1993

A comparison between hypercube and binary de Bruijin networks

Ato Y. Arkaah
Lehigh University

AUTHOR:

Arkaah, Ato Y.

TITLE:

A Comparison Between Hypercube and Binary De Bruijn Networks

DATE: January 16, 1994

A Comparison Between Hypercube and Binary de Bruijn Networks

by

Ato Y. Arkaah

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Engineering

Electrical Engineering and Computer Science Department

Lehigh University

December 10, 1993.

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

12/7/93
Date

Thesis Advisor

Chairperson of Department

Table of Contents

List of Figures

List of Tables

Abstract

Application specific multiprocessor networks tend to achieve the maximum possible performance for their class of applications. However, the task of comparing networks that differ in physical attributes on a specific class of applications is difficult in nature.

The intention of this study is to perform a comparison between the hypercube and binary de Bruijn multiprocessor networks. This class of networks differ in almost every physical design aspect. Although the hypercube network is known to be particularly well suited to the distributed sorting class of applications, little is known of the de Bruijn network within this class.

This study presents the differences and similarities between the two networks. It also presents results obtained from a hybrid distributed sorting application implemented on both networks for increasing network and list sizes. If the de Bruijn network at the very least matches the hypercube in low network fabrication cost, simplicity, and sorting efficiency, then the de Bruijn network can be targeted as a possible network of the future.

CHAPTER 1

An Introduction to Multiprocessor Networks

1.1 Introduction

The demand for systems that possess faster execution times and higher throughput has increased steadily over the last decade. Prior to the design of parallel systems conventional computers were approaching the upper bound of their physical switching limits. This limit, the natural delay time for signals to travel from one point to another, is bound by our present day technological standards. To further emphasize the point, a single sequential processor capable of operating at the speed of light could not achieve more than a couple billion instructions per second [14]. Such a system does not meet the requirements of applications derived from fields such as physics of fluid dynamics.

To increase computer throughput, it was realized that the sequential bottleneck problem of. Von Neumann architecture had to be avoided. One of the solutions to this problem is to create systems with multiple processors. In other words, systems that could process information concurrently or in parallel.

Parallel computers are constructed with the assumption that a large number of applications are inherently parallel. Thus, if an application or parts of an application can be executed in parallel we can achieve faster execution and an increased throughput.

It is the intention of this study to perform a critical comparison between the *de Bruijn multiprocessor network* and the *hypercube multiprocessor network.* During the course of this document, the advantage and disadvantages of both systems will be examined in detail. Also, some conclusions are drawn from empirical data obtained from timed sorting algorithms implemented on both networks.

Throughout the course of this document, the term *multiprocessor network* will be used to denote a parallel computer with a specific topological design for its communication links. The multiprocessor networks utilized in this study are static in nature. The communication links of a static network, once configured, remain passive throughout the network's lifetime. On the other hand, dynamic networks possess the ability to reconfigure their communication links by resetting their network's communication channel switching elements at run-time [12].

Static networks are usually represented as undirected graphs, where vertices are processing nodes and edges correspond to communication channel links between nodes [8,11,13,17]. In addition, each processing node consists of a processor with its local memory module . Once a graph is established for a network the attributes of the graph can be analyzed.

1.2 Attributes of a multiprocessor network

In a multiprocessor network, primary focus is placed on the following attributes:

- ***diameter.*** This denotes the maximum number of hops any message traverses for communication between nodes. This length is desired to be kept at a minimum value because, the shorter the message path the less time communication will consume, which in turn contributes to a faster system.
- ***node degree.*** The degree of a node corresponds to the number of input/output (I/O) communication links each node possesses. Obviously, the greater the degree the more

complex and expensive the node. Thus, node degree is desired to be at a minimal, constant number, or should avoid increasing as rapidly as the machine size. A constant node degree for increasing/decreasing network sizes does not require the design and manufacture of new nodes, but simply a reconfiguration of its communication channel links.

- ***routing algorithms.*** Routing algorithms are responsible for the efficient and safe transportation of messages/data between processors. The complexity of the routing algorithm of a multiprocessor network is directly related to its design. Algorithms for message routing are preferred to be as simple and as optimal as possible. (i.e. using the shortest path)
- ***uniformity and symmetry.*** A uniform and symmetric network permits a single node design with its fixed node degree to be utilized for all vertices of the network. This desired attribute contributes significantly to the systems cost factor, and when adhered to can lead to a standard routing algorithm with a balanced communication traffic flow.
- ***fault tolerance.*** It is every designers wish that their multiprocessor network be able to operate in the midst of faults. This attribute is not addressed throughout the course of this document primarily because of the vast range of its sub-topics.
- ***expansibility and tearing ability.*** These attributes consider the case of reducing or increasing the network size. It is the goal of every designer to create a network that is easy to increase/decrease in node size.
- ***absorbability.*** This particular attribute is concerned with the network's ability to emulate another network. If a network possesses this attribute for a different network an

argument can be made that the type of network absorbed is no longer needed. Usually, this attribute can be used to classify the designed network as a general-purpose or a special-purpose network.

In some cases, the above attributes can contradict each other. For example, a multiprocessor network with a small diameter might imply a network with nodes of a greater degree. This is understandable because to obtain an optimal diameter of *1*, one would expect a fully connected network (i.e. each node is directly connected to all other nodes). These contradictions contribute to the trade-off between cost and performance.

Multiprocessor network designers realize that an optimal network is created when it optimizes as many of the above mentioned attributes as possible. In fact, it is from these attributes that the ideas for both the hypercube and de Bruijn multiprocessor networks were created.

To perform a comparison of the two networks, it is necessary to define other networks that are used for analyzing the absorption properties of the hypercube and de Bruijn multiprocessor networks.

1.3 Relevant multiprocessor networks

Multiprocessor networks that are examined are the *linear array network,* the *ring network,* the *binary network,* and the *shuffle-exchange network.*

1.3.1 The linear array network

The linear array network, denoted ***L(n),*** is a multiprocessor network consisting of *n* processing nodes connected in a chained fashion. (see figure 1.1)

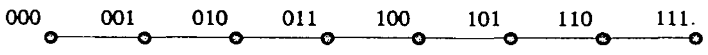


Figure 1.1 A L(8) array network

The linear array network possesses a diameter of ***n-1,*** and a maximum node degree of *2*. Outer nodes possess a node degree of ***1,*** and inner nodes possess a node degree of *2* [12,17].

The advantage of such a system is its ability to process data in an overlapped pipelined fashion [9]. In other words, a partially completed task can be passed on to an adjacent processor. In this case, each of these processors can be assigned a specific portion of the algorithm to perform on the incoming data.

The linear array's disadvantage is noticed when a slow node, due to a complex partial algorithm, affects the execution time of the pipelined process. The size of a partial algorithm must therefore be carefully selected in an attempt to avoid a *load imbalance.*

1.3.2 The ring network

The ring network, denoted ***R(n),*** is a multiprocessor network consisting of *n* processing nodes linked in a circular fashion.

The advantage of a ring network over a linear network is that it possesses a diameter of $\lceil \frac{n-1}{2} \rceil$, and a constant node degree of *2* [12]. Another advantage over its linear counterpart is its ability to continue recycling data past the n^{th} stage of.

processing.

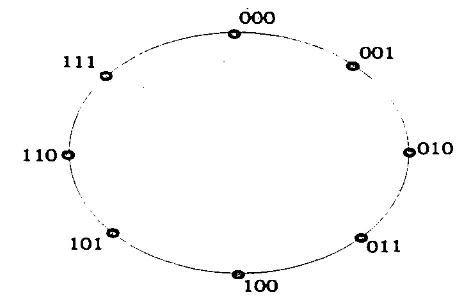


Figure 1.2 A R(8) network

The ring network also possesses the same disadvantage as its linear counterpart if it is used to implement a pipelined process.

1.3.3 The binary tree network

The binary tree network, denoted ***T(2,n)***, is a multiprocessor network consisting of 2^n-1 processing nodes linked in a fashion similar to that of a tree. (see figure 1.3)

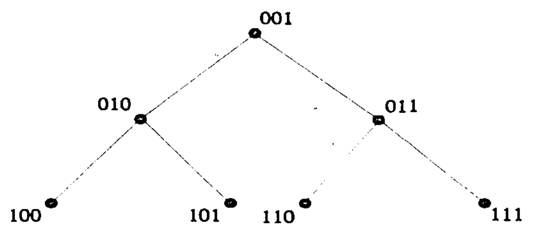


Figure 1.3 A T(2,3) network

Processing nodes of the network are labeled from *1* to 2^n-1 in binary notation. The following steps outline the main steps of the tree construction:

step 1. Designate the root node as $x_{n-1},x_{n-2},\ldots,x_1,1;$ where $x_*=0$.

step 2. The left child node *identifier (id)* of any node is obtained by multiplying the node's id by a factor of *2*, which is identical to a left-shift operation *(L_s)*.

step 3. The right child id of any node is obtained by adding a value of *1* to the left child's id.

step 4. Repeat steps *2* and *3* until all *2^n-1* nodes have been utilized in the binary tree's construction.

The binary tree network possesses a diameter of $2\ (\lceil \log_2 n \rceil - 1)$, and a maximum node-degree of *3*. The root node is of degree *2*, inner nodes are of degree *3*, and outer nodes are of degree *1* [12,17].

1.3.4 The shuffle-exchange network

A shuffle-exchange network, denoted ***ShX(n)***, is a multiprocessor network consisting of *2^n* processing nodes labeled from *0* to *2^n-1* in binary notation. (see figure 1.4)

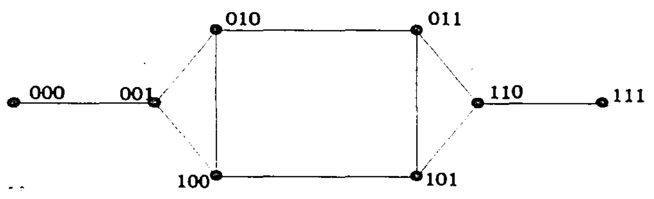


Figure 1.4 A ShX(2,3) network

The communication channel links are constructed utilizing the following rules and definitions:

definition 1.1: A node is said to be *shuffled* when a L_s operation with a wrap-around is performed on its node id. The shuffle operation, denoted ***Sh(m)***, is further illustrated as follows:

$$\mathbf{Sh(m)} = \mathrm{Sh}(m_{n-1}, m_{n-2}, \ldots, m_1, m_0) = m_{n-2}, m_{n-3}, \ldots, m_0, m_{n-1}.$$

definition 1.2: A node is said to be exchanged when its LSB is complemented. The exchange process, denoted ***Ex(m),*** is as follows:

Ex(m) = Ex($m_{n-1},m_{n-2},\ldots,m_1,m_0$) = $m_{n-1},m_{n-2},\ldots,m_1,m'_0$; where m'_0 is the complemented LSB.

With the following definitions in place, the shuffle-exchange network is defined as follows:

definition 1.3: A node, x, is connected to a node, y, if $y = Sh(x)$, or $x = Sh(y)$, or $y = Ex(x)$. In other words, x is connected to y if $y_{n-1},y_{n-2},\ldots,y_1,y_0 = x_{n-2},x_{n-3},\ldots,x_0,x_{n-1}$, or $x_{n-1},x_{n-2},\ldots,x_1,x_0 = y_{n-2},y_{n-3},\ldots,y_0,y_{n-1}$, or lastly, $y_{n-1},y_{n-2},\ldots,y_1,y_0 = x_{n-1},x_{n-2},\ldots,x_1,x'_0$.

The shuffle-exchange network possesses a diameter of $2\log_2 n - 1$, and a maximum node degree of *4* [12,17].

1.4 Organization of thesis

This paper intends to establish a clear understanding of the binary hypercube and de Bruijn networks before applying practical sorting applications on the defined networks. Each chapter contributes to a greater understanding of how the sorting algorithms are implemented on these multiprocessor networks.

