Parallel Implementation of Clique Partitioning Using Artificial Neural Networks

by

Hisham Hajj

Submitted in partial fulfillment of the requirements

for the Degree of Master of Science

Thesis Advisor: Dr. H. Harmanani

Department of Computer Science

LEBANESE AMERICAN UNIVERITY

Jully 2000

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the project of

Hisham Hajj

candidate for the *Master of Science* degree*.

(signed) [redacted]

(chair) [redacted]

date 3/7/2000

*We also certify that written approval has been obtained for any proprietary material contained therein.

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Contents

List of Figures

List of Tables

Acknowledgments

Special thanks to

My Family and Mariette for their support, and specially my father who provided me the best education.

Dr. Walid Keirouz for the resources, ideas, and the 24 hours, 7 days a week support.

Dr. Haidar Harmanani for all the ideas and the resources he provided.

Mr. Mussallam, Mr. Khalife, and all my teachers at LAU for the best education.

Mr. Mujaber for making the center and its resources available at all times.

Abstract

This work presents a parallel algorithm that simulates a neural network. The neural network solves the Clique Partitioning Problem, an NP-complete problem. Clique partitioning is used to solve the Data Path allocation problem, an important part of High Level Synthesis in VLSI circuits design.

Chapter 1

Introduction

The increasing level of integration in electronic devices leads to highly complex systems. Circuits with over 1 million devices are now common (e.g., Intel Pentium). Such circuits are called VLSI circuits (Very Large Scale Integration). VLSI circuits are almost impossible to repair, thus it is very important to design these circuits correctly. Designing complex circuits is capital-intensive. Furthermore, achieving a zero-defect design requires larger efforts.

The creation of an integrated circuit is divided into four stages: design, testing, fabrication, and packaging. The design stage is divided into three major steps: Modeling, Synthesis and Optimization, and Validation. Modeling starts with an idea, and transforms this idea into an abstract model that captures the functionality of the circuit. Synthesis and Optimization transform the abstract model into a detailed model with all the features required for fabrication. Validation verifies the consistency of the model used during design.

Synthesis techniques speed up the design stage and reduce the human effort. Optimization techniques enhance the design quality. The data path allocation problem is an important part of Synthesis. The data path allocation problem is the distribution of resources to the operations of the circuit being designed. One efficient way to solve the data path allocation problem is the

use of Clique partitioning The difficulty with the clique partitioning problem is that it is an NP complete problem [2], that is the time needed to solve the problem grows exponentially when the size of the problem grows.

This work develops a Hopfield neural network model to solve the data path allocation problem using clique partitioning, and then simulates the neural network using a parallel algorithm. The report is organized as follows: Chapter 2 provides background literature. It introduces High Level Synthesis, Neural Networks, Parallel Processing,and Beowulf clusters, topics needed and used in this work. Chapter 3 presents the Neural Network model and the parallel algorithm that simulates the neural network. Chapter 4 presents tests results. Chapter 5 provides the conclusion and suggests approaches for future research.

Chapter 2

Background

This chapter introduces topics related and used in this work. The topics introduced are High-Level Synthesis, Neural Networks, Parallel Processing, and Beowulf clusters.

2.1 High-Level Synthesis

High-Level Synthesis (HLS) generates a structural view from a behavioral description. Behavioral description is the description of the circuit using a discriptive language, like English for instance. The structural view is the description of the circuit using a structural language, such as VHDL. HLS begins with a behavioral description of a circuit then schedules the operations to come out with a data path. The data path describes an interconnection of resources, steering logic circuits, and registers or memory to store data. Resources implement the arithmetic functions or the logic functions specified in the behavioral description, and steering logic circuits are used to send data to the appropriate destination.

The scheduling of the resources is set according to constraints specified in the behavioral description of the circuit. Two major constraints exist: time constraint (or delay constraint) and resource constraint (or area constraint). The time constraint considers the issue of getting

a solution in the shortest time, ignoring the issue of having enough area in the design to store the resources. When dealing with time constraint, the depth of the data path is minimal, and most operations will be executing in parallel. Figure 2.1 illustrates the result of scheduling the Data Path under time constraint. Every operation is denoted by a vertex V_i. An edge exist between V_i and V_j if V_i depends on V_j, that is V_i cannot execute unless V_j has done executing. Table 2.1 contains a summary of the operations time. The design under time constraint needs 4 multipliers, and 2 ALUs. On the other hand, resource constraint minimizes the use of resources, thus reducing the area of the design. The resource constraint ignores the time needed to reach a solution. Figure 2.2 illustrates the result of rescheduling the graph in Figure 2.1 under resource constraint. In this case, the design needs only one ALU and one multiplier (see Table 2.2).

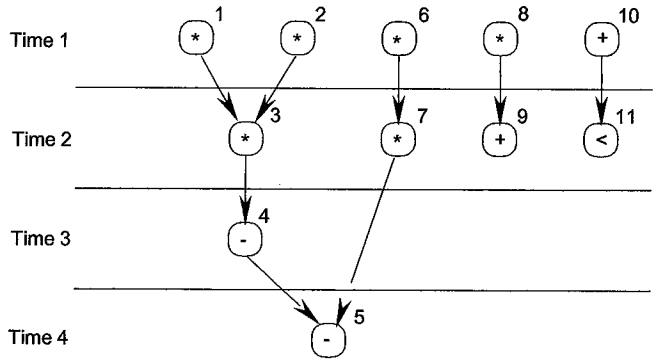

Figure 2.1: Data Path Under Time Constraint

Multiply	ALU	Start Time
V_1, V_2, V_6, V_8	V_{10}	1
V_3, V_7	V_9, V_{11}	2
-	V_4	3
-	V_5	4

Table 2.1: Operation Time Under Time Constraint

A composition of the two constraints is very familiar, since usually there are a time constraint, and a resource constraint to satisfy. Several scheduling algorithms, like ASAP, or the ALAP, ad-

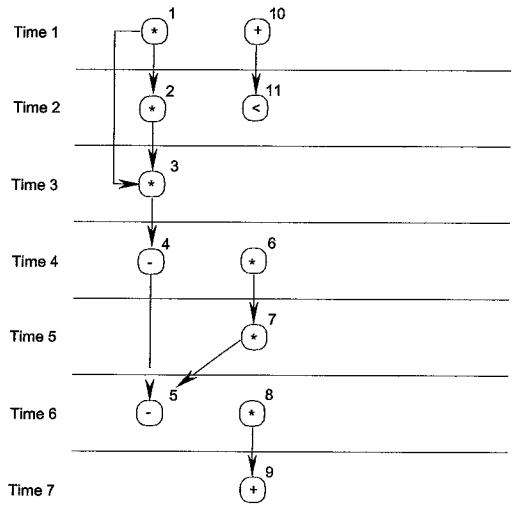

Figure 2.2: Data Path under Resource Constraint

Multiply	ALU	Start Time
V_1	V_{10}	1
V_2	V_{11}	2
V_3	-	3
V_6	V_4	4
V_7	-	5
V_8	V_5	6
-	V_9	7

Table 2.2: Operation Time under Resource Constraint

dressing unconstrained and constrained problems were proposed. After the data path is complete, the allocation of the resources takes place. A resource is assigned to one or several operations, by allowing multiple non-concurrent operations to share the same hardware operator resource. The sharing of resources leads to the problem of finding a mapping between the operations and the resources such as to minimize the number of resources used. Several algorithms can be used, among which is the clique partitioning problem.

2.2 Neural Networks

Warren McCulloch and Walter Pitts [8] presented the first model of artificial neurons presented in 1943. Since, more sophisticated proposals were made. Artificial neural networks try to model the information processing capabilities of the nervous system. Although biological neurons are slow compared to electronic logic gates, the brain is capable of solving problems that no computer can solve effectively.

To help understand how artificial neurons model biological neurons a brief introduction about biological neurons is hereby presented.

2.2.1 Biological Neurons

The general structure of a biological neuron is shown in Figure **??**. Biological neurons receive signals and produce a response. The Dendrites are the channels for receiving information. Dendrites receive the signal at the contact regions with other cells. These contact regions are called the Synapses. In the body of the cell, Organelles produce the chemicals necessary for the work of the neuron. The mitochondria produce chemicals consumed by other cell structures. It can be thought of as part of the energy supply of the cell. The axon transmits the output signal. Every cell has at most one axon.

2.2.2 Artificial Neurons

Artificial neurons adopt the minimal structure of a biological neuron: an input channel, a cell body, and an output channel. Contact points between the cell body and the input or output connections simulate synapses. Figure 2.3 shows an abstract neuron with n inputs. Each input channel i can have a value x_i. f is a primitive function selected arbitrarily. Usually, every input channel i has a corresponding weight w_i. Incoming information x_i is multiplied by the weight, and the primitive function is then evaluated. Artificial networks could be thought of as networks of primitive functions. Figure 2.4 illustrates a typical artificial neural network.

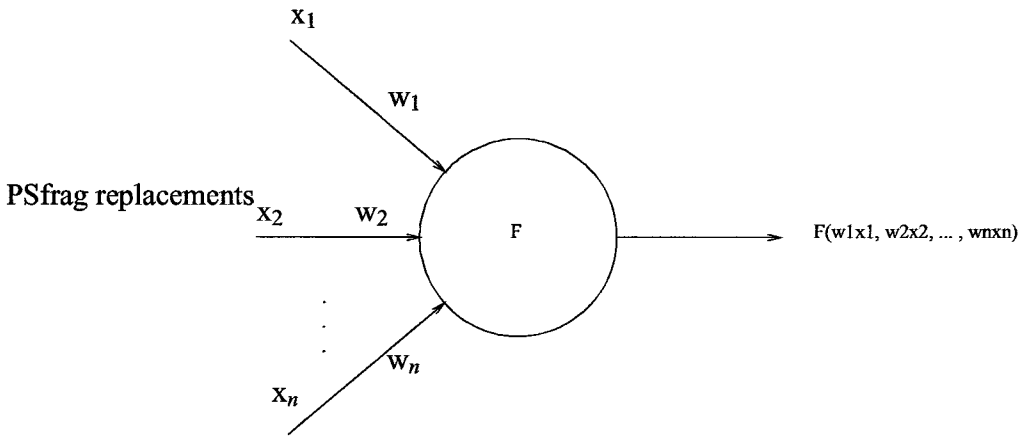

Figure 2.3: Artificial Neuron

Φ is the function to be evaluated, and x, y, z are the input values. Each node, or neuron, implements a primitive function $f_1, f_2, f_3, \ldots$ which are combined to produce Φ. The function Φ is the network function. Different weight values produce different network functions. Thus, the most important elements of an artificial neural network are:

- the structure of the nodes,
- the topology of the network, and
- the learning algorithm used to find the weights.

