Distributed and Multithreaded Neural Network Algorithms for Stock Price Learning

by

Mohammad Rashedur Rahman

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Computer Science

Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

DISTRIBUTED AND MULTITHREADED NEURAL NETWORK ALGORITHMS FOR STOCK PRICE LEARNING

BY

MOHAMMAD RASHEDUR RAHMAN

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Sience

Abstract

In this thesis, we focus on the problem of learning the stock price movement using neural networks in distributed and shared memory environments. We parallelize the Backpropagation Neural Network (BPNN) algorithm on an 8 node Beowulf cluster with Message Passing Interface (MPI) and on an 8 processor Pentium III Symmetric Multi-Processor (SMP) machine using OpenMP. We have developed algorithms for two types of BPNN: *neuron parallelism* and *training set parallelism* on the distributed architecture. In neuron parallelism the hidden nodes are partitioned and distributed among the various processors while in training set parallelism, the input data (stock prices) is partitioned and distributed among the processors. On SMP, we exploit two different approaches for parallelizing neural network with multithreading. The first approach, *loop-level parallelism* is a fine-grained algorithm where the iterations of a loop are divided dynamically among the threads by the compiler. In the second approach, *coarse-grained parallelism*, the user intervenes and creates a limited number of threads where each thread is given an equal amount of workload.

We have conducted various experiments to study the performance of the distributed and multithreaded algorithms. We have compared with a traditional autoregression model to establish accuracy of our results. In loop-level parallelism we observed that there is a limit to the number of threads that the system can handle on a specified number of processors to produce reasonable performance. The coarse-grained approach on OpenMP improved the performance to a certain extent. The comparison between our MPI and OpenMP results suggest that the training set parallelism outperforms all the other types of parallelism considered in the study. There are opportunities to improve the performance such as pipelining the network, which is left as future work. As one of the first attempts to parallelize the neural networks for financial forecasting applications, the current results are, however, encouraging in relation to the overall timings for execution of traditional autoregression models or sequential BPNN algorithms.

Acknowledgements

First, I would like to express my utmost gratitude to almighty Allah for His creation and making me submissive to Him.

I feel gratified to my thesis supervisor Dr. Ruppa K. Thulasiram, for providing me all out support during the last one year. His inspiration and enthusiastic participation has been the driving force for me. It has been greatest pleasure to find an enthusiastic mentor like him, who is interested in Neural Networks and Parallel Computing and is not afraid of to explore new and uncharted territories of Neural Networks and High Performance Computing. I would like to give special thanks to Dr. Parimala Thulasiraman for her inspiration in my research work. Both of them not only introduced the arena of parallel computing to me but also provided valuable remarks and suggestions through out my graduate study. Without their proper guidance, advice, continual encouragement and active participation in the process of work, it would have not been possible. They have let me free to explore, yet they have always guided me so that I never lose my way.

I would also like to thank Dr. Peter Graham and Dr. Rasit Eskicioglu for making Beowulf available for this study and for their timely help in system upgradations.

I am also thankful to Prof. Milton Boyd, Department of Agricultural Economics and Prof. David Scuse, Department of Computer Science for being in my thesis committee and providing valuable comments and criticisms.

I would like to acknowledge the financial support from Natural Sciences and Engineering Research Council, Canada and University Research Grant Program (University of Manitoba).

Finally I express my heartfelt gratitude to my parents, my only sister who gave me support and encourage during my research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Financial institutions use forecasting for two reasons: to decrease loses and to increase return on investment. Not long ago such forecasting was done with primitive, intuitive rules of the trade. For example, the premise behind the technical analysis is that variables (internal and external) affecting the market are already factored into the market price. Important indicators of technical analysis are filter and momentum indicators, cycle theory, volume indicators and pattern analysis. Fundamental analysis is another technique where trading decisions were made by forecasting market direction based upon underlying economic factors affecting the particular stock market. Sometimes, these two methods were not transparent to the front end traders themselves. Hence, decision making is not straight forward using these two techniques. It requires considerable effort from the front end traders in using these techniques. Averages of the consecutive values of the variables were found to follow some invariant pattern and were found useful. Therefore, moving averages of some quantities were found to be transparent for analysis and became prominent in predicting future values not only in finance but in other fields

such as weather prediction. In finance, moving averages were initially used for stock price forecasting. One of the important models developed is called *Auto Regressive Moving Average (ARMA)* which is more scientific with reproducible results and hence is applied successfully for decision making. Moving averages follow a statistical inference model, which allow the system to deduce or compute the coefficients for the variables from observed data. There are systems for which the invariant behavior is not observable, at least not with a naked eye. There are two complementary tasks in this area: estimating the model from observed data and analyzing the properties of the results generated from estimated model. Model estimation is done in such cases with statistical inference techniques such as autoregression. *Artificial Neural Networks (NN)* are essentially statistical devices for performing inductive inference. For statisticians neural networks are analogous to non-parametric, non-linear regression models.

As the market generated huge amount of data, using the moving averages to predict future prices became a daunting task and knowledge of neural networks came to bear fruit in finance. Instead of using all the past data available to predict future prices, neural networks were developed wherein a network of nodes (or neurons) was trained with a known set of past data to predict future prices with good accuracy. After this training phase, the new set of data were used to accurately predict future values. Ever since this development of neural networks for predicting future prices, neural networks have been used in many areas within the financial field such as: credit authorization screening, mortgage risk assessment, regularities in security price movements, portfolio

selection and diversification, simulation of market behavior.

One of the major hurdles with the neural networks is the training times. Sixty hours of training with certain parametric conditions of the given neural networks is not uncommon. This much of waiting is beyond the accepted norm these days for decision processes in certain financial sectors where the dynamics of the market dictates quick and accurate decisions.

The overall objective of this thesis is to design and develop parallel neural network algorithms for stock price forecasting with the focus of expediting the training process to facilitate quick decision making. Our aim is not in rigorous statistical testing of the forecasted results. In other words, our study aims at training the neural network by parallel processing. In the following chapter we present some of the forecasting models used in finance. We introduce parallel computing environments in chapter 3. In chapter 4 we discuss the sequential Backpropagation Neural Network (BPNN) algorithm and its parallelization. We have developed parallel algorithms in both distributed and shared memory architectures. In this study, we have developed two versions of the backpropagation algorithm in each of the two parallel computing approaches. In chapter 5 we present the theoretical analysis of our algorithms. We present our experimental results in chapter 6 and discuss the results elaborately. Concluding remarks we present in chapter 7 together with some possible future works.

Chapter 2

Related Work

Forecasting has been applied in many areas, from weather to business and economic forecasting. Weather forecasts are of interest to the general public, farmers as well as travelers because it helps them to plan and make decisions for the next day. Managers in businesses use forecasts in their day-to-day planning and control of company operations. Reliable forecasts help managers to make timely decisions on: company financial planning, investment in plant and machinery, acquisition of materials, human resources, set production, inventory levels and so on.

The forecasting techniques used in business, industry and finance are *surveys*, the *Delphi method* and various *extrapolation methods. Surveys* are commonly used in market research for industrial and financial products. This method relies on a questionnaire administered by mail, telephone or personal interview. The responses help to find out the views of consumers regarding future demand of a product and accordingly the managers

could predict the future sales. On the other hand, the *Delphi method* relies on combining the views of a number of experts. For this reason it is also called "jury of executive opinion" [12]. Initially a group of experts is asked independently to forecast some particular event. The results of this outcome are collected and discussed together. These experts are then asked to explain their standing for prediction and after further discussion, a second survey is conducted, a discussion follows and the process is repeated until the experts reach a decision which is acceptable to every one. The *extrapolation method* tries to identify the past patterns and expects that these previous patterns in the data will be repeated in the future under similar business condition. In this technique data are observed at regular time intervals, say daily, weekly, quarterly or annually. A *Time-series* is an extrapolation method which is an important tool used for forecasting. Time-series method has been widely used across many fields. We present a brief description of one of the traditional time-series forecasting called *ARMA* model in the following section.

2.1 Time Series Forecasting

A time series is a set of observations ordered in time. Time Series analysis is concerned with data which are not independent, but serially correlated, and where the relations between consecutive observations are of interest. With the advent of widespread computer applications, the much more general and statistically based methods of time-series analysis known as *autoregressive moving averages (ARMA)* are developed and applied in forecasting.

2.1.1 ARMA Model

Time-series data refers to observations on a variable that occurs in a time sequence. We use the symbol X_t to stand for the numerical value of an observation; the subscript t refers to the time period when the observation occurs. Thus, a sequence of n observations could be represented as: $(X_1,X_2,X_3,....,X_n)$. ARMA is a statistical tool that is used to explain the behavior of time-series data using only past observations on the variable in question. In ARMA analysis, the time-sequenced observations in the data $(...,X_{t-1},X_t,X_{t+1})$ are statistically dependent. We use the statistical concept of correlation to measure the relationships between observations within the series. In autoregression analysis we would like to examine the correlation between X at time t (X_t) and X at all earlier time periods $(X_{t-1},X_t,X_{t-3},...)$.

A general AutoRegression $AR(p)$ model is defined as follows [20]

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \phi_3 X_{t-3} + ... + \phi_p X_{t-p} + e_t$$

where the ϕ_i's are the autoregression coefficients, X_t's are the series under investigation and p is the order of the model. Before an AR model can be used, its order p must be specified. The appropriate value for p specifies the number of terms to be included and it is much less than the length of the series. The noise term or residue e_t is known as Gaussian white noise.

The current term of the series can be estimated by a linear weighted sum of previous terms in the series. The weights are the autoregressive coefficients ϕ_i's. The problem in AR analysis is to derive the "best" values for ϕ_i given a series X_t. The autoregressive

coefficients could be found using nonlinear least square methods. The most common method for deriving the coefficients involves multiplying the definition above by X_{t-p} and taking the expectation values and normalizing. This gives a set of linear equations called the Yule-Walker system of linear equations that can be solved numerically [20] .

2.1.2 The Method of Adaptive Filtering

Adaptive filtering can be applied as an AR method of the form described above. This method starts with an initial set of ϕ_i values that are calculated with the procedure described above and proceeds to adjust them according to the method of steepest descent

$$\acute{\phi}_{it} = \phi_{it-1} + 2Ke_tX_{t-i}$$

$$i = 1, 2, ..., p,$$

$$t = p + 1, p + 2, ...n,$$

where $\acute{\phi}_{it}$ is the new adaptive parameter, ϕ_{it-1} is the old parameter, K is the learning constant that determines the speed of adaptation, and e_t and X_{t-i} have same definitions as before. It is shown [20] that by repeatedly using the above equation under the necessary conditions, parametric values (ϕ_i's) that give successively smaller Mean Squared Error (MSE) can be easily attained.

2.2 Neural Network Forecasting

A Neural Network is one of the applications of Artificial Intelligence [9]. Neural networks were developed through studies of biological neuron perceptions [10]. In a sense, neural networks are computer programs that are trained to behave and learn like a human brain and remember some information from the past. A network is built with many neurons called *nodes* which are similar to neurons in the biological field. It is a system composed of many simple processing elements whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes. The network looks for patterns, learns the patterns and develops the ability to forecast.

Though neural networks have been studied since 1940's [21] they are relatively new methods for modeling and forecasting financial data, for example, stock/asset price. Using neural networks, it is possible to search for regularities in the historical data that could help predict the current asset price.

Kimoto [16] introduced modular neural networks that could learn the relationships between the past technical and economic indices and predict the timing to buy and sell a stock. Simulation of buying and selling stocks using the prediction system showed an excellent profit. Stock price fluctuation factors could be extracted by analyzing the neural networks. The neural networks learned the data well enough to show a very high correlation coefficient, i.e. 0.991 whereas the multiple regression analysis showed a lower coefficient, for example, 0.543.

