Proceedings of The National Conference
On Undergraduate Research (NCUR) 2006
The University of North Carolina at Asheville
Asheville, North Carolina
April 6 – 8, 2006

A Parallel Artificial Neural Network Implementation

Ian Wesley-Smith
Center for Computation and Technology
Louisiana State University
Baton Rouge, Louisiana, 70803

Faculty Adviser: Dr. Gabrielle Allen

Abstract

Traditional computational methods are highly structured and linear, properties which they derive from the digital nature of computers. These methods are highly effective at solving certain classes of problems: physics simulations, mathematical models, or the analysis of proteins. Classical computational methods are not effective at solving other problems, such as pattern recognition, adaptive learning, and spam filtering. Some biological systems, however, excel at the latter class of problems. For example, the human mind can quickly identify a face, even if it has changed heavily from the last time it was seen, while traditional computational systems are unable to accomplish facial recognition efficiently and accurately even if minor facial or environmental alterations occur. Attempts to create facsimiles of these biological systems electronically have resulted in the creation of artificial neural networks. Similar to their biological counterparts, artificial neural networks are massively parallel systems capable of learning and making generalizations. The inherent parallelism in the network allows for a distributed software implementation of the artificial neural network, causing the network to learn and operate in parallel, theoretically resulting in a performance improvement. This paper will address a parallel neural network implementation in Cactus, a high performance computing framework, the network's relative strengths and weaknesses, and conclude by considering future improvements to the system.

1. Introduction

The term 'Artificial Neural Network' (ANN) is not well defined, with many texts having varying definitions of it. Most texts, however, do agree that a system of simple computational nodes “networked” together constitutes a basic (and somewhat generic) definition of ANNs. The small computational nodes in an ANN are commonly referred to as neurons. For each input a neuron multiplies the input by its respective weight value then sums this value with the values of all the other inputs for this neuron, finally passing the result to an output function.

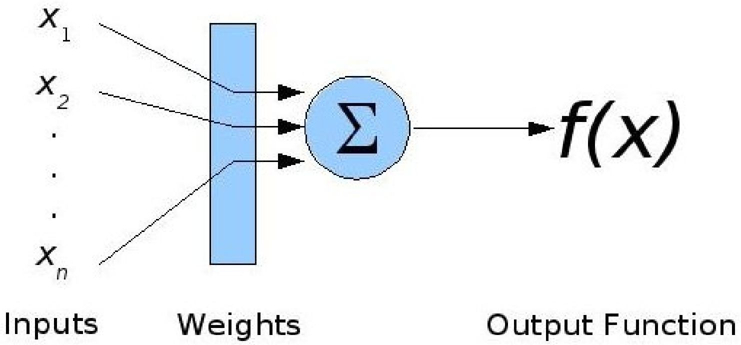

Figure 1. A Basic Neuron

Figure 1 An illustration of a basic neuron. Inputs (1-*N*) are multiplied by their respective weights, then summed and

passed to the output function.

The input is a vector (aptly named the input vector) consisting of scalar data that either originated from user input, or, in the case of multi-layer networks, previous layers of neurons. The weight is also a scalar vector, but it is actually the row vector of an M x N dimensional matrix, where M is the number of inputs and N is the number of neurons in a layer. This results in each neuron having a unique weight vector, and each input having a unique weight value per neuron. The output function can be a variety of functions, from a symmetric hard limit in simple binary classifiers to log-sigmoid, often utilized in more complex multi-layer networks. A perceptron, the simplest form of an ANN[1], is built with only one neuron and as such is only capable of classifying two linearly separable patterns. However, the number of neurons in a perceptron can be increased, in turn increasing the number of patterns that can be classified[1] (though they still must be linearly separable). Although perceptrons are capable of a wide variety of tasks, there are many more advanced ANNs that use multiple layers to analyze even more complex sets of data.

While the classification capabilities of ANNs are impressive, their most useful feature is their learning capability. ANNs can be taught in several different manners, with the simplest being a method known as learning with a teacher. In this method, we give the ANN a sample input and a target output. The ANN then compares the output it generates for the given input to the target output and adjusts its weights accordingly. This method is also known as generating an input-output mapping; we supply an input and tell the system our target output. After some number of iterations, the network reaches convergence, meaning it produces the correct output given the test input. Although learning with a teacher is sufficient for a variety of situations, it requires a substantial amount of knowledge regarding possible inputs. Another method, unsupervised learning, allows a neural network to modify its weights actively (during computation), allowing for networks capable of handling substantial variation with respect to input.