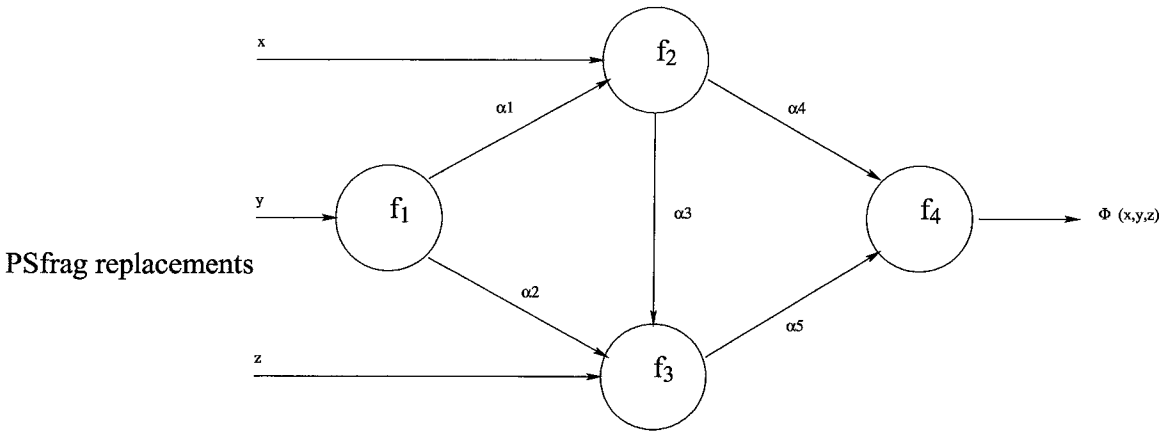

Figure 2.4: Artificial Neural Network

McCulloch-Pitts neurons use binary signals, zeros and ones. The input is divided into two types, excitatory and inhibitory forces. The inputs are represented by edges, where the inhibitory forces are marked with a circle at the end of the edge. A threshold value is provided for every neuron. Figure 2.5 shows an abstract presentation of the McCulloch-Pitts neuron. X_1 to X_n are the inputs, that arrive at the white half of the neuron, where Θ is specified, and the black half gives the result.

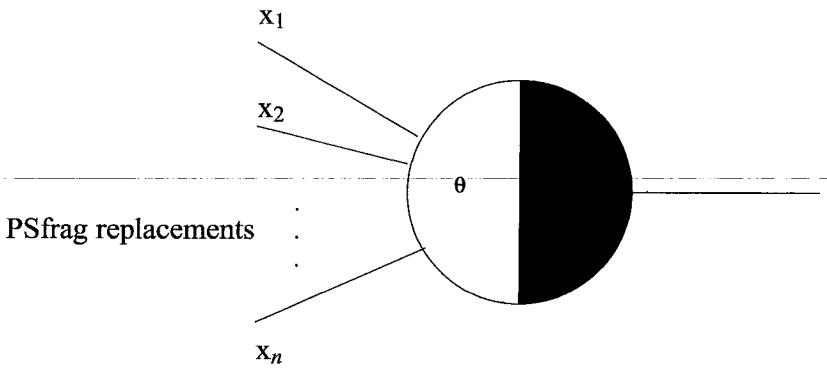

Figure 2.5: McCulloch-Pitts Neuron

The following rules evaluate the output of the McCulloch-Pitts neuron:

- Each neuron receives input on the excitatory edges X_1 to X_n, and on the inhibitory edges Y_1

to Y_m.

- The result is inhibited to 0 if at least one of the inhibitory edges Y_1 to Y_m is 1.
- Otherwise, compute the sum of all excitatory edges, $S = \sum_{i=1}^{n} X_i$ and then compare S to Θ. If $(S <= \Theta)$ then the output is 1, otherwise the output is 0.

Figure 2.6 illustrates some logical functions and their implementation using McCulloch-Pitts neurons.

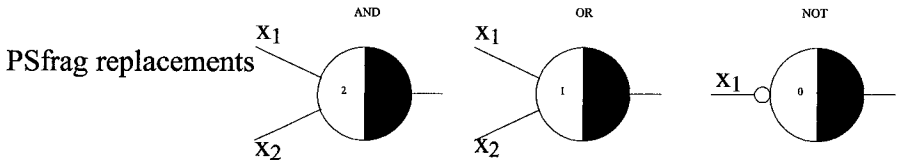

Figure 2.6: McCulloch-Pitts Neurons Implementing Logical Functions

Frank Rosenblatt, proposed in 1958 a more general computational model than McCulloch-Pitts neurons, the perceptron[8]. The perceptron introduced essentially two novelties, numerical weights and a special interconnection pattern. Weights are used for learning purposes, and the connections are determined stochastically.

2.2.3 Learning

Neural Networks use a learning algorithm to organize the neurons in order to implement a desired behavior. Certain input-output mapping of the network are presented, and a correction step is executed iteratively until the desired response is reached. In order to reach a solution, the Artificial Neural Network parameters are changed and adapted by the learning algorithm according to previous experience results.

Learning algorithms are divided into two categories, supervised and unsupervised. Supervised learning is achieved using a teacher process that observes the output of the network, and

computes the derivation from the expected answer. The learning algorithm defines the magnitude of the error according to which the weights are corrected.

Unsupervised Learning

When the result of a certain input is not known, unsupervised algorithms are used to reach the solution. In this case, the network organizes itself and corrects its weights to be able to reach the solution. Two classes of unsupervised learning exist: reinforcement and competitive learning. In the first method, the input produces a *reinforcement* for the weights enhancing the reproduction of the desired result. In competitive learning, each neuron *competes* to provide an output. The neuron that produces the answer inhibits all others.

For example, consider a set of vectors are to be classified into clusters. In Figure 2.7 is a bundle of vectors that needs to be clustered. It is possible to find three vectors V_1, V_2, and V_3, that represent the vectors in each of the clusters. The representative vectors are not far apart form the vectors they represent. Each weight vector V_i corresponds to a neuron.

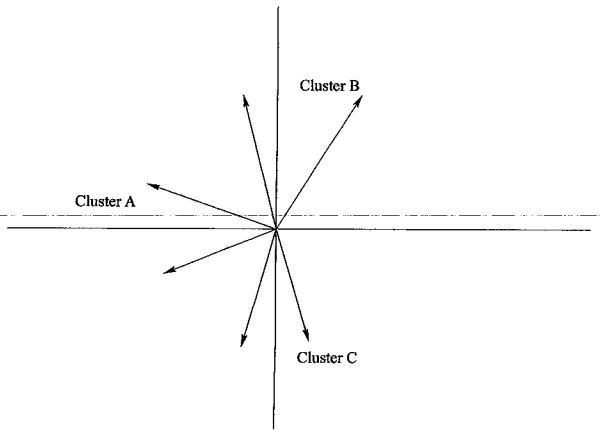

Figure 2.7: Bundle of Vectors to Be Clustered

In this case, the number of clusters is known in advance. In general, the number of clusters is unknown. If the number of clusters is unknown, then so is the number of weight vectors, and thus

the number of computing units. This is what is known by the clustering problem. What follows is a solution to this problem using unsupervised competitive learning. The Neural Network in Figure 2.8 solves the clustering problem illustrated in Figure 2.7. Since the number of clusters in this case is known in advance, a unit is assigned to each cluster thus the use of 3 units. The inputs x_1 and x_2 are processed by the units, and only the unit with the largest excitation is allowed to fire a 1, while inhibiting the other units through the lateral connections. A firing units implies that the vector is part of the cluster the unit represents. This solution assumes that the number of clusters is known in advance. In the case where the number of clusters is not known, the neural network should contains as many units as there are vectors (i.e assume the worst case, each vector is in a separate cluster). If the number of clusters turns out to be less than the number of units, and this is the most probable case, the unused units are called dead units.

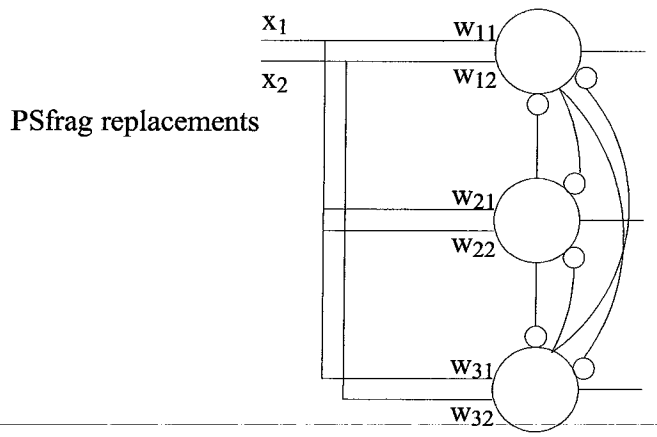

Figure 2.8: Neural Network to Solve the Clustering Problem

2.2.4 Hopfield Neural Networks

In 1982, Hopfield introduced the collective computational property of an artificial neural network and later proposed in 1984 a continuous output model and showed the circuit for implementing it. He used an NP-complete problem, the Traveling Salesman Problem (TSP), to illustrate his

model.

A look at Bi-directional Associative Memory (BAM) helps understand the Hopfield model. BAM is a network constituted of two layers of neurons that send information between them using bi-directional edges. Figure 2.9 illustrates an n-dimensional row vector x, and a k-dimensional row vector y. The neurons compute the sign function, and information is encoded using bipolar values.

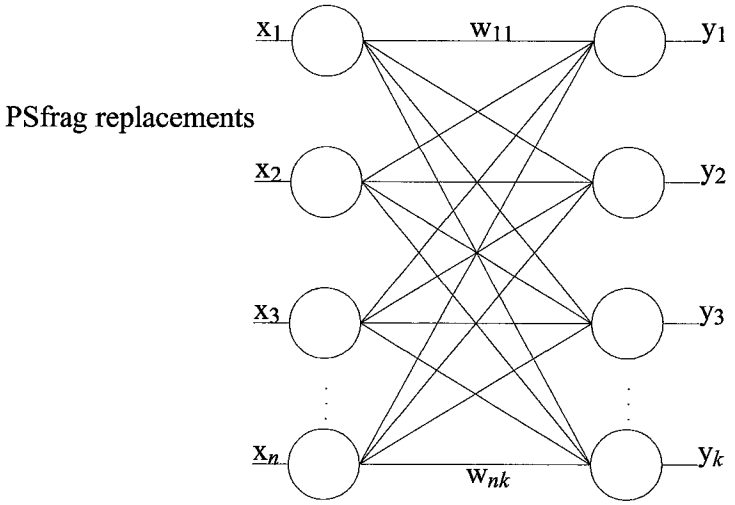

Figure 2.9: Bi-directional Associative Memory

W is the $n * k$ weight matrix. The computed value in the first step is written as follows:

$$y_0 = sign(x_0 W)$$

Now, y_0 is the input for x, and the new computation will be:

$$x_1^T = sign(W y_0^T)$$

Therefore, the new computation of y produces:

$$y_1 = sign(x_1 W)$$

At iteration m, the system has already computed m+1 vector pairs (x_0, y_0), ..., (x_1, y_1). All the vector pairs fulfill the following constraints:

$$y_i = sign(x_i W)$$

and

$x_{i+1}^T = sign(Wy_i^T)$

A fix point is reached after several iterations, when the following condition is satisfied:

$y = sign(xW)$

and

$x^T = sign(Wy^T)$

With this in mind, now is the time to introduce the concept of the energy function. Assume that for a certain BAM, the vector pair (x, y) is a stable state. Starting with the initial vector x_0 the network converges to (x, y) after a certain number of iterations. The vector y_0 is computed according to $y_0 = sign(x_0W)$. Now, y_0 is used for a new iteration from the right, and the excitation of the left layer units will be summarized in the excitation vector e, computed according to the following:

$e^T = Wy_0$

The network is stable for the vector pair (x_0, y_0) if $sign(e) = x_0$. Vectors e close enough to x_0 will fulfill this condition. These vectors differ from x_0 by a small angle such that the product $x_0 * e^T$ is larger than the product of other further away vectors with the same length. Therefore, if (Wy_0^T) lies closer to x_0, then the product

$E = -x_0e^T = -x_0Wy_0^T$

will be smaller (because of the negative sign). E is the Energy function of the network, and is used as the guideline to convergence. The Energy function E is given by the following formula:

$$E = -\frac{1}{2}x_iWy_i^T \tag{2.1}$$

The $\frac{1}{2}$ factor is just a scaling constant.

