

Methods for Parallelizing the Probabilistic Neural Network on a Beowulf Cluster Computer

Jimmy Secretan, Michael Georgiopoulos, Ian Maidhof, Philip Shibly and Joshua Hecker

Abstract—In this paper, we present three different methods for implementing the Probabilistic Neural Network on a Beowulf cluster computer. The three methods, Parallel Full Training Set (PFT-PNN), Parallel Split Training Set (PST-PNN) and the Pipelined PNN (PPNN) all present different performance tradeoffs for different applications. We present implementations for all three architectures that are fully equivalent to the serial version and analyze the tradeoffs governing their potential use in actual engineering applications. Finally we provide performance results for all three methods on a Beowulf cluster.

I. INTRODUCTION

The Probabilistic Neural Network (PNN), introduced by Specht [1], is an effective neural network architecture, derived from a sound theoretical framework. It can effectively solve a variety of classification problems. The PNN is an approximation of the well known Bayesian classifier. The Bayesian classifier is the best classifier that can be designed to solve any classification problem. However, Bayesian classification has the drawback that it requires knowledge of the class conditional probabilities of the data involved. In his paper, Specht uses the available data to estimate these class conditional probabilities. This approximation is based on the work by Parzen, who in his 1965 paper [2], provided a formula for the calculation of these class conditional probabilities.

Like many neural network algorithms, the PNN has a training phase and a performance phase. PNN's training phase has an obvious computational advantage compared to other neural network classification algorithms. In its training phase PNN needs to simply store all of the training data, which takes virtually no time, and quite often it is referred to as one-pass learning.

The inexpensive training phase of the PNN comes at the expense of a much more computationally intensive performance phase. In PNN's performance phase, in order for one to predict the label of a datum, whose label is unknown, some form of distance of this datum needs to be calculated to every data-point belonging to the training set. As a result, the number of computations required to produce

predictions for the labels of new data (belonging to a test set) is proportional to the number of points contained in the test set, the number of points contained in the training set (stored during PNN's training phase) and the dimensionality of the data. Consequently, the computational complexity associated with the label predictions of the data in the test set becomes high quickly, especially for problems of high dimensional data with significant training and testing set sizes.

To remedy this computational expense associated with the PNN's performance phase, researchers have offered a variety of solutions. One of these solutions relies on first clustering the training data prior to using them in PNN's training phase. Clustering reduces the number of training data-points that one has to deal with because it replaces groups of the original set of training points by their representatives. For instance, in [3], the author utilized Learning Vector Quantization (LVQ) clustering for grouping the training data. Alternatively, in [4], the authors accomplished data clustering by using a Self Organizing Maps (SOM) algorithm. Finally, the authors in [5] make a comparison of three methods (LVQ, General Grouping Method, and the Reciprocal Neighbors method) for reducing the training set size. It is worth pointing out that in [6], the focus is shifted from reducing the cardinality of the training set to reducing its dimensionality. The author proposes an iterative feature reduction algorithm to be used, whereby features that are deemed to be excessively noisy or contributing little to the final classifications are removed. However, the reduction of the dimensionality of the training set is not expected to have as much of an effect on the PNN's computational complexity as the reduction of the set's cardinality through clustering.

Whichever approach or approaches are followed for the reduction of PNN's computational complexity in the performance phase, one cannot avoid classification problems where the size of the training dataset (even after clustering) is large, or the dimensionality of the data is high (because all dimensions are important). Furthermore clustering of the training data and reduction of data dimensionality does not reduce the complexity related with the number of data points in the test set. The situation of dealing with a large size test set for which label predictions are needed in real-time or almost real-time mode is still grim when the PNN algorithm needs to be engaged to produce these predictions. In this

This work was supported in part by a National Science Foundation (NSF) grant CRCD: 0203446, National Science Foundation grant DUE:05254209, and a National Science Foundation Graduate Research Fellowship,

The authors are with the University of Central Florida, Orlando, FL 32816 USA (407-823-5338). All correspondence should be sent to Michael Georgiopoulos at michaelg@mail.ucf.edu.

case, PNN hardware implementations that speed-up its performance phase are usually employed. In [7], the authors design an FPGA-based system for processing satellite imagery with the PNN. The PNN was then able to outperform standard PC implementations by about an order of magnitude.

Due the popularity of inexpensive, coarse-grained parallel architectures such as Beowulf clusters, and computing grids, there has recently been a research emphasis into using this architectures for high performance, high volume data mining. This is the approach that we follow in this paper. All the methods that we propose, to parallelize the PNN's performance phase, are implemented on a Beowulf, coarse-grained parallel architecture.

It is worth mentioning that in [8], pipelined approaches to implement the training phases of several neural network architectures including, Fuzzy ARTMAP, Gaussian ARTMAP and Ellipsoidal ARTMAP were presented. Pipelining was used to reduce communication for optimal performance on high latency, low bandwidth interconnects. These designs provide a basis for the Pipeline PNN methods presented in this paper. Related work that has appeared in the literature is the effort conducted in [9] where the author provides code for a Beowulf parallel image processing system using the algorithm that we call in this paper PFT-PNN. In this paper, in addition to the PFT-PNN implementation of the PNN on the Beowulf architecture, we implement two other parallelization methods, and we discuss their advantages and disadvantages.