Chapters 2 and 3 are dedicated to the definition and analysis of the hypercube and de Bruijn multiprocessor networks. Attributes such as diameters, node degrees, uniformity, symmetry, and routing algorithms are explored in detail. The topic of *message broadcasting* is also visited for both networks. Message broadcasting is a topic that considers the number of time units that specific messages take to reach their destinations.

Chapter 4 analyzes various sorting algorithms that may be beneficial to this study. After an in-depth study of the algorithms, a decision is made as to which sorting algorithms better serve our purposes.

For this study to be valid, the practical timing analysis must be performed on a parallel machine that can emulate both topologies. This machine is the SuperSetPlus.64 Transputer machine from Computer System Architects (CSA). Chapter 5 contains the architectural and functional information of the SuperSetPlus.64 Transputer machine. Also included in chapter 5 is a brief introduction to ***Occam 2,*** the parallel language that is utilized in this study.

Chapter 6 outlines the pseudo-code for the sorting algorithms as well as the actual configuration process for the multiprocessor networks.

Finally, chapter 7 contains the empirical data obtained from the sorting algorithms implemented on both networks. From this data and the previous chapters, a decision is made as to which of the two networks is more suitable for sorting applications. These results may not necessarily apply to other applications implemented on both networks.

CHAPTER 2

The Hypercube Multiprocessor Network

2.1 Definition of a hypercube network

A ***n-dimensional*** hypercube multiprocessor is a distributed-memory parallel computer consisting of 2^n processing nodes, connected in a *n-dimensional cube network* [4,5,7,10,12,16]. The hypercube is constructed by labeling the 2^n nodes by 2^n binary numbers from 0 to 2^n-1. Communication channel links are constructed by connecting any two nodes whose node *ids* differ by exactly one bit position. (see figure 2.1)

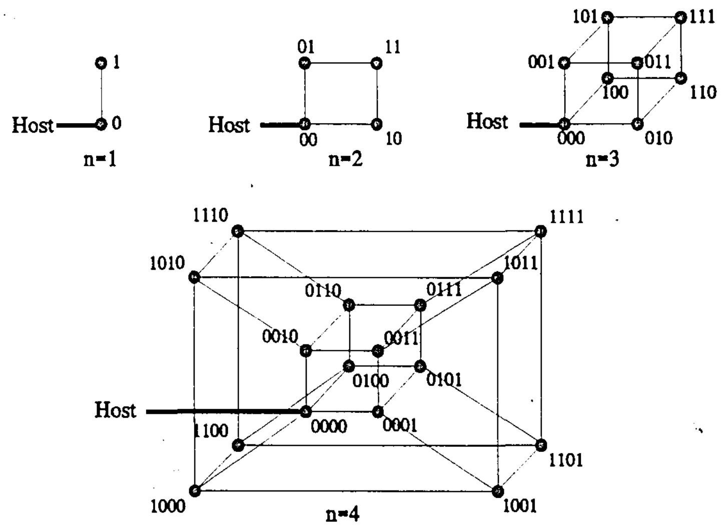


Figure 2.1 Sample hypercube topologies

In general, I/O capability is handled by a computer called the ***host,*** which is connected to any defined network by one or more nodes. This host is a conventional computer with standard I/O devices that acts as an intermediatory between the outside world and the network [6]. In this study, the host is the only link the hypercube network has to the outside world, and vice-versa. Whenever an application is to be run on the hypercube, the program is downloaded from the host. When the data has been transferred to the nodes of the system, the application is processed. Computational results are returned to the host by the active hypercube nodes.

Implementation of the distributed-memory architecture of a hypercube is accomplished by assigning independent portions of local memory to each processor, followed by the linking of each processor to its memory. Processors are granted direct access to their local memory, however, attempts to access or modify data outside of the processors local memory must be achieved via communication through channels. The communication channels are the actual physical linkages of the individual processors that are arranged in the dimensional cube network topology.

2.2 Dimensions and node degrees of the hypercube

The definition of a hypercube network implies that each node possess exactly *n-neighbors;* where a neighbor is defined to be any node with a direct 1-to-1 communication channel link to a given node [2,4,7,12].

proposition 2.1: The maximum degree of any node in a hypercube is ***n*** [4,7,12].

proof: The proof is by example. Assume a binary *3*-dimensional hypercube network, where x is a node that is a member of the network. If the individual digits within x are denoted $x_2,x_1,x_0,$ then by definition of a hypercube there must be channel links that support communication between $x_2,x_1,x_0,$ and the following nodes: (x'_2,x_1,x_0), (x_2,x'_1,x_0), and (x_2,x_1,x'_0).

proposition 2.2: The diameter of a hypercube network is ***n*** [2,4,7,10,12,16,18].

proof: It follows from the proof of proposition 2.1 that if the bit positions from one node to another can only differ by one bit position, the maximum distance ever travelled by a message in the network must occur when every bit position of the source node differs from that of the destination node. In other words, the source node (x_2,x_1,x_0) must be the total opposite of the destination node (x'_2,x'_1,x'_0). The difference in bits in this case is n.

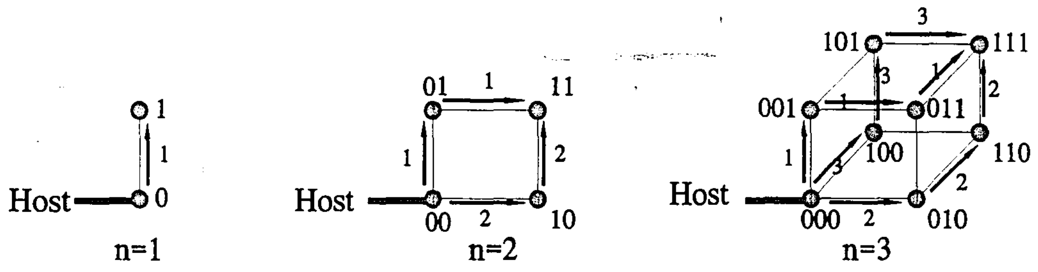
2.3 Hypercube properties

The hypercube network possesses several properties that can be attributed to its symmetric and uniform nature. For example, due to its symmetry and uniformity network routing traffic remains balanced. Other properties such as a singular routing algorithm, and a vast network absorbing capability also exist for this type of network.

2.3.1 Message routing

The general multiprocessor network concept of linking adjacent nodes with communicating channels requires that each node be able to send, route, and receive data (ideally in packets) to/from another node. In the hypercube network, the path from a source node to a destination node is obtained by a step-by-step examination of corresponding node bits. For example, if node $x_{n-1},x_{n-2},\ldots,x_1,x_0$ sends a message to node $y_{n-1},y_{n-2},\ldots,y_1,y_0,$ each bit of the source and destination nodes from the least significant bit (LSB) to the most significant bit (MSB) must be compared. If any bit of the source node, x_m $(0 \leq m \leq n-1)$, differs from its corresponding destination bit, y_m, this bit is complemented to obtain the next intermediate node. This process is repeated until a complete path from the source to the destination node is obtained [2,4,7,10,12]. For example, a valid path from source node *0110* to destination node *1001* is *(0110,0111,0101,0001,1001)*. This algorithm always obtains the shortest path for communication within the hypercube. It should also be noted that if the path is initiated from any of the n bits of the source node, n distinct paths for the message to traverse can be obtained. (see figure 2.2)

When a node sends, routes, or receives data to/from another node, a communication channel path (ideally the shortest path) must be obtained for the data packet to traverse. The minimum path length for any point-to-point node communication is obtained from the number of bits that differ between the source and destination nodes (known as the ***Hamming*** distance).



Source Node = 0	Source Node = 00	Source Node = 000
Dest. Node = 1	Dest. Node = 11	Dest. Node = 111
Path = {1}	Path = {1, 2}	Path = {1, 2, 3}
No. of paths = 1 = n	No. of paths = 2 = n	No. of paths = 3 = n

Key:

Source node: Origin of data/message.
Dest. node: Intended destination of data/message.
Path: Distinct paths that can be travelled.

Figure 2.2 Distinct communication paths

The Hamming distance is obtained as follows:

definition 2.1: ***Bsum()*** is defined as the function that computes the total number of *1s* present in any binary representation. Thus by definition

Bsum(01101) = 3.

definition 2.2: The shortest path length from one node to another is defined as follows:

BSUM((Source_Node_Id) ***XOR*** (Destination_Node_Id)); where XOR is defined as the exclusive-or function.

With the above definition, if one views the *4*-dimensional hypercube shown in figure 2.1 it can be verified that the shortest path length from node *1001* to node *0110* is:

BSUM((1001) ***XOR*** (0110)) = **BSUM**(1111) = **4**.

It would be improper to mention the use of communication channels from one node to another without discussing the possibility of a packet encountering a busy node. In other words, a packet enroute to a particular node may encounter a node already engaged in sending/receiving a packet to/from a neighboring node. This occurs when a transmitted packet, upon encountering a non-busy node, temporarily locks out that node's routing device from other packets to ensure the safe transmission of its data. Current architecture solves this problem by queuing the routed packet in a dedicated device of the busy node. A possible theoretic alternative is to re-route the path of that packet by selecting another communication path. However, re-routing a packet would only be beneficial if a node has a substantial number of messages queued and waiting to be forwarded.

2.3.2 Uniformity and symmetry

From the above sections it is apparent that the hypercube network is a totally uniform and symmetric network. When the topic of uniformity and symmetry is considered, several questions are raised regarding the redundancy of design, and the possibility of a singular routing algorithm for all nodes.

Redundancy in design examines the possibility of a network so well designed that each node of the system is identical in nature and design. A singular node design implies nodes will possess the same node degree, the same memory organization/capacity, and the same processing capabilities. The need for uniformity is better understood when one considers the cost associated with creating a non-singular node design for a network. The entire life-cycle for the creation of this network, in terms of cost and time, is proportional to the irregularity of the system [13].

The concept of a single and simple routing algorithm explores the possibility of a symmetric network. If a network is symmetric, nodes may pass packets/messages to neighbors in the same manner. If a network was non-symmetric a routing algorithm would become harder to derive and implement [13].

Both these concepts have been shown to exist within the hypercube network.

2.3.3 Expanding and tearing

With the definition of a hypercube comes several advantages that can be attributed to the symmetry of the hypercube network. One such advantage is the ability to construct with relative ease a cube of higher or lower dimensions.