The Hopfield model is a special case of a BAM. The Hopfield neural network is a one layer BAM where every neuron is connected to every other neuron except itself. Since there is a single bi-directional connection between neurons i and j, $w_{ij} = w_{ji}$. Furthermore, the diagonal of W (the weight matrix) is 0, since there is no connection between each neuron and itself. A simple Hopfield Network with four neurons is illustrated in Figure 2.10. A connection between two processors or neurons is established through a conductance T_{ij}, that transforms the voltage outputs of amplifier j to current input for amplifier i.

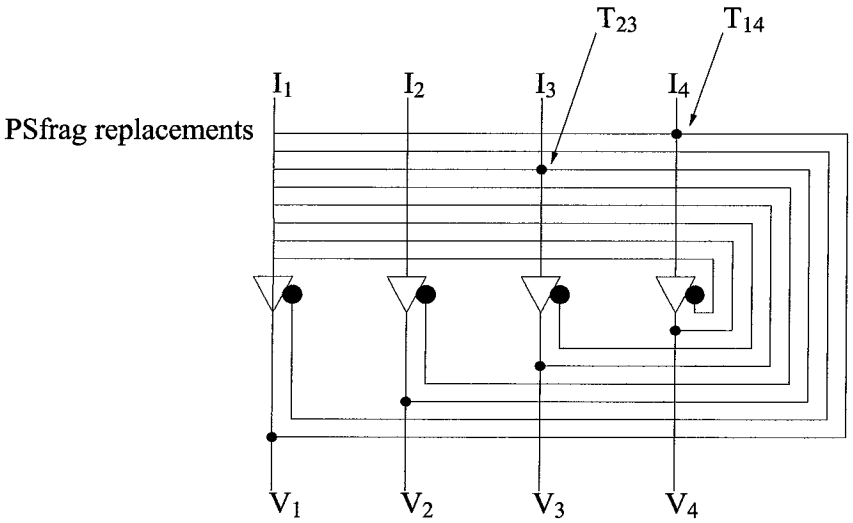

Figure 2.10: Hopfield Neural Network

Externally supplied bias current I_i is also present in every processor j. Each neuron i receives a weighted sum of the activation of other neurons in the network, and updates its activation according to the rule:

$$V_i = g(U_i) \tag{2.2}$$

Where $g(U_i)$ can be either a binary or a threshold function for the case of the McCulloch-Pitts

neurons.

Hopfield showed that in the case of symmetric connection ($T_{ij} = T_{ji}$) the motion equation for the activation of the neurons in a HNN always leads to a convergence to a stable state, in which the output voltages of all the amplifiers remain constant. The stable states of a network of N neuron units are the local minima of the energy function:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{ij}V_iV_j - \sum_{i=1}^{N} I_iV_i \tag{2.3}$$

Where V_i is the output of the i^{th} neuron and I_i is the externally supplied input or bias of the i^{th} neuron. E is referred to as the computational energy of the system. The equation of motion for the i^{th} neuron may be described in terms of the following energy function E as follows:

$$\frac{dU_i}{dt} = -\frac{U_i}{\tau} + \sum_{i \neq j} T_{ij}V_j + I_i \tag{2.4}$$

Where τ = RC is the time constant of the RC circuit connected to neuron i. Takefujji[6] showed that the above function performs the parallel gradient descent method. In fact, as long as the motion equation of the binary neurons is given by Equation 2.4 , the energy function E monotonically decreases. The state of the neural network is guaranteed to converge to the local minimum under the discrete numerical simulation.

Hopfield and Tank[6] were the first to use a neural network representation for solving NP-complete optimization problems. The difficulty in such optimization problems is that the best solution is computationally very hard and the time required to solve a problem on any computer, grows exponentially with the input size. Takefujji proposed several parallel algorithms to solve

a variety of optimization problems, including the graph planarization problem, the four coloring problem, and the tiling problem. Cimikowski and Shope[6] solved the graph layout problem that is similar to the planarization problem solved by Takefujji except that their algorithm yields an embedding that realizes the calculated number of crossings.

2.3 Parallel Processing and PVM

It is not a new idea to consider parallelism as a mean to speed up computation. In fact, since the earliest age of computers designers considered making several processors work together to resolve a problem. Technical developments in the 1980 s that parallelism feasible. The idea behind parallelism is to perform several operations simultaneously, in parallel. Parallel computing developed considerably, because of the increasing demand for computational power. This concept is applied, by most present-day computers. Parallel computing affected not only the architecture of computers, but also the connection and communication technology, algorithms, and languages.

The essential process of a computer is a sequence of instructions executed on a set of data. The multiplicity of the instruction stream and data are the criterion that classifies parallel computers. There are four classes of parallel computers:

- SIMD: Single Instruction flow, Multiple Data.
- SPMD: Single Process, Multiple Data.
- MISD: Multiple Instruction flow, Single Data.
- MIMD: Multiple Instruction flow, Multiple Data.

2.3.1 SIMD Architecture

In SIMD architecture, all the processing units receive the same instruction flow. Each processing unit execute the same instruction on a different set of data. This architecture is well suited for vector processing.

2.3.2 MIMD Architecture

Shared memory computers were the first really commercially successful computers. Imagine several processors sharing one large memory, where each processor executes independently from the others, and can exchange information with the others. The processors can execute several instruction flows on different data in parallel. Difficulties in accessing the memory limit the number of processors that can share the same memory. Performance degrades once there are more than ten processors. Distributed MIMD architecture, a solution to this problem, uses a rapid memory local to each processor, which is only connected to a certain number of neighboring processors. In the first MIMD model, processors communicate by sharing the data in the memory. In the latter model, the processors share data by exchanging messages, thus the execution time of a sequence no longer depends upon the execution time, but also on the communication cost between processors.

2.3.3 Interconnection Networks

The interconnection between processors and memory modules, or between processors themselves is a new arisen issue. Interconnection between processors can be achieved either by a direct connection (static network), or through shared-memory banks (dynamic network).

The simplest form of dynamic networks is the crossbar matrix. All the input lines cross all the output lines, and at every intersection, there is a switch. Figure 2.11 illustrates of a $4 * 4$ crossbar

matrix. Processor 1 is accessing memory bank 1, and processor 3 is accessing memory bank 2.

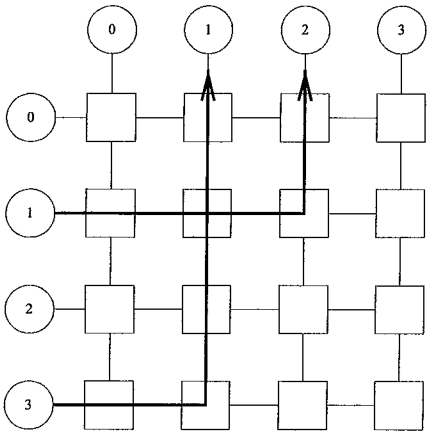

Figure 2.11: Crossbar Matrix

The C_{22} switch, illustrated in Figure 2.12, is widely used for implementing a crossbar matrix.

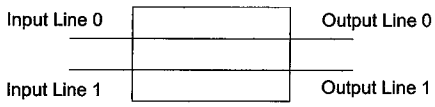

Figure 2.12: C_{22} Switch

In the case of a control of zero, the lines are connected directly, that is input 0 to output 0 and input 1 to output 1. If the control is one, then lines cross each other, that is input 0 to output 1 and input 1 to output 0. Figure 2.13 illustrates the main states of the switch.

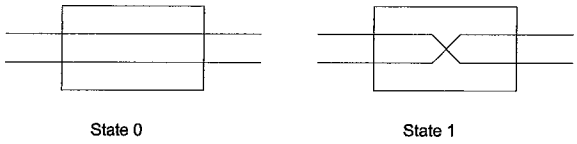

Figure 2.13: Main States of Switch

2.3.4 Static Networks

Most intercommunication networks are based on static topologies. Some of these topologies are discussed below.

Linear and Ring Topologies

Linear and Ring topologies are very popular due to their simplicity and the complexity of the results achieved by using them. In a linear network, nodes are connected to two of their neighbors, except for the first and last node. Figure 2.14 illustrates a linear network.

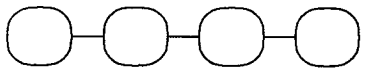

Figure 2.14: Linear Network

The result of connecting the first and last node together, is a ring topology shown in Figure 2.15.

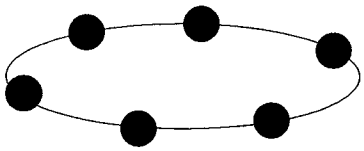

Figure 2.15: Ring Topology

Meshes and Toric Grids

Consider having several linear networks where the first node in the first network is linked to the first node in the second network, and so on. A Mesh is a network where every node has four neighbors, except for the first and last row and column, see Figure 2.16.

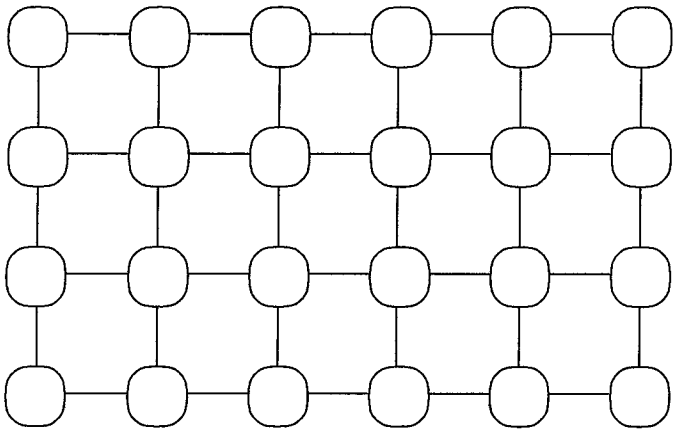

Figure 2.16: Mesh Topology

A toric grid is a mesh, where the first node in the first row is linked to the last node in the first row, and to the first node in the last row, such that every node has four neighbors (Figure 2.17).