More specifically, in Section II we briefly discuss the PNN algorithm. In Section III we discuss, in detail, all the proposed PNN parallelization methods. In Section IV we compare experimentally these three parallelization methods and emphasize their advantages and disadvantages. Finally, in Section V we summarize our work and provide conclusive remarks and directions for further research.

II. THE PNN ALGORITHM

The PNN approximates the Bayesian Classifier. From Bayes' Theorem, we have that the a-posteriori probability that an observed datum \mathbf{x} has come from class j is given by the formula:

$$P(c_j | \mathbf{x}) = (p(\mathbf{x} | c_j)P(c_j)) / p(\mathbf{x})$$

In order to make effective use of this formula, we must calculate the a-priori probabilities $P(c_j)$ and the class conditional probabilities $p(\mathbf{x} | c_j)$. The a-priori probability can be obtained directly from the training data, that is, $P(c_j) = PT_j / \sum PT_j$, where PT_j designates the number of points in the training data set that are of class j . The class conditional probability can be estimated by using the approximation formula provided in Parzen [2], that is $p(x | c_j) =$

$$\frac{1}{(2\pi)^{D/2} \sigma^D PT_j} \sum_{r=1}^{PT_j} \exp \left[-\frac{(\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j)}{2\sigma^2} \right]$$

where D is the dimensionality of the input patterns (data), PT_j represents the number of training patterns belonging to

class j , \mathbf{X}_r^j denotes the r -th such pattern, \mathbf{x} is the input pattern to be classified, and σ is the smoothing parameter. The PNN algorithm identifies the input pattern \mathbf{x} as belonging to the class that maximizes the above probability (for classification problems where the data are equally likely of belonging to any of the potential classes). The choice of the right σ parameter is beyond the scope of our work, and the assumption is made here that somehow the proper σ parameter been chosen. If the σ parameters are different per dimension and class then we make a reference to a matrix of sigma values, denoted as σ_{\cdot} .

Under the assumption that σ has been appropriately chosen we are only concerned of loading the training data into memory, prior to the initiation of the PNN's performance phase. Once the loading is complete, a set of class conditional probabilities (one for each class) is computed for each testing point. The label of the testing point is determined by the highest class conditional probability (under the assumption that the a-priori probabilities for the different classes are equal). If the classes are not equally probable the label of the testing point is the one maximizing the product of the a-priori probabilities and class conditional probabilities. Figure 1 shows the pseudo-code of the standard PNN algorithm. We first iterate over the testing points. Then, iterating over each training point, we sum the formula across the dimensions. After each training point, we add this summation to the **classProbabilities** array. For a particular testing point, once the training points have been iterated over, the **classProbabilities** has C entries, representing partially formed class conditional probabilities for each class. The program then loops over this array and performs the rest of this computation (the division by the constant terms is only done once, at the time when the final summation is calculated). After this, the **FindArrayMax** function returns the class for which this class-conditional probability is the maximum. This is added to the **results** array.

III. PARALLEL PNN ALGORITHMS

In the course of parallelizing the performance phase of the PNN, we focused on three different parallelization methods. They all have different strengths and weaknesses. Before providing the pseudo-code for these approaches, we introduce some necessary terminology (see Table I).

A. Parallel Full Training Set PNN (PFT-PNN)

The first approach may be the most obvious approach. It is called the Parallel Full Training Set PNN, because each

```

Procedure: SerialPNN( $T, \mathbf{X}, D, \sigma, \mathbf{PT}$ )
real array  $classProbabilities$ ;
real  $exponent$ ;
integer array  $results$ ;
foreach  $x_k$  in  $T$  do
  foreach  $X_r$  in  $X$  do
     $exponent = 0$ ;
    for  $i = 1$  to  $D$  do
       $exponent += (x_{ik} - X_{ir})^2 / \sigma_{ij}$ ;
       $classProbabilities_j += \exp(exponent)$ ;
    foreach  $j$  in  $classProbabilities$  do
       $classProbabilities_j /= (2\pi)^{D/2} PT_j \prod_{i=1}^D \sigma_{ij}$ ;
     $results_i = \text{FindArrayMax}(classProbabilities)$ ;
  return  $results$ ;

