PIPELINING OF ART ARCHITECTURES (FAM, EAM, GAM) WITHOUT MATCH TRACKING

JIMMY SECRETAN(*), JOSÉ CASTRO(**), AMIT CHADHA(*), BRIAN HUBER(*), JOE TAPIA(*), MICHAEL GEORGIOPOULOS(*), GEORGIOS ANAGNOSTOPOULOS(***), SAM RICHIE(*)
(*) Dept. of ECE, University of Central Florida, Orlando, FL 32816
(**) Comp Eng., Instituto Tecnológico de Costa Rica, Cartago, Costa Rica
(***) Dept. of ECE, Florida Institute of Technology, Melbourne, FL 32901

ABSTRACT

Adaptive Resonance theory was introduced by Grossberg to address the stability versus plasticity dilemma. That is, how can one design a learning system that is plastic enough to learn new information, and at the same time stable enough not to forget old, important information that it has already learned. In the past two decades a number of ART neural network architectures were introduced in the literature, based on the ART theory. These architectures can solve clustering and classification problems. Our focus in this paper is ART architectures that function like classifiers. ART classifiers have a number of desirable properties, such as guaranteed convergence to a solution for any classification problem of interest, fast convergence to a solution (i.e., they converge in a few epochs, where epoch is a single presentation of all the training data), they can be trained in an on-line fashion, they have the ability to recognize novel inputs, and they can explain the answers that they produce. One of their limitations is that for large database problems, where inevitably a lot of categories (clusters) are created to represent the input data, the convergence to a solution becomes excruciatingly slow, since ART's complexity is proportional to the product of the input patterns and the number of categories created. To address this problem, Castro had suggested a parallel implementation of Fuzzy ARTMAP (one of the ART classifiers) on a Beowulf cluster. Castro's implementation was efficient and general enough to apply to other ART architectures, such as Ellipsoidal ARTMAP and Gaussian ARTMAP, which are two other examples of ART classifiers. In this paper we validate this claim, that EAM and GAM can be implemented effectively on a Beowulf cluster, and we verify this claim by presenting appropriate experimental results. What is also worth noting is that Castro's Fuzzy ARTMAP Beowulf implementation can also be applied to other competitive classifiers, neural network based or not.

1. INTRODUCTION

The ART architectures that we are focusing on this paper are the Fuzzy ARTMAP (see Carpenter, et al., 1992), the Ellipsoidal ARTMAP (see Anagnostopoulos, et al., 2001), and Gaussian ARTMAP (see Williamson, 1996, and 1997). All of these architectures are classifiers and share the good ART properties that we mentioned earlier. Nevertheless they all suffer from the category proliferation problem, where, as the training data set grows in size, the number of categories formed to represent the data increases with it. The immediate impact of this category proliferation is that it takes a significant amount of time for these architectures to converge to a solution, despite the fact that convergence is guaranteed by passing the training data a few times through the

network. As a reminder, the categories formed in Fuzzy ARTMAP (FAM) are hyper-rectangles, in Ellipsoidal ARTMAP (EAM) are ellipsoids, and in Gaussian ARTMAP (GAM) are the parameters of Gaussian curves (such as mean and variance). The category proliferation problem leads us into ART structures with many categories, and consequently slows down the ART training process. To address this issue with ART, especially when the datasets are large, Castro, et al., (see Castro, 2004) has proposed an effective and efficient pipeline implementation of Fuzzy ARTMAP on a Beowulf cluster. In particular, Castro has implemented a variation of Fuzzy ARTMAP, called no-match tracking Fuzzy ARTMAP (NMT-FAM) that was easier to parallelize than FAM. The no-match tracking version of Fuzzy ARTMAP was first introduced and examined by Anagnostopoulos, et al., 2003. In his paper Anagnostopoulos illustrated that NMT-FAM achieves equivalent, and at times, better generalization than FAM at the expense of creating more categories. Furthermore, it was recognized by Castro (see Castro, et al., 2004) that the proposed pipeline implementation for NMT-FAM can be extended to other ART architectures. In this paper we are demonstrating this claim by implementing the NMT-EAM and NMT-GAM on the Beowulf cluster. Our experimental results, included here, illustrate the efficiency of our implementation. Our experiments were run on a Beowulf cluster and the database used for the experiments was the Forrest Covertype database (obtained from the UCI repository).