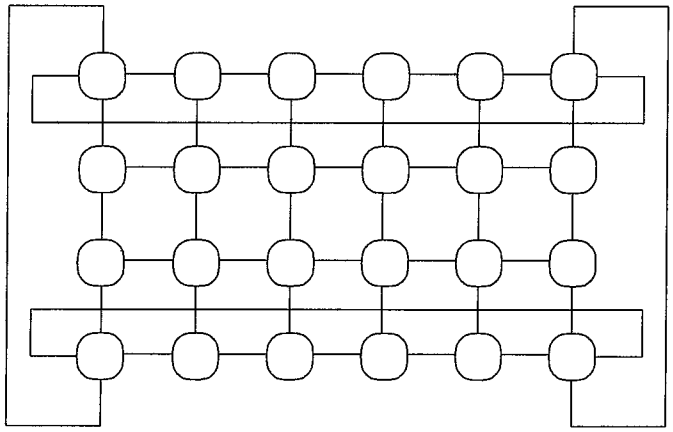

Figure 2.17: Toric Grid

Hypercube

A hypercube is a network of 2^d nodes where each node has d neighbors. A Hypercube with 2^d nodes is said to be of degree d. Figure 2.18 presents several basic illustrations of a hypercube, and Figure 2.19 illustrates a hypercube of degree 3.

Gray codes are a very convenient way to denote the nodes in a hypercube. d bits denote each

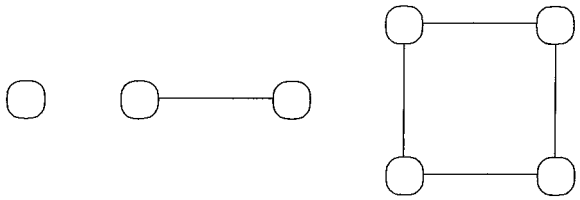

Figure 2.18: Several Hypercubes

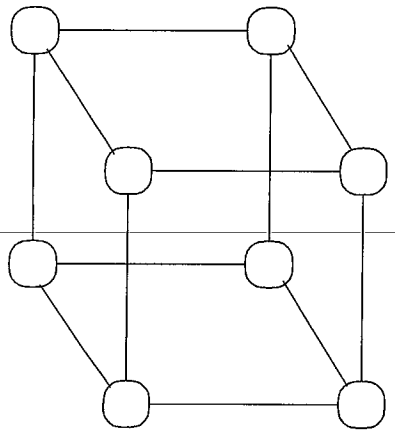

Figure 2.19: Degree 3 Hypercube

node in the network. All the neighbors of a certain node are obtained by complementing one bit at a time. For example, n_0 (000 in binary) has the following neighbors: n_1 (001), n_2 (010), and n_4 (100). Figure 2.20, illustrates a hypercube of degree 3, with the nodes denoted by Gray codes.

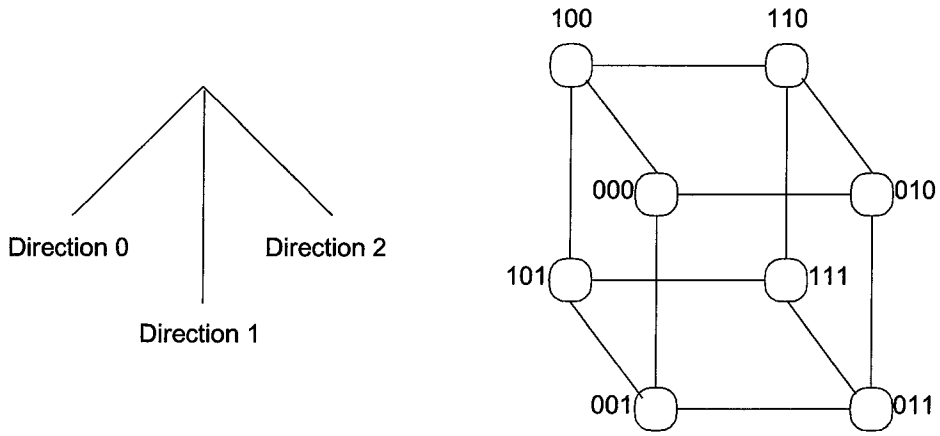

Figure 2.20: Hypercube using Gray Codes

Binary Tree

A binary tree is an acyclic connected graph, where every node has two children. A complete binary tree has 2^{n-1} vertices, and all the vertices, except for the leaves, have two children. Figure 2.21 shows a representation of a binary tree.

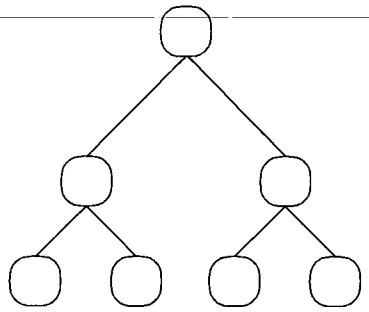

Figure 2.21: Binary Tree

The Binary ring tree is a binary tree, with the leafs connected to each other (Figure 2.22).

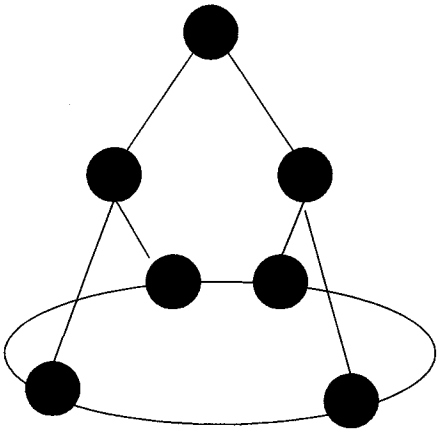

Figure 2.22: Binary Ring Tree

To denote the nodes in a binary tree, the two children keep the name of their parent, and append either a zero (for the left child), or a one (for the right child) as shown in Figure 2.23. As a result, the left nodes have a lower number than the nodes on the right, and this is convenient for several algorithms.

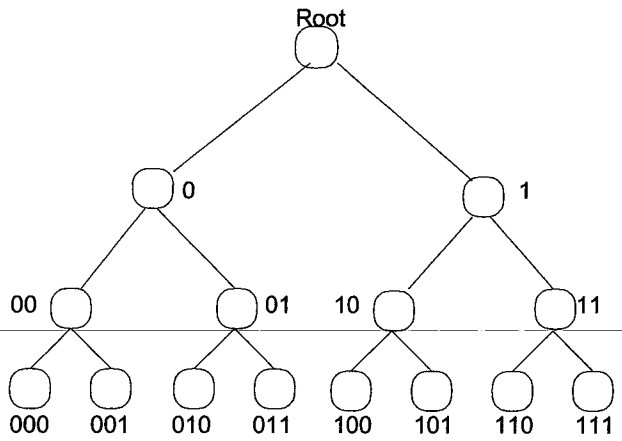

Figure 2.23: Denoting Binary Tree Nodes

Hybrid Topologies

By merging the topologies discussed above new hybrid topologies are created. Figure 2.24 illustrates a hybrid topology: a level 3 Hypercube where each node is replaced with a ring of 3

nodes.

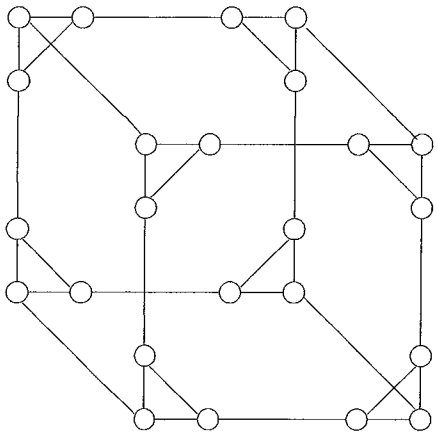

Figure 2.24: Merging Topologies

2.3.5 Message Switching

The weak point of parallel processing on distributed memory systems is communication. There exist several communication schemes taking advantage of the underlying network that connects the processors. What follows is a discussion of the most widely used communication schemes.

This method is also known as store and forward. Each processor stores an incoming message and checks if the message belongs to it. If it does not, the processor forwards the message to his neighbor. Usually, the message has two parts, the header that contains the path and destination, and the second part contains the data, as illustrated in Figure 2.25. Figure 2.26 illustrates the stages of transmitting a message from one processor to the other.

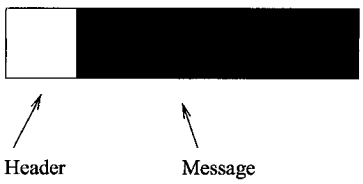

Figure 2.25: A basic message

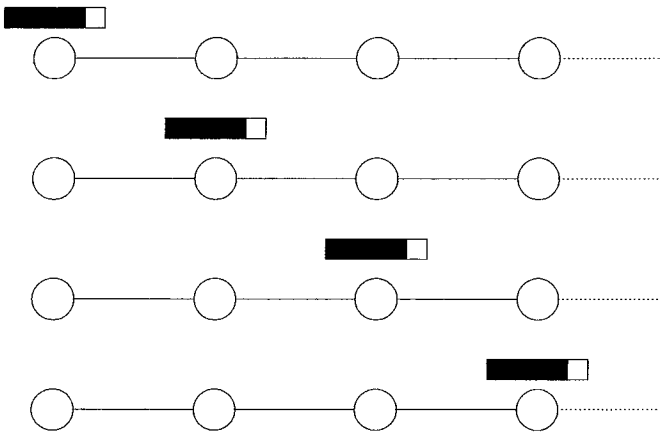

Figure 2.26: Stages of Transmission of a Message

Message switching, establishes a complete path between two processors, in order to send a message. The path is set according to information in the header of the message. When the path is free, the message will be sent along, and the path will be released only when the message reaches its destination. Dynamic circuit switching is an improvement on this technique. It starts the transmission before the path is established. If the path is not fully available, parts of the message are stored on intermediate processors. When the path is clear, all the packets are released, and as before, the path is released when the message reaches its destination. The Wormhole is a variant of dynamic circuit switching; the wormhole releases the path in stages as the transmission proceeds.

2.3.6 Communication Between Two Processors

Communication between two processors that are not necessarily neighbors is usually simple. In message switching mode, a message is split into packets that are transmitted over the network. This simple solution can be improved by pipelining the messages. The message is split into packets of equal size, which are dispatched over the network in a way to fill up all the nodes on the path. Figure 2.27 illustrates the concept of pipelining.