```

Fig. 1. Pseudo-code for the standard serial PNN algorithm

TABLE I
VARIABLES FOR PSEUDOCODE

Statement	Explanation
C	Number of output class labels in the problem.
D	Dimensionality of the data.
σ	$C \times D$, Matrix of sigma values. In our experiments all the entries of this matrix are the same, and equal to a fixed smoothing parameter σ .
PT	An array, indexed by class j , containing the number of points in the training set that belong to a particular class.
T	Set of testing points.
X	Set of training points.
k	The processor id on which the program is currently running, with $k = 0$ being the root node.
p	Total number of processors on which the program is currently running.
$nodes$	A set (of size p) containing all of the computational nodes that are part of the currently running system.
B	Number of points in a batch. When PST-PNN and PPNN nodes transfer testing points, they do so in sets of size B .
$batches$	The number of batches of size B into which the testing set is split.
$results$	An array of output class labels, resulting from the completion of the PNN test phase.
$classProbabilities$	An array of size C that contains the computed class conditional probabilities for a particular point.
PR	An array containing $classProbabilities$ arrays for multiple testing points.

node k of the p total nodes contains a copy of the entire training set. Suppose there is a test set that needs to be processed by the PNN. The test set is fully available at run time. Results for this test set are not needed as they are available, but are acceptable to be received all at once. The primary goal is to finish processing the entire test set as quickly as possible. Also assume that the training set fits easily into the main memory of a single node. If all of these requirements are met, it is possible to use the PFT-PNN. In the PFT-PNN, after the full training set is broadcast to the nodes, the test set is then split up equally among the nodes.

Note that this approach could be easily extended to a grid environment with heterogeneous nodes. The $calcPtsPerNode()$ function that is responsible for calculating the number of test points to be sent to the node could easily be modified to send a number of points to each node that is commensurate with its computational capability. In parallel, the nodes evaluate their portion of the test set and send the results back to the master node. Figure 2 illustrates the flow of training and testing points for the PFT-PNN.

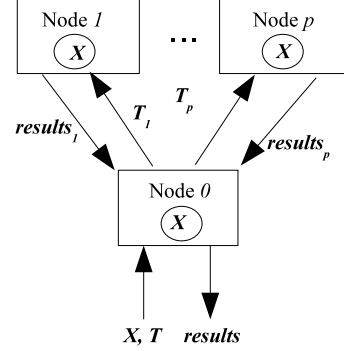


Fig. 2. Diagram for data flow in the PFT-PNN.

The pseudo-code for the PFT-PNN approach is given in figure 3. At the root node, the entire training set is loaded into X by the $LoadTrainingSet()$ function. The function $BroadcastSet()$ takes a set as an argument, broadcasting it from the root node, and receiving that set on all the other nodes. This is a tree structured communication, relying on all of the nodes to quickly propagate the information. Once the training set has been received, then the root splits up the testing set and sends it out to the nodes. The $calcPtsPerNode()$ functions computes the number of testing points that a particular node should have. The $SendSet()$ function sends to the specified node a specified set. The set it sends, in this case, is the one returned by $LoadTestSetPortion()$ which returns the specified number of testing set points. After this, the other nodes receive this set. Each node computes the standard Serial PNN and stores it in a local array. Finally, the $GatherResults()$ function executes, sending the results from the non-root nodes and receiving and merging the results at the root node.

This is fully equivalent to the serial PNN. A serial PNN algorithm will be run by node k , on the set T_k . Because

$$\bigcup_{k=0}^{p-1} T_k = T, \text{ this is equivalent to executing the serial PNN}$$

on the set T .

We now calculate the computation and communication complexity of the PFT-PNN architecture. First, the training set must be loaded and broadcast to all of the processor nodes, taking $O(D|X|)$ time. We assume that the broadcast operation is tree structured and therefore it takes

```

Procedure: PFTPNN( $k$ , nodes,  $\mathbf{T}$ ,  $\mathbf{X}$ ,  $D$ ,  $\sigma$ ,  $\mathbf{PT}$ )
Set  $\mathbf{T}_k$ ;
integer array results;
integer pointsPerNode;
if  $k == 0$  then
|  $\mathbf{X} = \text{LoadTrainingSet}()$ ;

BroadcastSet( $\mathbf{X}$ );
if  $k == 0$  then
| foreach nodesi in nodes do
|   pointsPerNode = calcPtsPerNode(nodesi);
|   SendSet(nodesi,
|   LoadTestSetPortion(pointsPerNode));
else
| nodeTestingPoints = ReceiveSet();
| resultsi = SerialPNN( $\mathbf{T}_k$ ,  $\mathbf{X}$ ,  $D$ ,  $\sigma$ ,  $\mathbf{PT}$ );
| GatherResults(results, resultsi);

```

Fig. 3. Pseudo-code for the PFT-PNN parallel algorithm.

$O(\log(D|\mathbf{X}|))$ operations. Then we must send to each node its portion of the test set. This operation is simply $O(D|\mathbf{T}|)$. After the test set has been broadcast, the serial PNN is executed in the standard way, requiring $O((D|\mathbf{X}||\mathbf{T}|C)/p)$ operations. Finally, the results are gathered back to the master node, which requires $O(|\mathbf{T}|)$ operations, because only a single label needs to be broadcast back for each test point. This leads to a cumulative complexity of $O(D|X| + (D|\mathbf{X}||\mathbf{T}|C/p) + \log(D|\mathbf{X}|) + D|\mathbf{T}| + |\mathbf{T}|)$ operations for the PTF-PPN approach. For storage requirements, each node must store the training set and a portion of the testing set, with a storage complexity for each node of $O(D|\mathbf{X}| + D|\mathbf{T}|/p)$.