Yoon and Swales [40] illustrated the neural networks approach to predict stock price

performance and compared its predictive power with that of multiple discriminant analysis (MDA) methods. Inputs for the network were a list of nine variables: confidence, economic factors outside the firm's control, growth, strategic gains, new products, anticipated loss, anticipated gain, long-term optimism, and short-term optimism. The outputs of the network were the predicted stock price performance of the firm; a firm whose stock price performed well was classified as Group 1 and a firm whose stock price performed poorly was classified as Group 2. The four layered network correctly classified 91% of the mean training data and appropriately predicted 77.5% of the mean testing data: for two sets 20 companies each, 18 were correctly classified into Group 1 whereas 13 companies were correctly classified into Group 2. The MDA method resulted in 74% correct classification for training set and 65% for the testing data set: again, for two sets of 20 companies each, about 12 and 14 companies were correctly classified into Group 1 and Group 2 respectively.

Dutta [8] tried to rate different industrial bonds using neural networks. Valueline Index and S&P Bond guide identified ten financial variables that had influence on the bond rating. These ten variables were used as input for the neural network. Various neural network configurations (two-layered, three-layered, different number of hidden nodes etc.) were experimented with the Berkeley ISP (the Interactive Statistical Package developed at the University of California, Berkeley), which is normally used for multiple regression analysis. neural network consistently outperformed the regression model in prediction for bond rating. The success rate of prediction with neural networks was

88.3% whereas with regression analysis it was only 66.7%. Besides, the total squared error for both training and testing samples for regression analysis was about an order of magnitude higher than that for neural network.

White [38] used neural networks for predicting one-day of return for holding IBM common stock. Of the available 5000 days of return data, 1000 days were selected for training purposes, whereas samples of 500 days before and after the training period were used for testing the neural network's performance. The three-layer neural network trained on the 1000 days detected the nonlinear structure of the market and a positive correlation coefficient was found between the actual rate of daily return and predicted rate of return.

Recently, Kanas [15] compared out-of-sample forecasts of monthly return for Dow Jones (DJ) and Financial Times (FT) indices generated by a nonlinear neural network and linear model. This comparison was carried out on the basis of two approaches: directional accuracy and forecast encompassing. Though the two models did not produce good performance in terms of predicting the directional change of the two indices, the neural network forecasts could explain the forecast errors of the linear model whereas the linear model could not explain the forecasts errors of the neural network for both indices.

In addition, neural networks have been applied to many problems in finance, including mortgage risk assessment, economic prediction, risk rating of exchange-traded fixed-income investments, portfolio selection/diversification, simulation of market behavior, index construction and identification of explanatory economic factors [34, 36]. Feedfor-

ward networks with a single hidden layer and trained by least-squares are statistically consistent estimators of arbitrary square-integrable regression functions under certain practically-satisfiable assumptions (regarding sampling, target noise, the number of hidden units, the size of weights, and the form of the hidden-unit activation function) [37]. Such networks can be trained as statistically consistent estimators of derivatives of regression functions and quantiles of the conditional noise distribution [39]. Here, we have briefly discussed the general course of development of neural networks for finance. However, there are many works using neural networks in finance and economic forecasting (for example, [14, 17, 22], among many other works using forecasting techniques) and are out of scope for reference in the current thesis.

Backpropagation(BP) is one of the most commonly used training algorithms for multilayer neural networks, in finance and other applications. The backpropagation algorithm is computationally intensive and training times on the order of days and weeks are not uncommon. As a result, some studies have been focused, though not for finance applications, on efficient parallel implementations of the backpropagation algorithm [32]. Two main paradigms used to parallelize the backpropagation learning algorithm are: partitioning the neural network called *network-based parallelism* and partitioning the training set of neural network called *training-set parallelism* [31].

In network-based parallelism, the neural network is partitioned and distributed among different processors. Each processor then simulates a group of neurons belonging to different layers of the neural network over the whole training set. In training-set parallelism

the neural network is duplicated on every processor of the parallel machine, and each processor works with a subset of the training set. Each processor has a local copy of the complete weight matrix of the neural network and accumulates weight change values for the given training patterns. Pipelining is another technique [26] that allows the training patterns to be "pipelined" between the layers. That is, the hidden and output layers are computed on different processors. While the output layer processor calculates the output and error values for the present training set, the hidden layer processor processes the next training set. The forward and backward phase of backpropagation algorithm could also be parallelized.

Rogers and Skillicorn [30] developed a cost model for different parallelization techniques to train the neural networks based on different architectural parameters using the Bulk Synchronous Parallelism (BSP) [35] cost model. They derived cost formulae for both training-set and network-based parallelism. They concluded from their experiments that the training-set parallelism is superior for almost all parallel architectures.

It is evident that parallelizing neural networks could help the training process for the given application. However, there is no such study reported in the literature for stock price forecasting with neural networks in parallel computing environments.

In the next chapter we introduce briefly several existing parallel architectures and return to the discussion of neural network algorithms in chapter 4.

Chapter 3

Parallel Computing Environments

Parallel computing is a coordinated, simultaneous use of a number of processors to solve a problem. It helps to perform large, complex tasks faster. A large task can be either performed serially, one step following another or can be decomposed into smaller tasks to be performed concurrently i.e., in parallel. Many large commercial parallel machines are based on Flynn's taxonomy of computer architecture [18]. According to Flynn [18], the main categories of computer architectures are SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data) machines. They are described briefly in the following sections. A detailed description could be found in [18].

3.1 Single Instruction Single Data (SISD) Model

In general, this type of machine is comprised of a processing unit and main memory. Traditional sequential computers are based on this model introduced by Von Neuman.

This computational model takes a single sequence of instructions and operates on a single sequence of data. Data and programs are moved between the memory and the processing unit. The main drawback of this architecture is that the speed of execution of programs not only depends on the processor speed but also on the speed at which the programs and instructions are brought from memory to the processor, referred to as memory latency. Therefore, the performance of programs is limited by the speed at which data are transferred from memory to the processor and vice-versa.

3.2 Single Instruction Multiple Data (SIMD) Model

This architecture has a single control unit and a number of processing units under that control unit. During each instruction cycle, the control processor broadcasts the same instruction to all of its subordinate processing units. Therefore, each processor performs the same computation on different data sets. The processors work in lock step, thereby requiring a global synchronization mechanism among the processors. Such parallel machines are called *synchronous* programming models. The Massively Parallel Processor [2] and MasParMP-1 [24] are examples of SIMD architectures. One of the drawbacks of SIMD computers is that the processors can not execute different instructions at the same clock cycle. For instance, in a conditional statement (if-then-else), the code for each condition must be executed sequentially. Under such circumstances, some processors may be idling while others remain active, thereby, degrading the performance.

3.3 Multiple Instruction Multiple Data (MIMD) Model

This architecture comprises of a number of autonomous processors: each processor is capable of executing a different program independent of other processors. MIMD are also referred to SPMD (Single Program Multiple Data) models if the same program is executed on every processor but with different data set. These are also called *asynchronous* programming models because they do not have a global clock to synchronize the different operations as in SIMD machine. MIMD systems may be divided into two sub-categories: distributed and shared memory multiprocessors.

3.3.1 Distributed-Memory MIMD Multiprocessors

In this model multiple processors operate independently and each has its own private local memory connected through an interconnection network as shown in Figure 3.1.

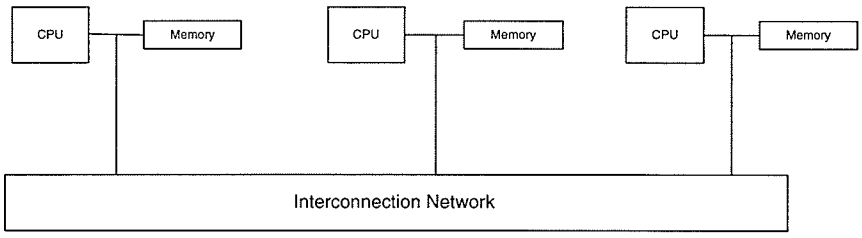

Figure 3.1: Generic Distributed-Memory System.

In this architecture, each processor can only address its own local memory and it is the programmer's responsibility managing partition, distributing and mapping data onto the processors. The processors communicate via message passing primitives through the interconnection network. Parallel Virtual Machine (PVM) and Message Passing Interface

(MPI) are two such message passing libraries. Intel Paragon [25], CM-5 [5], ncube [23] are few examples of distributed machines.

3.3.2 Shared-Memory MIMD Multiprocessors

In this model, all the processors share a common global memory (Figure 3.2).

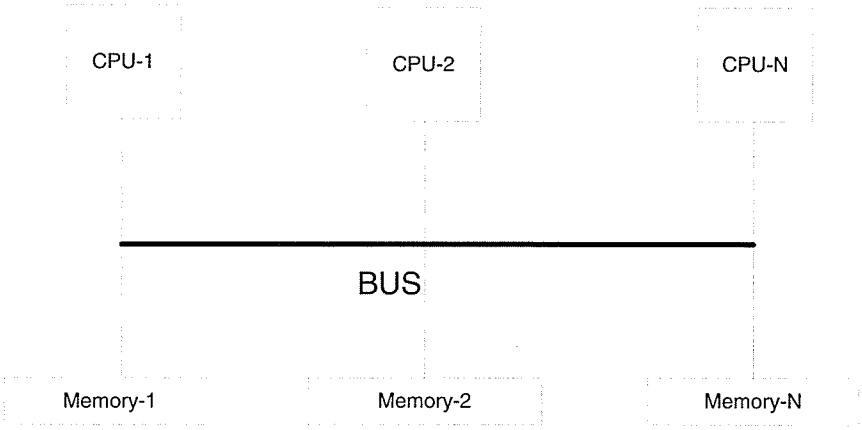

Figure 3.2: Generic Shared-Memory System.

Such machines are easy to build and program. Examples of such machines are SGI Origin 3000, BBN Butterfly [3]. The users need not worry about partitioning their data among the processors and only partition the computations. The drawback of shared memory machines is that due to single address space concept, variables may be shared, and resource sharing may limit the speed of the computation.

3.3.3 Multithreaded Paradigm

Massively parallel processors perform asynchronous computation by message passing through an interconnection network. In 1987, Arvind and Iannucci [1] realized two

fundamental latency problems that asynchronocity triggers: *communication* and *synchronization.* Communication, or remote access causes a processor to be idle until the necessary data is fetched from the destination processor. In order to realize that the necessary data is ready at the destination processor, the processors must synchronize.

There are many techniques to hide, reduce or tolerate latencies in hardware and software. The most general technique is *multithreading* which tries to overlap computation with communication by means of threads (a thread is a set of few instructions). The underlying principle in multithreading is that as long as there is enough parallelism in the algorithm, the processors are always busy executing threads.

There are different types of multithreaded constructs (languages, libraries or hardware). Tera [6] is a commercial multithreaded hardware architecture available at the San Diego Supercomputing Center. There are other research oriented multithreaded models such as Cilk [4, 19] which is a fine-grained multithreaded language developed at MIT. EARTH [13] is another example which is a fine-grained data flow multithreaded architecture developed at McGill University (a fine-grained thread is a thread with very few instructions). PThreads and Solaris threads are commercially available library routines developed at Sun Microsystems. Java is an object-oriented language that supports threads. These language based threads are coarse-grained (many instructions per thread), operating system dependent and slow.

Currently OpenMP is regarded as the standard language for multithreading. It is designed for shared-memory multiprocessors.

OpenMP supports parallelism by including parallel directives/constructs in the program. In general, an OpenMP program executes sequentially on a single thread (called a master thread), just like a serial program. When the program encounters the parallel constructs, a group of parallel threads (called slave threads) are created and each thread executes a copy of the body of code enclosed within the parallel directive. When each slave thread completes execution, the master thread gains control of the program and continues the program execution in sequential basis until it encounters another parallel directive.