2. **Problem Description**

This paper examines the implementation of a parallel ANN. A parallel ANN has potential benefits for numerous applications. Applications with large training or data sets could process those at a substantial speed increase, while other applications could utilize multiple networks simultaneously to ensure accuracy of any output (such a method is known as a committee of networks). As mentioned earlier, ANNs have a high level of inherent parallelism. Neurons on each layer perform their calculations independently of each other. As such, layers can be split into discrete 'clusters' of neurons residing on the same physical processor. After each layer completes its calculations, all the clusters need to pass the updated weights, bias and output back to the simulation. The computations could then continue for the next layer (or take whatever action would be most appropriate). The difficulty in creating a parallel ANN implementation involves correctly utilizing the inherent parallelism in a network. The relative simplicity of most network calculations makes the parallelization of smaller networks impractical, the added overhead of cross processor communication relative to performance gains would result in a negligible net performance increase (if not resulting in a performance decrease). Additionally, more complex networks have even greater amounts of cross processor communication, further degrading net performance. However, despite these apparent obstacles, achieving performance increases with a parallel ANN should be possible (and has been achieved in the past)[2].

3. **Solution Methodology**

Initially, a serial ANN was written, then moved to a parallel implementation. In an attempt to keep the code simple but functional, a perceptron was used in the construction of these networks. Most calculations performed by neurons in an ANN can be expressed as vector operations, operations which are very common in parallel computing. This factor resulted in a high performance toolkit with extensive parallel vector and matrix support, PETSc[3], being utilized when moving the serial code to a parallel implementation. PETSc also has the added benefit of being integrable with Cactus[4], a general purpose, high performance computing framework. Cactus offers access to a wide variety of HPC tools, including parallel file I/O and high precision timers, some of which will be utilized in future research.

The main area where performance improvements are expected are within the neuron calculations. Each neuron performs these calculations, and, as mentioned in section 2, the calculations are independent of the calculations performed by the other neurons on its layer. Additionally, the simplicity of these calculations allow for an entire layer to be generalized in a single operation: taking the dot product of the weights column vector with the input vector and passing the result to the output function. This results in simplification of neuron calculations, moving from a nested loop to a single loop.

Initial Serial Pseudo Code For Calculations:

```
for(x=0;x<OUTPUTS;x++)
        for(y=0;y<INPUTS;y++)
                value+=weights[x][y]*inputs[y];
        output[x]=outfunc(value);
```

Parallel Pseudo Code For Calculations:

```
for(x=0;x<OUTPUTS;x++)
        value[x]=outfunc(weights[x] <DOT> input);
```

The learning functions of an ANN are somewhat harder to implement in parallel. While the calculations in a neuron involves simple operations, learning requires adjusting the weight matrix. In sequential code this task is trivial, all weight data is present on the processor. In the parallel implementation, however, the weight matrix is divided evenly over the all processors. Essentially, the parallel code must ensure that each processor is only modifying data it has. PETSc supplies several functions to determine the local data on a processor, greatly decreasing the difficulty of this task and requiring only minor modifications to the serial code.

4. **Results and Discussion**

To test the parallel ANN, a set of inputs with known outputs were created, and then processed by both the parallel and non-parallel ANN. Several test files were run, ranging in size from 10 up to 10 million entries. The test environment was a Sun workstation with two AMD Opteron processors (essentially a shared memory environment) and 3 GBs of memory. Mpich-1.2.7 was used in conjunction with PETSc 2.3.1 on Fedora Core 3 x86_64, and all code was compiled with GCC 3.4.2 and glibc 2.3.3. There were several different expectations for the parallel ANN. First, the parallel implementation should always return the same result as the single processor implementation. Second, the performance of the parallel ANN compared to the non-parallel ANN should vary based upon the number of inputs in a given network and the number of entries in a test file, with test inputs having the larger number of inputs resulting in better performance of the parallel ANN. Similarly, test files with a larger number of entries should perform better than test files with a smaller number of entries[5]. As expected, the parallel ANN produced the correct output for the given input. While some performance degradation was expected due to MPI overhead, it was theorized that a sufficiently large number of inputs or entries in a test file would mitigate this effect, resulting in a net performance gain. This was not the case. The parallel ANN performed somewhat worse than the single processor network in all the tests, although the actual performance difference did decrease with larger inputs and entries (as was expected). There are several possible explanations for the performance degradation. One possible explanation is that the MPI overhead was substantially greater than expected. It should also be noted that PETSc is aimed towards solving partial differential equations in parallel. It is possible that coding the MPI by hand instead of using PETSc for MPI would improve performance.