In the construction of a higher dimension hypercube (expansion), two identical *(n-1)*-cube networks are obtained, and each of these sub-cubes are numbered *0* to 2^n*-1*. The higher dimension cube is then constructed by creating communication channel links between the nodes of the first *(n-1)*-cube to their identical nodes (node with matching ids) on the second *(n-1)*-cube. One must realize that each node of the first cube now has an id identical to that of the second cube. This problem is corrected by the concatenation (represented by |) of a *0* to the left-side of the first cube's node ids, and a *1* to the left-side of the second cube's node ids [2]. (see figure 2.3)

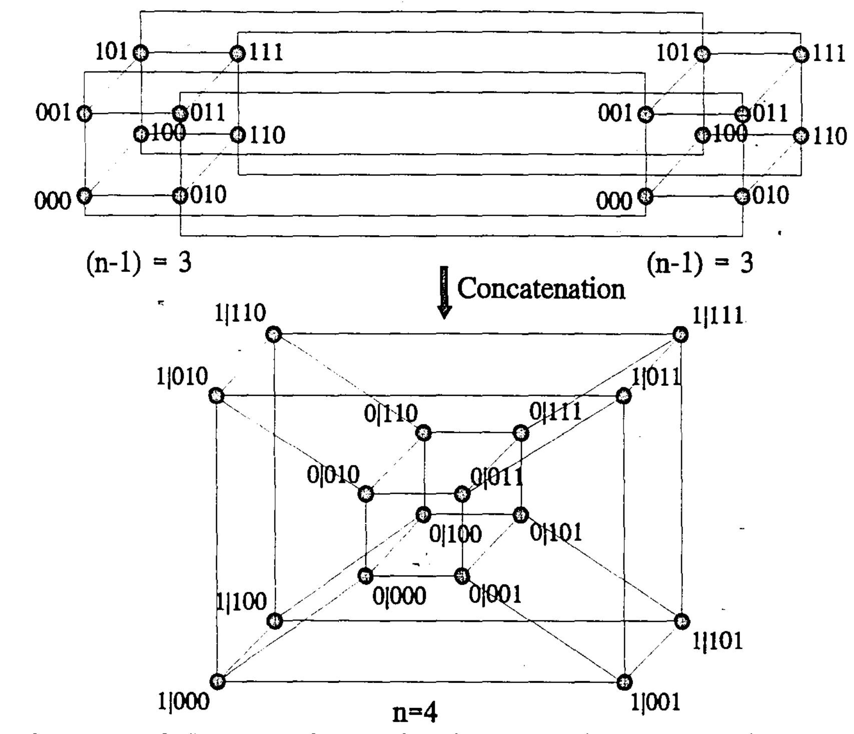


Figure 2.3 Expansion of a hypercube network

In a similar manner, a binary *n*-cube can be *torn* into two *(n-1)*-cubes, where one cube's most significant node id bits are *0s* and the other's are *1s.* The tearing of a binary *n*-cube does not necessarily have to occur along the MSB of the binary *n*-cube's node ids. In other words, the process of tearing can be applied to any bit position of the binary *n*-cube, with the stipulation that the sub-cubes created from the tearing of the i^{th}-bit position $(i < n)$ differ by the bit numbers *0* and *1* [2]. It can be inferred from the above that it is possible to tear a binary cube in *n* different ways. Each tearing permits a binary *n*-cube to be split into two sub-cubes. One sub-cube then possesses a *(n-1)*-bit node id whose i^{th}-bit is always a *0,* and the other sub-cube possesses a *(n-1)*-bit node whose i^{th}-bit is a *1.*

2.3.4 Absorbing networks into the hypercube

The main advantage of the hypercube network is it's ability to absorb several networks. This ability permits the binary *n*-cube to be recognized as a system suitable for several fields of interest.

2.3.4.1 Mapping rings and linear arrays onto a hypercube

Hypercube networks can be shown to handle with great efficiency any application suited for a linear array network or a ring network [16,18]. The mapping of a linear array is trivial, and similar to that of the ring network. Only the ring network is analyzed in this section.

The mapping of 2^n processing nodes in a ring network onto a hypercube network is obtained by traversing our cube in a *Hamiltonian* path which is defined as follows:

definition 2.3: Define the *n* bits of binary cube id of 2^n numbers to be ***Gray(n)*** [16]. The *binary-reflected* Gray code on *n* bits is defined recursively as follows:

$$\text{if } \mathbf{Gray(n)} = \{G_0, G_1, \ldots, G_{2^n-2}, G_{2^n-1}\}$$
$$\text{then } \mathbf{Gray(n+1)} = \{0|G_0, 0|G_1, \ldots, 0|G_{2^n-2}, 0|G_{2^n-1}, 1|G_{2^n-1}, 1|G_{2^n-2}, \ldots, 1|G_1, 1|G_0\}.$$

Thus for $n=2$ and $n=3$ the following is obtained:

$$\mathbf{Gray(2)} = \{00,01,11,10\},$$
$$\text{and } \mathbf{Gray(3)} = \{000,001,011,010,110,111,101,100\},$$

which translates into the following mapping shown in figure 2.4.

From figure 2.4, the simplicity of mapping 2^n processors of a ring network onto a hypercube network is easily realized.

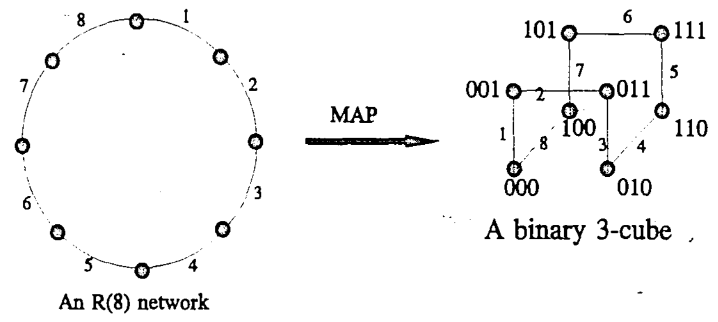

8
1
2
3
4
5
6
7
An R(8) network
MAP
101
111
001
011
100
110
000
010
A binary 3-cube
Key:
Dotted lines represent communication channel links which are not needed to implement the ring network

Figure 2.4 Mapping a ring network onto a hypercube network

2.3.4.2 Mapping a binary tree network onto a hypercube

The hypercube network consisting of 2^n processing nodes can only absorb a level binary tree of $2^{n-1}-1$ nodes [4,16,18,]. (see figure 2.5)

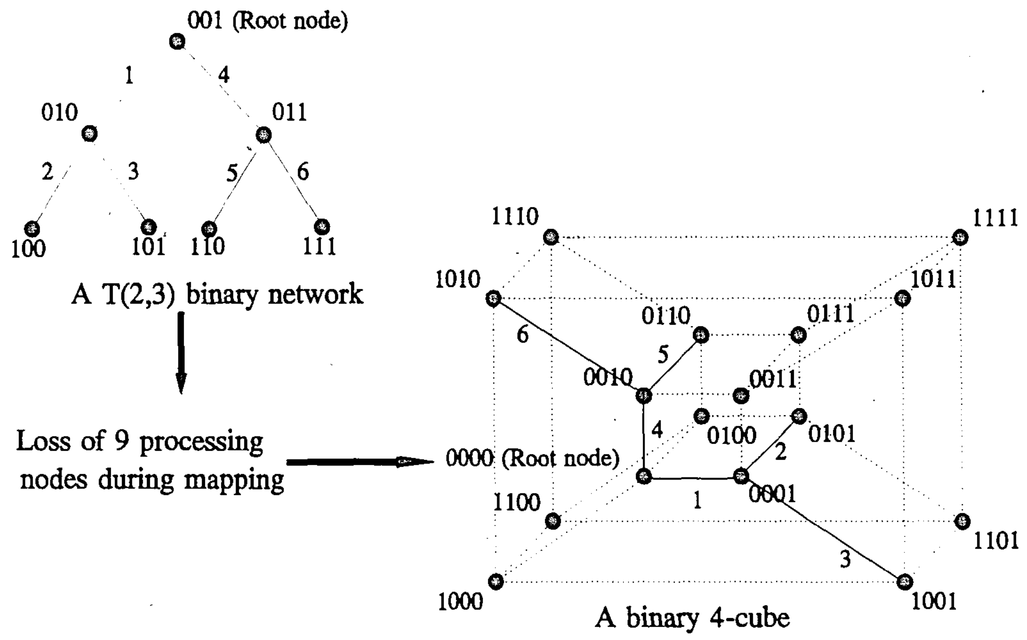


Figure 2.5 Mapping a binary tree onto a hypercube network

The failure to accommodate any other node is due to the symmetric nature of the hypercube network which contains redundant nodes at level $n-1$ of the given binary tree. This symmetry causes the waste of $2^n-2^{n-1}+1$ processing nodes in any hypercube network that attempts to emulate a binary tree network of $2^{n-1}-1$ nodes. Clearly, this waste indicates a weak point of the hypercube network.

2.4 Message broadcasting on a hypercube

Before progressing any further, the stage for analyzing broadcasting algorithms must be set. It is assumed that all data is transmitted in the form of a packet, and in the following algorithms the time taken for a packet to cross one communication channel link is *1* time unit. Packets may be transmitted in both directions on a communication channel, and for the sake of simplicity transmission of packets are completely error-free. Once a packet is placed on a communication channel line no other packets may be transmitted on that line. If more than one packet attempts to use a communication channel line, only one will be transmitted on that communication channel link for a period of *1* time unit while the other will be placed in a first-in-first-out (FIFO) queue. Each node is assumed to possess an infinite storage space. All communication channel links can receive and transmit packets simultaneously, this capability is termed the ***Multiple Link Availability (MLA)*** assumption [4].

Single node broadcasting involves the transmittal of an identical packet from a source node to all other nodes [4,10,16]. The approach to this problem is to transmit the packet along a ***directed spanning tree*** commencing from the source node [4,7,10,16]. A spanning tree is a tree which visits all nodes of a network by unique directed paths and is constructed as follows:

step 1. Select a source node. Create the first level by visiting each bit from the LSB to the MSB, invert each bit.

step 2. Traverse each *n*-node by progressing one bit to the right of the last bit inverted, each bit visited is inverted until the LSB is reached. After processing the LSB the progression is resumed from the MSB until the bit of origin for each node on level *1* is reached (see figure 2.6). This method eventually leads to a single node *n* times and all but one of these recurring sub-paths must be discarded.

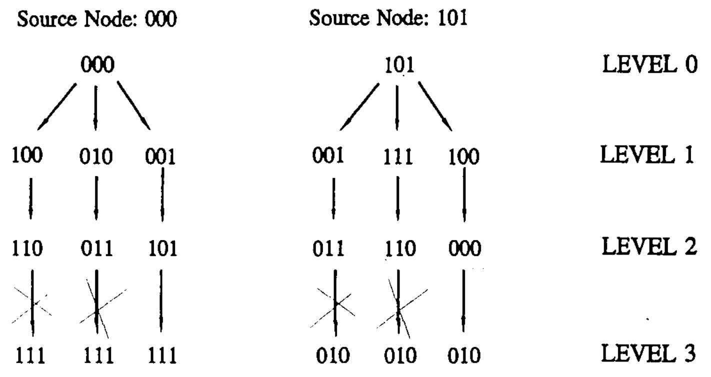


Figure 2.6 Creation of spanning trees for two nodes of a 3-cube

2.4.1 Timing analysis of a single node scatter broadcast

In this problem a single node attempts to transmit 2^n-1 different packets to 2^n-1 nodes. The transmission count is $n2^{n-1}$, because the selected node must send 2^{n-1} packets that must traverse n levels [4]. (see figure 2.6)

The lower bound time required is $\lceil \frac{2^n-1}{n} \rceil$; where each node receives a total of 2^n-1 packets over n communication channel links [4].

2.4.2 Timing analysis of a multinode broadcast

The multinode broadcast executes a single node broadcast simultaneously from all nodes. This problem can be solved by specifying one spanning tree per root node. However, problems arise when some communication channel links belong to more than one spanning tree. The timing analysis is affected as several packets arrive simultaneously at a node and all request transmission on the same communication channel link resulting in a FIFO queue.

The number of transmissions needed can be obtained by accepting the fact that each node sends a packet to 2^n-1 nodes and that there are 2^n nodes. The transmission count is given as $2^n(2^n-1)$ [4].

In figure 2.6, each transmitted packet eventually meets at a common node. Thus, all but one of the last transmission time units for the common node must be removed. The final timing analysis formula for the multinode broadcast is $\lceil \frac{2^n-1}{n} \rceil$ [4]. This value represents a lower bound for the time required by the specific broadcast algorithm.

2.4.3 Timing analysis of a total-exchange broadcast

Similar to the problem of a single node scatter, each node attempts to transmit a different packet to every other node. In the total-exchange broadcast, 2^n nodes transmit packets that will traverse n levels of a spanning tree. Assuming that each node transmits packets to 2^n-1 nodes, the transmission count is $n2^{2n}-1$ [4].

The lower bound on this transmission time is 2^n-1 [4]. This time is achieved due to the overlapped packet transmission in an attempt to keep all communication channels and ports busy.