Our work in this thesis concerns the development and implementation of parallel algorithms for neural network on MIMD model of computing with both distributed (MPI) and multithreaded shared-memory (OpenMP) approaches. As mentioned in chapter 1, the focus is not on the rigorous statistical testing of the results.

In the next chapter we return to the neural network starting with the description of sequential backpropagation neural network algorithm and its parallelization.

Chapter 4

Backpropagation Neural Network

4.1 Neural Network

A fully connected, layered, feedforward network is depicted in Figure 4.1. In this figure, x_i, h_i, o_i represent unit activation levels of input, hidden and output units, respectively.

Weights on the connections between the input and hidden layers are denoted by $w1_{ij}$, while weights on connections between the hidden and output layers are denoted by $w2_{ij}$. This network has three layers, although it is possible and sometimes useful to have more. Each unit in one layer is connected in the forward direction to every unit in the next layer. Activations flow from the input layer through the hidden layer, to the output layer. The knowledge of the network is encoded in the weights on connections between units.

A backpropagation network typically is initialized with a random set of weights. The network adjusts its weights each time it processes an input-output pair. Each pair re-

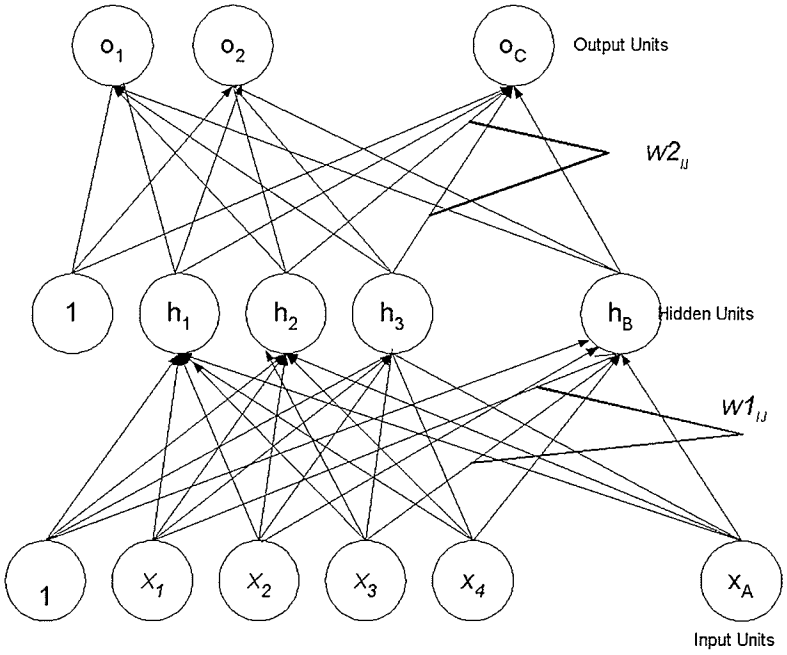

Figure 4.1: A Multilayer Feed-Forward Network.

quires two stages: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. During the backward pass, the network's output (from the forward pass) is compared with the target output and error estimates are computed for the output units. The weights connected to the output units are used in order to derive error estimates for the units in the hidden layers. Finally, errors are propagated back to the connections stemming from the input units. The backpropagation algorithm usually updates its weights incrementally, after seeing each input-output pair. After it has examined all the input-output pairs (and adjusted its weights many times) we say that one *epoch* has been completed. This is explained further with the algorithm in the next

section. This process is referred as neural network training/learning process. Training a backpropagation network usually requires many epochs. An individual unit sums its weighted inputs and produces a real value between 0 and 1 as output based on an activation function (sigmoid/bipolar). The activation function for a backpropagation network should have several important characteristics: it should be continuous, differentiable, and monotonically non-decreasing [9]. The computation of derivatives of the activation function is generally easy. For the most commonly used activation functions, the value of the derivative can be expressed in terms of the value of the function itself. The most common activation function is the binary sigmoid function which has a range of (0,1) and is defined as $output = 1/(1 + e^{-sum})$, where sum is the weighted sum of the inputs to a unit. Another common activation function is bipolar sigmoid, which has a range of (-1,1) and is defined as $output = 2/(1 + e^{-sum}) - 1$.

Learning algorithms in neural networks are generally categorized as *supervised* and *unsupervised* learning. In supervised learning, the correct results (target values or desired outputs) are known and are given to the neural network during training so that the neural network can adjust its weights to try to match its outputs to the target values. In unsupervised learning, the neural network is not provided with the correct results during training. Unsupervised neural networks usually perform some kind of data compression, such as dimensionality reduction or clustering. Since this study is not concerned with unsupervised learning, we do not discuss this learning category any further.

The backpropagation networks with supervised learning rules are the most popular

and useful for time series forecasting. Standard backpropagation employs an optimization method called the *gradient descent* method to map the inputs to the outputs of the network. This optimization method poses serious challenges when a large network is involved, in terms of solving the resulting matrix problem, which by itself is an active research problem requiring a lot of effort in scientific computing area (for example, see [33]).

4.2 General Backpropagation Algorithm

Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let B, C be the number of units in the hidden layer and output layer respectively. As shown in Figure 4.1 the input and hidden layers each have an extra unit used for thresholding; therefore, the units in these layers are indexed by the ranges *(0,.....,A)* and *(0,.....,B)*. We denote the activation levels of the units in the input layer by x_j, in the hidden layer by h_j, and in the output layer by o_j. Weights connecting the input layer to the hidden layer are denoted by $w1_{ij}$, where the subscript i corresponds to the input units and j corresponds to the hidden units. Likewise weights connecting the hidden layer to the output layer are denoted by $w2_{ij}$, with i referring to hidden units and j referring to output units.

The backpropagation [9, 29] algorithm is described as follows: This is also explained for financial forecasting in [14].

I Initialization:

1. Initialize the weights in the network (set to small random values) and the activations of the thresholding units. The values of these thresholding units will never change. For faster learning, we use Nguyen-Widrow [9] initialization. Weights from the hidden units to output units are initialized to random values between -0.5 and 0.5. The initialization of the weights from the input units to the hidden units is designed to improve the ability of the hidden units to learn. This is accomplished by distributing the initial weights and biases so that, for each input pattern, the net input to one of the hidden units will be in the range that will facilitate ready learning by the hidden neurons. Further details of this Nguyen-Widrow initialization can be found in [9].

II Feedforward:

2. Choose an input-output pair. Suppose the input vector is $x_i, (i = 1, ..., A)$ and the target vector is $t_i, (i = 1, ..., C)$. Each input unit receives the input signal x_i and broadcasts this signal to all units in the layer above, i.e., the hidden units.

3. Each hidden unit $h_j, (j = 1, ..., B)$ sums its weighted input signals, applies its activation function to compute its output signal and sends this signal to all units in the layer above (output units).

$$h\_in_j = \sum_{i=0}^{A} x_i w1_{ij}$$

$$h_j = f(h\_in_j)$$

4. Each output unit $o_j, (j = 1, ..., C)$ sums its weighted input signals, applies its

activation function to compute its output signal.

$$o\_in_j = \sum_{i=0}^{B} h_i w2_{ij}$$

$$o_j = f(o\_in_j)$$

III Backpropagation of error:

5. Each output unit o_j receives a target pattern corresponding to the input training pattern, computes its error/delta information term: $\delta 2_j = (t_j - o_j)f'(o\_in_j)$, and calculates its weight correction term $\Delta w2_{ij}$ given by $\Delta w2_{ij} = \eta \delta 2_j h_i$ (where η is called learning rate) and sends $\delta 2_j$ to units in the layer below. $\Delta w2_{ij}$ is used later to update $w2_{ij}$. The learning rate η gives an estimate for moving in the gradient direction for convergence towards the target value.

6. Each hidden unit h_j sums its delta inputs (from units in the layer above $\delta 1\_in_j = \sum_{i=1}^{C} \delta 2_i w2_{ji}$), multiplies it with the derivative of its activation function f' to calculate its error information term, $\delta 1_j = \delta 1\_in_j f'(h\_in_j)$. Each hidden unit then calculates its weight correction term (used to update $w1_{ij}$ later), $\Delta w1_{ij} = \eta \delta 1_j x_i$.

IV Update Weights:

7. Each output unit o_j updates weights, $w2_{ij}(i = 0, ..., B)$. Each hidden unit h_j updates weights, $w1_{ij}(i = 0, ..., A)$:

$$w2_{ij}(new) = w2_{ij}(old) + \Delta w2_{ij}$$

$$w1_{ij}(new) = w1_{ij}(old) + \Delta w1_{ij}$$

8. Go to step 2 and repeat. When all the input-output pairs have been presented to

the network, *one epoch is completed.*

9. Repeat steps 3 to 8 for as many epochs as desired or the error is acceptably low.

The sequential backpropagation algorithm runs well and has a good running time for a small network but when the network is large it takes a longer time to converge. Therefore, the main drawback of a backpropagation algorithm is it's training time. For example, 50 input variable combinations tested over three different hidden neurons with five sets of randomly selected starting weights and a maximum number of 4 thousand runs, results in 3 million iterations. The training time of 60 hours is not uncommon [11] for such problems. Parallel processing could definitely help in accelerating the training time of the network.

4.3 Parallel Backpropagation Algorithm: Distributed Approach

To expedite the training process, we have developed parallel neural network algorithms in this thesis and implemented them on distributed and multithreaded architectures and conducted various experiments to study the performance. In the distributed version we have followed a very coarse-grained approach, where the data is partitioned and distributed among the processors. In the multithreaded approach, we experimented with parameters such as thread size and number of threads in the system. We describe the details of these two algorithms in the following sections.

We have developed and implemented two types of parallelism in the distributed approach: *neuron parallelism*(NP) and *training set parallelism* (TSP). In NP, the neurons are subdivided and distributed among the processors. In TSP, the input data is divided among the processors but the network is replicated on every processor. Each processor trains its own part of the network on its local data set examples, and the weights are resolved by propagating the weights among the processors. The algorithms are implemented on a Beowulf cluster using MPI.

4.3.1 Neuron Parallelism

In our implementation we have used only one hidden layer, however for general understanding we describe the algorithm here with multiple hidden layers. In the backpropagation algorithm, the nodes within a hidden layer i are independent of one another. There is no communication required among these nodes. This lends itself easily to a parallel implementation. However, the nodes between the two hidden layers h_j and h_{j+1} (j refers to the number of hidden layers in the network) are dependent on one another and therefore, require communication.

Figure 4.2 shows the technique of NP for a network with one hidden layer and one output. The total training data set is N. We present $n < N$ subsets of data to the network at a time. We assume P processors and the number of nodes at each hidden layer is M. Every processor has a replicated copy of the input nodes. We partition and distribute $\frac{M}{P}$ hidden neurons to each of the processors. The edge weights corresponding

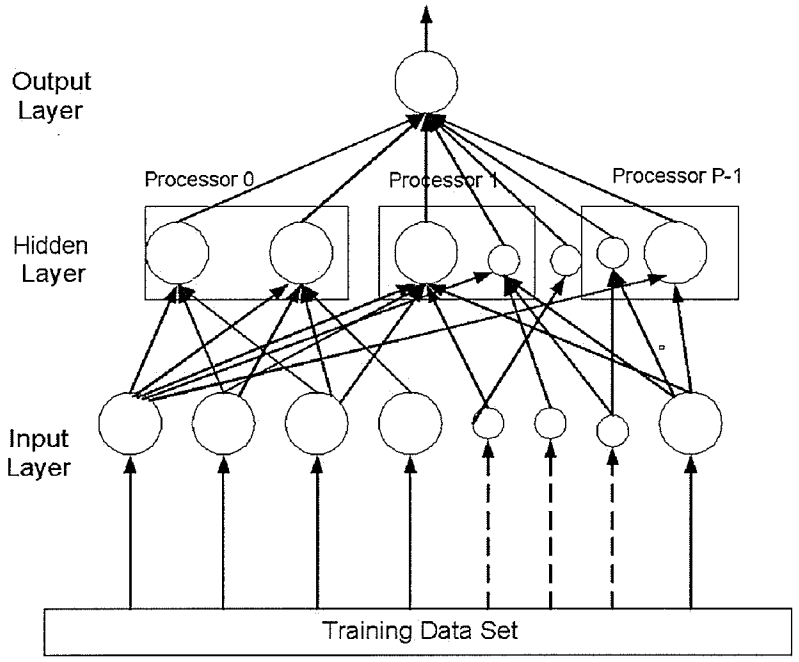

Figure 4.2: Parallelization of the Neural Network - Neuron Parallelism

to the $\frac{M}{P}$ nodes local to each processor, reside within that processor.

