Deployment of Neural Network on Multi-Core Architecture

Jigisha Gandhi[1]
Assistant Professor, Information Technology Department
Sarvajanik College of Engineering and Technology, Surat, India

Shitanshu Parekh[2]
Lecturer, MCA Department,
Sarvajanik College of Engineering and Technology, Surat, India

Abstract

Traditional computational methods are highly structured and linear, properties which they drive 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 theirbiological 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 project will address a parallel neural network implementation, the network's relative strengths and weaknesses, and conclude by comparing the performance using different Intel tools.

1. Introduction

In recent years there has been a great rising of interest in a method of computing that was originally investigated in the 1940s. This method is modelled generally after biological nervous system and is called neural networks (NN), artificial neural networks (ANN), parallel distributed processing (PDP) and perhaps others.

A parallel implementation of neural computations is a possible solution for memory and time consuming neural network applications (for instance real-time data processing). The two main ideas are to distribute the patterns that are used for training or to distribute the computation performed by the neural network. Pattern partitioning schemes require large pattern sets. Network partitioning schemes require large neural networks. Due mostly to their learning capability, artificial neural networks are increasingly recognized in academic and engineering communities as powerful tools for complex problem solving tasks. Unfortunately, their use in time-critical applications often demands high performance, and therefore high cost hardware systems.

Obtaining optimal solution for engineering design problem is often expensive because the process typically requires numerous iteration involving analysis and optimization programs. Many researchers have shown that optimum solution can be obtained in less time by simulating a slow, expensive analysis with a fast inexpensive Artificial Neural Network from a process perspective. And on a hardware point of view this has led to two major directions – the accelerations of execution speed of microprocessor and the parallel application of more than one processor to the problem solution. The major reason of selecting ANN for parallel programming is its own basic parallel topology, which is easily viable to parallel processing. The proposed approach explores the parallelism in ANN on Decomposition of network, weight initialization, instance presentation, calculation of activation in a

MC (Multi-Core) environment for better performance.
Neural computation means organizing processing into a number of processing elements that are massively interconnected and that exchange signals. Processing within elements usually involves adding weighted input values, applying a (non-) linear function to the input sum, and forwarding the result to other elements. Since the basic principle of neurocomputation is learning by example, such processing must be repeated again and again, with weights being changed until a network learns the problem. As matrix-vector operations are at the core of many neuroalgorithms, processing is often organized in such a way as to ensure their efficient implementation.

In this research we have developed some of the neural network models with the help of OpenMP and C++ language. The evaluation and results are compared using different intel tools – intelVtune performance analyser, intel thread checker, intel thread profiler.

Here the main concentration can be on how the object-oriented programming style can be used in the context of OpenMP and hoe to exploit C++ language features to improve scalability. The beauty of OpenMP is that it provides an abstract model. Users can develop OpenMP program on any piece of hardware with OpenMP compliant compiler and then run it on any parallel system. Possibly users need to recompile if we change architecture.

2. Deployment of Neural Network on Multi-core Architecture

To achieve the objective here we are using parallel programming concept which is implemented by OpenMP programming. We have chosen three of the neural network models which are mostly used to give solutions to complex problems in digital communications due to their nonlinear processing, parallel distributed architecture, and self-organization, capacity of learning and generalization, and efficient hardware implementation. These are single layer Feed-Forward Perceptron, NN with Back-propogation algorithm and SOM (Self-Organizing Map). We are gathering the statistical data for each model with the help of different intel tools. These data helps us to compare the performance of each model.

With the help of parallel programming a problem can be solved in a reasonable time; situations arise when the same problem has to be evaluated multiple times with different input values. This situation is especially applicable to parallel computers, since without any alteration to the program, multiple instances of the same program can be executed on different processors/computers simultaneously.

OpenMP programming is helping in the following way, at run-time; the application will go parallel at the point where the OpenMP part comes. The threads are created and the work is distributed over the threads. In this case “work” means the various loop iterations. Each thread will get assigned a chunk out of the total number of iterations that need to be executed. At the end of the loop, the thread synchronizes and one thread (the so-called “master thread”) resumes execution.