CHAPTER 3

The Binary de Bruijn Multiprocessor Network

3.1 Definition of a de Bruijn network

A de Bruijn multiprocessor network is a distributed-memory parallel computer consisting of R^n processing nodes; where R represents the radix of the system and n is the number of digit positions used to identify all nodes within the system. The de Bruijn network, denoted ***dB(R,n)***, possesses a bi-directional communication channel link between two nodes x and y, if the n-1 last digits of node x are equal to the n-1 first digits of node y. In other words, there exists a communication channel link between the two nodes $x_{n-1},x_{n-2},\ldots,x_1,x_0$ and $x_{n-2},x_{n-3},\ldots,x_0,x_{-1}$; where x_{-1} is any digit in the given radix [8,13,17]. With this definition in place a de Bruijn network can be easily constructed by creating R^n processing nodes labeled from 0 to R^{n-1} of the given radix. Communication links are placed between nodes that meet the above specified criteria. (see figure 3.1)

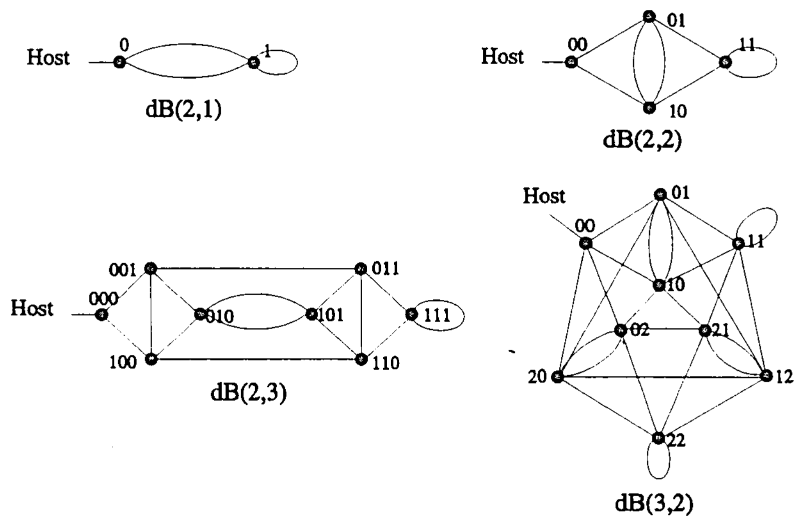


Figure 3.1 Sample de Bruijn topologies

From the definition of a de Bruijn network there must be at least two redundant self loops on the beginning and ending nodes of the system. These links are of no particular benefit to the system and as a result are disconnected from the network, or in the case of the root node, connected to the host computer.

The distributed-memory architecture of the de Bruijn network is constructed in the same fashion as that of the hypercube network in chapter 2. It is accomplished by the assignment of a local memory module to each processing node for its individual use. Access to data outside of a nodes range is achieved by message passing.

Although several other radix-based networks exist for the de Bruijn network, the main focus in this paper is on the binary de Bruijn network which is denoted ***dB(2,n)***.

3.2 Dimensions and node degrees of the de Bruijn

It follows from the definition of a de Bruijn network that each node within a de Bruijn network must have a maximum of *2R* neighbors.

Proposition 3.1: The maximum node degree in a binary de Bruijn network is *4* [13,17].

Proof: The proof is by example. Assume a *dB(2,3)* network, where x is a node that is a member of the network. If the individual digits within x are denoted as $x_2,x_1,x_0,$ then according to the definition there must exist channel links to support communication between x_2,x_1,x_0 and the following nodes: x_1,x_0,x_* and x_*,x_2,x_1; where x_* is a binary digit. In the case of a binary de Bruijn network this can only occur in four ways: $(x_1,x_0,0)$, $(x_1,x_0,1)$, $(0,x_2,x_1)$, and $(1,x_2,x_1)$, for all sizes of n.

Proposition 3.2: The diameter of any binary de Bruijn network is ***n*** [8,13,17].

Proof: The proof is similar to that of *proposition 2.2*.

3.3 de Bruijn network properties

At this point, it should be apparent that a network's properties are what makes a network amiable to system designers. The properties of a network eventually make or break the system, and as a result the search to discover new topologies which encompass an even greater set of attributes continues. As discussed earlier, several property trade-offs exist within any system, and the de Bruijn multiprocessing network is in no way an exception to the rule.

3.3.1 Message routing

With any multiprocessor topology, the designer's goal is to create a simple routing algorithm that will not sacrifice other attributes of the system. This goal is easily accomplished in the de Bruijn network via node id shifting. For example, to send a message from source node $x_{n-1},x_{n-2},\ldots,x_0$ to destination node $y_{n-1},y_{n-2},\ldots,y_0$ one need only perform n L_s operations on both nodes. Intermediate nodes are created from the incoming least significant bit (LSB) of the source node; which is the most significant bit (MSB) of the destination node. Thus the path from $x_{n-1},x_{n-2},\ldots,x_0$ to $y_{n-1},y_{n-2},\ldots,y_0$ is as follows: $(x_{n-1},x_{n-2},\ldots,x_1,x_0)$, $(x_{n-2},x_{n-3},\ldots,x_0,y_{n-1})$, ..., $(x_0,y_{n-1},\ldots,y_2,y_1)$, $(y_{n-1}, y_{n-2},\ldots,y_1,y_0)$ [13,17].

Utilizing the previous message routing principle, it follows that another trivial path can be obtained from performing n right shifts on the source and destination nodes. In particular, the path obtained is as follows: $(y_0,x_{n-1},\ldots,x_1,x_0)$, $(y_1,y_0,\ldots,x_2,x_1)$, ..., $(y_{n-2},y_{n-3},\ldots,y_0,x_{n-1})$, $(y_{n-1}, y_{n-2},\ldots,y_1,y_0)$. It should be apparent that these paths are not necessarily distinct. For example, the two paths from source node *100* to destination node *101* are:

path 1: *(100), (001), (010), (101).*
path 2: *(100), (110), (010), (101).*

Although not plainly obvious, these routing algorithms reinforce the concept of the diameter of a binary de Bruijn network. This is due to the fact that a valid path between two nodes will be obtained in at most *n* shifts.

One must realize that the previous routing algorithms, although relatively simple, do not necessarily obtain the shortest path between two nodes. For example, the shortest path from source node *001* to destination node *100* is not *(001, 011, 110, 100)*, but rather, the direct path from *001* to *100*. In theory, several algorithms can be obtained to extract the shortest paths between nodes, however, these paths are obtained at an increase in complexity to the system.

3.3.2 Uniformity and symmetry

From the previous de Bruijn network definitions and routing algorithms it can be seen that this kind of multiprocessor network is indeed uniform. Uniform in the sense that there is a singular design for all nodes [13]. In particular, the node design and degree will remain the same for all nodes. (see figure 3.2)

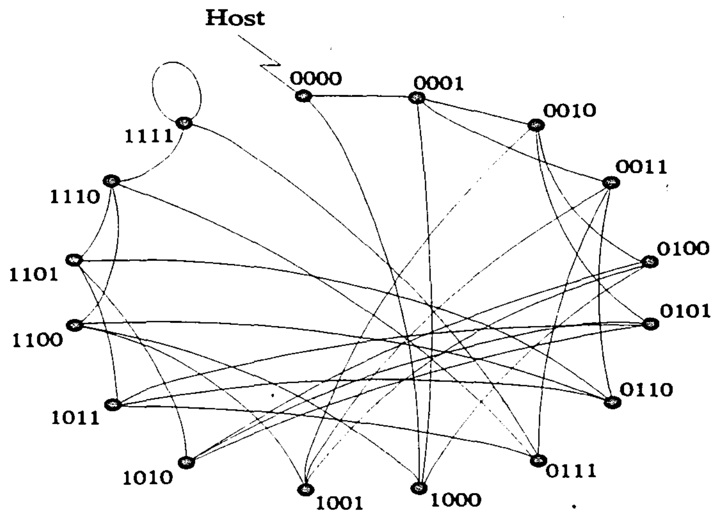


Figure 3.2 Sample dB(2,4) network

The de Bruijn network, however, is not fully symmetric due to such attributes as the redundant self loops of the system [13]. This partially un-symmetric behavior becomes more apparent when one views the *dB(2,4)* network shown in figure 3.2.

3.3.3 Expanding and tearing

The advantage of any de Bruijn network is fully realized when one considers the topic of expanding and tearing the network.

Expansion is achieved by placing R^{n+i}-R^n additional nodes in the network and numbering them from R^n to R^{n+i}-1; where $i=n_{new}-n_{old}$, n_{new} is the new node id digit length and n_{old} is the old node id digit length [13,17]. The definition of the communication channel link is applied to the new system to obtain its complete topology. (see figure 3.3)

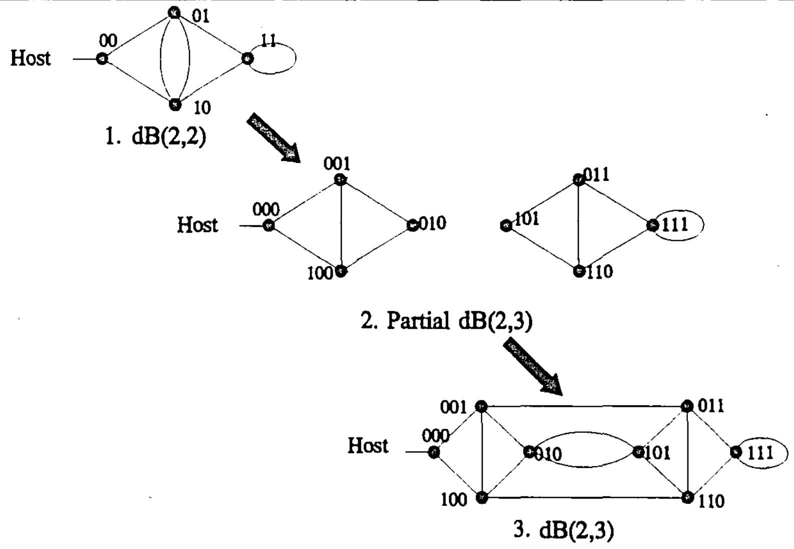


Figure 3.3 Expansion of a dB(2,2) network to a dB(2,3) network

In the case of tearing, the nodes of the network need not be removed from the system, instead communication channel links are simply re-routed.

Unlike the hypercube, an increase or decrease in the number of nodes does not necessitate a change in the design of the processing nodes. All that is needed in the expansion case is the re-configuration of the communication channel links with the additional nodes. For tearing, the system required a re-configuration of the communication channel links, leaving excess nodes temporarily inactive. If the de Bruijn system is designed properly, the cost of physically upgrading/downgrading the number of processing nodes will be at a mere fraction of the cost of other systems.

3.3.4 Absorbing networks into the binary de Bruijn

As mentioned previously a goal of every multiprocessor network designer is to create a network that is able to imitate other topologies. In the following sections the binary de Bruijn network is shown to succeed in absorbing the following networks: a linear network, a ring network, a binary tree network, and a shuffle-exchange network.

Of the above listed networks only the shuffle-exchange network can not be absorbed by the hypercube topology, this is due to the hypercubes inability to create a communication channel link between two nodes whose bit positions differ by more than one bit. However, in the scope of this paper, this is not a shortcoming of the hypercube as one could easily argue that the binary de Bruijn network rejects some topologies a hypercube easily absorbs. (i.e. the torus network)

3.3.4.1 Mapping rings and linear array onto a binary de Bruijn

It is simple to establish that a binary de Bruijn network possesses both a hamiltonian path as well as a hamiltonian cycle [13,17]. In fact, the binary de Bruijn network possesses several such paths.

The mapping of both the linear array network and the ring network onto a binary de Bruijn network is trivial, and similar to the mapping discussed in the hypercube section (chapter 2).