B. Parallel Split Training Set PNN (PST-PNN)

In this approach, the training set is initially split among the processor nodes, hence the name Parallel Split Training Set PNN. If the training set is of large size, this approach allows for the training set to be more easily stored on a single processing node. With training sets that are large relative to the node memory, the node may not be able to fit the full set or may have to do a great deal of swapping to the hard disk, slowing the computations significantly. In batches of size B , test points are broadcast to all of the nodes. This packet size B can be adjusted to modify the system's latency and to accommodate the availability of data. For instance, this may be used in a system where data arrives in bursts or as single points. This method offers low latency for obtaining the label of a few test points. The flow of data for the PST-PNN is pictorially illustrated in figure 4.

In the PST-PNN pseudo-code in figure 4, the root node

```

Procedure: PSTPNN( $k$ , nodes,  $B$ ,  $\mathbf{T}$ ,  $\mathbf{X}$ ,  $D$ ,  $\sigma$ ,  $\mathbf{PT}$ )

Set testBatch;
Set  $\mathbf{X}_k$ ;
integer array results;
real array PR;
integer batches;
integer pointsPerNode;
if  $k == 0$  then
| foreach nodesi in nodes do
|   pointsPerNode = calcPtsPerNode(nodesi);
|   SendSet(nodesi,
|   LoadTrSetPortion(pointsPerNode));
else
|  $\mathbf{X}_k = \text{ReceiveSet}()$ ;
| batches =  $\lceil |\mathbf{T}|/B \rceil$ ;
for  $i = 1$  to batches do
| if  $k == 0$  then
|   | testBatch = LoadTestSetPortion( $B$ );
|   BroadcastSet(testBatch);
|    $PR_i = \text{PartialPNN}(\text{testBatch}, \mathbf{X}, D, \sigma, \mathbf{PT})$ ;
|   ReduceResults(results,  $PR_i$ );

```

Fig. 4. Pseudo-code for the PST-PNN parallel algorithm.

starts by calculating the number of training points that each node will have (similar to the *calcPtsPerNode()* function mentioned previously). The *LoadTrSetPortion()* function loads the specified number of points of the training set. The *SendSet()* function then sends this subset to other nodes. The root node loads a batch of the testing set, and broadcasts this to all nodes. Each node computes the function *PartialPNN()* which operates similarly to the *SerialPNN()* function except it stops short of making the final classification. Instead it returns a set of *classProbabilities* arrays in PR_i . Finally, the *ReduceResults* function executes to send the partial results from the non-root node and receive and merge the results at the root node.

This is fully equivalent to the serial PNN. Every node receives an identical batch of testing points. For each testing point in the set, on every node k , the following is calculated:

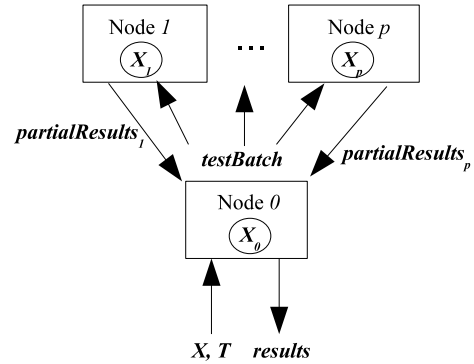


Fig. 5. Diagram for the data flow in PST-PNN.

$\sum_{r=1}^{PT_j^k} \exp((\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j) / (2\sigma^2))$ where PT_j^k is the set of training points in class j that are stored on node k . These class probabilities are summed together when they are returned to the root node. This yields

$$\sum_{k=0}^{p-1} \sum_{r=1}^{PT_j^k} \exp((\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j) / (2\sigma^2)).$$

Because we split the training set fully among the nodes, for every class j

$$\bigcup_{k=0}^{p-1} PT_j^k = PT_j.$$

Multiplying this by

$1 / ((2\pi)^{D/2} \sigma^D PT_j)$ then yields the original PNN formula. The root node makes the decision based on this formula, giving the same results as the serial PNN.

For the time complexity of the PST-PNN, we first start with the splitting of the training set among all of the processor nodes. This is an $O(D|\mathbf{X}|)$ operation. Then, we have the broadcast, reduce loop that is executed *batches* times. Over the course of this loop's execution, the loading of testing set values will account for $O(D|\mathbf{T}|)$ operations. There will be a broadcast for each batch, totaling $O(batches(\log(BD)))$. Then, there is the partial PNN execution, with complexity $O(D|\mathbf{T}||\mathbf{X}|/p)$. Finally, there is the reduction that takes $O((\log(CB) + CB)batches)$ operations because it is a tree structured operation, taking the partial result arrays of the nodes and turning it into a final result. After the tree structured gather, there is a final search for the maximum probability. All of these calculations yield a complexity of $O(D|\mathbf{X}| + D|\mathbf{T}| + batches(\log(BD)) + D|\mathbf{T}||\mathbf{X}|/p + (CB)batches)$ operations for the PST-PNN approach. Because each node only has to store a part of the training set, and batches of the testing set, the storage complexity per node is only $O(D|\mathbf{X}|/p + DB)$.