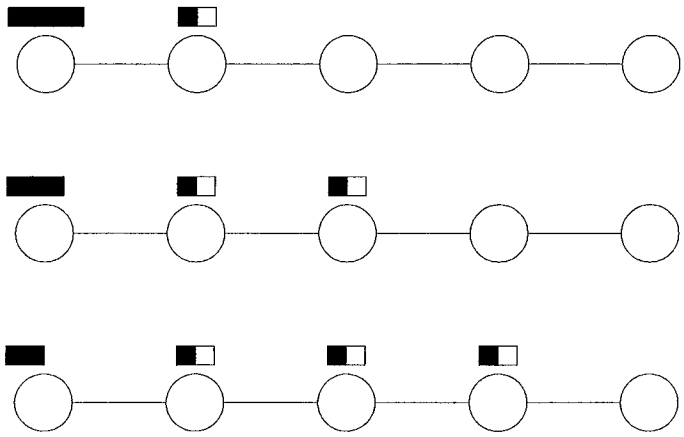

Figure 2.27: Pipelining

Lets us now consider a 2 dimensional toric grid network. Each processor has a reference (*i, j*). If the two processors are on the same horizontal or vertical grid, then a single, straight path exists. Pipelining is used to pass the message. If the processors are on different grids, there exist two minimal length paths to pass the message. The two paths are illustrated in the following Figure 2.28. In this case, the packets are sent along the two paths.

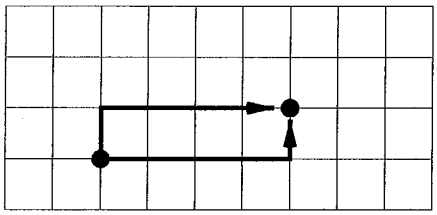

Figure 2.28: Message Passing Along a Toric Grid

In the case of a hypercube, the approach is a bit more complex. Let k be the length of the path to the destination processor. The source processor divides the message by k and sends the message to its neighbours, who will repeat the process until the desired processor is reached. Figure 2.29 illustrates the process.

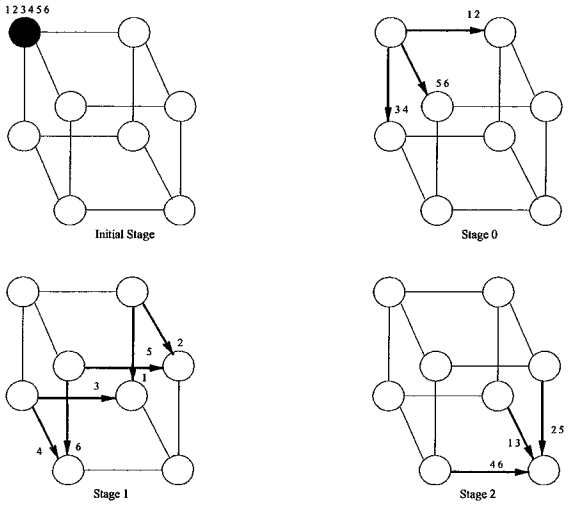

Figure 2.29: Message Passing in a Hypercube

Broadcasting

In Broadcasting, one processor sends a message to all processors on the network. In a ring, the processor wishing to broadcast starts by sending the first packet to its immediate neighbors. The neighbors then forward the message to their immediate neighbors and wait for the next packet. Figure 2.30 shows a simulation of the broadcasting procedure on a ring. The numbers shown are the number of the packet being passed from one node to the other.

One broadcasting scheme on a toric grid broadcasts the message vertically first, and in a second stage, broadcasts horizontally. Figure 2.31 illustrates the two phases of the broadcast.

This broadcasting technique can be improved by dispatching the packets on the two directions, horizontally and vertically.

A scheme used for broadcasting in a hypercube involves sending a pipelined message along a spanning tree. The spanning tree represents the hypercube. Figure 2.33 is an illustration of the algorithm on a hypercube of degree 3.

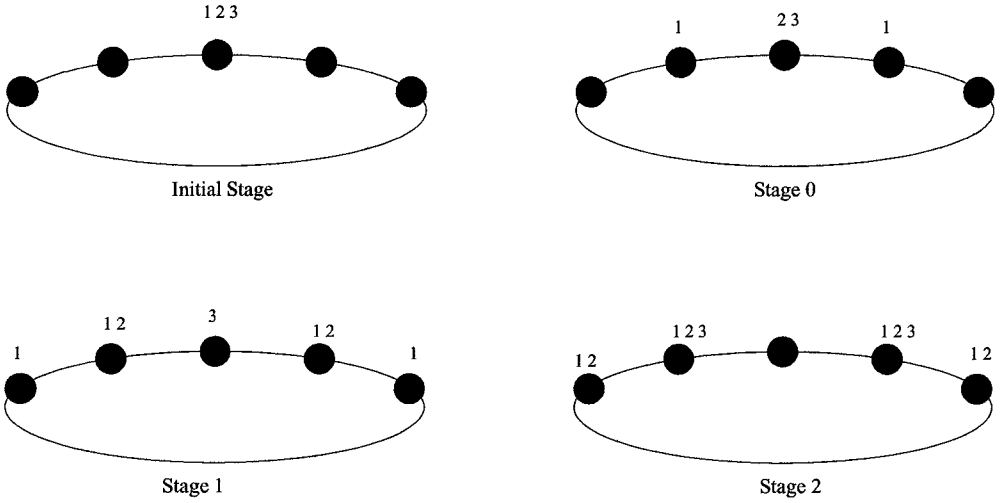

Figure 2.30: Broadcasting on a Ring

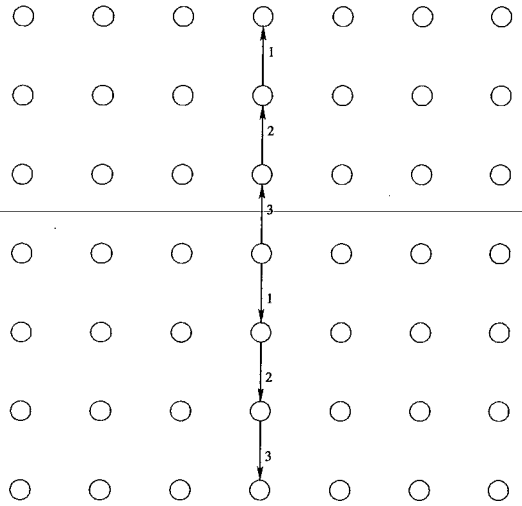

Figure 2.31: Broadcasting on a Toric Grid stage1

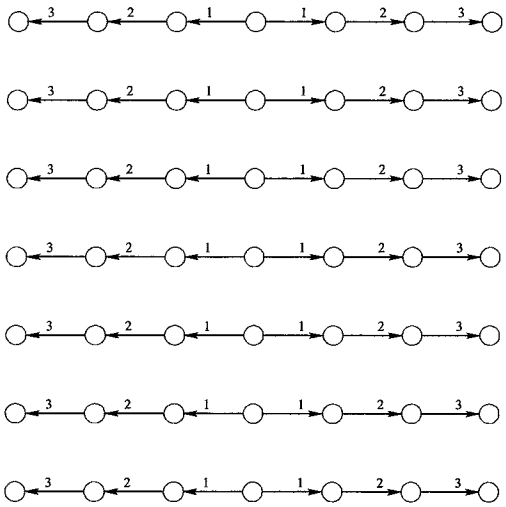

Figure 2.32: Broadcasting on a Toric Grid stage2

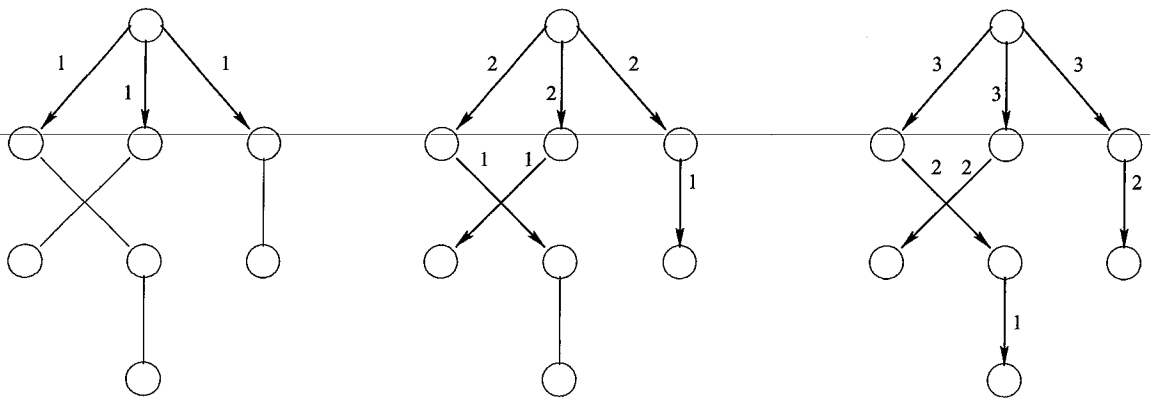

Figure 2.33: Broadcasting in a Hypercube

Total Exchange

In the Total Exchange communication scheme, every processor sends a message to every other processor on the network. In other words, every processor on the network is broadcasting a message to all processors.

Consider a ring with p processors, assume for simplicity that p is odd and that the processors are numbered from $-\frac{p}{2}$ to $\frac{p}{2}$. The following algorithm is used by the processors: Every processor sends its message to the immediate neighbors, and receives their messages. Then, $for\ i = 2\ until\ \frac{p}{2}$ send the message received from the right neighbor to the left neighbor, and send the message received from the left neighbor to the right neighbor. Receive messages from right and left neighbors and repeat the procedure. Figure 2.34 illustrates the stages of the algorithm from the point of view of processor 0. The number are the messages of the processors.

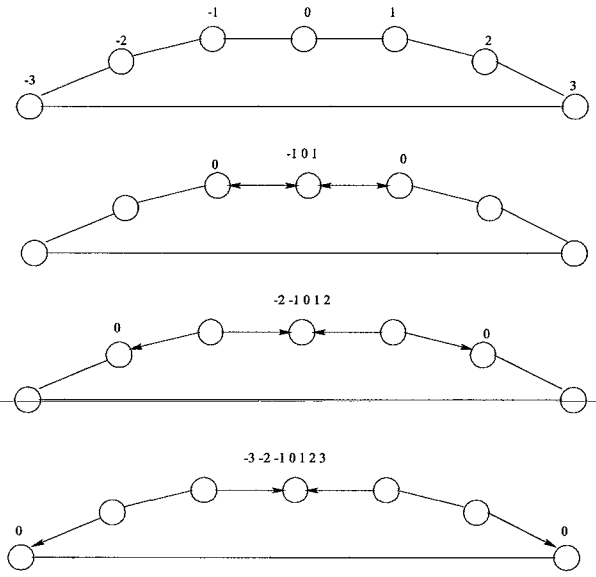

Figure 2.34: Total Exchange on a Ring

The algorithm used for total exchange on a ring can be used for total exchange on a toric grid. For simplicity, assume a square toric grid, $p * p$ where p is an odd number. The fist phase of the algorithm is to exchange the first half of the message on the horizontal ring, and the second half

on the vertical ring using the total exchange algorithm on a ring. Every processor at this stage knows the first half of the messages of all the processors on its row, and the second half of all the messages of the processors on its column. It is enough as a second stage to totally exchange the second half of the messages on the horizontal ring, and the first half of the messages on the vertical ring. The two stages are illustrated in Figure 2.35

