information Engineering* (Vol. 2, pp. 1–6). KES'03.

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 diagnosys: 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., Fragkou, V. G., & Kehagais, A. (2001). Text classification using the σ–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 27th – December 1st). 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.icf.uci.edu/mlearn/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.
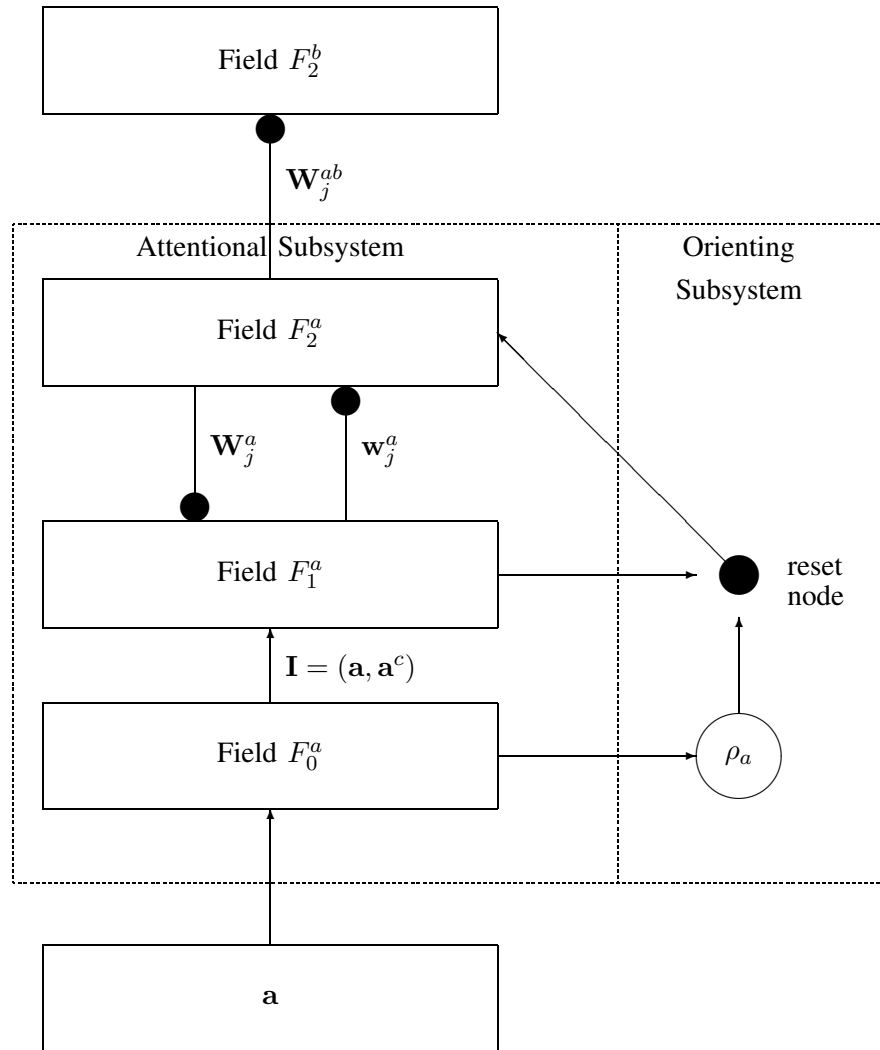
Fig. 1. Block Diagram of the FS-FAM Architecture.

FS-FAM-LEARNING-PHASE$\left(\left\{\mathbf{I}^1, \mathbf{I}^2, \ldots, \mathbf{I}^{PT}\right\}, \bar{\rho}_a, \alpha, epochs, \varepsilon\right)$

  1 $\mathbf{w}_0 \leftarrow \underbrace{(1, 1, \ldots, 1)}_{2M_a}$

  2 *templates* $\leftarrow \{\mathbf{w}_0\}$

  3 *iterations* $\leftarrow 0$

  4 **repeat**

  5   *modified* $\leftarrow$ FALSE

  6   **for each $\mathbf{I}^r$ in** $\left\{\mathbf{I}^1, \mathbf{I}^2, \ldots, \mathbf{I}^{PT}\right\}$

  7   **do** $\rho \leftarrow \bar{\rho}_a$

  8   **repeat**

  9            $T_{max} \leftarrow 0$

10   *status* $\leftarrow$ 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* $\leftarrow$ Allocated

20   **else** *status* $\leftarrow$ Matchtracking

21                      $\rho \leftarrow \rho(\mathbf{I}^r, \mathbf{w}_{j_{max}}^a) + \varepsilon$

22   **until** *status* $\neq$ Matchtracking

23   **if** *status* $=$ 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* $\leftarrow$ TRUE

27   **else** *templates* $\leftarrow$ *templates* $\cup \{\mathbf{I}^r\}$

28   *modified* $\leftarrow$ TRUE

29   *iterations* $\leftarrow$ *iterations* $+ 1$

30 **until** (*iterations* $=$ *epochs*) or (*modified* $=$ FALSE)