C. Pipelined PNN (PPNN)

In this architecture, communication time between nodes is minimized by allowing each node to communicate with its 2 neighbors. First of all, as in the PST-PNN, the training set is split evenly among the nodes. Test points are loaded at the first node in the pipeline. For each batch B , a partial list of class conditional probabilities is calculated. This partial list and the test point batch are passed on to the next node in the pipeline that does the same for the batch with its set of training points. A label is finally assigned at the end of the pipeline. This requires only point to point communication as opposed to the switched communication of the other two methods. It is worth noting that this approach is similar to the approach used for the parallel implementation of Fuzzy

ARTMAP and the no match tracking (NMT) Fuzzy ARTMAP (see [8]). Figure 6 shows the logical layout of the pipeline and the data flow between nodes.

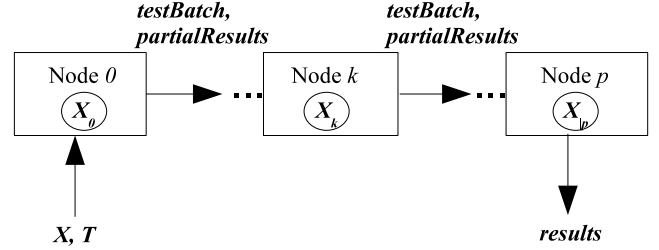


Fig. 6. Diagram for data flow in the Pipeline PNN.

The pseudo-code for the pipeline PNN is provided in figure 7. The first part of the Pipeline PNN works identically to the PST-PNN. The loop, however, is different. First, the *testBatch* is loaded by the root node. All of the other nodes receive a test batch and their partially computed results from the neighbor with id $k-1$ through the *RecvPrev()* function. As in the PST-PNN, a partial *classProbabilities* array is computed based on the current testing batch and the node's available training points. The partial results returned are merged with the partial results received from the previous node. Finally, if the node is not the last in the pipeline, these are sent on to the neighbor with id $k+1$. If this is the last node in the pipeline, a final classification is made from the now complete class conditional probability array with the *calcFinalResults()* function.

This is computationally equivalent to the serial PNN. At each node k (stage in the pipeline), the partial results of a particular input point \mathbf{x} in batch i is given by

$$PR(k, i) = \sum_{r=1}^{PT_j^k} \exp((\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j) / (2\sigma^2)) + P(k-1, i),$$

with $P(0, i, r) = 0$, and PT_j^k being the set of training points in class j that are stored on node k . Finally,

$$P(p, i) = \sum_{k=0}^{p-1} \sum_{r=1}^{PT_j^k} \exp((\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j) / (2\sigma^2)),$$

reflecting the fact that we add the partial results at every stage. Because we split the training set fully among the

nodes, then for every j , $\bigcup_{k=0}^{p-1} PT_j^k = PT_j$. This allows the

partial results formula to be rewritten as

$$P(p, i) = \sum_{r=1}^{PT_j} \exp((\mathbf{x} - \mathbf{X}_r^j)^T (\mathbf{x} - \mathbf{X}_r^j) / (2\sigma^2)).$$

Multiplying this by $1 / ((2\pi)^{D/2} \sigma^D PT_j)$ gives the original formula on which the classification decision is made. This same classification decision is made on the last node, $p-1$.

```

Procedure: PipePNN( $k, \text{nodes}, B, \mathbf{T}, \mathbf{X}, D, \sigma, \mathbf{PT}$ )
Set testBatch;
Set  $\mathbf{X}_k$ ;
integer array results;
real array PR;
integer batches;
integer pointsPerNode;
if  $k == 0$  then
  foreach nodesi in nodes do
    pointsPerNode = calcPtsPerNode(nodesi);
    SendSet(nodesi,
    LoadTrSetPortion(pointsPerNode));
  else
     $\mathbf{X}_k$  = ReceiveSet();
    batches =  $\lceil |\mathbf{T}|/B \rceil$ ;
    for  $i = 1$  to batches do
      if  $k == 0$  then
        | testBatch = LoadTestSetPortion(B);
      else
        | RecvPrev(PRi, testBatch);
        | PRi += PartialPNN(testBatch,  $\mathbf{X}, D, \sigma, \mathbf{PT}$ );
      if  $k == |\text{nodes}| - 1$  then
        | resultsi = calcFinalResults(PRi);
      else
        | SendNext(PRi, testBatch);

```

Fig. 7. Pseudo-code for the pipeline PNN algorithm

For the Pipeline PNN time complexity, we first start, as before with the splitting of the training set among all of the processor nodes, which is $O(D|\mathbf{X}|)$. Each node executes its loop *batches* times. However, the last node waits $p-1$ additional timesteps for the first computation to reach it. The communication at each cycle will take $O(BD)$. Then, there is the partial PNN execution, with complexity $O(DB|\mathbf{X}|/p)$ on each processor, each cycle. There is also a final labeling of the batch, taking $O(BC)$. All of these add up to yield a computational complexity of $O(D|\mathbf{X}| + (\text{batches} + p - 1)(BD + DB|\mathbf{X}|/p + BC))$ operations for the pipeline approach. As in PST-PNN, the storage complexity per node is $O(D|\mathbf{X}|/p + DB)$.