Each node i in a given hidden layer h_{j+1}, requires the value of all the nodes in h_j. Therefore, each processor has to distribute its local node values calculated at h_j to its neighboring nodes in different processors to calculate the value of output neuron at layer h_{j+1}. This communication can be performed as one *broadcast* operation. However, to proceed from one hidden layer to the next layer requires barrier synchronization. The following steps comprise the parallel implementation.

1. The processors calculate the activations for the hidden layer neurons residing locally.

2. Each processor sends the activations of its neurons to all other processors as soon

as these are computed.

3. Once the activations of all the hidden neurons are received, each processor calculates the output value for the output layer neuron.

4. All processors calculate the *deltas* for the output neuron residing in them. (As mentioned before, *delta* is error computed in the output layer based on the network's actual output and the target output value. This error is directly proportional to the derivative of activation function with respect to the weights, the constant of proportionality being the learning rate. In other words, delta is defined as the product of learning rate and the rate of change of the squared error in the output neuron with respect to the change in the weights of the connections).

5. All processors send these delta values to all other processors.

6. The processors calculate the deltas for all the hidden neurons residing in them as soon as they receive the deltas for the output neuron.

7. Finally the processors calculate the weight increments and update the incoming and outgoing weights for those neurons that reside in them.

8. Repeat steps 1-7 to present N data set to the network to complete one epoch.

4.3.2 Training Set Parallelism

We describe here the TSP technique for training a parallel network. Each processor maintains its own copy of the neural network and local weights. This style of parallelism

is different from neuron parallelism in the sense that the input data is partitioned, rather than the hidden nodes.

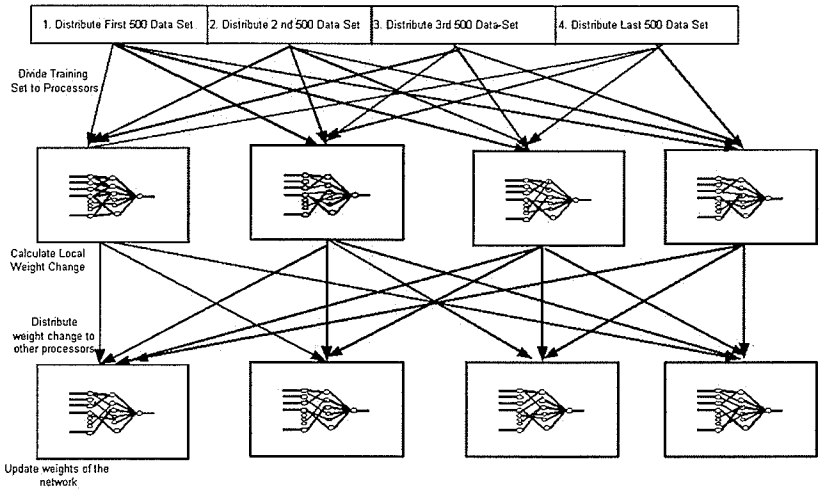

Figure 4.3: Parallelization of the Neural Network - Training Set Parallelism

Figure 4.3 illustrates the TSP technique. Given N data set, and P processors, we partition the data set into $\frac{N}{P}$ data sets and distribute each set to the processors. Each processor iterates through every data in its local data set $\frac{N}{P}$ and calculates the error gradients or weight changes to the network. Upon completion of computation, each processor broadcasts the error gradients to the other processors. Once each processor receives the entire gradient information from all the other processors, it updates the weights of its network. This process continues for many epochs or until the error is acceptably low. In effect, each network looks as if it had been trained with the entire data set.

For training set parallelism, the complete training set data is distributed among different processors. In this paradigm, each processor maintains its own network and own local weights. For example, in our case of 2080 data sets, we could distribute first 520 data sets among P processors so that each processor gets (520/P) data points. Weights remain the same for the first (520/P) data sets. Each processor accumulates the weight changes for each training pair (data-set) and keeps weight changes in its own local memory. After completing the weight updates of the network for (520/P) data sets, each processor broadcasts its local weights and weight changes to other processors. Each processor calculates an average of the initial weights of P processors and required weight change for the first 520 data sets. The new weight of the network is calculated by adding the average weight change to the average weight. Each processor broadcasts these updated weights to other processors so that for the next 520 data sets each processor works with this new weight. This process continues until one epoch is completed, that is, 2080 data sets are presented to the network. After one epoch, the algorithm described above repeats itself until the error is acceptably low.

4.4 Parallel Backpropagation Algorithm: Shared Memory Multithreaded Approach

OpenMP supports both *fine-grained parallelism* (FGP) and *coarse-grained parallelism* (CGP) using the concept of parallel regions. We describe in this section the neural

network multithreaded algorithms for these types of parallelism.

4.4.1 Fine-Grained/Loop Level Parallelism

This structure exploits the inherent loop structure in the program by parallelizing the loops individually. The iterations of a loop are divided among the threads. In this approach, we flood the system with as many threads as we require and allow the dynamic scheduler in the system to allocate the threads among the processors. These threads can be executed concurrently and asynchronously.

The parallel/distributed version of the neural network algorithm described in the previous section, (for N input data, M hidden neurons and one output) requires a barrier synchronization to perform the reduction operation at the output layer. Hence, all processors have to broadcast their final values to the master processor before the master processor can compute the output. In a shared memory model, this extra overhead is not required. Instead, we explicitly synchronize the output variable by placing it in a *critical section* and allow each processor to access this critical section without any read-write conflicts.

A portion of the actual computations of the neural network algorithm using loops for the activation calculations of the hidden neurons (the weight update calculations are also performed using loop-level parallelism without critical section) is given below in OpenMP.

```
#pragma omp parallel for
{
     for(j = 0; j < M; j++) {
        for (i = 0; i < n; i++) {
        /* Some Computation is performed here*/
     }
     #pragma omp critical
     {
        /* Output is calculated here*/
     }
}
```

In the above code, there are n input data and M hidden nodes. This code can be easily explained through the execution diagram in Figure 4.4. Each vertical arrow represents an executable thread. The master thread starts the algorithm and executes the serial portion of the code before encountering the *parallel for* directive. The master thread invokes the loop creating *X-1* slave threads, X may be specified by the user. We assume in this diagram that $X = m$. The *X-1* slave threads together with the master thread creates X threads. These X threads divide the iterations of the *for loop* among themselves, with each thread executing a portion of the iterations. The parameter X can be varied according to the needs of the application. In this algorithm, when $X = M$, we obtain pure fine-grained parallelism.

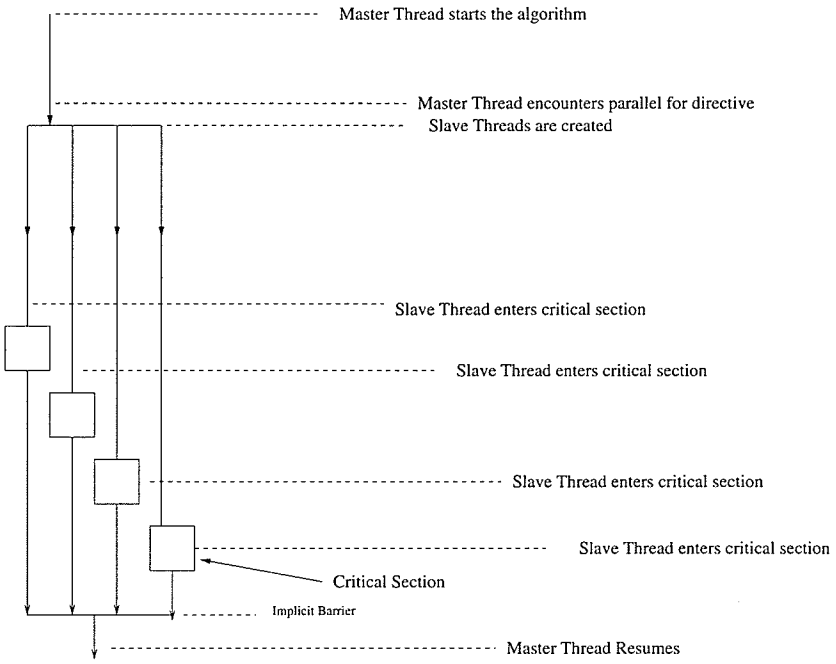

Figure 4.4: Loop Level Parallelism in OpenMP

At the end of every iteration, the slave threads enter the mutually exclusive critical section to calculate the output. Note that we have explicitly inserted this directive for synchronization purposes to control access to the shared variable. After each thread has completed its portion of the iterations, the thread waits at the end of the *parallel for construct* for all the other threads to finish. OpenMP adds an implicit barrier at the end of the parallel constructs. The threads are released of this barrier condition, once all the threads finish and synchronize at the barrier. At this point, the slave threads no longer exist and the master thread resumes execution of the code at the end of the *parallel for* construct.

4.4.2 Coarse-Grained Parallelism

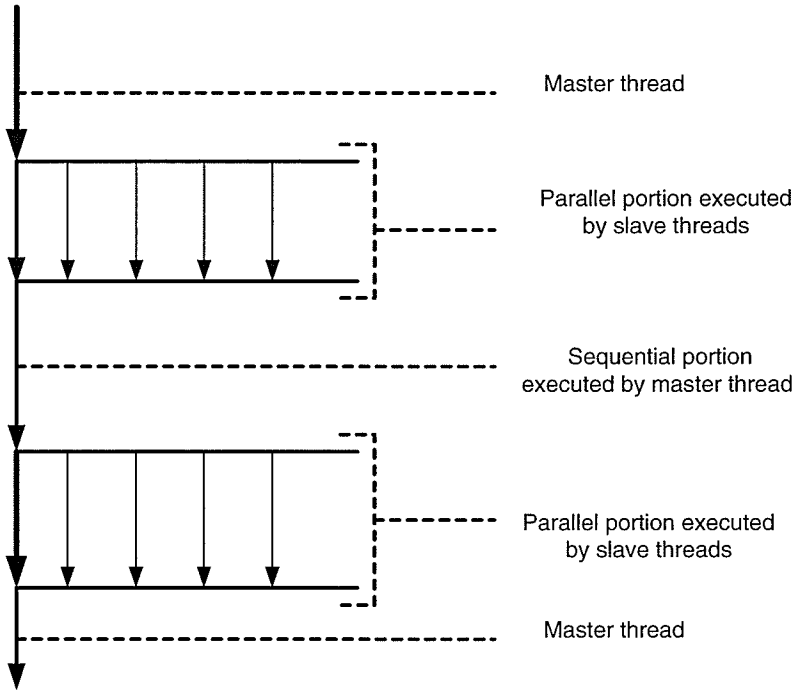

Figure 4.5: Parallel Region in OpenMP

In this approach, we explicitly intervene by partitioning the workload into various sections and distributing them to the threads. This is performed in OpenMP using the concept of *parallel region*. The *parallel* directive defines a parallel region shown below. The number of threads to execute the parallel region can be defined by the user within the code. The code within the *parallel* directive is executed concurrently and asynchronously by the threads. For example, with T threads and n nodes, each thread can be allowed to execute on $\frac{n}{T}$ nodes. The same code is replicated on each thread. Each thread, therefore, has its own data environment. The threads and the data environments disappear at the

end of the parallel region and the master thread resumes execution.

```
for i = 1 to number_of_epochs do {
  #pragma omp parallel
    {
       id=omp_get_thread_num() /* number of the thread
                           specified by user*/
     /* Calculation of activations of hidden neurons done in
        parallel fashion */
    }
    /* Activation calculation of output neuron and error*/
    /* calculation of output and hidden neurons are done*/
    /* sequentially by the master processor */
    #pragma omp parallel
      {
        id=omp_get_thread_num();
        /* Weight update of the weights connecting to*/
        /* hidden-input and hidden-output neurons are*/
        /* done in parallel fashion.*/

      }
}
```

There are two parallel regions and one sequential region. The above code can be best explained using Figure 4.5. The master thread starts the serial execution as before. When it encounters the first parallel region to compute the activations of the hidden neurons, a team of slave threads is created to execute this parallel region concurrently. The code within this parallel region is replicated on every thread. At the end of the parallel region, the master thread performs the activation calculations of the output neuron and the error calculations of the output and hidden neurons sequentially. When the master thread encounters the parallel region for the second time, a team of slave threads are again created. The code within this parallel region, updating weights of its neurons, is replicated on each slave thread as before.