31 **return** *templates*

Fig. 2.   FS-FAM off-line training phase algorithm

FS-FAM-on-line-Learning$\left(\left\{\mathbf{I}^1, \mathbf{I}^2, \ldots, \mathbf{I}^{PT}\right\}, \bar{\rho}_a, \alpha, \varepsilon\right)$

1   $\mathbf{w}_0 \leftarrow \underbrace{(1, 1, \ldots, 1)}_{2M_a}$

2   $templates \leftarrow \{\mathbf{w}_0\}$

3   **for each** $\mathbf{I}^r$ **in** $\left\{\mathbf{I}^1, \mathbf{I}^2, \ldots, \mathbf{I}^{PT}\right\}$

4   **do** $\rho \leftarrow \bar{\rho}_a$

5   **repeat**

6           $T_{max} \leftarrow 0$

7           $status \leftarrow \mathtt{NoneFound}$

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** $\mathbf{w}_{j_{max}}^a \neq uncommitted$

15   **then if** $label(\mathbf{I}^r) = label(\mathbf{w}_{j_{max}}^a)$

16   **then** $status \leftarrow \mathtt{Allocated}$

17   **else** $status \leftarrow \mathtt{Matchtracking}$

18                    $\rho \leftarrow \rho(\mathbf{I}^r, \mathbf{w}_{j_{max}}^a) + \varepsilon$

19   **until** $status \neq \mathtt{Matchtracking}$

20   **if** $status = \mathtt{Allocated}$

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.   FS-FAM on-line training phase algorithm

FS-FAM PERFORMANCE PHASE($\mathbf{I}^r$, *templates*, $\bar{\rho}_a, \beta$)

```
 1   T_max ← 0
 2   j_max ← NIL
 3   for each w_j^a in templates
 4   do
 5       if ρ(I^r, w_j^a) ≥ ρ̄_a and T(I^r, w_j^a, β) > T_max
 6         then
 7                 T_max ← T(I^r, w_j^a, β)
 8                 j_max ← j
 9
10   if w_{j_max}^a ≠ w_0
11       then return label(w_{j_max}^a)
12       else return NIL
```

Fig. 4.   FS-FAM performance phase algorithm

FS-FAM-NO-MATCHTRACKING-LEARNING($\left\{ \mathbf{I}^1, \mathbf{I}^2, \ldots, \mathbf{I}^{PT} \right\}, \rho, \alpha$)

```
 1   w_0 ← (1, 1, …, 1)
                └──┬──┘
                  2M_a
 2   templates ← {w_0}
 3   for each I^r in {I^1, I^2, …, I^PT}
 4   do
 5       repeat
 6               T_max ← 0
 7               status ← NoneFound
 8               for each w_j^a in templates
 9               do if [ρ(I^r, w_j^a) ≥ ρ]  and  [T(I^r, w_j^a, α) > T_max]
10                     then
11                             T_max ← T(I^r, w_j^a, α)
12                             j_max ← j
13
14               if w_{j_max}^a ≠ w_0 and label(I^r) = label(w_{j_max}^a)
15                 then w_{j_max}^a ← w_{j_max}^a ∧ I^r
16                 else templates ← templates ∪ {I^r}
17       until
18   return templates
```