3.3.4.2 Mapping a binary tree onto a binary de Bruijn

A binary tree of 2^n-*1* nodes can be absorbed into a binary de Bruijn network, *dB(2,n)*, by selecting a root node and applying the following algorithm:

step 1. To create the left child of a parent node perform a left-shift operation on the parent's node id.

step 2. The right child of a parent node is obtained by performing *step 1* and then adding the value *1* to the resulting node id.

step 3. Repeat *step 1* and *step 2* at each level of the binary tree until all but one node of the binary de Bruijn network have been utilized.

This absorption is shown in figure 3.4.

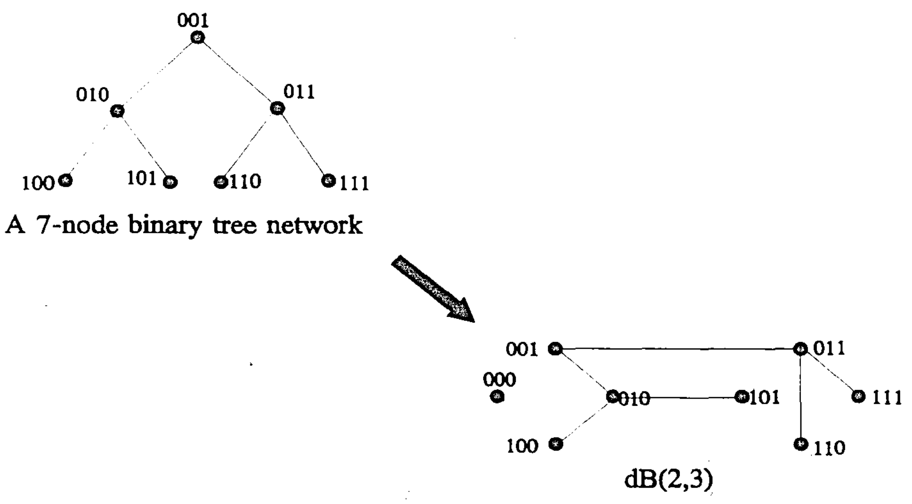


Figure 3.4 Mapping a binary tree onto a binary de Bruijn network

The reason a binary tree is absorbed by a binary de Bruijn network is not easily comprehensible. The topology and node id placement of a binary tree is remarkably similar to that of the binary de Bruijn network. From the previous definitions, it was stated that a communication channel link exists between nodes x and y, if the $n-1$ last digits of node x were equal to the $n-1$ first digits of node y. In other words, there was a communication channel link between the two nodes $x_{n-1},x_{n-2},\ldots,x_1,x_0$ and $x_{n-2},x_{n-3},x_0,x_{-1}$; where x_{-1} for the binary de Bruijn case, is a binary digit (0 or 1) [17]. These statements are similar to those utilized in the labelling of a binary tree.

3.3.4.3 Mapping a shuffle-exchange onto a binary de Bruijn

As defined earlier, the shuffle-exchange network possesses a communication channel link between two nodes x and y, if-and-only-if: $y=Sh(x)$, or $x=Sh(y)$, or $y=Ex(x)$ [17]. Therefore, node *010* is connected to nodes *(100, 001,011)*. Likewise, the node *000* can only be connected to the node *001*. (see figure 3.5)

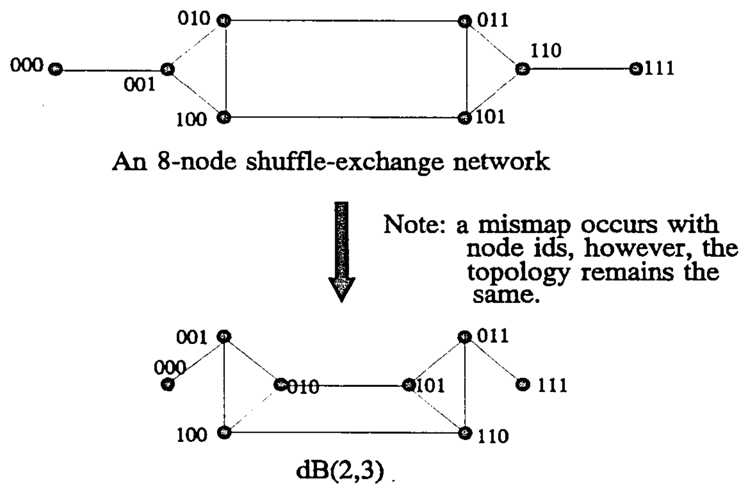

010
011
000
001
110
111
100
101
An 8-node shuffle-exchange network
Note: a mismap occurs with node ids, however, the topology remains the same.
001
011
000
010
101
111
100
110
dB(2,3)

Figure 3.5 Mapping a shuffle-exchange network onto a binary de Bruijn network

The definition of the shuffle-exchange network closely resembles that of the de Bruijn network, and as result it is no surprise that the shuffle-exchange network can be absorbed by the binary de Bruijn topology.

3.4 Message Broadcasting

In accordance with the MLA protocol and broadcasting rules discussed in chapter 2, the broadcast analysis of a de Bruijn network is complex in nature. In particular, broadcasts are placed in a best-case or worst-case category for cases were nodes must transmit 2^n-*1* different packets. The creation of these categories is necessary because the routing algorithm employed for the network is restrictive in nature.

Worst-case broadcasts are achieved from the root node and last node of the binary de Bruijn network. This can be attributed to the redundant self-loops of the network which restrict parallel transmissions/receptions to/from other nodes. All remaining nodes within the network fall under the category of best-case broadcasting nodes. (see figure 3.6)

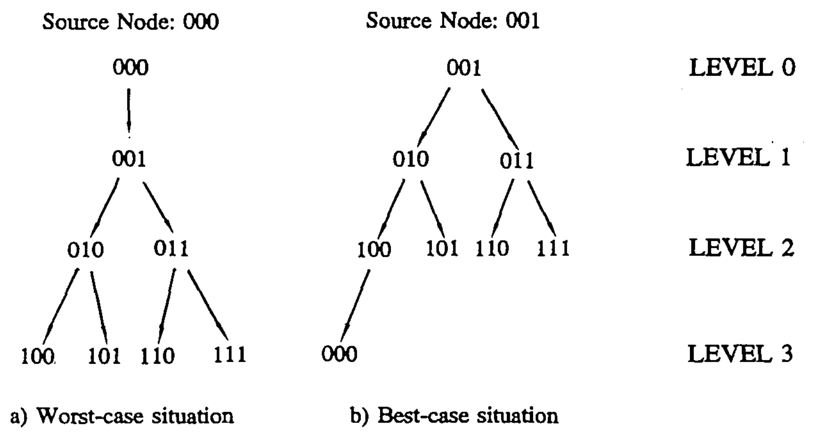


Figure 3.6 Worst-case and Best-case spanning trees of a dB(2,3) network

In single node broadcasting packets take *n* time units regardless of the best or worst case spanning trees. The proof for these broadcast times is embedded in the expansion and packet distribution properties of binary tree networks [12,13,17].

3.4.1 Timing analysis of a single-node scatter broadcast

Broadcast times are achieved by combining the two categories of broadcast times together. A single-node scatter entails the transmission of 2^n-1 different packets to 2^n-1 different nodes within the network.

If the node is identified as a worst-case node the broadcast time is given as 2^n-1, because 2^n-1 packets must be transmitted in sequential (see figure 3.6.a) order from the node. Best-case nodes, however, achieve broadcast times of 2^{n-1}.

3.4.2 Timing analysis of a multi-node broadcast

A multi-node broadcast entails the broadcast of the same packet to all nodes from each node in the network.

This broadcast is simply a combination of the total time for worst-case and best-case single node broadcasting in the entire network. The broadcast is therefore expressed as $2(2^n-1) + n(2^n-2)$, which is equivalent to $2(2^n + n2^{n-1} - n - 1)$.

3.4.3 Timing analysis of a total-exchange broadcast

Neglecting node contention time for buffered packets on communication channels, the best-case total-exchange broadcast on a binary de Bruijn network is expressed as $2(2^n-1) + (2^n-2)(2^{n-1}) = 2^{2n-1}-2$. In this broadcast, the worst-case time is 2^n-1, hence the $2(2^n-1)$ term. Likewise, the best-case broadcast is achieved by 2^n-2 processing nodes where each node's broadcast takes 2^{n-1} time units.

CHAPTER 4

An Analysis of a Select Group of Sorting Algorithms

4.1 Introduction

Sorting algorithms play a major role in a significant number of computational algorithms. It is for this reason that sorting algorithms were utilized for the comparison between the hypercube and de Bruijn multiprocessor networks.

Sorting, as used in these computational algorithms, entails the organization of data (not necessarily numeric in value) into some logical structured order. In the case of an application which utilizes several thousand numerical values, it is apparent that processed data must eventually be arranged into an easily comprehensible form. This form can be obtained by multiprocessor networks at a faster rate than that of any conventional computer system. Examining the case of several thousand numerical values, a single sequential processor is inefficient due to its inability to compare more than two values at one point in time. However, these comparisons are overlapped by a multiprocessor network, and are only hampered by the number of available processing nodes.

In this study, as multiprocessor networks are implemented to utilize the distributed-memory organization, the question of where sorted data finally resides must be addressed. Sorted data may eventually reside in a single node or may remain scattered in some order throughout the network [5]. It is important, however, that the final placement of the sorted data is dependent on the chosen application.

Initially, *M* data items in the root node are split into *M/N* lists, where *N* is the number of processors in the network. The lists are then distributed for sorting throughout the network. When all distributed sorting is complete, sorted items are returned (in order) to the root node to be merged into a single sorted list.

In the following chapters, selected algorithms presented in the next sections are utilized to perform the comparison between the hypercube and de Bruijn networks. In an attempt to obtain unbiased execution times the algorithms implemented on the hypercube and binary de Bruijn networks must be identical. This stipulation implies that the difference obtained from the execution times depends solely on the efficiency of the routing algorithms of both networks.

The bubble sort, insertion sort, shell sort, and quick sort are all sorting algorithms worth examining in this chapter. The knowledge obtained from the implementation and the efficiency of each sort is utilized to select a sorting algorithm that best suits this study's need.

4.2 The bubble sort algorithm

The bubble sort is a well known algorithm primarily because of its simplicity, however, it turns out that this sort is the least efficient [19]. The general concept of the bubble sort can be described as follows. Given an initial unsorted list of M elements, pass through the list and compare adjacent pairs of items. Whenever a pair of items are out of order with respect to each other, swap them. The first pass through a list of M items ensures that the last item (in logical value) will be deposited in the last location of the list. (see figure 4.1.) From this point it is easy to see that the last item in the list need no longer be included in the sort. The second pass now only needs to visit the first

M-1 items in the list, and the third pass only visits the first *M-2* list items. This process

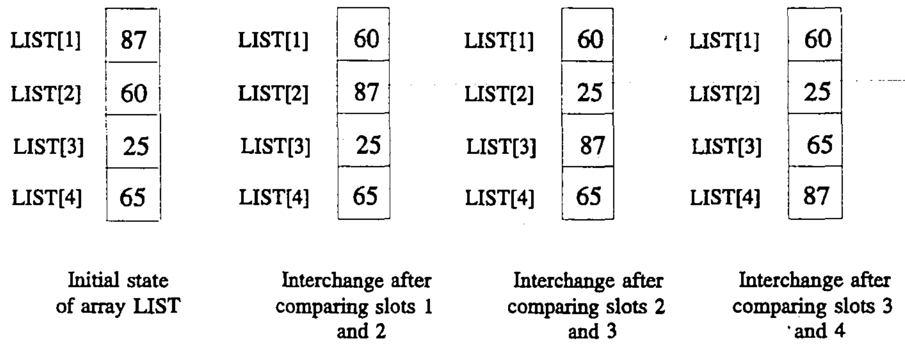


Figure 4.1 First pass through a list using a bubble sort

is continued until *M-1* passes have been completed.