The amount of computation and communication required for the neural network training process is determined theoretically first. This is done in order to get an idea of the feasibility of implementation of the algorithms on parallel architectures for their practicality.

In the next chapter we derive the theoretical limits of these four algorithms based on the computation and communication requirements.

Chapter 5

Theoretical Analysis

In this section we present some of the analytical results of the algorithms.

5.1 Distributed Approach

In an SPMD model, a parallel backpropagation algorithm consists of the following steps for both training set and neuron parallelisms: each processor performs the computations on its local data set; communicates and exchanges information between the processors; and applies a barrier synchronization to perform the final computation.

The processors in a Beowulf cluster are connected by a ring and every processor requires to broadcast its message to every other processor, say to $P-1$ processors. To analyze the time complexity for two algorithms in the distributed approach, we assume that t_s is the startup time and t_c is the communication cost for a fixed length message (floating point) to traverse the interconnection network. We also assume that t_b is the

average time required for the parallel machine to complete a barrier synchronization.

The training process consists of several batches of inputs from initial data set to the network. This requires several epochs. The training time is the product of the number of epochs and the time consumed for each epoch. During each epoch, the following computations are performed: calculation of the activation for each neuron; calculation of error gradient with respect to weight; and updating the weights.

Given n neurons in the input layer and M neurons in the hidden layer, each hidden neuron computes a weighted sum with n inputs and evaluates a bipolar sigmoid activation function. Each hidden neuron is connected by an edge from all n input neurons with a weight w_i. Therefore, for M neurons, the total number of weights connecting the input layer to the hidden layer is $Mn = W_1$. The number of weights connecting from the hidden layer to the output is $M = W_2$. Since W_2 is essentially much less than W_1, we will ignore W_2. Therefore, the total number of weights in the network is $W = Mn$. The computation of the error gradient at each neuron, involves each weight in the network, which is approximately W. The time required to update each weight, in the final step, is therefore W.

5.1.1 Training Set Parallelism

For accuracy of the neural network results, each of the the $\frac{N}{P}$ data set is further subdivided into l sets and distributed to the P processors. The computation time T_{c1}, for evaluating the network and computing the error gradient is $\theta(\frac{N}{P}W)$. Each processor then broadcasts

the calculated error gradient to each of the $P-1$ processors to update every weight in the network. The total communication time is, therefore, $T_{comm} = \theta(Wl(P-1)(t_s+t_c))$. The total time required to update the weights, W, for l steps is $T_{c2} = \theta(Wl(P-1))$. Time required for barrier synchronization is l. Therefore, the total time for the training set parallelism (TSP) approach, including communications and computations is:

$$\begin{aligned} Total_{TSP} &= T_{c1} + T_{c2} + T_{comm} + T_{barrier} \\ &= \theta(\frac{N}{P}W) + \theta(Wl(P-1)(t_s+t_c)) + \theta(Wl(P-1)) + \theta(lt_b). \end{aligned}$$

5.1.2 Neuron Parallelism

In this algorithm, M hidden neurons are partitioned so that $\frac{M}{P}$ hidden neurons are distributed to P processors. The total computation time is $\theta(\frac{W}{P}N)$. Each processor accumulates the values of its hidden neurons and distributes the sum to the other $P-1$ processors for activation calculation of output neuron. This communication phase takes $\theta((t_s+t_c)(P-1)N)$. Before broadcasting the accumulated sum, a barrier is introduced to perform the final computation at the output neuron. The total synchronization required is Nt_b. Thus, the total time for the NP is:

$$\begin{aligned} Total_{NP} &= T_{comp} + T_{comm} + T_{barrier} \\ &= \theta(\frac{W}{P}N) + \theta((t_s+t_c)(P-1)N) + \theta(Nt_b). \end{aligned}$$

5.2 Shared-Memory Multithreaded Approach

In this section, we derive the time complexity of both fine-grained/loop level parallelism and coarse-grained parallelism.

5.2.1 Fine Grained or Loop-Level Parallelism

For the code discussed in section 4.4.1, the number of threads created is the number of hidden neurons, M. Only the outer loop is parallelized. In loop-level parallelism, the scheduler within the system distributes the work load among the processors trying to evenly balance the load. Each thread then enters a critical section. However, OpenMP does not provide fair access to the critical section, meaning that the threads are not guaranteed access in some fashion such as first-in first-out or round robin. OpenMP makes sure that a thread will be granted access to the critical section eventually. Under this circumstance, a thread may be starved since other threads may get repeated access to the critical section.

In our problem, threads enter the critical section only once. In the worst case, a thread T, will enter the critical section after all the other $M-1$ threads have had access. Then the waiting time to compute the critical section (CS) is the product of the time to compute the instructions within the section, t_{CS}, and the number of threads ahead of T $(= T \times t_{CS})$.

In shared memory, there is no communication between processors or a barrier synchronization. Each thread is fine-grained and performs computation independently and

concurrently. As mentioned in section 4.4.1, all threads have an implicit barrier (IB), however. This is additional overhead. Let us assume that each thread T takes time t to finish the whole algorithm. Let t_{IB} be the associated time for all threads to reach the end of the loop. As we explain later, the overhead of the critical section waiting time and the implicit barrier, play a major role in the performance of the multithreaded algorithm in loop-level parallelism.

Therefore, the total computation time for fine-grained/loop level parallelism (LLP) with the overhead incurred is:

$$\begin{aligned} Total_{LLP} &= T_{comp} + T_{CS} + T_{overhead} \\ &= (t + t_{CS} * (M-1) + t_{IB})N \end{aligned}$$

5.2.2 Coarse-Grained Parallelism

In this approach, given M neurons, we predetermine the number of threads required and allocate data to these threads. For example, with T threads and n nodes, each thread can be allowed to execute on $\frac{n}{T}$ nodes. Each thread executes the same code. In section 4.4.2, we have shown the code for this approach. We notice that there are two parallel regions and one sequential region. The activation calculation of the hidden neurons is performed in parallel. The time required for this is $\frac{W}{T}N$. At the end of the activation calculations, all threads join the master thread. At this point the master thread resumes execution of the serial part. OpenMP implicitly adds a barrier at the end of the parallel region

directive. Let t_{PRT} be the time required by all threads to join the master thread. The time taken for sequential step to calculate the activations of output neuron and error calculation of output and hidden neurons is $2M$. Finally the weight updates are done in parallel. This takes time $\frac{W}{T}N$. Since there is no critical section, there is no additional overhead and there is no communication because of shared-memory. Therefore, the total computation time for coarse-grained parallelism (CGP) for N data set is:

$$\begin{aligned} Total_{CGP} &= T_{parallel} + T_{serial} + T_{overhead} \\ &= (2 \times \frac{W}{T} + 2M + t_{PRT})N \end{aligned}$$

This is a first attempt of neural network implementation on OpenMP. It is very difficult to analyze the multithreaded algorithm in general. There are no theoretical models such as those (BSP [35], LOGP [7]) available for SPMD. However, we use the above derived complexity results as a basis to analyze the performance results we present in the next chapter.

Chapter 6

Performance Results

In this section we describe the performance results of the parallel implementations on an 8 node Beowulf cluster with MPI and on an 8 processor SMP with OpenMP. Each node in a Beowulf cluster consists of two Pentium III processors with speed of 501.146 MHz and has a memory of 526 MB with cache size of 512 KB. The SMP has Pentium III processors with speed of 700.011MHz and has a memory of 764 MB with cache size 1024 KB. The inputs for the neural network for NP approach are daily stock prices for 200 consecutive days. The output is the predicted stock price for the following day. The number of hidden neurons is about 75% of input neurons, that is 152 neurons. For TSP we use a small network of 25 input nodes representing stock prices of 25 days, 16 hidden nodes and 1 output node that predicts the output for the following day. We used bipolar sigmoid as the activation function. Using the data set of 2080 daily prices for a real-life stock with a mean and standard deviation, the inputs are normalized and fed to the input

units of the neural network. The data were obtained from Great-West Life. We initialize the weights using Nguyen-Widrow initialization [20] which helped the neural network for fast learning rather than a fully random initialization. Some of our results are already published [27] and under review for publication [28].

6.1 ARMA Model

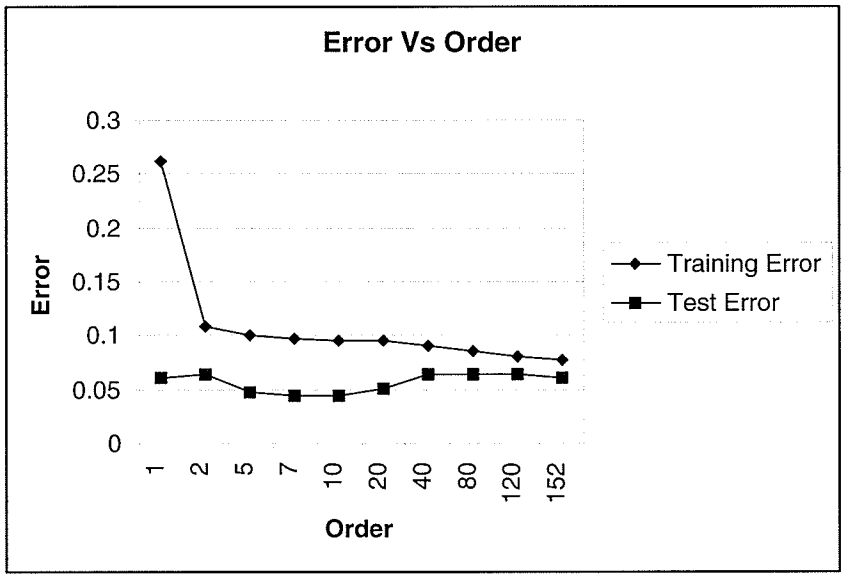

Figure 6.1: Autoregression Model with varying order

Figure 6.1 shows the comparison of our results with one of the traditional time-series models namely the ARMA model. The figure depicts the error with respect to order p of the AR(p) model. The order p is the total number of previous stock prices included for the prediction of the current stock price. We run the adaptive filtering AR model 1000 times to get the optimal set of autoregressive coefficients. It clearly shows that the

testing error decreases as we consider more lagged variables. For our 2080 data set, we get the lowest training error when we consider 152 lagged variables. The error in this case is only 0.076. With the help of autoregressive coefficients, we tried to predict stock prices for successive seven days and obtained a testing error of 0.061463. Both these errors are calculated from the difference of square of the predicted stock and the actual stock price for that day. We get an optimal test error for successive seven days with 10 lagged variables with an error of 0.043908.