Fig. 5.   Anagnostopoulos' No–matchtracking on-line training FS-FAM algorithm
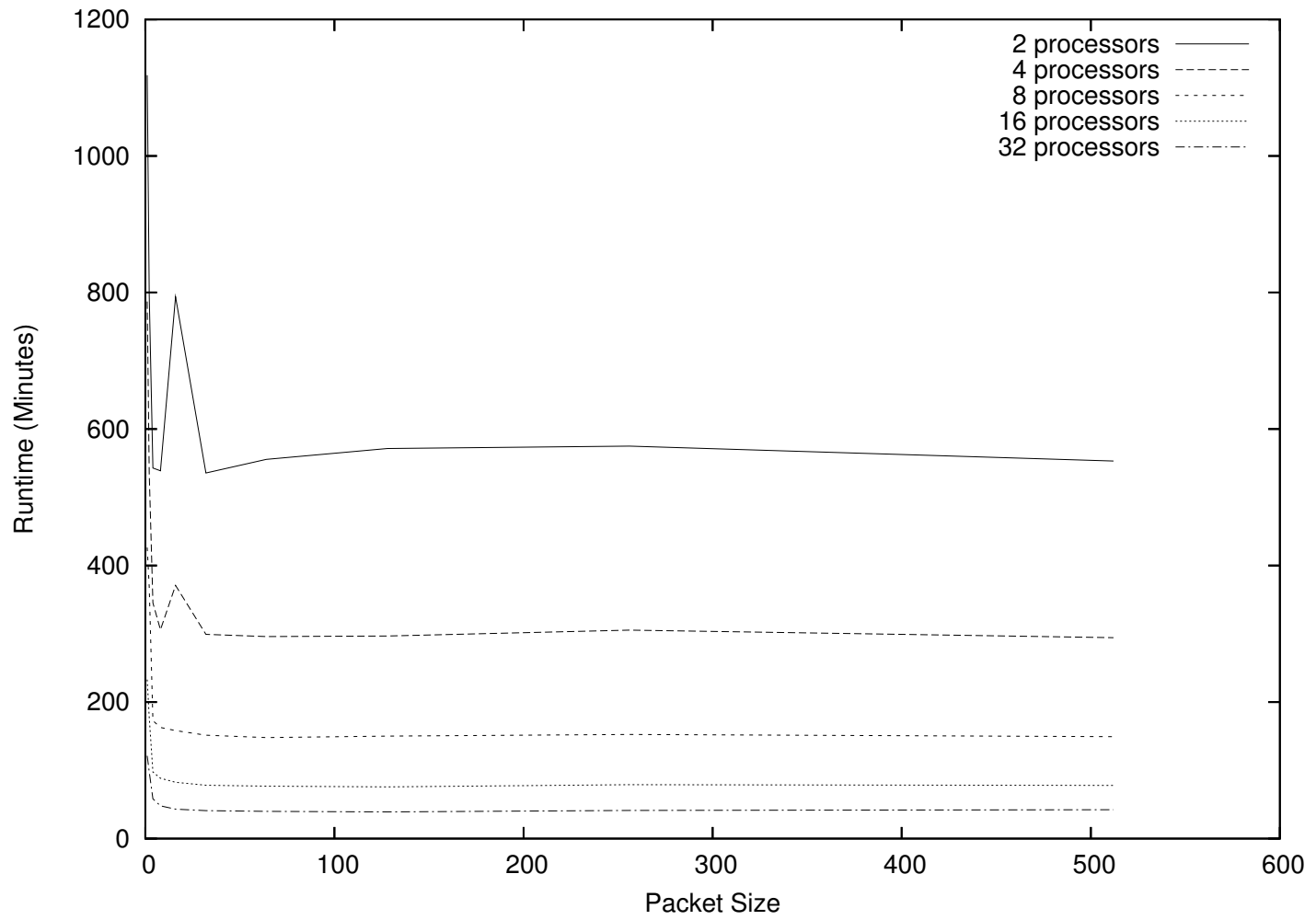
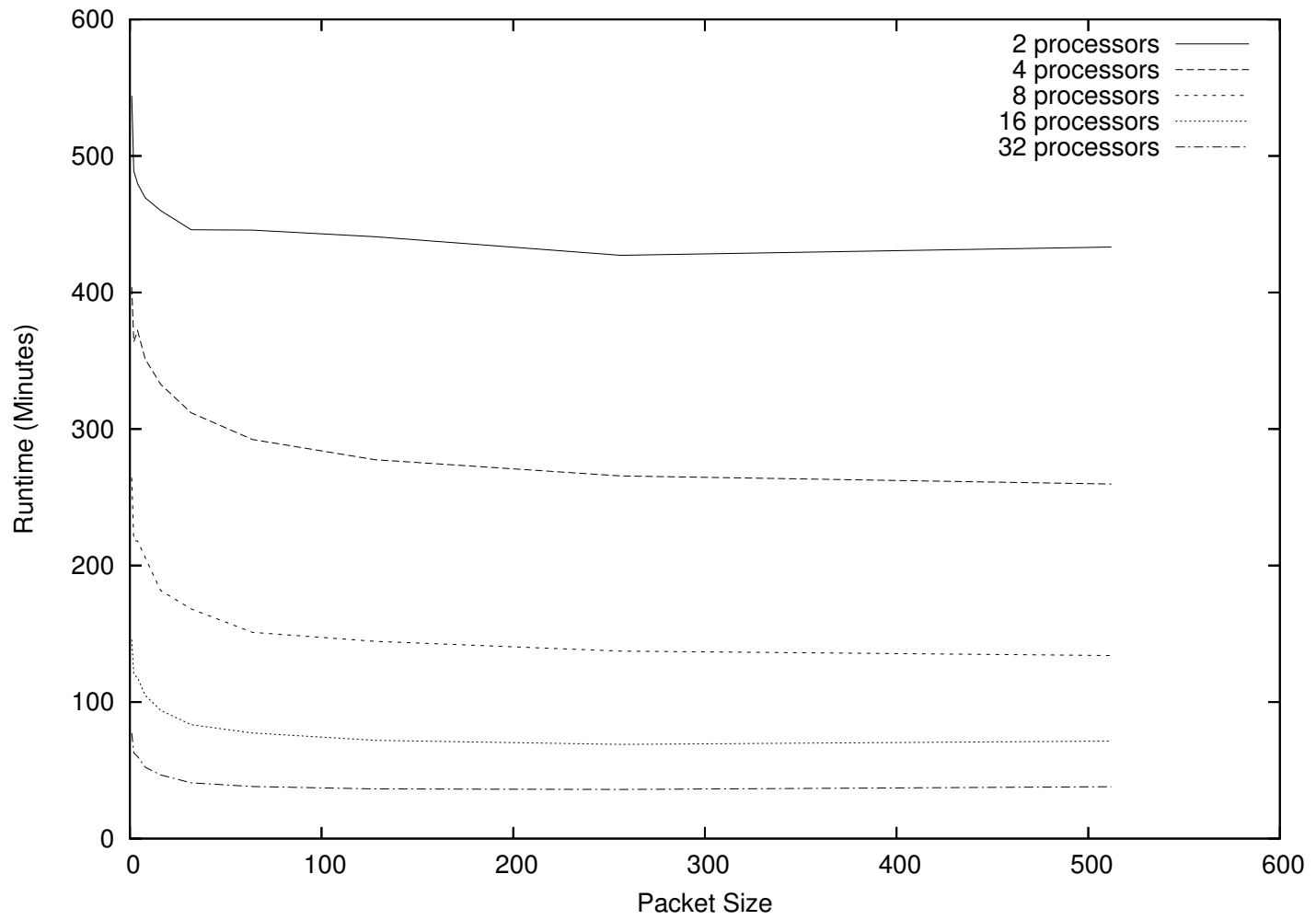Fig. 6. Runtime versus packet size for the Covertype database.