4.2.1 Efficiency of the bubble sort

The efficiency of the bubble sort can be measured by the number of comparisons it requires to sort a list of *M* items. Initially, *M-1* comparisons are made, however the number of comparisons decreases by a factor of one on each pass until, in the final pass, only one comparison is made. On average, as *M/2* comparisons are made per pass, the actual efficiency of the bubble sort algorithm is expressed as $(M-1)(\frac{M}{2})$ [19] .

For large *M,* however, the M^2 term prevails. The efficiency of the bubble sort is therefore accepted to be $O(M^2)$.

4.3 The insertion sort algorithm

It is the goal of the insertion sort to insert in the *i*th pass the *i*th element in *LIST[1], LIST[2], ..., LIST[i]* in its correct location. (see figure 4.2)

LIST[M]	i = 2	i = 3	i = 4
87	60	25	25
60	87	60	60
25	25	87	65
65	65	65	87
Initial state of array LIST	Interchange after comparing slots 1 and 2	2 interchanges after comparing slots 2 and 3	Interchange after comparing slots 3 and 4

Figure 4.2 Insertion sort algorithm applied to a list

In *Naps et al.* *[19],* the sorting process is expressed in the following steps:

step 1. Set $j = 2$, where j is an integer.

step 2. Check if *LIST[j]* < *LIST[j-1]*. If so interchange them; set $j = j - 1$ and repeat *step 2* until $j = 1$.

step 3. Set $j = 3, 4, 5, \ldots, M$ and keep on executing *step 2.*

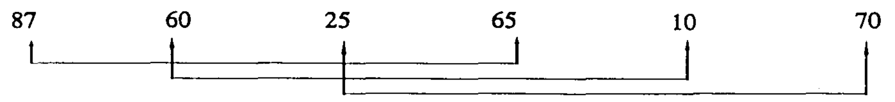
4.3.1 Efficiency of the insertion sort

Although the insertion sort is almost always better than the bubble sort, the time element in both methods remains at $O(M^2)$ [19] .

In the case of partially sorted data, the insertion sort normally takes less time than the bubble sort. The number of interchanges needed in both the methods is on the average $\frac{M^2}{4}$, and in the worst cases about $\frac{M^2}{2}$ [19] .

4.4 The shell sort algorithm

The shell sort algorithm is based on the concept that partially sorted lists require less comparisons and interchanges to achieve the goal of becoming a sorted list [18,19]. Instead of sorting the M items of a list immediately, the list is divided into smaller segments which are then separately sorted using the insertion sort [19,20]. (see figure 4.3)



First divide the list into 3 segments of 2 elements each.

87	65	-----	segment 1
60	10	-----	segment 2
27	70	-----	segment 3

and then sort each of the segments:

65	87
10	60
27	70

Figure 4.3 First pass of a shell sort on a list

The key to the shell sort algorithm is that the whole array is first fragmented into K segments for some number K, where K is preferably a prime number [19,20]. If the size of the list array *LIST* is M, then the segments are:

*LIST[1], LIST[K+1], LIST[2*K+1], ..., LIST[M/K+1]*
*LIST[2], LIST[K+2], LIST[2*K+2], ..., LIST[M/K+2]*

.

.

.

*LIST[K], LIST[2*K], LIST[3*K], ..., LIST[M/K+K].*

As each segment is already sorted, the whole array is partially sorted after several passes. In following passes, the value of K is reduced, which increases the size of each segment, and also reduces the number of segments. The next value of K is also chosen so that it is *relatively prime* to its previous value. (Two integers are said to be relatively prime to each other if they have no common factor greater than 1.) This process is repeated until K = 1, at which point the list is sorted [19]. Each segment is sorted by the insertion sort method, so that each successive segment is partially sorted. During the later phases of the sort the insertion sort increases in efficiency, which in turn increases the overall efficiency of the shell sort [19,20].

4.4.1 Efficiency of the shell sort

The shell sort is also called the *diminishing increment sort* because the number of segments, *K*, continually decreases. The method is considered to be more efficient if the successive integers are relatively prime to each other [19,20]. D. E. Knuth in [20] estimates the average execution time of the shell sort with relatively prime integers to be proportional to $O(M(\log_2 M)^2)$. The shell sort also works for any value of *K* greater than 1. However, when the values of *K* are not relatively prime, then the efficiency of the sort is given as $O(M^r)$, $where\ 1<r<2$ [19,20].

The shell sort is most efficient on arrays that are already nearly sorted. In fact, the first chosen value of *K* should be large to ensure that the whole array is fragmented into small individual arrays, for which the insertion sort is highly effective [19].

4.5 The quick sort algorithm

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in a list [18,19]. Utilizing a divide-and-conquer approach, this algorithm avoids the unnecessary swapping of data items by moving an item a great distance in one move. A pivotal item is selected and moves are made so that data items on one side of the pivot are smaller than the pivot, and data items on the other are greater than the pivot [18,19]. The pivot is now in its correct position. The algorithm is applied recursively to the parts of the list on either side of the pivot until the whole list is sorted. (see figure 4.4)

LIST[1], LIST[2], ..., LIST[10]

15*	20	5	8	95	12	80	17	9	$\overline{55}$
9	$\overline{20}$	5	8	95	12	80	17	( )	55
9	( )	5	8	95	12	80	$\overline{17}$	20	55
9	12	$\overline{5}$	8	95	( )	80	17	20	55
9	12	5	8	( )	95	$\overline{80}$	17	20	55
9	12	5	8	15	95	80	17	20	55

* - indicates selected pivot item

— - indicates next item in list to be positioned in respect to pivot

( ) - indicates open slot position of item selected for swap

Figure 4.4 Quick sort of a list

4.5.1 Efficiency of the quick sort

The quick sort algorithm only performs efficiently if its pivots are selected as close as possible to the median of the list to be sorted. If care is not taken to select the appropriate pivot, the sorting efficiency and speed are hampered by several re-orderings of the list.

The average run-time efficiency of the quick sort is given as $O(M\log_2 M)$ [19]. However, the quick sort efficiency can drop to that of $O(M^2)$ due to the continuous right to left scan all the way to the last left boundary in the case of a badly selected pivot

[19].

The quick sort algorithm is hampered by its need to maintain a stack for the temporary storage of pivoted items [19]. This need, however, can be avoided by use of a language that permits recursion.

4.6 Selecting sorting algorithms for multiprocessor networks

Although it is the intention of this study to compare the hypercube and binary de Bruijn networks, it is also a goal of this study to prove the efficiency of multiprocessor networks over single-processor computers. This goal can be achieved by implementing an inefficient sorting algorithm, such as the bubble sort, on a single-processor computer and comparing the results to that of the hypercube and binary de Bruijn multiprocessor networks.

In this manner, a list of M items can be divided into a list of $M/2^n$ items which are distributed and sorted concurrently on each node of the multiprocessor networks via the bubble sort algorithm. However, when the partial lists have been sorted and returned they still need to be merged.

Although the quick sort is the most efficient algorithm discussed, its need for a stack or a recursion capable language for the temporary storage of its pivot points increases its complexity. In fact, the language utilized in this study, *Occam 2* (chapter 5), does not support recursion and for this reason the quick sort must be discarded from the final sort/merge selection process.

It has been shown that the insertion sort is merely a subset of the shell sort and that the shell sort is particularly well suited to sorting a partially sorted list. The shell sort was therefore the appropriate choice for the merging process of the 2^n lists. As the shell sort is selected for the merge task it is implemented on the root node. (i.e. processor collecting the sorted lists)

The pseudo-code for implementing this hybrid sorting algorithm on the multiprocessor networks is shown in chapter 6 of this thesis, and the final results obtained are listed in the concluding chapter, chapter 7.

CHAPTER 5

Transputers and Occam

5.1 Introduction to transputers

The transputer is a single-chip reduced instruction set computer (RISC) designed with the principle of message passing [3]. This device is composed of a high-speed processor with local memory and four inter-processor communication channel links. Developed by SGS-THOMSON Microelectronics (formally INMOS Ltd.), the transputer is intended for use in parallel processing environments and applications. The major features of transputers include:

- High speed integer (and for T800 and T9000-floating point) processor;
- On-chip fast static memory;
- Four serial bi-directional communication channel links;
- Internal timers, and
- External memory interface.

Communication is achieved on a point-to-point basis where a transputer is connected to another through one or more serial communication links. Once these links are established, transputers may communicate with each other via defined protocols.

The transputer family consists of the IMS T212, IMS T225, IMS T400, IMS T414, IMS T425, IMS T800, IMS T801, IMS T805, and the latest T9000 floating point transputer. These transputers differ mainly by their data and address bus widths. Others such as the T800, T801, T805 and T9000 also differ by their increased internal random access memory (RAM) and hardware floating point units.

The central processing unit (CPU) of a transputer possesses extra functionality in high-level languages and timers, which raises some questions as to its RISC classification. Also, it should be noted that the transputer does not possess memory management capabilities, and only implements multitasking via hardware as opposed to the conventional operating system method.

In this study, the T805 transputer is utilized in the SuperSetPlus machine to implement the sorting algorithms on the hypercube and binary de Bruijn networks.

5.2 The IMS T805 transputer

The IMS T805 is a transputer which possesses a 32-bit CMOS microcomputer with a 64-bit floating point unit [15]. The T805 is equipped with 4 Kbytes of RAM, a configurable memory interface and four INMOS communication links. (see figure 5.1)

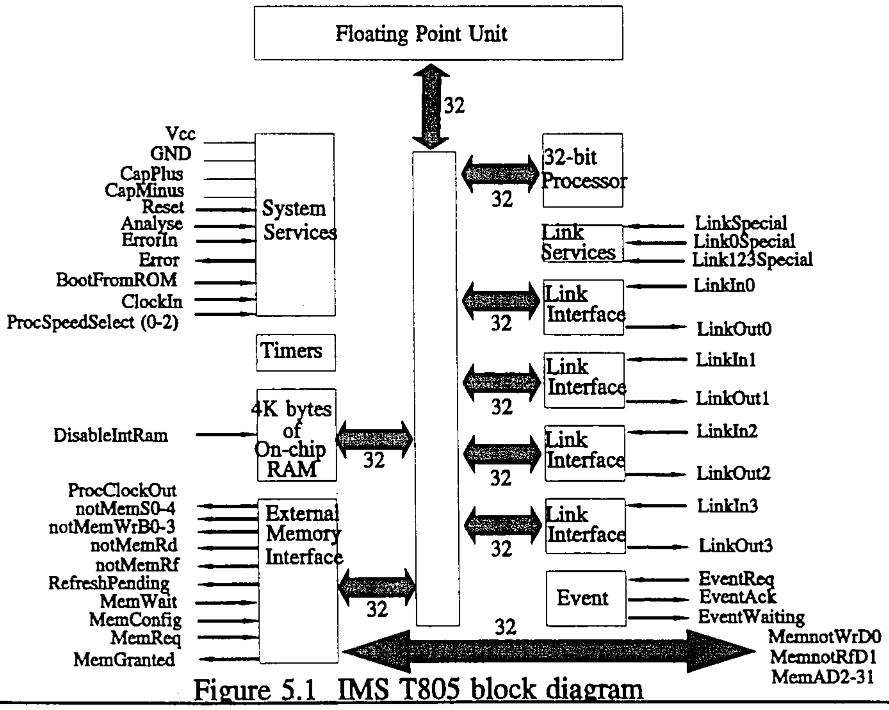


Figure 5.1 IMS T805 block diagram

Equipped to handle high performance arithmetic and floating point operations, the T805 is able to perform floating point arithmetic concurrently with the processor. This concurrency obtains a rate of 2.2 Mflops at a processor speed of 20 MHz and 3.3 Mflops at 30 MHz. Limited to a memory address space of 4 Gbytes, the 32-bit wide T805 memory interface uses multiplexed data and address lines to provide a data rate of up to 4 bytes every 100 nanoseconds (400 Mbytes/sec) for a 30 MHz processor speed [15].