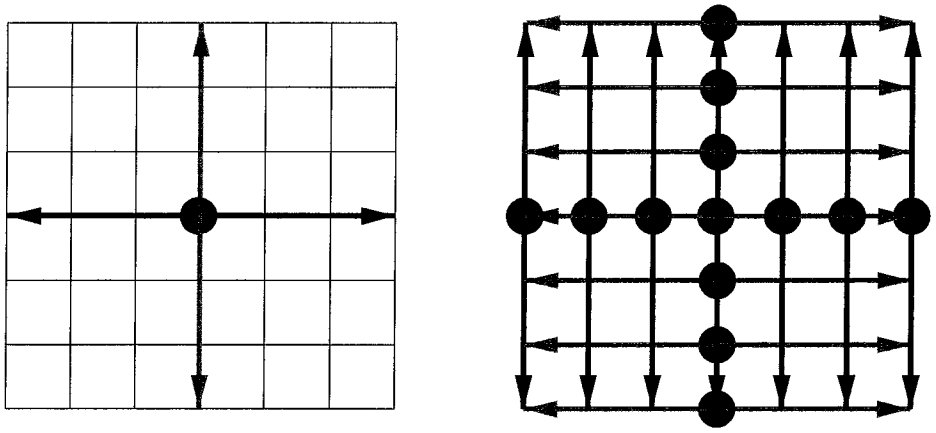

Figure 2.35: Total Exchange on a Toric Grid

On a hypercube, the total exchange algorithm is based on a two-by-two exchange of the whole message. Each processor exchanges all his known messages successively in each direction. Figure 2.36 illustrates the steps of the algorithm. From the point of view of processor 1, in the first stage, processor 1 exchanges all its known messages (i.e., message 1) with processor 2. At the end of this stage, processor 1 now has 2 messages, 1 and 2. In the second stage, processor 1 exchanges all its known messages (i.e messages 1 and 2) with processor 3. Now, processor 1 has messages 1, 2, 3 and 4. As a final step, processor 1 exchanges its messages with processor 5.

2.3.7 Parallel Virtual Machine, PVM

Parallel processing solves a large problem by dividing it into small tasks and solving the small tasks in parallel. Two major developments are fostering parallel processing: massively parallel

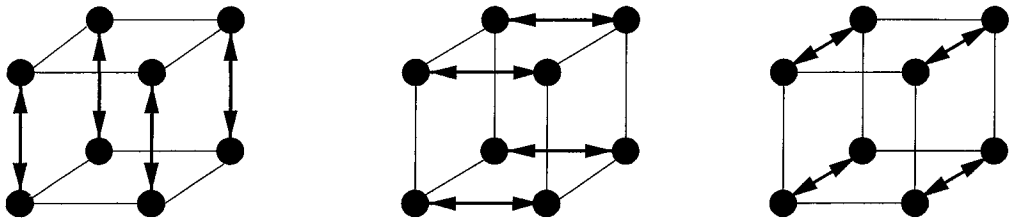

Figure 2.36: Total Exchange on a Cube

processors (MPPs) and distributed computing. MPPs combine a few hundred to several thousand CPUs in a single large machine connected to hundreds of Gigabytes of memory, thus offering an enormous computational power. In some cases, several MPPs were combined using distributed computing to produce unequaled computational power.

Distributed computing offers many advantages. Cost is very low since existing hardware is used, and since different architectures exists, performance is optimize by assigning the task to the most appropriate architecture, thereby exploiting the heterogeneous nature of a computation. Furthermore, the virtual computer resources can grow in stages taking advantage of newly available technologies. Moreover, program development uses a familiar environment, and individual computers and workstations are usually reliable and stable, and expertise in their use and maintenance is already available. To be effective, distributed computing requires high communication speed. Here are some of the most notable advances in computer networking technology:

- Fast Ethernet: is a 100 Mbps broadcast bus technology with distributed access control.
- Myrinet: achieves rates up to 1.1 Gbps.
- Gigabit Ethernet: achieves 1 Gbps bandwidth.

PVM provides a simple and efficient way to develop parallel programs using existing hardware. PVM treats a heterogeneous collection of computers as a single virtual parallel machine. It handles message routing, data conversion, and task scheduling. The PVM computing model is

quite simple: the user can access PVM resources through a library of standard interface routines. These routines allow the user to start and end tasks across the virtual machine, and handles communication and synchronization. Furthermore, the user can add or remove computers from the virtual machine. PVM also supplies routines to investigate the architecture of computers in the virtual machine.

The principles upon which PVM stands are summarized as follows:

- A user configured host pool: the application the user is running executes on the machines specified and selected by the user. Both single-CPU and multiprocessor machines may be part of the host pool. The user can alter the host pool by adding or removing machines during operation.

- Translucent access to hardware: the application program can view the host pool as a collection of similar computers, or may choose to exploit the capabilities of certain computers in the pool by assigning them certain tasks.

- Task-based computation: the unit of parallelism in PVM is a task. A task is often a Unix process, but not always. The mapping of a process to a processor is not enforced by PVM; multiple tasks may run on a single processor.

- Explicit message-passing model: A collection of tasks can cooperate by explicitly sending and receiving messages to each other. The message size is limited only by the size of the memory.

- Heterogeneity support: PVM supports heterogeneous machines, networks and applications. PVM permits messages containing different types of data to be exchanged between machines of different data representation.

- Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of underlying hardware.

2.3.8 PVM *vs* MPI

MPI (Message Passing Interface), like PVM, provides functionalities to implement parallel algorithms on MPPs or on a cluster of PCs. The following is a comparison of PVM and MPI features.

PVM is the fruit of a research project, while MPI was specified by a committee of high performance computing researchers, and industry experts. MPI is a standard that an MPP vendor implements in their own way. Although PVM and MPI have comparable performance on the Cray T3D and the IBM SP-2, MPI is always expected to perform better than PVM on MPPs [5].

MPI was designed to be portable. An MPI program written on one architecture can be compiled and executed on another architecture, without any modification. PVM programs can be copied, compiled and executed on different architectures without any change, but PVM executables also can communicate with each other over different architectures. In other words, PVM programs can run on different architectures and cooperate with each other. MPI applications can only run on a single architecture. MPI takes advantage of the underlying library to use native hardware, thus eliminating unnecessary waste for that architecture, and increasing the performance. PVM sacrifices this performance in favor of flexibility, but when communicating locally or with another host of the same architecture, PVM uses native communication functions just like MPI.

PVM is dynamic in nature, you can start and stop processes, find out which processes is running, and where it is running. Computing resources can be added or deleted dynamically from the console, or from within an application. In contrast, MPI has static design, aiming to

improve performance. Hosts cannot be added during a computation in MPI, and only the new release MPI-2 is able to start and stop a group of tasks.

It is usual for simulations to run for days, without any supervision. Fault detection and recovery in this case is a critical issue, since if an error occurs, long hours of computation could be lost. PVM supports a basic fault notification scheme, where a given task can be notified when a task fails or a host exits the virtual machine. MPI did not cover fault tolerance since it considered hosts, and tasks to be static. MPI-2 contains a notification scheme similar to PVM's.

In conclusion, MPI has an advantage over PVM if an application runs on a MPP. However,PVM has the advantage over MPI if the application runs on a cluster of computers.

2.4 Beowulf Clusters

Beowulf is a sixth century hero who freed the Danes from an oppressive monster called Gendel. "Beowulf" is now the name of a new strategy in high performance computing that exploits mass market technologies. Beowulf overcomes the cost in time and money of supercomputers, thus freeing us from the oppression of the cost of supercomputers.

A Beowulf is a collection of personal computers (PCs) interconnected by any available networking technology. Most of the Beowulf programs are written in C and Fortran, using a message passing model of parallel computation, but other approaches are possible, among which are languages like Java and Lisp, and other communication strategies like RPC, RMI, or Corba[1].

Beowulf uses mass market commodity off the shelf, thus it is not subject to time constraint waiting for custom hardware to be produced, and can benefit from new technology.

The building block of any cluster is a node. A node could be a small desktop system costing around a thousand dollars, or a mighty multiprocessor costing millions of dollars. For a Beowulf, a node is usually a PC derived from mass market technology, exhibiting an excellent

price/performance ratio.

Second to the node, the most important part of a Beowulf cluster is the underlying network. Beowulf usually, but not always, uses Ethernet and TCP/IP. Fast Ethernet achieve 100 Mbps bandwidth. Greater bandwidth is achieved through Myrinet and Gigabit Ethernet, but these are expensive technologies. Processors of different nodes in a cluster communicate with each others by means of the network.

The operating system is the software that access the processor and the memory, that provides services to applications and to the user, and that manages the devices. Linux is a multitasking, virtual memory, POSIX compliant operating system. Linux is available at no cost on the interned, and at low cost on CD-ROM. Linux comes with the GNU programming environment, which is also free, and includes gcc, g++, gdb, emacs, as well as lex, yacc, bison, and other development tools.

Chapter 3

Neural data-path allocation

The work described in this chapter is based on a research by H. Harmanani[6].

The examples describing the method for solving the clique partitioning problem are based on the Data Flow Graph (DFG) illustrated in Figure 3.1.

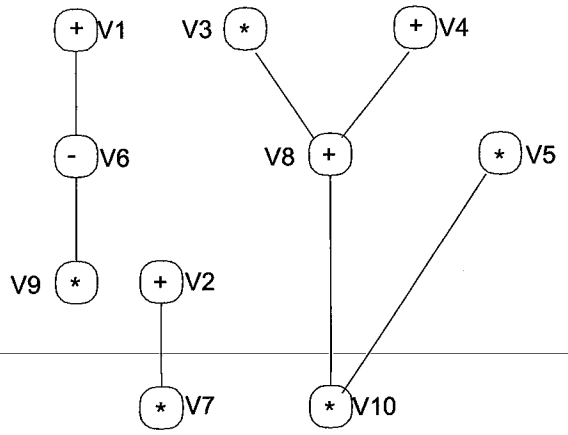

Figure 3.1: Data Flow Graph

Two nodes are defined to be compatible if they can share the same resource. Two nodes have to satisfy the following conditions to be able to share the same resource:

1. The nodes should be homogeneous, such that there exists an operator to execute them both. For example, an ALU can execute both an addition and a subtraction. Both V_1 and V_6 could

be assigned to an ALU, thus they are homogeneous.

2. Both nodes should not be concurrent, and that is that they should not be at the same level because they have to be executed at the same time.

In Figure 3.1, V_1 and V_4 are homogeneous. However, they are not compatible because they are concurrent, and as such they cannot share the same resource, and are not considered as compatible. V_1 and V_6 are homogeneous, and they are in two different time stamps, thus compatible.

The compatibility graph is a graph $G(V, E)$ where there exists an edge E connecting every two nodes V_i and V_j which are compatible. The compatibility graph is derived from the DFG by adding all the vertices, then creating an edge between every two compatible vertices. The graph in Figure 3.2 corresponds to the compatibility graph of the DFG in Figure 3.1.