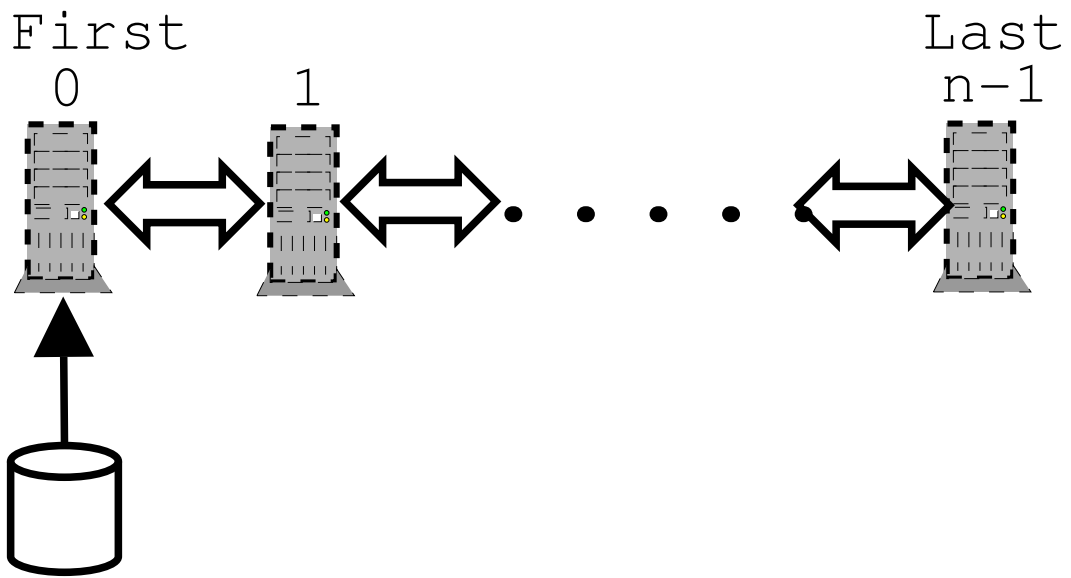Fig. 7. Runtime versus packet size for the Gaussian 5% database.
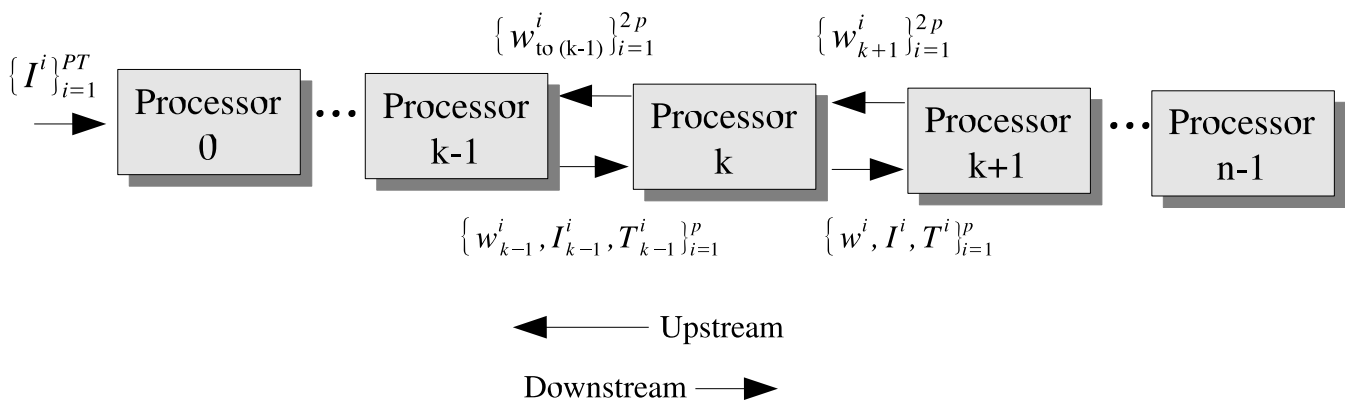
Fig. 8. Pipeline Structure

Fig. 9. Exchange of packets between processors. Note, packets are listed for processor k only.