The organization of the paper is as follows: In section 2 we discuss some of the specifics of the ART architectures that are applicable to FAM, EAM and GAM. Furthermore, the high level operation of no-match tracking ART architectures is outlined. In Section 3, we describe some of the characteristics of the Beowulf cluster and we discuss the specifics of the Beowulf pipeline implementation. In Section 4, we present the experimental results conducted and we depict the speed-up obtained by the pipeline implementations of NMT-FAM, NMT-EAM and NMT-GAM. Finally, in Section 5 we provide a summary of our work and directions for future work.

2. ART ARCHITECTURES

The ART architecture has three major layers. The first is the input layer (F_1^a) where the input patterns are presented. Next, there is the category representation layer (F_2^a) where compressed representations of these input patterns, called templates are formed (designated as $\mathbf{w}_j^a$). Finally, there is the output layer (F_2^b) that holds the labels of the categories formed in the category representation layer (designated by $\mathbf{W}_j^{ab}$). In FAM, EAM or GAM only one of the components of this weight vector is equal to 1 and the rest of the components are 0. The component that is equal to 1, designates the label of the first input pattern that committed this node, for the first time. Normalization of the input patterns (so that their components lie in the interval [0, 1]) is a frequent pre-processing strategy for all of these architectures. The number of nodes in the input layer of FAM is equal to $2M_a$, where M_a is the dimensionality of the vector (note that for EAM and GAM the number of nodes in the input layer is equal to M_a).

It is worth mentioning that the vector of weights $\mathbf{w}_j^a$ (template), emanating from node *j* in the category representation layer, differs from one ART architecture to another. For instance, in FAM, $\mathbf{w}_j^a$ represents the lower and upper endpoints of a hyper-box enclosing the input patterns that chose this box as their representative box. On the other hand, in the case of EAM, $\mathbf{w}_j^a$ represents the center of an ellipsoid, and the direction of

the ellipsoid's major axis. This ellipsoid encloses within its boundaries all the input patterns that chose this ellipsoid as their representative ellipsoid. Finally, in the case of GAM, $\mathbf{w}_j^a$ represents the mean and the variances of the input patterns that chose category (node) *j* as their representative category.

ART operates in two phases: The training phase and the performance phase. In the training phase of ART we have a collection of input/associated labels pairs (called training set), and we present it to ART, in a repeated fashion until the network learns this collection or until an upper limit on the number of epochs is reached. The training phase of the no-match tracking FAM, EAM and GAM follows the following simple rules:

1. Find the nearest category (node) in the category representation layer of ART that resonates with the input patterns.
2. If the label of the input pattern matches the label of the category, learning ensues according to the specific ART learning rules (FAM, EAM, GAM).
3. If the label of the input pattern does not match the label of the category, the category is reset, and an uncommitted category is activated that learns the input/output pair according to the specific ART learning rules (FAM, EAM, GAM).

The learning rules in each one of the ART architectures are different. In FAM when a new input pattern is coded by a category its hyper-box representation expands so that it encloses within its boundaries the new input pattern. In EAM when a new input pattern is coded by a category its ellipsoidal representation expands so that it encloses within its boundaries the new input pattern. Finally, in GAM when a new input pattern is coded by a category its mean and variances vectors are appropriately updated to reflect that the new input pattern is now a member of the cluster of points that this category represents.

For the performance phase, a new input pattern excites the input layer of ART and finds the nearest category in the category representation layer that resonates with it, and uses the label of this category as the predicted label of the input pattern presented. If an existing category that satisfies these conditions cannot be found, then ART flags this input pattern as a novel input pattern whose label cannot be reliably predicted.