Although the parallel ANN proved successful during testing, further tests on larger distributed systems would help to accurately gauge the systems' performance in more diverse environments. Additionally, analysis of the code in a variety of environments might allow for improvements in the parallel ANN's performance. A few obvious questions arise with regard to the parallelism of other types of ANNs. Will more complex multi-layer neural networks show similar results as this single layer single neuron network? Will the message passing overhead cause significant performance degradation? Utilizing back-propagation methods would result in even more message passing, however, these types of networks are some of the most powerful. Could methods be found to mitigate message passing overhead? Currently, the parallel 'learning' functions are actually slower than the serial implementation due to file I/O (and the subsequent message passing). Utilizing parallel file I/O might help to alleviate the message passing overhead, possibly resulting in performance benefit. Finally, the question of applying this code to real world problems presents itself. The current code is not sufficiently general to handle a wide variety of problems, further modification and utilization of more advanced techniques would be required.

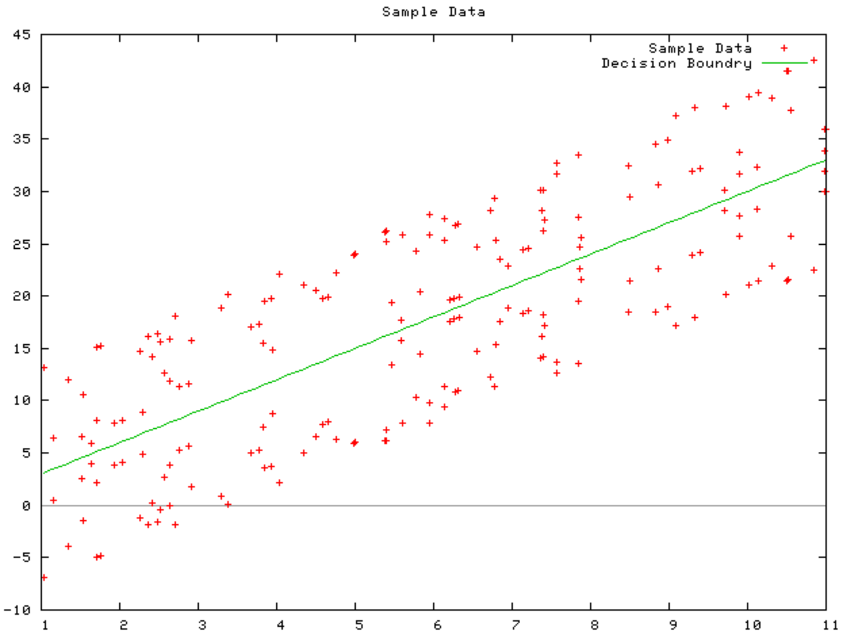

Figure 2. Sample test data

Figure 2 Sample test data used in training and verification of the networks output. The data consisted of pseudo-randomly generated points on a 2D graph, with the green line representing the 'decision boundary'. Any point above this line was in one category, while any point below was in another.

5. **Acknowledgments**

The author would like acknowledge Yaakoub Y. El Khamra for his dedication and support of this work, Kathy Traxler for her advice and encouragement, and Dr. Gabrielle Allen for supporting undergraduate research. Additionally, the author would like to thank the Center for Computation and Technology and the Computer Science Department at Louisiana State University.

6. **References**

1. Simon Haykin, *Neural Networks: A Comprehensive Foundation* (New Jersey: Prentice-Hall, 1999), 117.
2. C.L Wilson, “Massively parallel neural network recognition” *International Joint Conference on Neural Networks, Vol. 3* (1992): 11
3. Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith and Hong Zhang, PETSc Web Page, http://www.mcs.anl.gov/petsc
4. Cactus Code Webpage, http://www.cactuscode.org/
5. S.W. Aiken, M.W. Koch, and M.W Roberts, “A parallel neural network simulator” *1990 International Joint Conference on Neural Networks, Vol. 2* (1990): 21
6. Wang Guoyin and Shi Hongbao, “Parallel neural network architectures” *IEEE International Conference on Neural Networks, Vol. 3* (1995): 1239