The INMOS communication channel links also allow the T805 transputer to be configured in several multiprocessor network topologies via point-to-point connections. Each bi-directional link between T805 transputers can operate at a selectable rate of 5, 10, or 20 Mbits/sec.

Although Occam is the main language utilized on the T805 transputer, other high level languages such as C, and FORTRAN are supported [3,15].

5.3 The SuperSetPlus.64 system

Equipped with programmable link switches, the SuperSetPlus.64 system consists of 64 T805 transputers operating at a speed of 20 MHz. Memory is implemented by a 4 Kbyte on-chip RAM and either 1 Mbyte or 4 Mbyte of external memory [5]. Multiprocessor networks on the SuperSetPlus.64 system are specifically created via the configuration of communication channel links of the system into the desired network's topology.

5.4 The Occam programming language

The Occam language is named after William of Occam, a fourteenth century

English scholar and philosopher. Developed by David May at INMOS, England, Occam is a parallel programming language that supports explicit hardware concurrency and is based on C.A.R. Hoare's Communicating Sequential Processes (CSP). Occam is intended for use by transputer-based systems and is considered to be the assembly language of the transputer.

Initially, Occam was restricted to single data (integer) types and could not handle floating point operations (Occam 1). However, the latest version, Occam 2, is designed to handle features such as floating point representations, block memory transfers on a single transputer, and block memory transfers via a link from one transputer's memory to another. Occam 2 also supports mixed language programming which allows module processes of different languages to be re-used as part of an overall program [3].

Each process in Occam may communicate concurrently with another process via channels. This communicating concurrency of processes is exactly what was needed for transputer-based systems. Applications of several independent processes can be mapped onto the transputer-based system and their communication is achieved through the communication channel links of the system.

An Occam based program can be fully developed and tested on a single transputer before being implemented on a network of transputers. The two main advantages of Occam are its parallel construct ***(PAR)***, and its ability to prioritize processes through its *PRI* construct. Processes may be executed in parallel by preceding them with the PAR construct, and prioritization is achieved by placing the PRI construct in front of the PAR construct.

In Occam, parallel processes are separate entities and a process may only communicate with another via message passing. However, message passing in Occam utilizes the broadcast-and-wait technique where a process attempting to send a message to another must wait for the other process to acknowledge its readiness to receive the message. This technique is also implemented for an early process read. In this technique, a message is never lost within the system, however, processes are capable of waiting indefinitely (deadlocked) for another process [3].

Chapter 6 contains the sorting pseudo-code algorithms that are eventually implemented in the Occam 2 programming language.

CHAPTER 6

Implementing a Hybrid Sorting Algorithm on Multiprocessor Networks

6.1 Introduction

Currently, several versions of algorithms for the sorting algorithms discussed in chapter 4 exist. However, each of these algorithms can be generalized into standard pseudo-code form in order to further clarify the steps required for the entire sorting process.

6.2 Distributing and timing sorting algorithms

As the sorting algorithms are executed on multiprocessor networks where the root node is the only node connected to the host, there must exist an algorithm that efficiently broadcasts $M/2^n$ portions of the original list to each processor within the network. Prior to this action, timing of the sorting process must commence.

The global timer is implemented on the root node and only terminates when all the returned sorted lists have been merged into the final list. The algorithm to handle this portion of the tasks is expressed in the following modules.

```
proc_begin root_node_download_and_timer( Obtain list, distribute list, and time sort )
declare_int dimension;         (network's dimension)
declare_int root_node_id;      (root node id (i.e. if n=4, then root_node_id=0000))
declare_int counter, ptr;      (local vars. to keep track of distribution & loops)
declare_global_int M;          (number of items in partial lists)
declare_global_int n_nodes;    (number of nodes in the network)
declare_global_time timer;     (sorting timer)
declare_global_buf max_list;   (original list buffer)
        [set ptr = 0]
        [set n_nodes = $2^{dimension}$]
        [get max_list]
        [set M = sizeof(max_list)/n_nodes]
        [initialize timer]
        for_begin (counter = 0 to (n_nodes - 1))
            ptr  ←  (counter * M)
            [send max_list[ FROM ptr FOR M ] to PROCESSOR_counter]
        for_end
proc_end


proc_begin root_node_retrieve_timer( Retrieve partial lists and  shell sort merged list )
declare_int counter, ptr;      (local vars. to keep track of retrieval and nodes)
        [set ptr = 0]
        for_begin (counter = 0 to (n_nodes - 1))
            ptr  ←  (counter * M)
            [set max_list[ FROM ptr FOR M ] = list from PROCESSOR_counter]
        for_end
        proc_call shellsort( max_list )
        [stop timer]
        [report results]
proc_end
```

6.3 Implementing the bubble sort algorithm

As stated in chapter 4, each processor is responsible for sorting its portion of the list. These lists are distributed to each processor which in turn must obtain and sort its partial list. The partial lists are sorted utilizing the bubble sort algorithm which is expressed as follows:

```
declare_global_int nitems;          (number of items in partial list)
declare_global_buf partial_list;    (partial list buffer)

        proc_begin bubblesort( partial_list )
        declare_int i, j;           (local tracking vars.)
        declare_int temp;           (temp. storage var.)
            for_begin (i = 0 to (nitems - 1))               (number of passes)
                for_begin (j = 0 to ((nitems - 1) - i))     (number of comparisons)
                    if_begin (partial_list[j] > partial_list[j+1]) then
```

```
                        temp ← partial_list[j]
                        partial_list[j] ← partial_list[j+1]
                        partial_list[j+1] ← temp
                    if_end
                for_end
            for_end
    proc_end

    main_begin
            [obtain partial_list from comm. channel]
            proc_call bubblesort( partial_list )
            [return sorted partial_list to root_node over comm. channel]
    main_end
```

6.4 Implementing the shell sort algorithm

In chapter 4, the shell sort algorithm was shown to be well suited to handling the task of sorting partially sorted lists. The returned list from each processor, when merged, creates 2^n partially sorted lists where every $M/2^n$ locations in the list is the start of a partially sorted list.

The following pseudo-code extracted from *Naps at al. [19]* expresses the steps required for the effective execution of the shell sort algorithm. This algorithm is only executed on the root node after all nodes have returned their sorted partial lists.

```
proc_begin shellsort( max_list )
declare_int n_items;                    (number of items in original list)
declare_int i, j, k, temp;              (manipulative tracking vars.)
        [set i = n_items/n_nodes]       (partial lists pointer)
        while_begin (i ≥ 0)             (arrays start at location 0)
            j ← i
            repeat_begin
                j ← (j + 1)
                k ← (j - i)
                while_begin (k ≥ 0)
                    if_begin (max_list[k] > max_list[k+i]) then
```

```
                            temp ← max_list[k]
                            max_list[k] ← max_list[k+i]
                            max_list[k+i] ← temp
                            k ← (k - i)
                        else
                            k ← -1        (force loop termination)
                    if_end
                while_begin
            repeat_until_end (j = n_items)
            i ← (i / n_nodes)
        while_end
proc_end (bubblesort)
```

CHAPTER 7

Results and Conclusions

7.1 Introduction

Statements can almost always be made that justify the specific attribute of any multiprocessor network. However, if a statement insinuates a network is more efficient than another it must have substantial evidence to support its claim. In fact, the task of proving a particular network is more efficient than another is extremely arduous. The complexity of such a task is increased when multiprocessor networks designed for a specific class of applications are compared. This class of application specific networks tend to achieve the maximum performance available for their class of applications as opposed to other general-purpose networks.

This chapter presents the differences, results, and conclusions obtained from the study of the hypercube and binary de Bruijn multiprocessor networks.

7.2 Attribute differences

As stated earlier, several statements can be made to support each network's attributes, however, to avoid the risk of counter-statements that prove or dis-prove the efficiency of one network over another, the differences between the hypercube and binary de Bruijn networks are listed in table 7.1. This table is constructed in part from the previous chapters, *Ganesan et al. in [1],* and *Samantham et al. in [17].*

	Hypercube Network	**Binary de Bruijn Network**
Number of nodes	2^n	2^n
Node degree	n (subject to network size)	*4* (constant)
Network diameter	n	n
Fault tolerant	*yes* (up to $n - 1$ nodes)	*yes* (only 1 node)
Routing under node/link faults	*yes*	*yes*
Longest path length w/single fault	$n + 1$ (max.)	$n + 4$ (max.)
Easy detours around faults	*no*	*yes*
Extensibility	*difficult*	*easy*
Mapping/absorption		
Binary tree network	*difficult* (loss of 2^n-2^{n-1} nodes)	*easy* (loss of 1 node)
Shuffle-exchange network	*no*	*yes*
Mesh network	*yes* (even w/faults)	*no*
Linear array network	*yes*	*yes* (even w/faults)
Ring network	*yes*	*yes*

Table 7.1 Differences of the hypercube and binary de Bruijn networks

7.3 Implementing the networks

This section examines the two differences encountered in the implementation of the hypercube and binary de Bruijn networks. Of these two differences, the dynamic node degree of the hypercube for increasing and decreasing network sizes proved to be the most disturbing. The second difference, which was minor, was in that of the packet routing header protocols[1].

7.3.1 The effect of a dynamic node degree on network software

A major difference between the hypercube and binary de Bruijn network is the node degree attribute of each network. The impact of a network's node degree on that network's software is significant in the case of tearing and increasing the network's size. If the node degree attribute for a network is not constant in nature, an alteration to the network's size necessitates a major alteration throughout the network. (Recall that the node degree of a hypercube network is dynamic in nature as opposed to the static node degree of the binary de Bruijn network.)

In this study, increasing or decreasing the size of the binary de Bruijn network required no extensive modifications to the network software. Apart from the required physical network alterations and notifying the root node of its new dimension, the software for the binary de Bruijn network remained unaltered.

However, porting the software of the hypercube network proved to be a daunting task. After the usual physical network re-configuration and the root node dimension notification process, the network software had to be adapted for cases in which the network was extended or torn. This task although not complex in nature, proves to be

[1] Occam 2 Source code for both networks is archived in the EECS department of Lehigh University located in Packard Laboratory. (Rm. 304)

cumbersome and is required for all modifications to the dimension of the hypercube. In fact, two additional channels, for the transmission and reception of packets, were required each time the dimension of the hypercube was increased by a factor of one. These additional channels must be reflected in all cases throughout the hypercube's communication channel software modules.

7.3.2 Packet routing header protocols

To implement successful node-to-node communications on each network, a packet, consisting of the partitioned list of numbers to be sorted, is required to possess its destination node id, origin node id, and list size. This requirement, however, is not sufficient in the case of the binary de Bruijn network, as each intermediate network node needs to maintain a pointer to the next bit of the destination id that will be absorbed. (Recall that in chapter 3, a message route from node $x_{n-1},x_{n-2},\ldots, x_0$ to node $y_{n-1},y_{n-2},\ldots,y_0$ on a binary de Bruijn network is as follows: $(x_{n-1},x_{n-2},\ldots,x_1,x_0)$, $(x_{n-2},x_{n-3},\ldots,x_0,y_{n-1})$, $(x_{n-3},x_{n-4},\ldots,y_{n-1},y_{n-2})$, $(x_0,y_{n-1},\ldots,y_2,y_1)$, $(y_{n-1},y_{n-2},\ldots,y_1,y_0)$.) It should be apparent that intermediate nodes within the de Bruijn network require a knowledge of the next y_m bit to be absorbed from the destination node id, $where \quad 0 \le m < n-1$. (see figure 7.1)

The routing header for the de Bruijn network for a node-to-node communication is shown in figure 7.1. In figure 7.1.a, 7.1.b, 7.1.c, the routing header of a packet is modified at each intermediate node to point to the next y_m bit position to be absorbed by the next intermediate node.