IV. EXPERIMENTS

A. Hardware/Software

To experiment with the different parallel PNN approaches, an implementation for each approach was done in C++ using the MPICH libraries for message passing. A serial version of the algorithm was also implemented.

All of versions of the PNN were run on the Cerberus

Beowulf cluster at the University of Central Florida. Cerberus has 40 nodes, each with a 400Mhz Pentium II processor and 384MB of RAM. They are connected together by a fast Ethernet switch and managed by a combination frontend/fileserver.

B. Experimental Database

The database used in the testing of the three different parallelization PNN approaches, mentioned above, consisted of a 2-class, 16-dimensional Gaussian data with 15% overlap between the data belonging to the two different classes. The meaning of the 15% overlap is that if we were to design a Bayesian classifier to separate the 2-class data in this dataset the error rate of the Bayesian classifier would be 15%. Two datasets were generated, to be alternately used for training and testing; one of size 512,000 data-points and the other of size 16,000 data-points. Although this was an artificially generated database it is sufficient for the purposes of this paper. In this paper we are primarily concerned on comparing the speed of the three different PNN parallelization approaches, and because of that and the fact that all of these are equivalent to the serial algorithm, the actual dataset that we use to perform our experiments is not important. What is important is the dimensionality of the dataset and the size of the training/testing sets.

C. Test Description

In the first test, 16k training points were used, while the test set contained 32k, 128k and 512k points, respectively. Then, the size of the training set was kept fixed at 16k, while the size of the training set varied from 32k, to 128k and finally to 512k. Each of these tests was done for 1, 2, 4, 8, 16 and 32 number of processor nodes.

For the second set of tests, we examined the latency of processing a single test data-point using the PST-PNN and the Pipelined PNN (after initial set up and loading of the training data was completed). In this second set of tests we used a training set size of 512k points. The PFT-PNN approach was omitted from these tests because the single test point can never be split among the nodes. The number of nodes in this second set of tests was 1, 2, 4, 8, 16, 32.

D. Parameters Chosen

A nominal sigma (σ) value of 0.5 over all dimensions and classes was chosen, because it worked well in the initial experimentation with the Gaussian dataset. Because this paper only focuses on the computational efficiency of the PNN and not on its classification performance and because the 3 implementations are computationally equivalent to the serial PNN, no effort was expended to optimize sigma. Values of B that worked well were established by experimentation (20 for the PST-PNN approach and 5 for the Pipeline PNN approach).

V. RESULTS

The results obtained from the parallel runs were mostly expected. For the PFT-PNN (figure 8), the scaling was

mostly related to the product, $|\mathbf{T}\|\mathbf{X}|$ (for a fixed D). The higher this product was, the better the algorithm scaled for large numbers of processors. This is not surprising, as the large computational tasks make up for the overhead of transmitting the training and partial testing sets.

In the graph of the PST-PNN, figure 9, some interesting conclusions can be drawn. As with the PFT-PNN, there is clearly some dependence between the linearity of the scaling and the size of the product $|\mathbf{T}\|\mathbf{X}|$. However, there is a clear advantage to smaller testing sets and larger training sets. This can be presumably attributed to two effects. First, there are fewer communications to split the training set among the nodes than the numerous separate communications to broadcast batches of testing points and receive their results. On the high latency, low bandwidth interconnect typical to clusters, less frequent communication is favored. Another effect is the role of the CPU's cache in the computation. In the single processor case, for each testing point, the algorithm must iterate over all of the training points. This is typically significantly larger than the cache, and the training points are swapped frequently in and out. However, as the training points are split into smaller pieces shared among the nodes, it is possible to keep many of these in cache.

This allows additional speedup over the single processor, in some cases outweighing the overhead of the parallel algorithm and leading to super-linear scaling. An effect similar to that of the PST-PNN is seen for the Pipeline PNN in figure 10. The splitting of the training set allows some of the same caching benefits.

For products $|\mathbf{T}\|\mathbf{X}|$ that are too small, larger pipelines make less sense in terms of scaling, because the overhead of starting up the pipeline becomes too great.