In the proposed methodology we are comparing the performance of Neural Network using OpenMP with sequential programming on dual core architecture. It also ensures parallelism of ANN on Multi-Core (MC) environment in the following levels of implementation of ANN:

1. First level parallelism can be achieved through the topology of ANN by decomposing the ANN into sub-networks depending on available cores.
2. Once subnet have been defined advantage of thread level parallelism can be taken into picture for achieving parallelism at following basic stages of ANN:
   a. Weight initialization
   b. Instance presentation to input layer
   c. Calculation of activation on different layers according to the application and ANN used.

To improve the computation capability of Neural Network we are trying to implement it on dual core by parallelizing the unit of the program which seems to be easily parallelized, because as matrix-vector operations are at the core of many neuroalgorithms, processing is often organized in such a way as to ensure their efficient implementation (parallel implementation)[5].

3. Neural Network

An artificial neural network is a massively parallel distributed processor made up of simple processing units (neurons), which has the ability to learn functional dependencies from data. It resembles the brain in two respects:

1. Knowledge is acquired by the network from its environment through a learning process.

2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

A typical feedforward neural network will consist of a set of nodes. Some of these are designed input nodes, some output nodes, and those in between hidden nodes. There are also connections between the neurons, with a number referred to as a weight associated with each connection. When the network is in operation, a value will be applied to each input node – the values being fed in by a human operator, or from environmental sensors, or perhaps from some other program.

Each node than passes it's given value to the connections leading out from it, and on each connection the value is multiplied by the weight associated with that connection. Each node in the next layer than receives a value which is the sum of the values produced by the connections leading into it, and in each node a simple computation is performed on the value – a sigmoid function is typical. This process is then repeated, with the results being passed through subsequent layers of nodes until the output nodes are reached.

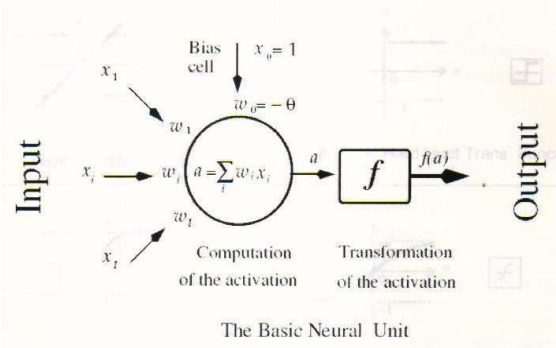

Figure 1. Graphical representation of a neuron

Each neuron is a simple processing unit which receives some weighted data, sums them with a bias and calculates an output to be passed on (Figure 1). The function that the neuron uses to calculate the output is called the activation function.
O = f(x1.W1+x2.W2+x3.W3+….+x1w1+b) = f($\sum_{j=1}$to n xj Wj + b) where f is the activation function[1].
Typically, activation functions are generally non-linear having a "squashing" effect. Linear functions are limited because the output is simply proportional to the input.

3.1. Types of Neural Networks

Neural Networks can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.
Based on the connection pattern (architecture), Neural Networks can be grouped into two categories (Figure 2):

1. **Feed-forward networks:** Feed-forward networks, in which graphs have no recurrent (or feedback) networks, in which loops occur because of feedback connections. Feed-forward networks are static, that is, they produce only one set of output values rather than a sequence of values from a given input. These networks are memory-less in the sense that their response to an input is independent of the previous network state.
   There are three types of networks in this category:
   a) Single-layer perceptron
   b) Multilayer perceptron
   c) Radial basis function nets

2. **Recurrent or feedback networks:** Recurrent, or feedback networks on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state.
   There are four types of networks in this category:
   a) Competitive Networks
   b) Kohonen's SOM
   c) Hopfield Networks
   d) ART models

3.2. Learning

A learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task.

There are three main learning paradigms: supervised, unsupervised, and hybrid.