One important observation from this figure is that the testing errors are better than the training errors, a counter intuitive phenomenon. This could be due to the fact that test data size is small and very structured. Since our aim is only on the parallel neural network results, we do not pay attention to this phenomenon.

6.2 Distributed Approach

Figure 6.2 presents the execution time of the NP in MPI with respect to the number of processors at various epochs. There is a gradual decrease in execution time as we increase the number of processors for different epochs. In NP, we partition and distribute the hidden neurons to the processors. The processors execute the neural network algorithm on their local data sets for a set of 200 data each time until all the data is fed in. In this approach, there is a communication overhead of $(t_s + t_c)$*7*2080=14560*$(t_s + t_c)$ for 8 processors and 2080 data. This communication is required in each of the 1000 epochs and hence leads to a very large overhead for the overall performance of the algorithm.

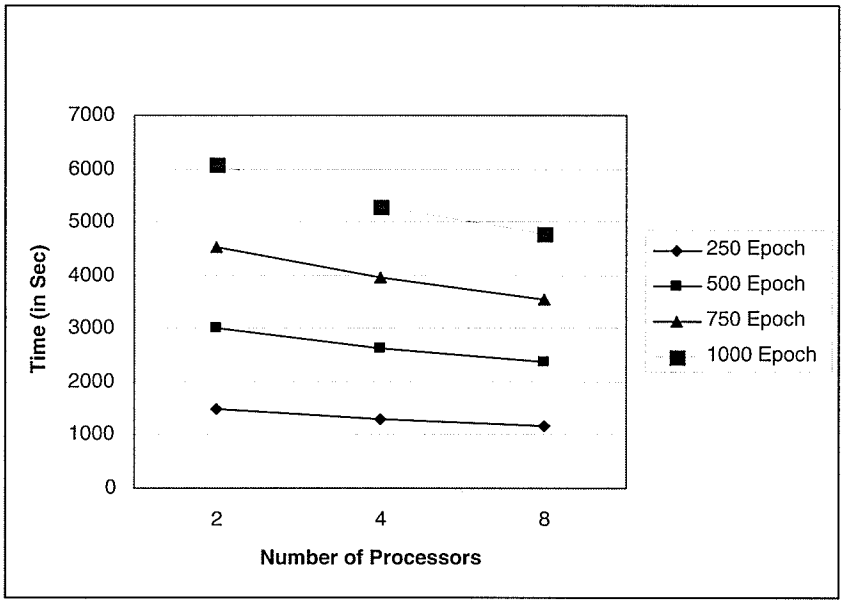

Figure 6.2: Execution Time vs. Number of Processors for Neuron Parallelism in MPI

Also, the time required for synchronization per epoch is $2080t_b$. Again, for 1000 epochs (last epoch) this is a large overhead. However, the parallel implementation does produce good performance as we increase machine size from 2 to 8 processors.

Figure 6.3 presents the execution time of the TSP algorithm for MPI with respect to number of processors for various epochs. In this approach, the input data set (2080) is partitioned and distributed to the processors. We observe a significant difference in execution time compared to the NP algorithm in MPI shown in Figure 6.2. On 8 processors, the execution time for TSP with 15000 epochs (last epoch) is only 700 secs and in NP with only 1000 epochs, it is already close to 5000 seconds. (see Figure 6.2). The communication time (25 inputs and 16 hidden neurons) is only $Wl(P-1)(t_s+t_c) =$ (25*16) $\times$ (4) $\times$ (7) $\times(t_s+t_c) = 11200 \times(t_s+t_c)$, where l is 4 and the synchronization time

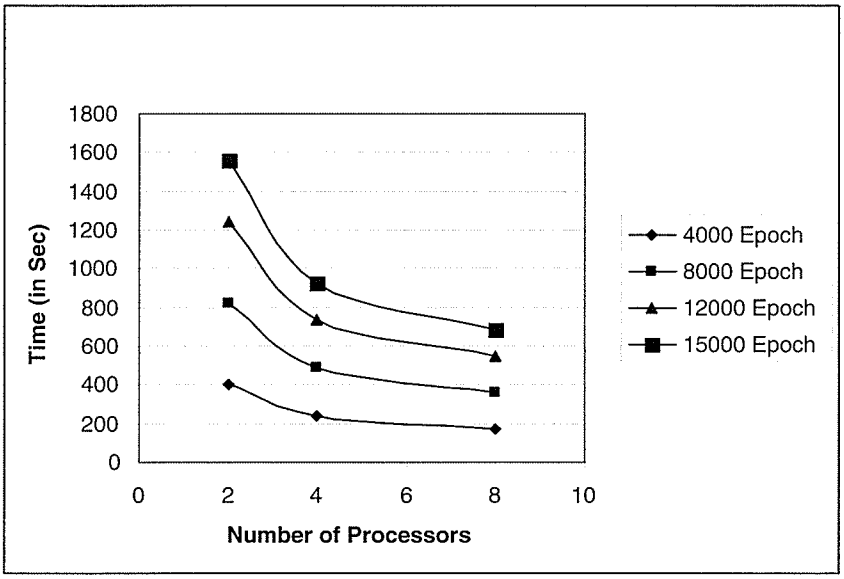

Figure 6.3: Execution Time vs. Number of Processors for Training Set Parallelism in MPI

compared to NP is only $4 \times t_b$. This has a significant impact on the overall execution time of algorithm. In addition, since the network size is small, due to domain decomposition, the computation time is smaller than that of a larger network as in NP. This implies better scalability of the TSP algorithm.

A useful indication of the convergence of a neural network is the monitoring of the *correlation* between the target and observed output layer values over each epoch during training given by [9]:

$$r = \frac{\sum_{i=1}^{N_e} T_i O_i - N_e \bar{T} \bar{O}}{\sqrt{\left(\sum_{i=1}^{N_e} O_i^2 - N_e \bar{O}^2\right)\left(\sum_{i=1}^{N_e} T_i^2 - N_e \bar{T}^2\right)}}$$

where O_i is defined as the output for input dataset i, T_i is defined as the target output for input dataset i, N_e is the total number of input datasets, $\bar{O}$ is the average output

and $\bar{T}$ is the average target output. The coefficient r will lie in the range [-1,1]. A small value of r near 0 implies no association and +1 means the output and target values are strongly correlated and indication of the convergence of neural network towards a global minimum.

Experiments with NP in MPI indicated that the correlation coefficient gradually increases as we increase the number of epochs. For example, in the second epoch, the correlation coefficient is 0.77 while at the last epoch (1000), the correlation is 0.999005. Also, note that the coefficients are positively correlated, which implies that the neural network converges as desired.

6.3 Shared-Memory Multithreaded Approach

Figure 6.4 shows the timing results for fine-grained algorithm for NP in OpenMP for five epochs. We achieve the best performance with approximately 60 threads after which the execution time starts to increase. This we attribute to the load imbalance and the synchronization cost of entering the critical section. In loop-level parallelism, only the outer loop is parallelized (section 4.4.1). Also, there is an explicit barrier at the end of each loop, which is added by the system (OpenMP compiler). The critical section that we need to add explicitly for synchronization purposes is an additional overhead. For a large number of threads, the implicit barrier time t_{IB} is greater than that required for a small number of threads. The computation time decreases when we adopt a large number of threads. However, the overhead incurred due to synchronization barrier dominates the

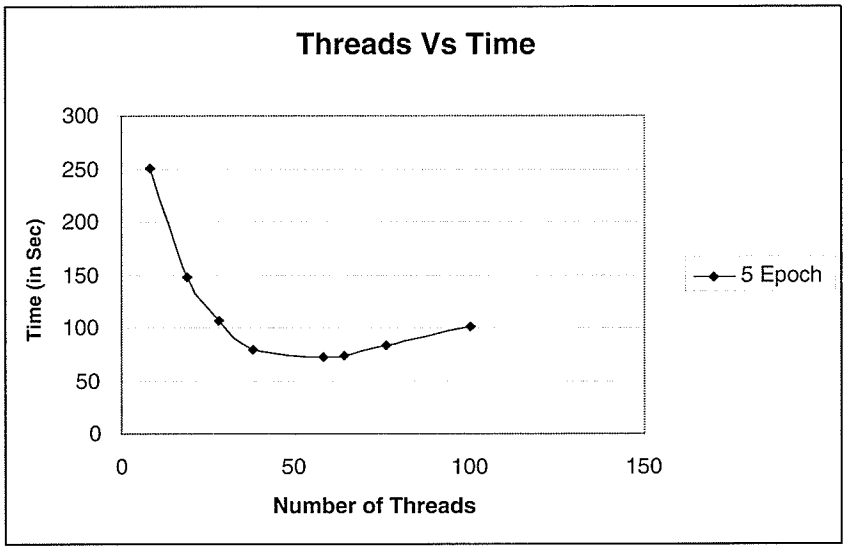

Figure 6.4: Execution Time vs. Number of Processors for Fine-Grained Parallelism in OpenMP

entire computation, for example if we used 100 threads. All threads have to wait until the critical section is completed. The time taken for this is 152*t_{CS} for 152 threads. One of the reasons for the downtrend in the execution time for this approach is that the parallelized loops are not distributed evenly (by OpenMP scheduler) among the processors. Due to the poor performance results of this approach, we have conducted all our experiments (training and testing) with coarse-grained algorithm in OpenMP and compared our results with MPI on this algorithm. Note that since we did not find any significant improvement in the accuracy of the results for more epochs, the expertimental results are given using 5 epochs.

Figure 6.5 presents the timing results for coarse-grained parallelism in OpenMP with respect to number of processors and various epochs. We notice in relation to the NP in

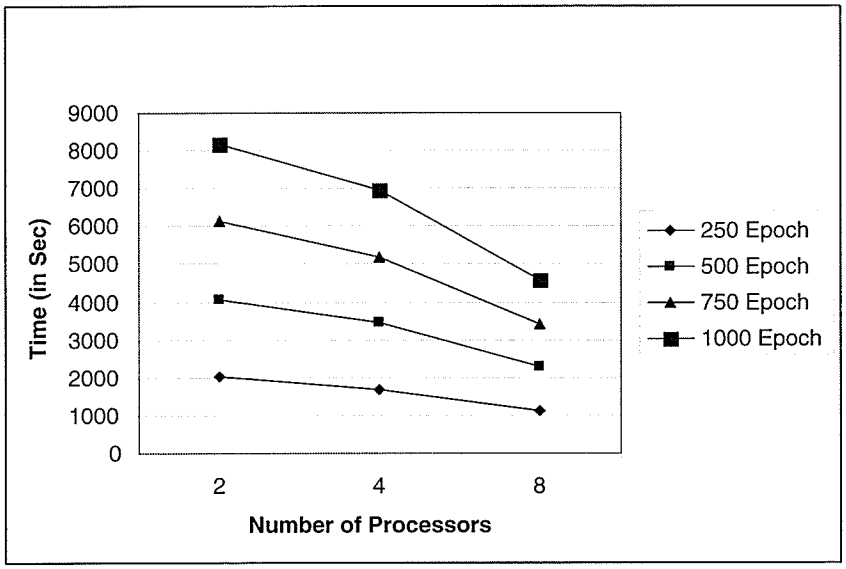

Figure 6.5: Execution Time vs. Number of Processors for Coarse-Grained Parallelism in OpenMP

MPI (Figure 6.2), the execution time decreases as we increase the number of processors. The execution time for 1000 epochs (last epoch) on 8 processors is approximately the same in both cases.