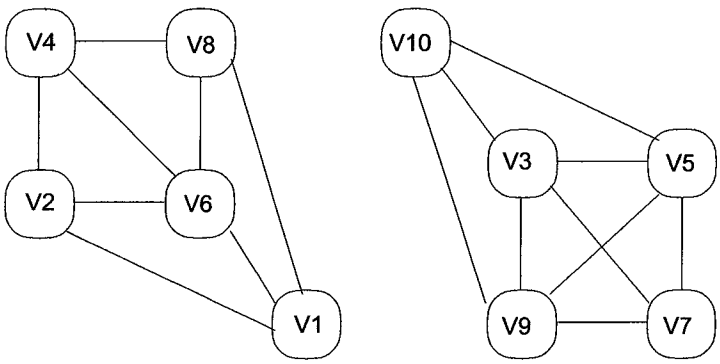

Figure 3.2: Compatibility Graph

An undirected graph $G(V, E)$ is defined to be a complete graph, if for every vertex V in G there exist an edge E that connects V to every other vertex in G. A clique C in a graph G is a complete graph where every node of C is a part of the graph G, and C is not contained in any other complete graph in G. The clique partitioning problem is to find the minimal number of cliques in a certain graph. The solution of the clique partitioning problem for the compatibility graph is the minimal number of resources needed. One resource for every clique, shared by the

vertices of the clique. Since the clique partitioning problem is NP complete, Neural networks are used to solve the problem.

Consider V to be a matrix, where the number of rows is equal to the minimal number of cliques, and the number of columns equal to the number of vertices in the DFG. If a vertex V_i is part of clique j, then V_{ij} is 1, otherwise it is 0.

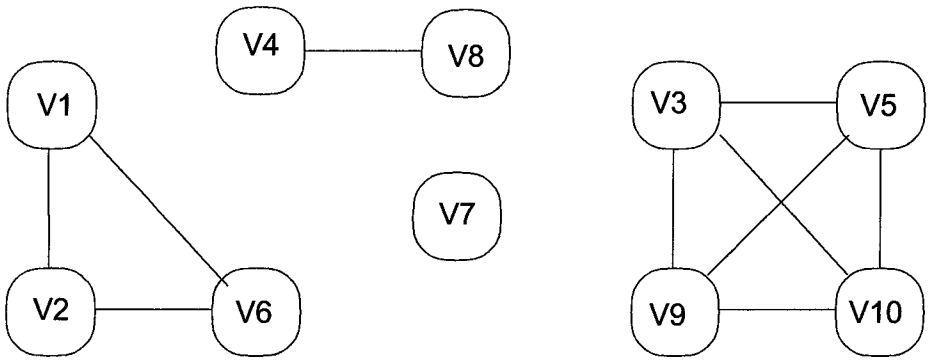

Figure 3.3: Clique Solution

Figure 3.3 shows a solution for the clique partitioning problem for the compatibility graph, and Figure 3.4 shows the table representation of the solution. Since this behavior is similar to the behavior of neurons, let us assume that every location in table V is a neuron. Every neuron computes its new state using a motion equation.

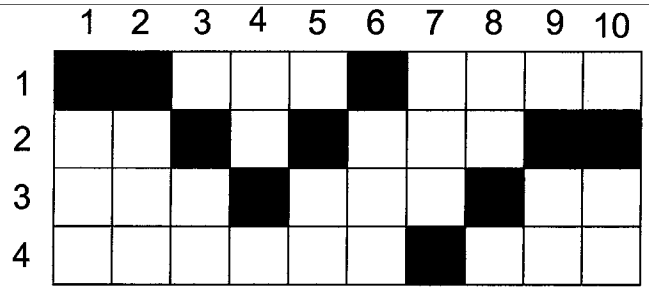

Figure 3.4: Table Representation for the Solution

3.1 Motion Equation

In order to enable the neurons to compute a solution to the problem, an energy function that describes the motion of a neuron is in order. What follows is the description of the motion equation.

If a vertex V_i doesn't belong to a clique C_j, then neuron V_{ji} is encouraged to fire. This is fulfilled using the following term:

$$Term1 = \begin{cases} 1 & \text{if } \sum_{j=1}^{Clique} V_{xj} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Equation 3.1 adds all the terms in the column of the vertex x, thus finding to how many cliques this vertex belongs. If the result is 0, then the vertex belongs to no cliques, thus the term will take an excitatory value, 1.

Vertices are encouraged to fire in cliques where there are adjacent nodes. Equation 3.2 fulfills this requirement.

$$\sum_{j=1}^{Nodes} V_{(j,y)} * Adj_{(j,x)} \tag{3.2}$$

Equation 3.2 returns the number of vertices that are part of the clique, and are adjacent to the vertex.

Vertices should belong to one and only one clique. Equation 3.3 acts as an inhibitor forces, that inhibits any vertex which belongs to more than one clique.

$$Term3 = \begin{cases} -1 * \sum_{j=1}^{Clique} V_{xj} & \text{if } \sum_{j=1}^{Clique} V_{xj} \text{ ¿ } 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.3)$$

The sum in Equation 3.3, as in Equation 3.1, is the sum of all the cliques to which the vertex belongs. The result of multiplying the sum by -1 is an inhibitory force if the vertex belongs to more than one clique.

A clique cannot contain non-adjacent vertices. The following term ensures that there all the vertices are adjacent.

$$\sum_{j=1}^{Nodes} V_{(j,y)} * (Adj_{(j,x)} - 1) \quad (3.4)$$

Equation 3.4 is the sum of all the vertices in a clique that are not adjacent to the node. The result of subtracting 1 from $Adj_{(j,x)}$ is -1 if the nodes are not adjacent (since $Adj_{(j,x)}$ is 0 in this case) or 0 if the nodes are adjacent (since $Adj_{(j,x)}$ is 1 in this case). Thus the result of Equation 3.4 is inhibitory if there are non-adjacent vertices in the clique.

Finally, the motion equation is derived to be:

$$\begin{aligned} \frac{dU_x}{dt} &= A * (Term1) + \\ & B * (Term2) + \\ & C * (Term3) + \\ & D * (Term4) \end{aligned}$$

Where A, B, C, and D are the connection weights. Now that the motion equation for every neuron is defined, the parallel algorithm to simulate the neural network is introduced.

3.2 Parallel Algorithm

So far, the representation of the neurons is given in the V matrix, and the motion equation for the neurons is set. The parallel algorithm that simulates the neural network is derived from a sequential algorithm. In a sequential algorithm, each neuron computes its new state sequentially, while the others neurons wait. Since the neurons are organized in the V matrix, then one iteration to compute the new state looks as follow:

$for(x = 0;\ x < nodes;\ x++)$

$for(y = 0;\ y < cliques;\ y++)$

$computedU_{xy}$

$UpdateUbysettingU_{ij} = U_{ij} + dU_{ij}$

The algorithm iterates until it reaches a stable state, or after the completion of a certain number of iterations. In the latter case, the system did not converge to a solution. In every iteration, the neurons use the same data that describes the current state to compute the new state. Parallelizing the work of the neurons, reduces the time cost for each neuron waiting for the others. Consider a neuron to be a simple process that determines its new state using the current state and the motion equation described earlier. A new problem is how do neurons communicate their new state to the other neurons. Two solutions present themselves:

1. all neurons communicate their new state to all other neurons.

2. the neurons communicate their new state to a process that will update the state of network (i.e., update the U matrix) and then send the new state back to all the neurons.

3.2.1 Manager Process Model

This model uses the total exchange communication scheme discussed earlier. This model is divided into two parts, the manager and the computing process or neuron. The manager process handles all initializations, starting the neurons, and displaying the result on the screen. Each process, or neuron, computes its new state and communicates the result to all the neurons (i.e, all the neurons are exchanging their new states). One and only one neuron sends the new state of all the neurons to the manager, so that the manager can print the output (Figure 3.5). What follows, is a behavioral description of the manager and the neuron process.

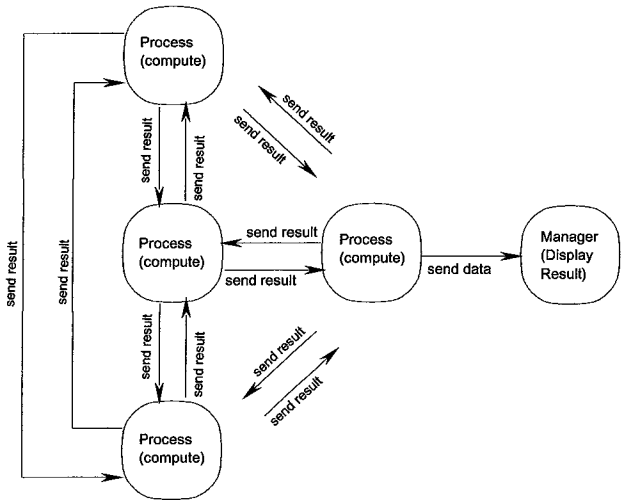

Figure 3.5: Manager Process Model

Manager Program

The manager process computes n, the number of the vertices in the graph, initializes U with a random value in the range $(-w,0)$, and sets Adj, the representation of the compatibility graph in the form of a table. Adj_{ij} is 1 if V_i and V_j are adjacent, or compatible, otherwise, Adj_{ij} is 0. The manager process computes the optimal number of cliques c by: running the Left Edge Algorithm (LEA) for register allocation and then running the concurrency test on the DFG. Choose c to

be equal to the maximum of the results returned by the LEA and the concurrency test. LEA is an allocation algorithm that allocates the registers used by the operations in the DFG. The concurrency test returns the maximum number of concurrent operations in the DFG.

1. Initialize. Set $t = 0$, $\Delta t = 1$. Randomize the initial values of $U_{ij}(t)$ in the range $(-w,0)$, where $i = 1, 2, ..., n$ and $j = 1, 2, ..., n$ and $w = 50$.
2. Evaluate $V_{xi}(t)$ based on the binary function $V_{xi} = \begin{cases} 1 & if\ U_{xi} > 0 \\ 0 & otherwise \end{cases}$
3. Start the neuron processes.
4. Send *Adj*, *c*,and *n* to the neurons.
5. Send *U*, and *V* to the neurons.
6. Wait for a response from one of the neurons.
7. Output the current state.
8. Increment *t* by Δt. If the system is in equilibrium, or $t = T$, then terminate all the processes and exit, else, go to 6.

Neuron Process Program

The neuron process has a simple task to do. When the neuron process starts, it waits for the data from the manager, when the data arrives, the neuron process computes its new state, then send its new state to all other neurons, and waits for the states of the other neurons. When all the new states are received, loop again until the stop signal is received from the manager.

1. Wait for *Adj*, *c*,and *n*.

2. Wait for U, and V.

3. Use the motion equation to compute $dU_{ij}(t)$.

4. Send the result to all the neurons.

5. Wait for dU_{ij} of $neuron_{ij}$.

6. Update U.

7. Go to 3.

3.2.2 Master Slave Model

This model uses the broadcasting communication scheme discussed earlier. The Master Slave model, as its name implies, is divided into two parts: the Master process and the Slave process. The master process is responsible of any initialization, starting and terminating the slaves, sending data, collecting and displaying results, and in some cases, managing the workload of the slaves. A slave process performs a specific task, or computation, and reports the result back to the master process (Figure 3.6).