1. In supervised learning, or learning with a "teacher", the network is provided with a correct answer (output) for every input

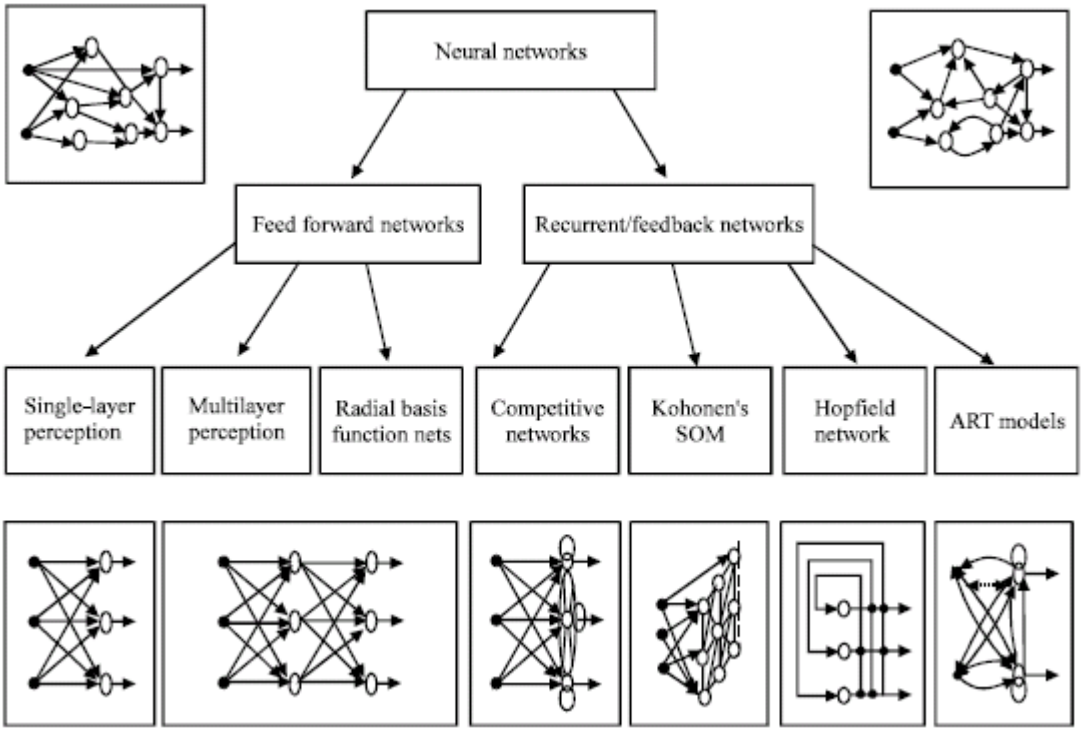

Figure 2. A taxonomy of feed-forward and recurrent/feedback network architectures

pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers.
Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves.

2. Unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations.
3. Hybrid learning combines supervised and unsupervised learning. Parts of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

In this research paper we are trying to implement three basic learning algorithms of neural networks. These are perceptron learning algorithm, backpropogation learning algorithm and SOM (self-organizing maps) learning algorithm [2].

4. OpenMP

What is OpenMP?

OpenMP is a shared-memory application programming interface (API) whose features are based on prior efforts to facilitate shared-memory parallel programming.
OpenMP uses a directive based approach to parallelize an application. The one limitation of OpenMP is that an application can only run within a single address space. In other words, we cannot run an OpenMP application on a cluster. This is a difference with MPI. OpenMP is built on top of a native threading model and therefore adds overhead, but the additional cost is fairly low. Unless we use OpenMP in the "wrong" way. One golden rule is to create large portions of parallel work to amortize the cost of the so-called parallel region in OpenMP.

Creating an OpenMP Program

OpenMP's *directives* let the user tell the compiler which instructions to execute parallel and how to distribute them among the threads that will run the code. An OpenMP directive is an instruction in a special format that is understood by OpenMP compilers only. In fact, it looks like a comment to a regular Fortran compiler or a pragma to a C/C++ compiler, so that the program may run just as it did beforehand if a compiler is not OpenMP-aware. The API does not have many different directives,

but they are powerful enough to cover a variety of needs.

- The first step in creating an OpenMP program a sequential one is to identify the parallelism it contains. Basically, this means finding instructions, sequences of instructions, or even large regions of code that may be executed concurrently by different processors.
- The second step in creating an OpenMP program is to express, using OpenMP, the parallelism that has been identified. A huge practical benefit of OpenMP is that it can be applied to *incrementally* create a parallel program from an existing sequential code. The developer can insert directives into a portion of the program version has been successfully compiled and tested, another portion of the code can be parallelized. The programmer can terminate this process once the desired speedup has been obtained [3].