6.4 Comparison

Figures 6.6, 6.7, 6.8 and 6.9 capture the training and testing errors in four different implementations against epochs. Since the number of epochs required in neuron and training-set parallelism is different, the training errors are plotted in separate figures. We compare the errors of the algorithms (neuron parallelism, coarse grained parallelism) in Figure 6.6 with AR Model. As it is clear from this figure neural network gives better

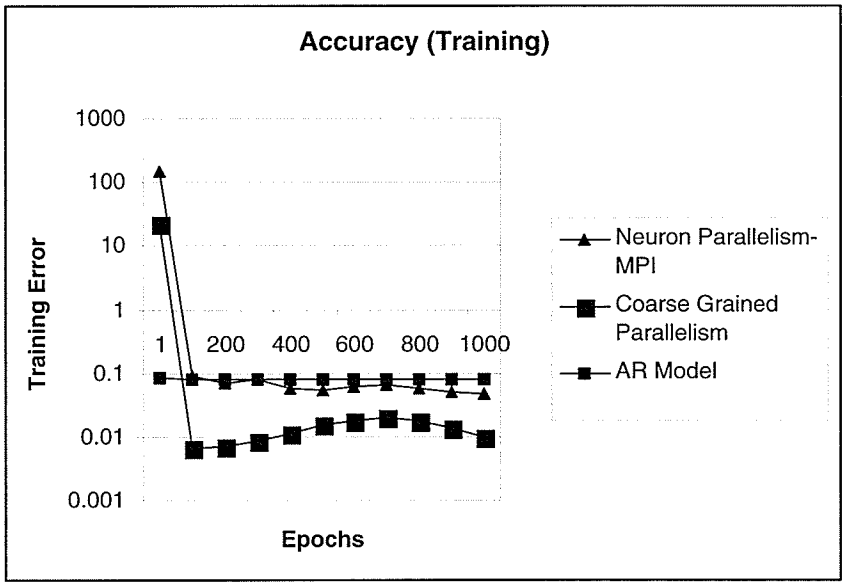

Figure 6.6: Training Errors for Neuron, Coarse Grained Parallelism and AR Model

accuracy when compared to the traditional AR model. This is also true with training set parallelism (as shown in Figure 6.7) though the scale is different in terms of number of epochs. For 1500 epochs the training set parallelism gives better accuracy than the AR model. In addition, we observe the following points from these figures with respect to the parallel implementation.

Figure 6.6, 6.7 demonstrate the training error for the parallel implementations in MPI, OpenMP and the AR model at various epochs. We notice that OpenMP produces the best training results. The AR model stagnates at about 0.08. Adaptive filtering of the AR model gives a good acceleration if the coefficients are randomly chosen. Here, the coefficients for regression analysis given by Wales Yulker [20] equation yields good estimation of the coefficients for regression analysis. We see that adaptive filtering does not

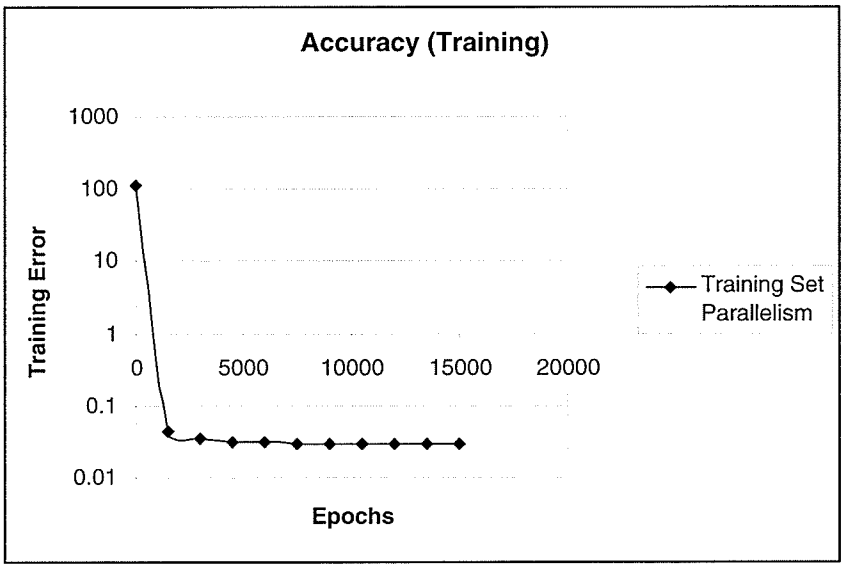

Figure 6.7: Training Errors for Training-set Parallelism

improve the initial estimation of coefficients. Further, in order to remove discrepancies, the data was normalized by taking the difference from the mean. For the calculation of coefficients of regression analysis, we used the difference values of successive stock prices. We observe from this figure that all three algorithms train the neural network with better accuracy. In other words, our algorithms do not compromise on the training errors of the neural network. However, since the focus is on the development of parallel algorithms for neural network training process we do not pay attention to the testing error as depicted in Figure 6.8, 6.9. However, we make the following observations.

Figure 6.8, 6.9 demonstrate the testing error with 8 processors at various epochs. The network was trained with 2080 data set but tested with 7 data set each with 200 inputs. The network training depends on the weight initialization as well as gradient error. We notice that in MPI, NP version converges very fast. For NP, each processor

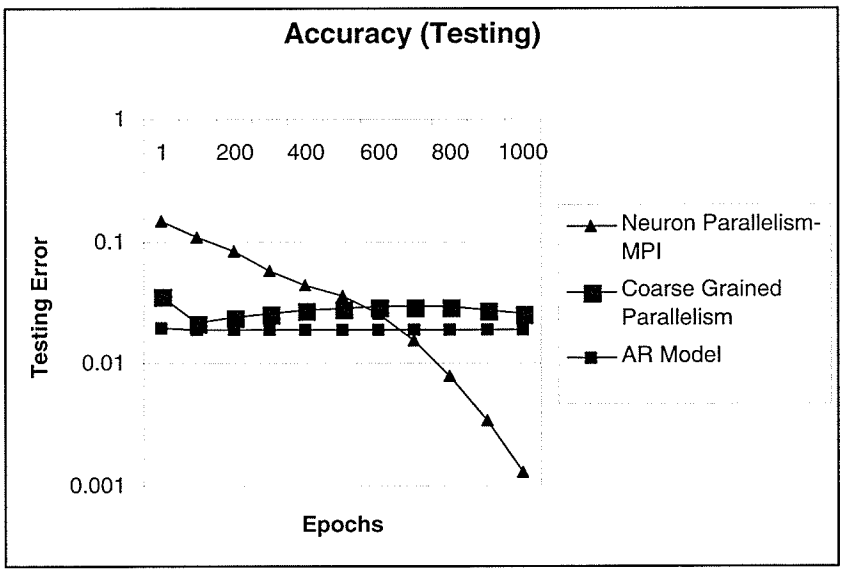

Figure 6.8: Testing Errors for Neuron, Coarse Grained Parallelism and AR Model

initializes its weight matrix and updates its weights in each iteration independently. Since the data is distributed among the processors in MPI implementation, the independent weight initialization helps in reducing the error accumulation and hence helps in faster convergence. In other words, with independent weight initialization, successive stock price computations produce higher correlation value in relation to the target value.

Figure 6.10 presents the comparison of the three algorithms on 8 processors and at the end of the last epoch in each case. For NP in MPI and CGP in OpenMP, the last epoch is 1000 and for the TSP, the last epoch is 15000 (for 1000 epochs we noticed that this approach was too fast, with very small execution time). Even with 15000 epochs, we notice that the TSP gives the best execution time.

For NP in MPI and OpenMP the backpropagation algorithm trains a big network of 200 input nodes, 152 hidden nodes and one output node, which requires a large com-

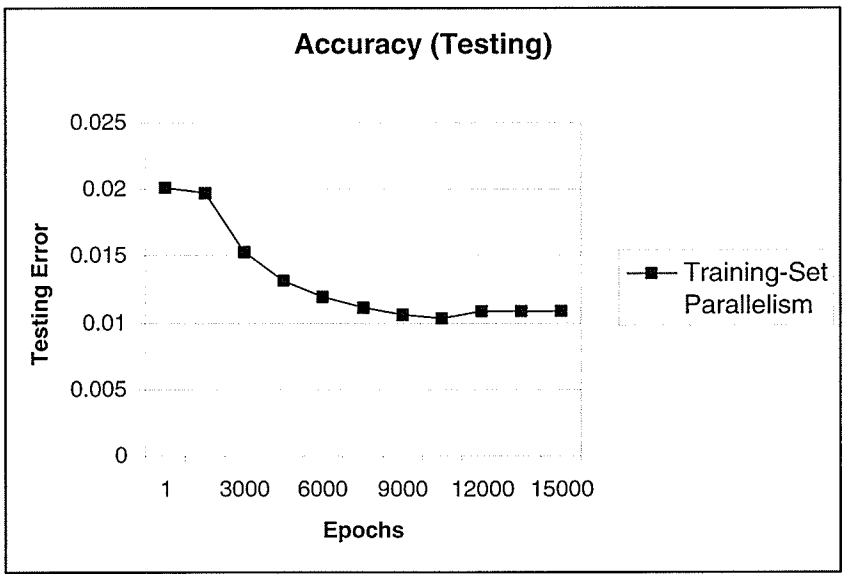

Figure 6.9: Testing Errors for Training-set Parallelism

putation. On the other hand, in TSP, the network is of small size, so the computation time is less than that of a bigger network. For NP in OpenMP, we get a better execution time than NP in MPI because there is less associated communication cost in the shared memory architecture as broadcast is eliminated by a common shared variable in critical section.

Algorithm Type	Exec. Time (sec)	Speedup (T_{seq}/T_P)	Training error	Testing error
MPI(NP)1000 epoch	4773.65	2.4	0.048	0.00132
MPI (TSP)15000 epoch	683.28	2.32	0.029	0.01808
OpenMP(CGP)1000 epoch	4574.23	2.5	0.01	0.019

Table 6.1: Comparison of the three techniques

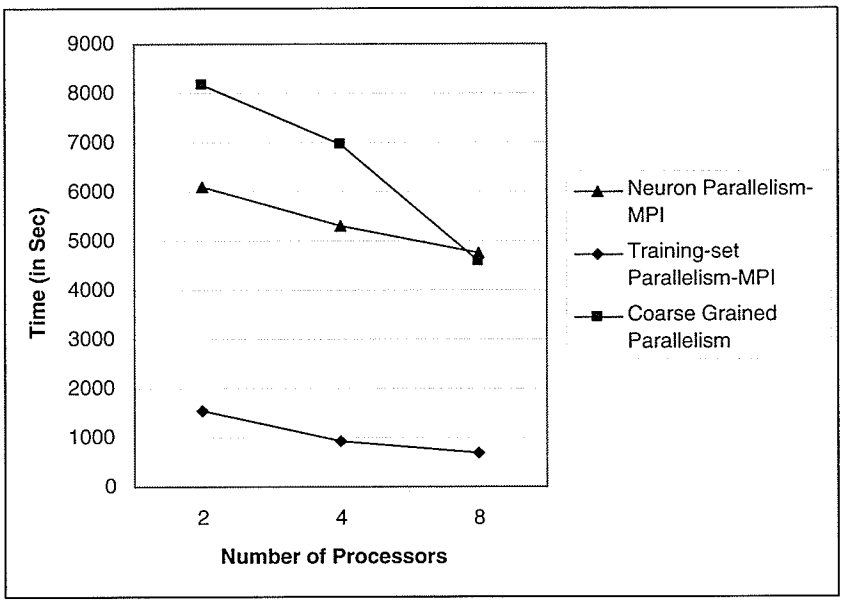

Figure 6.10: Execution Time vs. Number of Processors for all Three Parallelisms

Table 6.1 shows the comparison of the three algorithms at the end of the last epoch in each case for 8 processors. The learning rate for NP and CGP is 0.01 and the momentum is 0.04; for TSP the learning rate is 0.7 and the momentum is 0.6. We define the learning rate as a scale factor that informs how far to move in the direction of gradient. Momentum is a parameter that can be defined as the fraction of previous weight change that should be added to the current weight to proceed with a constant weight change in the same direction. The absolute speedup for TSP with 15000 epochs is 2.32 while for NP in MPI and OpenMP it is 2.4 and 2.5 respectively. For a given architecture of neural network and backpropagation algorithm we have to work with a fixed problem size (for TSP) or fixed hidden neurons (for NP). In order to get a close to linear speed up, the problem size (or neurons) should be increased significantly in proportion to the machine size (number

of processors) so that a large amount of computation will compensate the increased communication overhead. Accuracy is another concern of neural network. In TSP, data is divided into smaller chunks and distributed among a large number of processors. As a result, the computation time decreases but the communication cost dominates the overall efficiency. So we get a less speedup with TSP. Since the network is large, each processor has to do a large amount of computation, we get a better speed up of 2.4 for NP in MPI. The large computation compensates for the communication cost to some extent in this case. The same argument can be made for NP in OpenMP. Moreover, a relatively better speedup is achieved due to absence of broadcast operations and communication delay that are inherent with distributed approach in MPI. However, it does not provide very good speedup as the shared memory machine has its own problems like context switching between the threads, implicit synchronization, barrier for parallel regions, and improper load balance, which contribute to the overall performance degradation.