$\textsc{Init}(p)$

1   *nodes* $\leftarrow 0$

2   *myTemplates* $\leftarrow \{\}$

3   $\forall_{i=1}^{2p}\left(\mathbf{w}_{to(k-1)}^{i} \leftarrow \texttt{none}\right)$

4   $\forall_{i=1}^{p}\left(\mathbf{w}^{i} \leftarrow \texttt{none}\right)$

5   $\forall_{i=1}^{p}\mathbf{I}^{i} \leftarrow \texttt{none}$

6   *myShare* $\leftarrow 0$

7   *newNodes* $\leftarrow 0$

8   *newNodes*$_{k+1} \leftarrow 0$

9   *continue* $\leftarrow \texttt{TRUE}$

Fig. 10.   Initialization procedure for pipelined parallel no-matchtracking FS-FAM implementation

PROCESS$(k, n, \rho_a, \alpha, p)$

1   INIT(p)
2   **while** *continue*
3   **do**
4   **while** $|myTemplates| > myShare$
5   **do**
6   EXTRACT-TEMPLATE $\left(myTemplates, \left\{ \mathbf{w}^i_{to(k-1)} \right\} \right)$
7   SEND-NEXT $\left(k, n, \left\{ \left(\mathbf{w}^i, \mathbf{I}^i, T^i \right) : i = 1, 2, \ldots, p \right\} \right)$
8   RECV-NEXT $\left(k, n, \left\{ \mathbf{w}^i_{k+1} : i = 1, 2, \ldots, 2p \right\}, newNodes_{k+1} \right)$
9   SEND-PREV $\left(k, \left\{ \mathbf{w}^i_{to(k-1)} : i = 1, 2, \ldots, 2p \right\}, newNodes \right)$
10   RECV-PREV $\left(k, \left\{ \left(\mathbf{w}^i_{k-1}, \mathbf{I}^i_{k-1}, T^i_{k-1} \right) : i = 1, 2, \ldots, p \right\} \right)$
11     $newNodes \leftarrow newNodes_{k+1}$
12     $\mathcal{S} \leftarrow \left\{ \mathbf{w}^i_{k+1} \right\}$
13   **for each** i **in** $\{1, 2, \ldots, p\}$
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** $\mathbf{I}^i_{k-1} = \texttt{EOF}$
17   **then** *continue* $\leftarrow$ FALSE
18   **else** $\mathcal{S} \leftarrow \left\{ \mathbf{w}^i_{to(k-1)} \right\}$
19   **for each** i **in** $\{1, 2, \ldots, p\}$
20   **do** FIND-WINNER$(\mathbf{I}^i_{k-1}, \mathbf{w}^i_{k-1}, T^i_{k-1}, \rho_a, \alpha, \mathcal{S})$
21              $\left(\mathbf{I}^i, \mathbf{w}^i, T^i \right) \leftarrow \left(\mathbf{I}^i_{k-1}, \mathbf{w}^i_{k-1}, T^i_{k-1} \right)$
22   **for each** i **in** $\{1, 2, \ldots, p\}$
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                  $myShare \leftarrow \left\lceil \frac{nodes}{n} \right\rceil$
36   SEND-NEXT $(k, n, \{(\texttt{none}, \texttt{none}, 0)\})$
37   RECV-NEXT $\left(k, n, \left\{ \mathbf{w}^i_{k+1} : i = 1, 2, \ldots, 2p \right\}, newNode_{k+1} \right)$
38   $myTemplates \leftarrow myTemplates \cup \left\{ \mathbf{w}^i_{k+1} : i = 1, 2, \ldots, 2p \right\}$

Fig. 11.   Pipelined FS-FAM ring implementation for parallel processing

FIND-WINNER$(\mathbf{I}, \mathbf{w}, T, \rho_a, \alpha, \mathcal{S} = \{\mathbf{w}^i\})$

```
 1   idx ← −1
 2   for each wⁱ in 𝒮
 3   do if [ρ(I, wⁱ) ≥ ρₐ]
 4        then
 5              if [T(I, wⁱ, α) > T]
 6                then
 7                       T ← T(I, wⁱ, α)
 8                       idx ← i
 9                else if [T(I, wⁱ, α) = T] and index(wⁱ) < index(w)
10                       then T ← T(I, wⁱ, α)
11                              idx ← i
12   if idx ≠ −1
13      then
14              EXTRACT(wⁱᵈˣ, 𝒮)
15              ADD(w, 𝒮)
16              w ← wⁱᵈˣ
17              return TRUE
18      else
19              return FALSE
```

Fig. 12.   Utility function to find best candidate template in a template list. Needed by parallel no-matchtracking FS-FAM ring implementation