3. PARALLEL, NO-MATCH TRACKING FAM IMPLEMENTATION

3.1 Beowulf Cluster Preliminaries

A Beowulf cluster computer is a collection of standard PC's connected together by a fast network interconnect and programmed in parallel, usually with open-source software. 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.

We have two choices for parallelization design. We can request from each node in the network to process a different input pattern during a slice of time. 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 scatter/gather approach where all nodes communicate to a gathering master node. This approach was explored in (Malkani and Vassiliadis, 1995).

In this paper, the authors propose a hypercube network design, where each node has a subset of the templates. A single input pattern is broadcast to all of the nodes and through several synchronization operations they find the template with the maximum bottom up input. However, because the results must be synchronized with a master node, this approach can limit scalability. 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 scatter/gather approach tends to degrade communication performance as the number of processing elements 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, and to find out the size of a packet a receiving process would have to incur an extra (and expensive) communication.

3.2 Parallel Implementation of ART Architectures

Castro, et al., (see Castro, 2004) has demonstrated that Anagnostopoulos's NMT-FAM variant is amenable to production-line style pipeline. We extended Castro's idea to the parallel implementation of the NMT-EAM and NMT-GAM. For the implementation of the no-match tracking FAM, EAM, GAM, we first introduce a number of definitions. The algorithm itself (parallel, no-match tracking ART implementation) is shown in the Appendix, after the definitions are introduced. In the description of the parallel no-match tracking ART (FAM, EAM, GAM) the initialization procedure *(INIT(p))* and *WINNER* are not described due to lack of space. More details about these procedures as well as the algorithm presented here can be found in (Castro, et al., 2004).

n: number of processors in the pipeline

k: index of current process, $k \in \{0,1,\ldots,n-1\}$

p: packet size, number of patterns sent downstream, 2p = number of templates sent upstream

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

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 process is aware of (its own plus the templates of other processors)

myShare: amount of templates that the current process should have.

w_{k-1}^i: template i coming from previous process in the pipeline.

w_{k+1}^i: template i coming from next process in the ring.

w^i: template i going to next process in the ring.

$w_{to(k-1)}^i$: template i going to previous process in the pipeline.

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

w.class: class label associated with a given input template.

$index(w)$: sequential index assigned to the template.

newNodes: number of created nodes on a given iteration to communicate upstream in the pipeline.

$newNodes_{k+1}$: number of created nodes on a given iteration communicated from processor k+1 in the pipeline.

The exchange of packets between processors is pictorially illustrated in figure 1. 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 ART (FAM, EAM or GAM) is shown in the Appendix. The pseudocode, shown in the Appendix, is the main heart of the parallel algorithm. 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 we enter the loop of lines 2 through 35 (see Appendix). 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, do not 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,…,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 receipt of information.

On line 30 of the Appendix we add 2 templates to the template set *myTemplates*. This is because a new template was created and the current candidate winner *w* is not of the correct category and has to be inserted back into the pool of templates. The function *Find-Winner* is also important. This function searches through a set of templates *S* to find if there exists a template $\mathbf{w}^i$ that is a better choice (using ART 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 *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 Appendix). On line 12 we set *S* to represent the set of templates that have been sent upstream to node *k* by node *k+1*. We loop through each pattern, template pair $(\mathbf{I}, \mathbf{w})$ to see if one of the templates, sent upstream, has a higher activation (bottom-up input) than the ones that were sent downstream; if this is true then the template will be extracted from *S*. The net result of this is that *S* ends up containing the templates that lost the competition, and therefore the ones that process *k* should keep (line 15).

The converse process is done in lines 18 to 21. On line 18 we set *S* to represent the set of templates that are sent *back* to the previous node k-1 in the pipeline. In 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 *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 *S* is discarded since it contains the losing templates and therefore the templates that process *k-1* keeps.