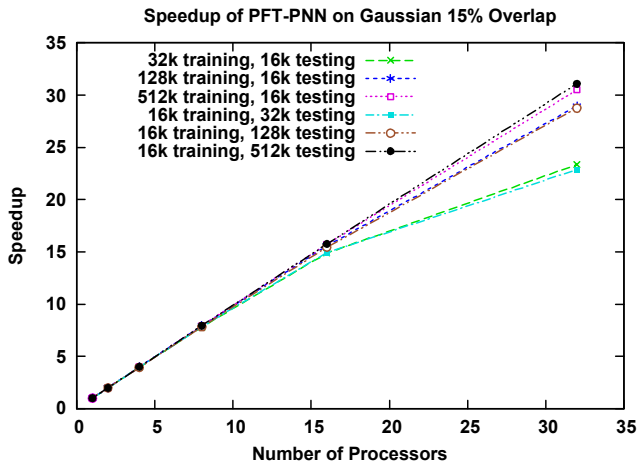


Fig. 8. Speedup of PFT-PNN for 15% overlap Gaussian database.

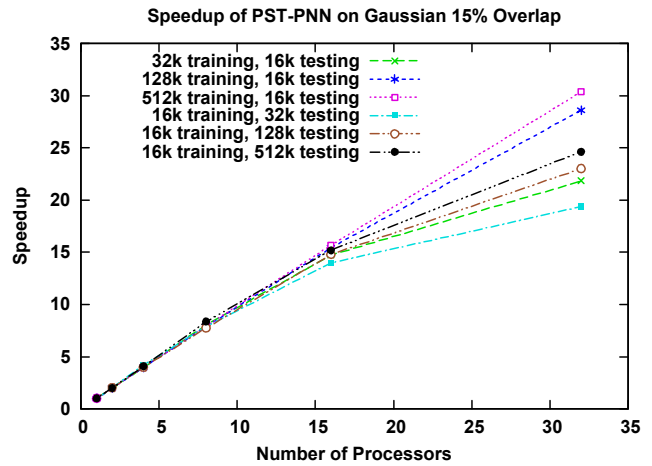


Fig. 9. Speedup of PST-PNN for 15% overlap Gaussian database.

In figure 11, we compare all three implementations in scenarios where they all have 512k training points and 16k testing points, as well as 16k training points and 512 testing points.

For the 512k training point scenario, all three architectures appeared to perform similarly. However, their performances diverge for the scenario of 16k training points and 512k testing points. The pipeline PNN has the greatest advantage, because of its efficient communication and, in our case, greater ability to retain frequently used training points in cache. The PFT-PNN performs well also, which is not surprising because of infrequent communication requirements. However, it becomes evident that the frequent broadcasts and gather operations affect the scaling of the PST-PNN approach. The large number of testing points used in this experiment makes this effect more evident.

Figure 12 shows the latency of testing a single point with 512k training points. The pipeline PNN and PST-PNN are shown. The PFT-PNN, as specified before, is omitted, because running a single point on the PFT-PNN is equivalent to executing the serial version of the algorithm.

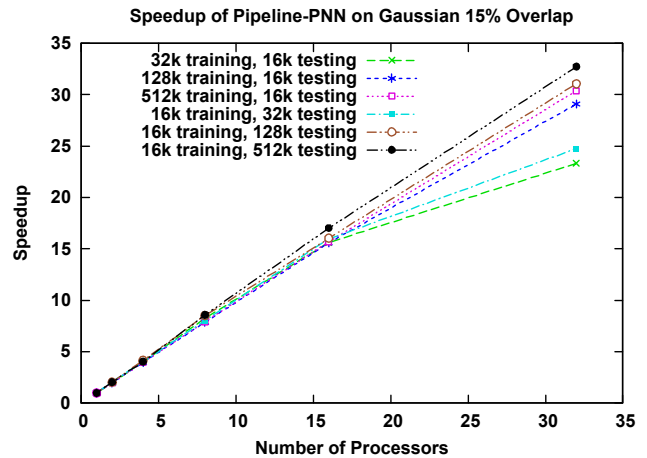


Fig. 10. Speedup of Pipeline-PNN for 15% overlap Gaussian database.

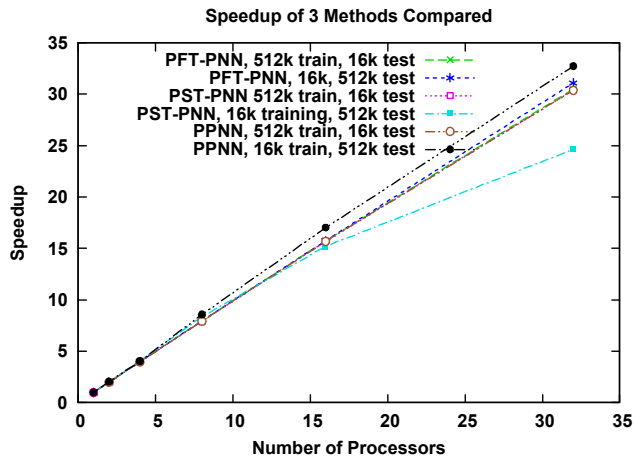


Fig. 11. Comparison of all three approaches for 512k training points and 16k testing points as well as 16k training points and 512k testing points. Note that, while all 6 cases are shown, the lines are closely overlaid.