PROCESS(ρ_a, α)

```
 1   myTemplates ← {}
 2   ∀ᵖᵢ₌₁Iⁱ ← none
 3   newNodes ← 0
 4   continue ← TRUE
 5   RECV-PREV (k, {(wⁱₖ₋₁, Iⁱₖ₋₁, Tⁱₖ₋₁) : i = 1, 2, . . . , p})
 6   newNodes ← 0
 7   for each i in {1, 2, . . . , p}
 8   do FIND-WINNER(Iⁱ, wⁱ, Tⁱ, ρₐ, α, myTemplates)
 9      if class(Iⁱ) = class(wⁱ)
10         then
11                 myTemplates.ADD({Iⁱ ∧ wⁱ})
12
13         else
14                 newTemplate ← Iⁱ
15                 index(newTemplate) ← newNodes
16                 myTemplates.ADD({Iⁱ, wⁱ})
17                 newNodes ← newNodes + 1
18
```

Fig. 13.   Partial evaluation of parallel no matchtracking FS-FAM using number of processors $p = 1$
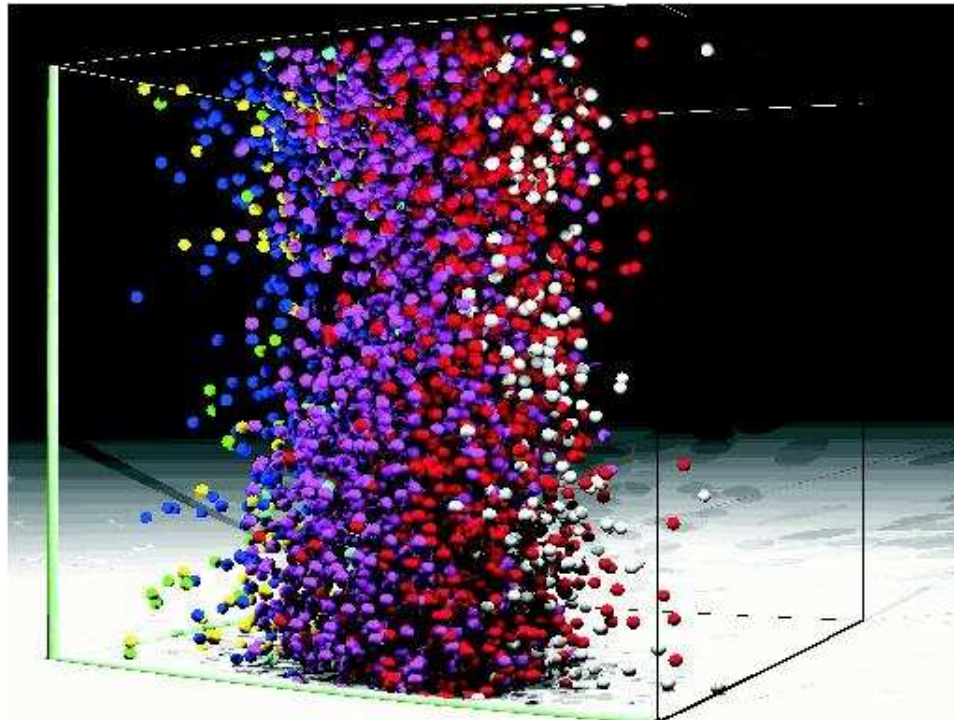
Fig. 14. A random sample of 5,000 Forest Covertype data-points out of the available 581,012 data-points is shown. The data-points are projected to the first 3 dimensions of the database. Different Colors for the data-points represent different class labels.
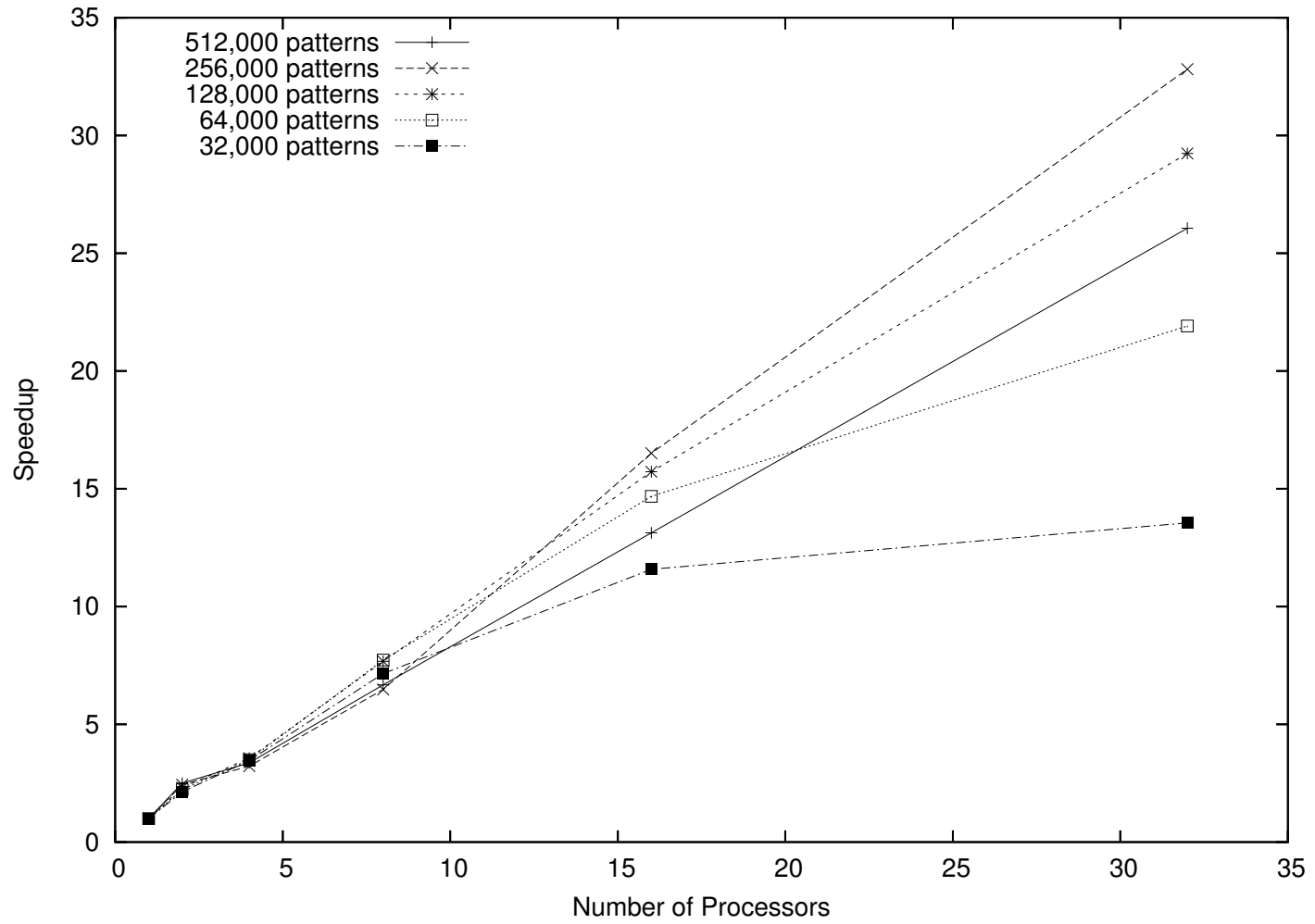
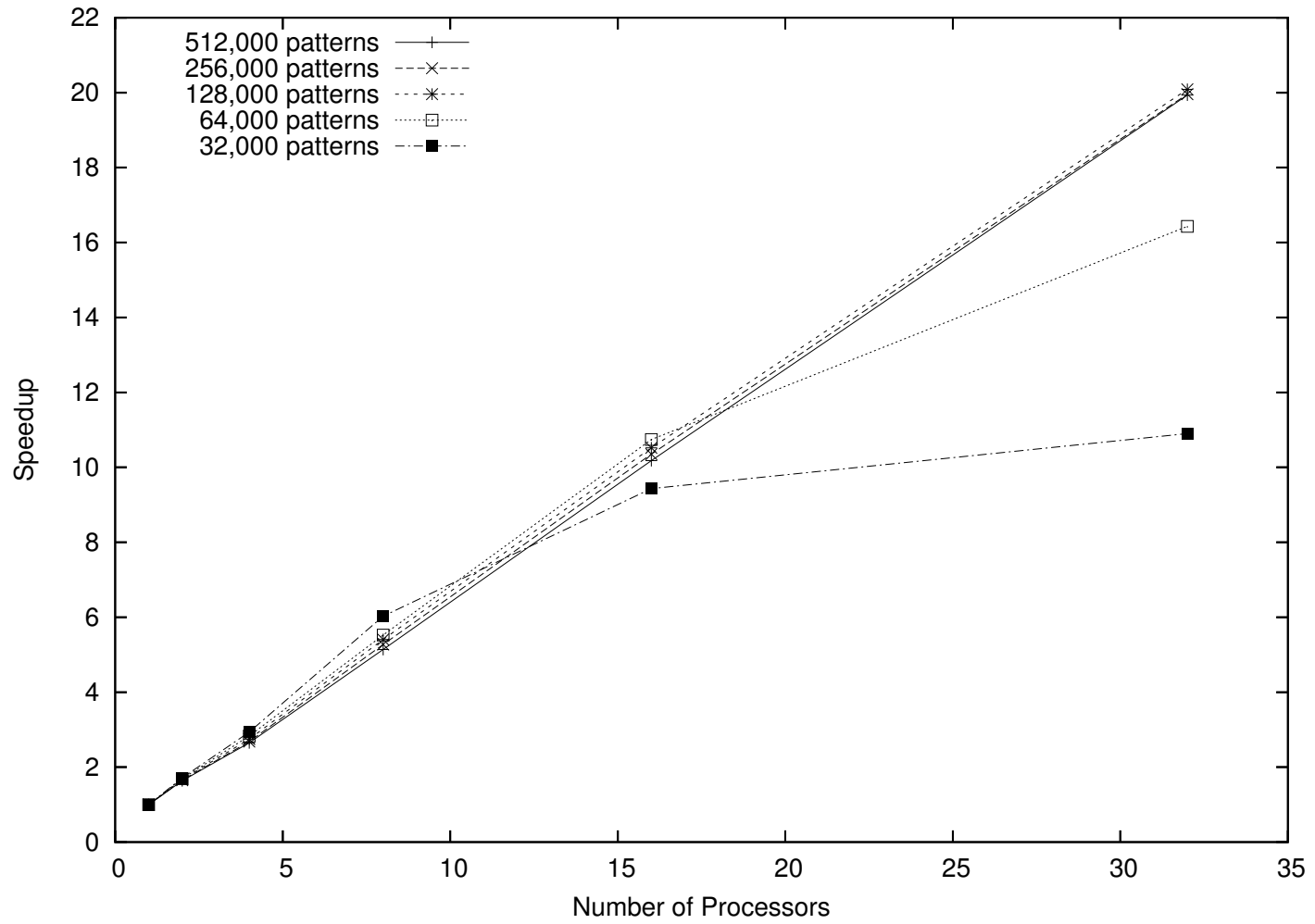Fig. 15. Parallel speedup versus number of processors for Covertype database.