OpenMP Language Features

OpenMP provides directives, library functions, and environment variables to create and control the execution of parallel programs.

- *OpenMP Directive* – In C/C++, a #pragma and in Fortran, a comment, that specifies OpenMP program behaviour.
- *Executable Directive* – An OpenMP directive that is not declarative; that is, it may be placed in an executable context.
- *Construct* – An OpenMP executable directive (and, for Fortran, the paired end directive, if any) and the associated statement, loop, or structured block, if any, not including the code in any called routines, that is, the lexical extent of an executable directive [6].

This set comprises the following constructs, some of the clauses that make them powerful, and (informally) a few of the OpenMP library routines [7]:

- Parallel Constucts
- Work-Sharing Constructs
  1. Loop Construct
  2. Sections Construct
  3. Single Construct
  4. Workshare Construct (FORTRAN only)
- Data-Sharing, No wait, and Schedule Clauses
- Other constructs
  1. Barrier Construct
  2. Critical Construct
  3. Atomic Construct
  4. Locks
  5. Master Construct

5. Tools Used

5.1. Intel VTune Performance Analyzer

This tool helps to streamline the code in just a few clicks. It locates and removes performance bottlenecks with low overhead through a graphical interface on Windows platforms, with strong Visual Studio .NET integration.

5.2. Intel Thread Checker

The tool is designed to observe the execution of a program and to inform the user of places in the application where problem may exist. The problems detected are specific to the threads. These include incorrect use of the threading and synchronization API functions.

5.3. Intel Thread Profiler

The tool is very useful for analysing bottlenecks in our threaded code. Thread Profiler quickly pinpointed problem areas and showed us the reasons for the slowdown, so user is able to restructure the code for better threaded performance [4].

6. Results and Discussion

In this research paper we have implemented perceptron algorithm of neural network model. There are two types both sequential and parallel methodologies have been used to develop this algorithm. Here Intel VTune Performance Analyzer is used to evaluate this program.

The sequential and parallel both programs have the same input values. The performance is checked by intel tools. Intel VTune Performance Analyzer finds out the hotspots in the program. In hotspots analysis it views results of time and event sampling on multiple levels, drilling down to the exact operating system process, thread, module executable, function/method, individual line of source code, or individual machine/assembly language instruction to identify specific bottlenecks.

Evaluation: Neural Network model with Perceptron algorithm

The program contains one class named neuron having four member functions as:

- Initialization
- Calculation of activation

- Weight change
- Weight adjustment

Statistical data of perceptron algorithm:

Table 1. Statistical data of sequential program – Perceptron algorithm

Execution time : 0.03100 sec	
Time statistics	
1) Clockticks	1,778,000,000 events
2) Processor Time	0.64 sec
Characterization data	
1. System CPI	7.47 Clockticks per Instruction Retired
2. Parallel activity	16.99 %
3. Processor Utilization	58.5 %

Table 2. Statistical data of parallel program – Perceptron algorithm

Execution time : 0.0460 sec	
Time statistics	
1) Clockticks	1,156,400,000 events
2) Processor Time	0.41 sec
Characterization data	
1. System CPI	4.49 Clockticks per Instruction Retired
2. Parallel activity	28.82 %
3. Processor Utilization	64.41 %

7. References

[1] Alexandra Oliveira, "Neural network software tool development: exploring programming language options", INEB – Instituto de EngenhariaBiomedica FEUP/DEEC, RuaDr. Roberto Frias, 4200-645 PORTO

[2] Anil K.Jain, Michigan State University, Jianchang Mao, K.M.Mohiuddin, ZBMAZmaden Research Center, "Artificial Neural Networks : A tutorial"

[3] Christian Terbovan, "C++ and OpenMP" and "OpenMP and C++", Center for Computing and Communication, RWTH Aachen University, Germany

[4] Information about intel tools. [Online]. Available: www.intel.com

[5] LiMin Fu, "NEURAL NETWORKS IN COMPUTER INTELLIGENCE", University of Florida, Gainesville.

[6] OpenMP and C++, article of MSDN Magazine Available: http://www.indopedia.org/Neuaral_network.html

[7] Tim Mattson of Intel Corporation, "OpenMP C/C++ Application Program Interface" version 1.0, October 1998