Finally, on line 30 of the Appendix 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 case 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 $(\mathbf{w}^i, \mathbf{I}^i, T^i) = (none, none, 0)$ to the next processor (see line 36 of Appendix). 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.

4. EXPERIMENTS

The database used for testing the efficiency of the parallel, no-match tracking ART (FAM, EAM, GAM) implementations was the Forest Covertype database, provided by Blackard, and donated to the UCI Repository. The experiments were run on Cerberus, a 40 node Beowulf cluster, connected by a fast Ethernet network. 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 through 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 (ARTMAP architectures require normalization of input values, so that they lie in the interval [0, 1]). Hence, we normalized the input values.

For testing the parallel efficiency of all three algorithms, patterns 1 through 256,000 were used for the training. Patterns 561,001 to 581, 000 (20,000 of them) were used for testing. The number of processors in the pipeline varied from $p=1$ to p=32, in powers of 2. For these runs, the primary concern was the speed-up of the pipelined NMT-FAM, NMT-GAM and NMT-EAM versus their sequential counterparts. Results of the speed-up for this database can be seen in Figures 2, 3, and 4. We observe from these figures that the speed-up is approaching linear. For small numbers of training patterns, the speed-up trails off for higher number of processors, creating a knee in the curve. This is because for smaller numbers of input patterns, there are fewer templates created and thus less processing to be done. With too many processors in the pipeline, these relatively small computational tasks are too finely split, resulting in too much communication versus computation. This effect is very commonly seen in parallel algorithms. The problems for the algorithm need to be large enough to justify parallelism. For some of the training set sizes, the speed-up is slightly above linear. This behavior is attributed to caching effects. When there are greater numbers of processors in the pipeline each processor has fewer templates. This means that more of the templates can remain cached for the competition loop versus a processor with more templates.

5. CONCLUSIONS

In this paper, we have extended the implementation of one of our no match tracking ARTMAP architectures (Fuzzy ARTMAP) to 2 other variations of the ARTMAP neural network (Ellipsoidal ARTMAP and Gaussian ARTMAP). These no match tracking variants allowed us to focus on the parallelization of the competition process in ARTMAP. We have showed that this parallel implementation of the FAM variant is theoretically sound (results were reported in Castro, et al., 2004, and omitted due to lack of space) and exhibits good workload balancing properties. We also showed experimentally (by working with the Covertype database) that this algorithm exhibited linear speed-up when the number of processors in the pipeline is increased for all these no-match tracking ART variations. We expect that the parallelization strategy introduced for the no-match tracking ART structures discussed in this paper can be readily extended to other classifiers, neural network based or not, that share commonalities with the ART classifiers. In particular, these are classifiers that rely on an exemplar structure to compress (group) their input patterns and on a competitive loop that chooses the exemplar that best matches the incoming input patterns.

ACKNOWLEDGEMENTS

Jimmy Secretan and Michael Georgiopoulos would like to acknowledge the partial support of the NSF CRCD grant, no: 0203446. Georgios Anagnostopoulos and Michael Georgiopoulos would also like to acknowledge the partial support of the NSF CCLI grant, no: 0341601.

REFERENCES

Anagnostopoulos G., and Georgiopoulos, M., "Ellipsoid ART and ARTMAP for incremental clustering and classification," *IEEE-INNS International Joint Conference on Neural Networks 2001 (IJCNN 2001)*, Washington, DC, July 14-19, 2001, pp. 1221-1226.

Anagnostopoulos, G. C., and Georgiopoulos, M., "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, 2003.

Carpenter, G. A., Grossberg, S. Markuzon, N., Reynolds, J. H., Rosen, D. B., "Fuzzy ARTMAP: A neural network architecture for incremental learning of analog multi-dimensional maps," *IEEE Transactions on Neural Networks*, Vol. 3, No. 5. pp. 698-713, 1992.

Castro, J., "Modifications of the Fuzzy-ARTMAP Algorithms for Distributed Learning in Large Data Sets," PhD. Dissertation, University of Central Florida, Summer 2004.