As expected, we see that the latency drops with a greater number of processors in the PST-PNN, because comparison against portions of the training set happen concurrently. However, it is clear that this effect is diminishing as the overhead of broadcasting the points and merging the results begins to overshadow the speedup gained by the parallelism. In the pipeline PNN, the latency of processing a single point is approximately constant. The computational task of each processor becomes smaller, but these tasks are not done concurrently, thus eliminating the advantages of parallelism when a single test point is processed.

VI. SUMMARY/CONCLUSIONS

We have designed and implemented three different approaches for the parallelization of the PNN's performance phase. In the PFT-PNN approach each node has a complete copy of the training set and processes a portion of the testing set. In the PST-PNN approach each node has a portion of the training set and the entire testing set (received in batches) and merging of the results is required. In the PPNN approach

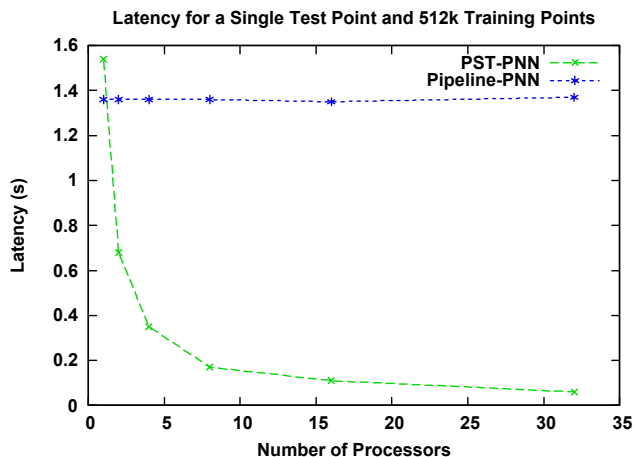


Fig. 12. Latency for a single testing point and 512k training points.

each node has a portion of the training set and processes the

testing set in a pipeline fashion.

All three of these approaches show reasonable scaling. In a production system utilizing PNN, a decision of which architecture to use is dependent on the problem at hand. If there are very few points to split up amongst the processor nodes, the overhead of any of the parallel algorithms will overshadow the speedup. If there are large batches of test points to process, with reasonably sized training sets, the PFT-PNN can provide efficient processing that is easy to control and implement. For training sets that are prohibitively large to fit in a single node's memory, either the PST-PNN or the Pipeline PNN may be more beneficial. If a very large training set is used with the PFT-PNN or serial approaches, the amount of memory swapping of the training data will significantly reduce the performance of the algorithm. Finally, if points are needed in a low latency fashion, the PST-PNN approach is the algorithm of choice.

Future work will include speed-up analysis of the proposed parallelization approaches on a more extensive set of databases. In addition, these algorithms will be extended to heterogeneous grid environments. The nature of the way the PNN processes points allows easy load balancing of the PNN on multiple heterogeneous nodes.

REFERENCES

- [1] D. F. Specht, "Probabilistic Neural Networks," *Neural Networks*, vol. 3, pp. 109-118, 1990.
- [2] E. Parzen, "On estimation of probability density function and mode," *Annals of Mathematical Statistics*, vol. 33, pp. 1065-1073, 1962.
- [3] P. Burrascano, "Learning vector quantization for the probabilistic neural network," *IEEE Transactions on Neural Networks*, vol. 2, pp. 458-461, 1991.
- [4] J. Chen, H. Li, S. Tang, and J. Sun, "A SOM-based probabilistic neural network for classification of ship noises," presented at Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference, 2002.
- [5] M. H. Hammond, C. J. Riedel, S. L. Rose-Pehrsson, and F. W. Williams, "Training set optimization methods for a probabilistic neural network," *Chemometrics and Intelligent Laboratory Systems*, vol. 71, pp. 73-78, 2004.
- [6] C.-Y. Tsai, "An iterative feature reduction algorithm for probabilistic neural networks," *Omega*, vol. 28, pp. 513-524, 2000.
- [7] M. A. Figueiredo and C. Gloster, "Implementation of a probabilistic neural network for multi-spectral image classification on an FPGA based custom computing machine," presented at Proceedings of the 5th Brazilian Symposium on Neural Networks, 1998.
- [8] J. Secretan, J. Castro, M. Georgiopoulos, J. Tapia, A. Chadha, B. Huber, and G. Anagnostopoulos, "Parallelizing the Fuzzy ARTMAP Algorithm on a Beowulf Cluster," presented at International Joint Conference on Neural Networks, Montreal, Quebec, Canada, 2005.
- [9] K. D. Underwood, W. Ligon, and R. Sass, "An Analysis of the Cost Effectiveness of an Adaptable Computing Cluster," *Cluster Computing*, vol. 7, pp. 357-371, 2004.