We can summarize our results by stating that the TSP approach outperforms all other approaches with reduced execution time. This is due to the fact that we decomposed the large input data (stock prices), and distributed chunks of the job to the various processors and thereby, working with smaller networks.

Chapter 7

Conclusion and Future Work

We have studied stock price forecasting problem by developing four different parallel training algorithms for backpropagation neural networks. Neuron (NP) and training set parallelism (TSP) algorithms are developed for distributed memory machines and implemented on an 8 node Beowulf cluster using the standard programming method called message passing interface (MPI). Fine-grained and coarse-grained multithreaded algorithms are developed for shared memory machines for NP algorithm and implemented on an 8 node SMP using the standard shared memory language called OpenMP.

Training the neural networks was shown to be accurate by:

- establishing a strong correlation between the network output and target value
- comparing with one of the traditional time series methods called autoregression model.

Comparison of the parallel algorithms with the traditional AR model for the train-

ing errors showed that the OpenMP version produces the best results. The AR model begins to stagnate at an error value of 0.08. The overall comparison between all three algorithms on 8 processors indicates that the execution time of the TSP algorithm in MPI outperforms all other algorithms. This, we attribute to the fact that network size in TSP is smaller and the computation, therefore, is faster.

The absolute speedup for all three algorithms are approximately 2.5. In the TSP algorithm, since the implicit data size is distributed among the processors but the network is replicated in every processor, the computation time is less. However, since the values have to be broadcast to all processors, communication increases the cost of of this algorithm and hence leads to a reduced overall performance with a speedup of only 2.32. On the other hand, the neuron parallelism technique in MPI also produces a speedup of 2.4, slightly higher than TSP. In NP, the task (data) size per processor is large and leads to less communication overhead. However, the computation in each processor is coarse-grained. As the problem size increases, the computation time also increases. Therefore, we do not see a significant difference between speedups in TSP and NP. With the SPMD model of computation followed in a distributed memory machine, the TSP and NP algorithms are coarse-grained. It is a fact that in such machines, the algorithm should be designed to avoid extensive communications. We tried this using the NP method. However, in neural network, the problem has an inherent synchronization step when proceeding from one layer to the next. In this case, we have to block all processors from proceeding to the next layer until the computations in the current layer is completed. This synchronization

latency is expensive. This is the reason behind the observed minimal speedup in the NP and TSP algorithms with MPI.

The fine-grained algorithm did not produce good results in OpenMP. This is because we do not have control over the number of iterations that a thread can be allocated. This is performed dynamically by the system. In OpenMP, there are implicit barriers added by the system for the synchronization purposes and they add to the total overhead. The performance stagnated after 60 threads. For this application having more than 60 threads does not improve the performance.

In the coarse-grained algorithm for OpenMP, we experimented with different thread sizes. We distributed data to the threads. In other words, distribution of data to the threads is left to the user. The speedup of 2.5 in this case is slightly higher than the NP version with MPI. This is due to the shared memory architecture, where broadcasting is not necessary. While this is good, neural network still has synchronization step as mentioned before and this, we feel is the primary bottleneck for all the algorithms. Synchronization consumes a significant portion of the execution time of the algorithms. In addition, context switching between threads in OpenMP, which takes few cycles of operation, contributes to the overall performance degradation, when there are plenty of threads.

There are opportunities to improve the performance with other types of algorithms such as pipelining the network. However, as a first attempt to parallelize neural networks for financial forecasting application, the current results are very encouraging in relation

to the overall neural network training timings with traditional regression models.

For example, in *pipelining*, the different weight layers are computed in different processing elements. First, the hidden layer processor computes output values of training pattern (a). The output processor reads the values and computes and the error values of (a). The hidden layer processor concurrently processes the next training pattern (b). Then it reads the hidden error for (a) and both processors accumulate the weight change values for (a). This method pipelines the forward phase with the backward phase and therefore, improves the concurrency in the application. This requires a major modification in the algorithm and left as future work.

The theoretical analysis could be studied in depth to find the cost of the execution for a given platform on which the algorithm is implemented. This requires a study of the system level details to calculate the actual theoretical timings of the parameters mentioned in chapter 5, such as t_s, t_c, t_{IB} etc.

A rigorous statistical testing of the results is in order for practicability of the algorithms. In other words, the testing results need to be substantiated further which is left as our important extension of this thesis work.

Bibliography

[1] Arvind and R.A Iannucci. Two fundamental issues in multiprocessing. Technical report, Computation Structure Group Memo 226-6, Lab for Computer Science, MIT, MA, May 1991.

[2] K.E Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, pages 836–840, September 1980.

[3] BBN. Technical report, BBN Advanced Computers Inc., TC-2000 Technical Product Summary, Cambridge, MA, 1989.

[4] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Run Time System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, California, July 1995.

[5] CM5. The CM-5 Technical Summary - Thinking Machines Corporation. Technical report, Cambridge, MA, 1991.

[6] Cray. Cray Inc.: News and Higlights. Technical report, Cray: http://www.cray.com, 2002.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, San Diego, CA, May 1993.

[8] S. Dutta and S. Shekhar. Bond rating: a non-conservative application of neural networks. In *IEEE Ineternational Conference on Neural Networks*, pages 527–554, 1994.

[9] L. Fausett. *Fundamentals of Neural Network: Architecture, Algorithms, and Applications.* Prentice Hall, Upper Saddle River, NJ, 1994.

[10] F.M. Ham and I. Kostanic. *Principles of Neurocomputing for Science and Engineering.* McGraw Hill Higher Education, New York, NY, 2001.

[11] L. Hamm, B.W. Brorsen, and R. Sharda. Futures Trading with a Neural Network. In *Proceedings of the NCR-134 Conference on Applied Commodity Analysis, Price Forecasting, and Market Risk Management*, pages 486–296, 1993.

[12] K. Holden, D.A.Peel, and J.L Thompson. *Economic Forecasting: an Introduction.* Cambridge University Press, NewYork, NY, 1990.

[13] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[14] I. Kaastra and M. Boyd. Designing a Neural Network for Forecasting Financial and Economic Time Series. *Neurocomputing*, 10:215–236, 1996.

[15] A. Kanas. Neural network linear forecasts for stock returns. *Financial Economics*, 6:245–254, 2001.

[16] T. Kimito, K. Asakawa, and N. Takeoka. Stock Market Prediction System with Modular Neural Networks. In *IEEE Ineternational Joint Conference on Neural Networks*, pages 11–16, 1990.

[17] N. Kohzadi, M. Boyd, I. Kaastra, B.S. Kermanshahi, and D. Scuse. Neural Networks for Forecasting: An Introduction. *Canadian Journal of Agricultural Economics*, 43:463–474, 1995.

[18] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings, Redwood City, CA, USA, 1994.

[19] Leiserson C. Cilk. In *http://supertech.lcs.mit.edu/cilk*, 1999.

[20] S. Makridakis and S. Wheelright. *Forecasting Methods and Applications.* John Wiley Sons, New York, USA, 1978.

[21] W.S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Imminent in Nervous Activity.* In Neurocomputing: Foundations and Research (Ed: Anderson and Rosenberg) Chapter 2, pp.18-28, The MIT Press. (Reprint of the 'Bulletin of the Mathematical Biophysics' vol.5, pp.115-133, 1943), 1988.

[22] S. Moshiri, N. Cameron, and D. Scuse. Static, Dynamic and Hybrid Neural Networks in Forecasting Inflation. In *Proc. of 5th Symp. of Soceity for Non-Linear Dynamics and Econometrics*, April 1997.

[23] nCUBE Coporation. ncube 6400 processor manual. Technical report, Beaverton, OR, 1990.

[24] J. R. Nicolls. The design of the MasPar MP-1: A cost-effective massively parallel computer. *IEEE Digest of Papers-Comcom*, pages 25–28, 1990.

[25] Paragon. Paragon xp/s product overview - supercomputer systems division, intel corporation. Technical report, Beaverton, OR, 1991.

[26] A. Petrowski, G. Dreyfus, and C. Girault. Performance Analysis of a Pipelined Backpropagation Parallel Algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, Novemeber 1993.

[27] Mohammad R. Rahman, Ruppa K. Thulasiram, and Parimala Thulasiraman. Forecasting Stock Prices using Neural Networks on a Beowulf Cluster. In *Proceedings of the* 14^{th} *IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 470–475, Boston, MA, Nov 2002.

[28] Mohammad R. Rahman, Ruppa K. Thulasiram, and Parimala Thulasiraman. A Study of Parallel Neural Network Algorithms for Stock Price Forecasting on MPI and OpenMP. In *Proceedings of the International Parallel and Distributed Processing Symposium (Under Review)*, Nice, France, April 2003.

[29] E. Rich and K. Knight. *Artificial Intelligence.* Tata McGraw Hill, New Delhi, India, 1993.

[30] R. O. Rogers and D.B. Skillicorn. Using the BSP Cost Model for Optimal Parallel Neural Network Training. In *Proceedings of the BioSP3 Workshop held in conjunction with IPPS/SPDP98*, San Juan, Puerto Rico, April 1998.

[31] N. Sundararajan and P. Saratchandran. *Parallel Architectures for Artificial Neural Networks - Paradigms and Implementation.* IEEE Computer Society, Los Alamitos, CA, 1998.

[32] Y. Takefuji. *Neural Network Parallel Computing.* Kluwer Academic, Norwell, MA, USA, 1992.

[33] K. B. Theobald, R. Kumar, G. Agrawal, G. Heber, R. K. Thulasiram, and G. R. Gao. Implementation and Evaluation of a Communication Intensive Application on the EARTH Multithreaded Architecture. *Concurrency and Computation: Practice and Experience*, 14:183–201, March 2002.

[34] R. Trippi and E. Turban. *Neural Networks in Finance and Investing.* Probus, Chicago,IL, USA, 1993.

[35] L.G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.

[36] A.S. Weigend, Y.S. Abu-Mostafa, and A.-P.N. Refenes. *Decision Technologies for Financial Engineering.* World Scientific, NewYork, NY, 1997.

[37] H. White. Learning in Neural Networks: A Statistical Perspective. *Neural Computation*, 4:425–464, 1989.

[38] H. White. Economic Prediction Using Neural Networks: The Case of IBM Dalily Stock Returns. In *IEEE Ineternational Conference on Neural Networks*, pages 451–458, July 1990.

[39] H. White, A.R. Gallant, K. Hornik, and J. Wooldridge. *Artificial Neural Networks: Approximation and Learning Theory.* Blackwell, Cambridge, MA, USA, 1992.

[40] Yoon and Y. Swales. A comparision of discriminant analysis vs artificial neural network. *Operaltional Research Society*, 1:51–60, 1993.