Malkani, A. and Vassiliadis, C.A., "Paralllel implementation of the Fuzzy ARTMAP neural network paradigm on a hypercube," *Expert Systems*, Vol. 12, No. 1, 1995, pp 39-53.

Manolakos, E. S., "Parallel Implementation of ART1 neural networks on Processor Ring Architectures", in Parallel Architectures for Artificial Neural Networks, editors N. Sundararajan and P. Saratchandran, IEEE Computer Society Press, 1998.

Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multi-dimensional maps," *IEEE Transactions on Neural Networks*, Vol. 3, No. 5, 1992, pp. 698-713.

Grossberg, S., "Adaptive pattern recognition and universal recoding II: Feedback, expectation, olfaction, and illusions," *Biological Cybernetics*, Vol. 23, 1976, pp. 187-202.

Williamson, J. R., “Gaussian ARTMAP: A Neural Network for Fast Incremental Learning of Noisy Multi-Dimensional Maps,” *Neural Networks*, Vol. 9, No. 5, 1996, pp. 881-897.

Williamson, J. R., “A constructive, incremental-learning network for mixture modeling and classification,” *Neural Computation*, Vol. 9, 1997, pp. 1517-1543.

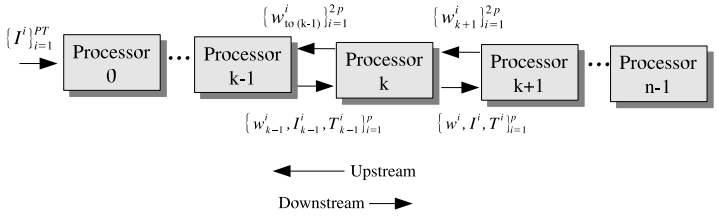

Figure 1. Processor pipeline architecture for NMT- ART.