In the algorithm proposed here, a slave process represents a neuron. The slave process computes the new state of the neuron, and reports the result back to the master that updates the state, and then broadcast it to all the slaves processes. What follows is a behavioral description of the master process and the slave process.

Master Program

The master process computes n, the number of the vertices in the graph, initializes U with a random value in the range $(-w,0)$, and sets Adj, the representation of the compatibility graph in the form of a table. Adj_{ij} is 1 if V_i and V_j are adjacent, or compatible, otherwise, Adj_{ij} is 0. The

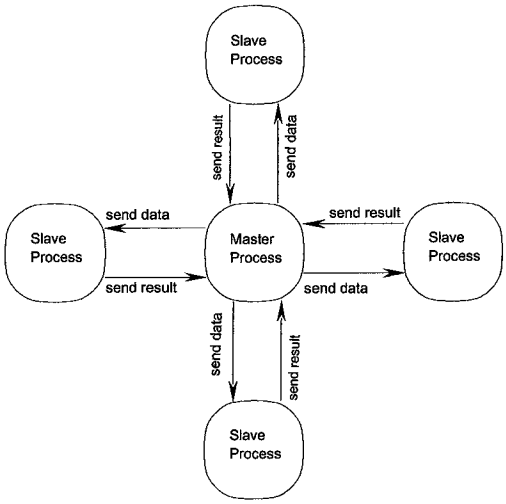

Figure 3.6: Master Slave Model

master process computes the optimal number of cliques c by: running the Left Edge Algorithm (LEA) for register allocation and then running the concurrency test on the DFG. Choose c to be equal to the maximum of the results returned by the LEA and the concurrency test. LEA is an allocation algorithm that allocates the registers used by the operations in the DFG. The concurrency test returns the maximum number of concurrent operations in the DFG.

1. Initialize. Set $t = 0$, $\Delta t = 1$. Randomize the initial values of $U_{ij}(t)$ in the range $(-w,0)$, where $i = 1, 2, ..., n$ and $j = 1, 2, ..., n$ and $w = 50$.

2. Evaluate $V_{xi}(t)$ based on the binary function

3. Start the slave processes.

4. Send to the slaves *Adj*,*c*, and n.

5. Send to the slaves U, and V.

6. Wait for a response from the $slave_{i,j}$, then Compute $U_{ij}(t+\Delta t)$ using the first order Euler Method: $U_{ij}(t+\Delta t) = U_{ij}(t) + \Delta U_{ij}(t)\Delta(t)$

7. Apply 2.

8. Increment t by (Δt) If the system is in equilibrium, or $t = T$, then terminate all the slave processes and exit, else, go to 5.

Slave Program

The slave process, like the neuron it represents, has a simple task to do. When the slave process starts, it waits for the data from the master process. When the data arrives, the slave process does the computation and reports the result to the master process.

1. Wait for Adj, c, and n.

2. Wait for U, and V.

3. Use the motion equation to compute $dU_{ij}(t)$.

4. Send the result to the master.

5. Go to 2.

3.3 Enhancing the Algorithm

The algorithm discussed earlier has a big communication overhead, due to congestion by the high number of messages being exchanged at each iteration. The reduce the problem of congestion, the neurons, or processes were clustered. A process is assigned to each host on the virtual machine. Each process will compute the new state of the neurons assigned to it, and then communicates the result, either to the master process (Master-Slave model), or to the other processes on the other hosts (Manager-Process model). The neurons are assigned to the hosts using the following algorithm:

Host Number	Starting Neuron	Ending Neuron
0	0	12
2	13	26
3	27	39

Table 3.1: Neuron Distribution

$quot = numberOfNeurons/numberOfHosts$

$rem = numberOfNeurons\%numberOfHosts$

$current = 0$

$for(i = 0;\ i < numberOfHosts;\ i++)$

{

$value = quot$

$if\ (i > 0\ and\ i < rem)$

$value = value + 1$

$hostsNeuron[i].beginAt = cur$

$hostsNeuron[i].endAt = hostsNeuron[i].beginAt + value - 1;$

$current = current + value$

}

For example, in the case where there are 40 neurons and 3 hosts, the neurons will be divided among the hosts in the following manner:

Chapter 4

Results

Both algorithm (Manager-Process and Master-Slave) were tested on a cluster of 6 hosts. Each host is a PII 300 Mhz. The host are connected through two 100 Mbps ethernet to two fast switches. Tests were done with 1, 2, 4, and 6 hosts in the virtual machine.

The following Figure illustrates the result of the old algorithm used on an example of 4 cliques and 10 vertices, that is 40 neurons.

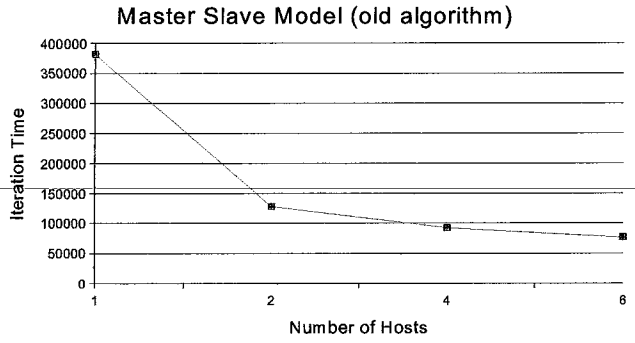

Figure 4.1: Master-Slave results

The fastest time produced by this algorithm, on a smaller example, is still to large to be compared with the slowest time produced by the new algorithm. What follows are the results of the improved algorithm, tested on a bigger example.

The example used has 7 cliques and 34 vertices, that is a total of 238 neurons. The following figure illustrates the result of the tests.

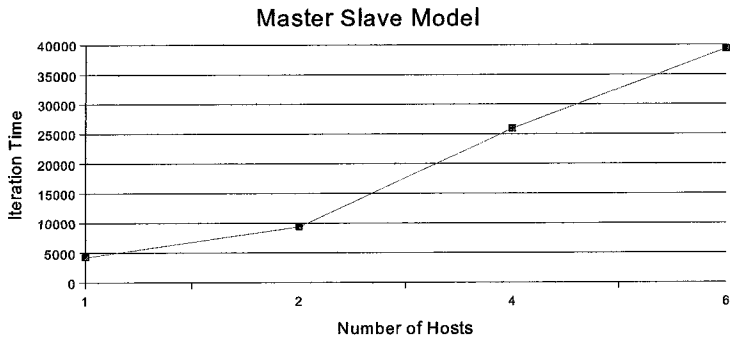

Figure 4.2: Master-Slave results

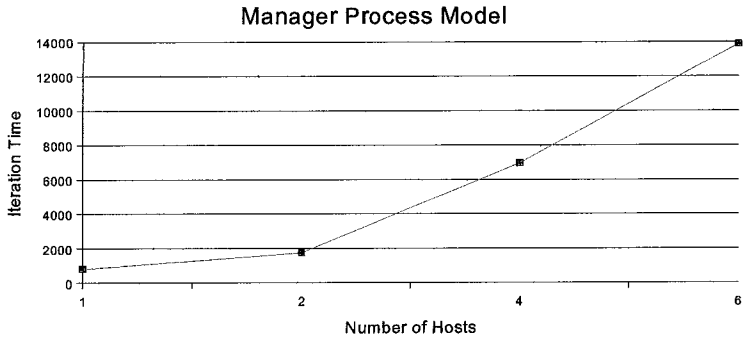

Figure 4.3: Manager-Process results

The iteration time is in μsecs. The iteration time of the master-slave model is calculated by the master, its the time interval between broadcasting the data, and collecting all the results from all the slaves. The iteration time of the Manager-Process model is calculated by a process, its the time interval between broadcasting its new state and collecting the states of all the other processes. In the case of one host, the iteration time of the manager process model is comparable to the iteration time of a serial algorithm.

The following Figures illustrate the speed up of the iteration time with respect to the number

of hosts in the virtual machine.

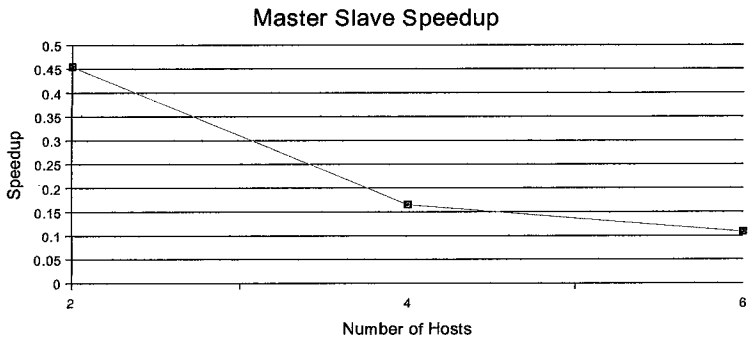

Figure 4.4: Master-Slave Speedup

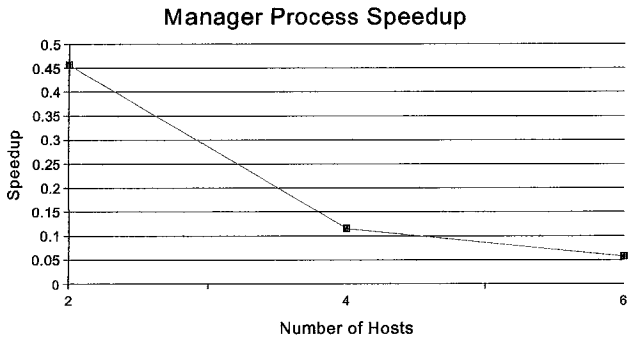

Figure 4.5: Manager-Process Speedup

The example used is a small example compared to real life. As a result, the communication overhead induced by increasing the number of hosts is larger than the speedup of the computation. In real life computations, the result of increasing the number of hosts is expected to produce a better speedup.

Chapter 5

Conclusion and Future Research

The results of the motion equation are satisfactory on small examples. On examples that involve a large number of neurons, the motion equation did not converge correctly. The reason is the fact that there is no hill-climbing term, and the motion equation got stuck in local minimas. Furthermore, the clustering algorithm of the neurons could be improved in a way to reduce further the communication overhead.

Bibliography

[1] www.omg.org/corba.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to ALGORITHMS.* McGraw-Hill, 1997.

[3] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures.* International Thomson Computer Press, 1995.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Prallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing.* Massachusetts Institute of Technology, 1994.

[5] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. Pvm and mpi: a comparision of features. Technical report, 1996.

[6] H. Harmanani. A method for data path synthesis using neural networks. Technical report, 1999.

[7] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[8] R. Rojas. *Neural Networks, A Systematic Introduction.* Springer, 1996.

[9] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. *How to Build a Beowulf.* Massachusetts Institute of Technology, 1999.