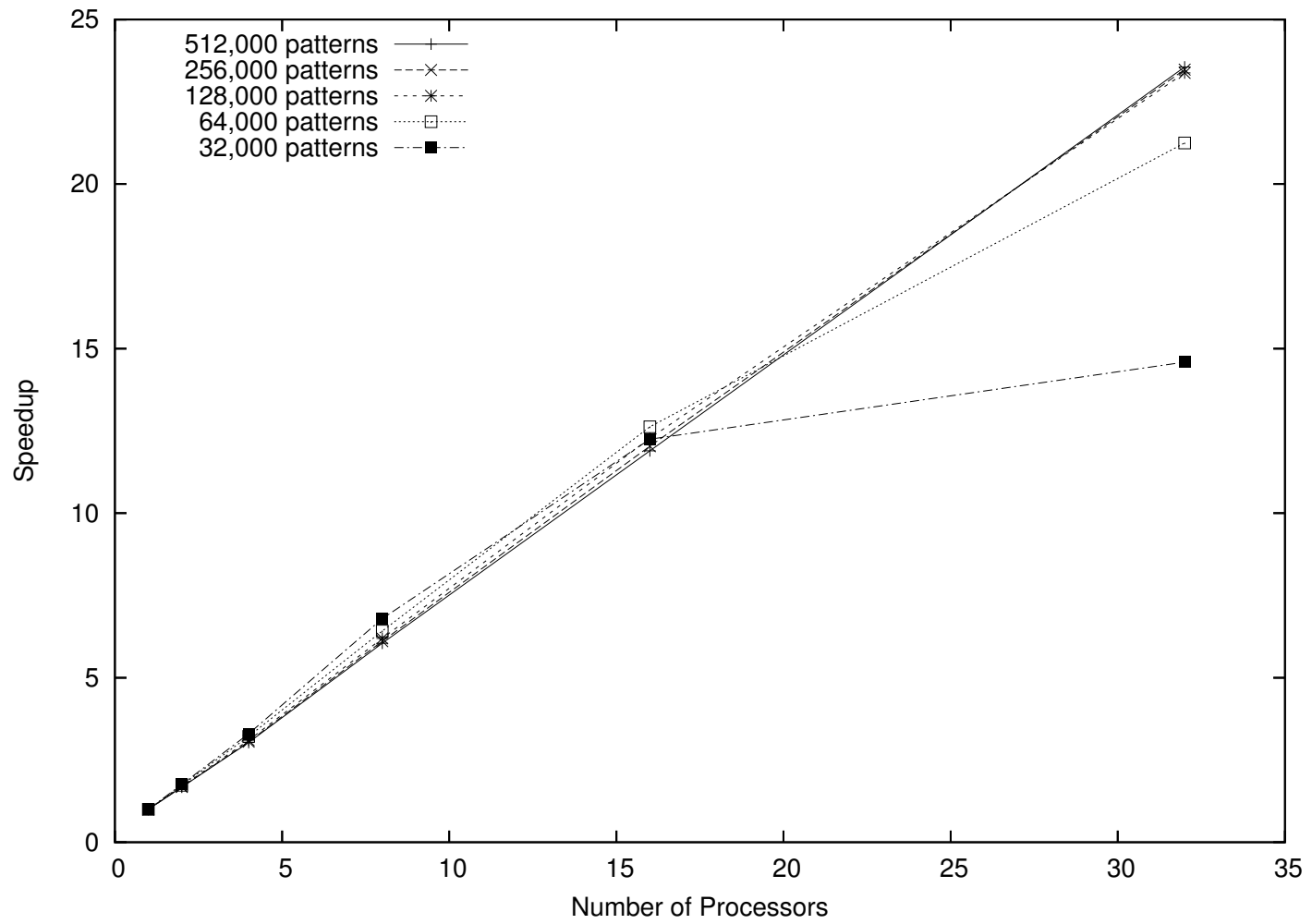Fig. 16. Parallel speedup versus number of processors for Gaussian 5% database.

Fig. 17.   Parallel speedup versus number of processors for Gaussian 15% database.

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

COVERTYPE RUN STATISTICS

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

GAUSSIAN 5% RUN STATISTICS

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

GAUSSIAN 15% RUN STATISTICS