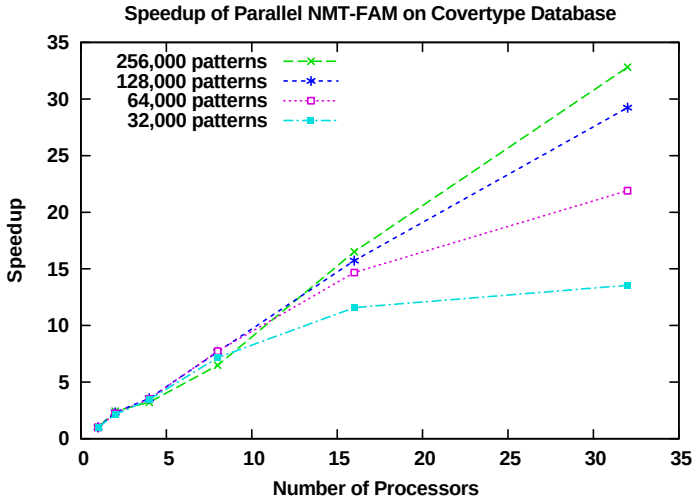

Figure 2. Forest Covertype Database Speed-Up Results for NMT-FAM

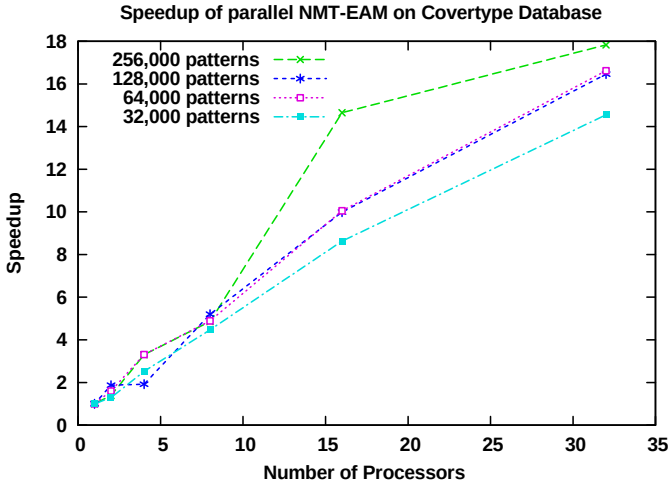

Figure 3. Forest Covertype Database Speed-Up Results for NMT-EAM

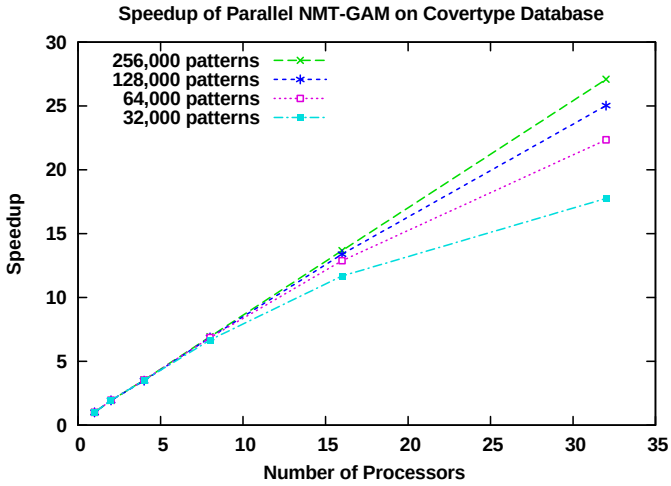

Figure 4. Forest Covertype Database Speed-Up Results for NMT-GAM

APPENDIX

Process $(k, n, \overline{\rho}_a, \beta_a, p)$

1 INIT (p)

2 while *continue*

3 do

4 while $|myTemplates| > myShare$

5 do

```
6        EXTRACT-TEMPLATE (myTemplates, {w^i_to(k−1)})
7        SEND-NEXT (k, n, {(w^i, I^i, T^i): i = 1,…, p})
8        RECV-NEXT (k, n, {w^i_{k+1} : i = 1,…,2p}, newNodes_{k+1})
9        SEND-NEXT (k, {w^i_to(k−1) : i = 1,…,2p}, newNodes)
10       RECV-NEXT (k, {(w^i_(k−1), I^i_{k−1}, T^i_{k−1}): i = 1,…, p})
11       newNodes ← newNodes_{k+1}
12       S ← {w^i_{k+1}}
13        for each i in {1,2,…, p}
14       do WINNER (I^i, w^j, T^i, ρ̄_a, β_a, S)
15         myTemplates ← myTemplates ∪ S
16         if I^i_{k−1} = EOF
17            then continue ← FALSE
18            else S ← {w^i_to(k−1)}
19                 for each i in {1,2,…, p}
20                 do WINNER (I^i_{k−1}, w^i_{k−1}, T^i_{k−1}, ρ_a, β_a, S)
21                      (I^i, w^j, T^i) ← (I^i_{k−1}, w^j_{k−1}, T^i_{k−1})
22                 for each i in {1,2,…, p}
23                 do WINNER (I^i, w^j, T^i, ρ̄_a, β_a, myTemplates)
24                      if k = n − 1
25                        then if class(I^i) = class(w^j)
26                                then
27                                   myTemplates ← myTemplates ∪ {I^i ∧ w^j}
28                                else newTemplate ← I^i
29                                  index(newTemplate) ← newNodes + nodes
30                                   myTemplates ← myTemplates ∪ {I^i, w^j}
31                                   newNodes ← newNodes + 1
32                 if newNodes > 0
33                   then
34                     nodes ← nodes + newNodes
35                     myShare ← ⌈ nodes / n ⌉
36       SEND-NEXT (k, n, {(none, none, 0)})
37       RECV-NEXT (k, n, {w^i_{k+1} : i = 1,…,2p}, newNodes_{k+1})
38       myTemplates ← myTemplates ∪ {w^i_{k+1} : i = 1,…,2p}
```