Routing headers of packets in hypercube network's are not required to maintain a pointer to the next bit for intermediate nodes, instead, intermediate nodes simply scan

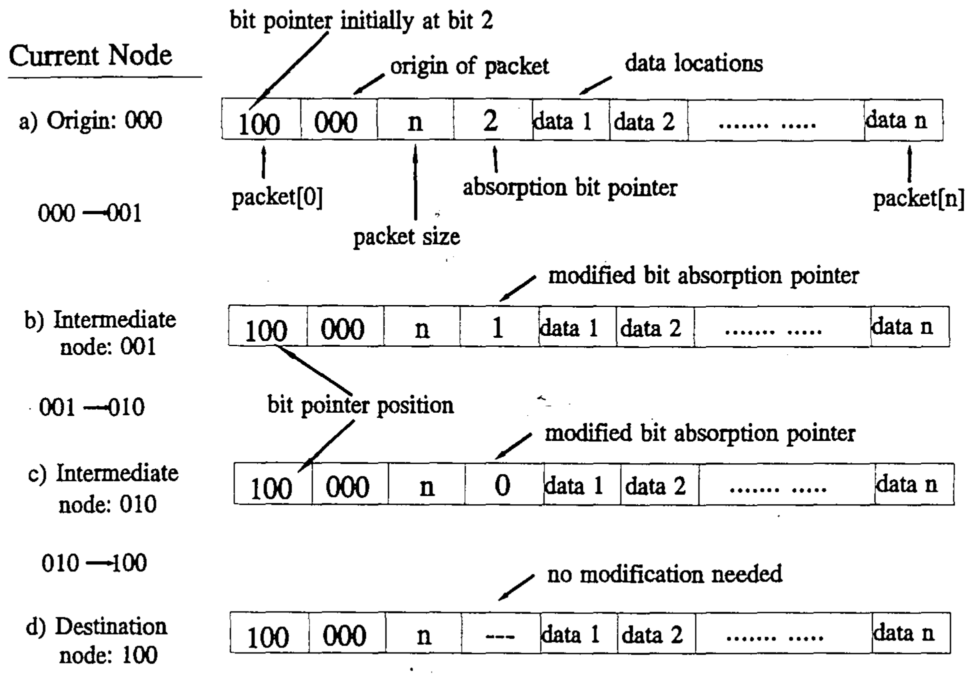


Figure 7.1 Routing a packet in the de Bruijn network

Steps for modification of a packet header from node 000 to node 100

from bit *0* to bit *n-1* of the destination node. When a differing bit is found the packet is routed to the node that differs in its bit position.

7.4 Results of the of the hybrid sort

The results obtained for the execution times of the hybrid sorting algorithm discussed in chapter 4 of this study are split into two classes: i) the broadcast-return category, and ii) the elapsed sort time category.

The broadcast-return category deals with the time elapsed for the distribution, partial sort and return of the segmented sorted lists. At this point no attempt is made to merge the sorted segmented lists. The elapsed sort time category progresses a stage further than the broadcast-return category by performing a shell sort on the merged but still partially sorted list. (see table 7.2)

The decrease in broadcast-return times for increasing nodes of each network can be attributed to the decreasing packet size. This elapsed time decreases due to the decrease in packet sizes, and the decrease in the number of comparisons each sorting node must perform.

	Execution times in seconds				
	list_sizes*				
	1K	**2K**	**4K**	**8K**	**16K**
Network					
Single node	2.612	10.276	41.463	165.496	663.623
dB(2,3)	[.192]	[.289]	[.691]	[2.329]	[8.881]
	1.710	6.073	24.400	96.376	387.624
3-cube	[.165]	[.290]	[.695]	[2.316]	[8.881]
	1.684	6.074	24.403	96.362	387.624
dB(2,4)	[1.112[f]]	[1.180[f]]	[1.333[f]]	[1.748[f]]	[3.377[f]]
	2.735[f]	7.379[f]	26.672[f]	102.522[f]	408.763[f]
4-cube	[1.107[f]]	[1.163[f]]	[1.335[f]]	[1.767[f]]	[3.399[f]]
	2.730[f]	7.362[f]	26.673[f]	102.541[f]	408.784[f]

[.192] - indicates achieved broadcast-return time
* - List sizes are in powers of 2. (i.e. 1K = 1024 items)
[f] - Times obtained for networks of 16 nodes include a one second delay in local processing nodes to allow the initial broadcast lists to be distributed without channel contention. This delay was necessary to prevent deadlock in the root node as the distribution process assumes that partially sorted lists will only be returned after all distribution is complete.

Table 7.2 Results of the hybrid sorting algorithm

The results of the hybrid sort on increasing multiprocessor network sizes is easier to comprehend in the form of a graph. In some cases, the elapsed sort times acquired for the multiprocessor networks are identical and plots are therefore difficult to disassociate from one another. (see figure 7.2)

It can be concluded that the increase in overall sorting times of 16 node networks is due to the nature of the shell sort. For example, the effect of splitting the list into 2^n segments results in an increase in the number of comparisons needed to obtain the lowest number of each segment in the merge process. (see chapter 4)

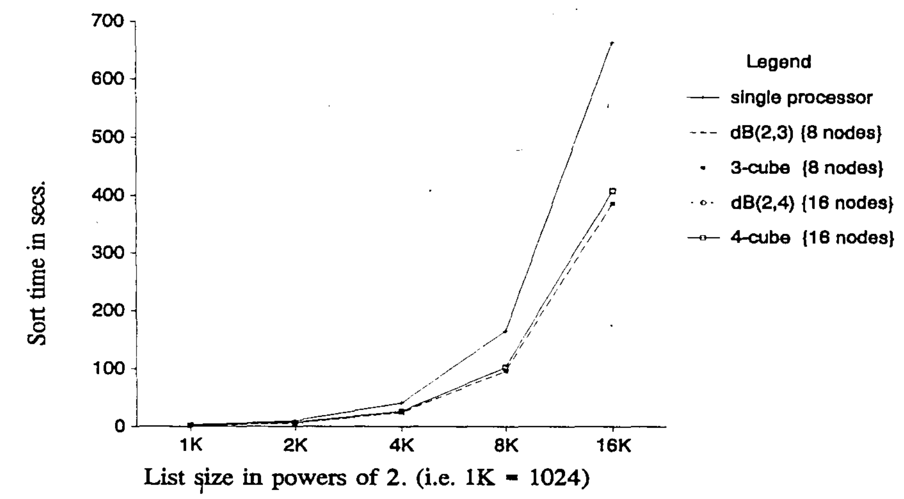


Figure 7.2 Plot of hybrid sort vs. bubble sort

It must be noted that the hybrid sort implemented on both networks is not communication intensive in nature. In other words, node-to-node communication did not occur extensively for either network. It is suspected that in the case of communication intensive classes of applications the hypercube network will triumph over the de Bruijn network. This statement is based on the broadcast-exchange bounds presented in chapters 2 and 3. However, for the hybrid sorting class of applications no significant timing differences were recorded between the hypercube and binary de bruijn network. In fact, the times are almost practically identical in value.

It is realized that from this portion of the study, multiprocessor networks do in fact triumph over a single processor network due to their concurrent nature. However, it must also be stated that if the hybrid sort times for the increasing network sizes continues to increase at its present rate, then it must also be assumed that the hybrid sort time will eventually surpass that of the single node sort for greater network sizes. This assumption implies that the hybrid sort is only effective for limited sizes of multiprocessor networks.

Finally, although the performance of the multiprocessor networks proved to be better than that of the single processor system, it must be realized that the efficiency of this hybrid sort is rather poor.

7.5 Conclusion

In general, several factors must be examined in an attempt to present a valid comparison between different multiprocessor networks. Some of these factors can be identified as the design and implementation complexity, absorption capability, and finally, the software development and hardware manufacturing cost of each network.

It is apparent that the design and implementation of both networks is simple in nature. However, significant differences exist in both their hardware and software development costs as well as their absorption properties. In this study, the cost factor of the hypercube network tends to be its downfall simply because expanding the network requires the re-design and alteration of both the network's hardware and software communication interfaces. This process is undesirable and may even lead to the introduction of errors into the network.

The absorption properties of the hypercube and binary de Bruijn networks are extensive in nature. However, this class of property can not be used as a comparison factor because some networks absorbed by the hypercube can not be absorbed by the binary de Bruijn network and vice-versa.

It has been ascertained that the performance of the binary de Bruijn network in this class of hybrid sorting applications is similar to that of the hypercube. Although the hypercube network is expected to surpass the binary de Bruijn network in communication intensive sorting applications, this still remains to be proven.

This study has proven that the binary de Bruijn network possesses network properties similar to that of the hypercube. In fact, in some cases the binary de Bruijn network's properties surpass that of the hypercube. This was evident in the binary de Bruijn network's ability to alter its network size without the re-design of nodes or software modules. In this regard, the binary de Bruijn network must be considered an adaptive, low-cost alternative to the hypercube for this particular class of sorting application.

REFERENCES

[1] E. Ganesan and D.K. Pradhan, *The hyper-de Bruijn networks: scalable versatile architecture,* IEEE Trans. Parallel Distrib. Systems, vol. 4, no. 9, pp 962-978, 1993.

[2] P. Gaughan and S. Yalamanchili, *Adaptive routing protocols for hypercube interconnection networks,* Computer, vol. 26, no. 5, pp. 12-23, 1993.

[3] T. Hazra, *Programming transputers,* IEEE Potentials, vol. 12, no. 3, pp. 47-49, 1993.

[4] D. Bertsek, C. Ozveren, G. Stamoulis, P. Tseng and J. Tsitsiklis, *Optimal communication algorithms for hypercubes,* J. Parallel Distrib. Comput., vol. 11, pp. 263-275, 1991.

[5] Computer System Architects, *SuperSetPlus.64: A multi-transputer, multi-user, processor,* SuperSetPlus.64 Tech. Manual, pp. 122-181, 1991.

[6] Prasad, V.V.R. and C. Murthy, *Downloading node programs/data into hypercubes,* Parallel Comput., vol. 17, pp.633-642, 1991.

[7] S. K. Das, N. Deo and S. Prasad, *Parallel graph algorithms for hypercube computers,* Parallel Comput., vol. 13, pp. 143-158, 1990.

[8] Z. Liu, *Optimal routing in the de Bruijn networks,* IEEE Int'l Conference on Parallel Processing, pp. 537-544, 1990.

[9] H. Stone, *High performance computer architecture,* Addison-Wesley Pub. Co., second edition, pp. 122-358, 1990.

[10] Q.F. Stout and B. Wagar, *Intensive hypercube communication,* J. Parallel Distrib. Comput., vol. 10, pp. 167-181, 1990.

[11] S. Akers and B. Krishnamurthy, *A group-theoretic model for symmetric interconnection networks,* IEEE Trans. Comput., vol. 38, no. 4, pp. 555-566, 1989.

[12] G. Almasi and A Gottlieb, *Highly parallel computing,* Benjamin/Cummings Publ. Co., pp. 279-411, 1989.

[13] J. Bermond and C. Peyrat, *de Bruijn and Kautz networks: a competitor for the hypercube?,* Hypercube and Distrib. Comput., pp. 279-293, 1989.

[14] G. Fox, A Ho, P. Messina and T. Cole, *Hands-on parallel processing,* Byte, October, pp.287-293, 1989.

[15] INMOS, *The transputer databook,* INMOS Tech. Manual, second edition, pp. 1-78, 1989.

[16] S. Johnson and C. Ho, *Optimum broadcasting and personalized communication in hypercubes,* IEEE Trans. Comput., vol. 38, no. 9, pp. 1249-1268, 1989.

[17] M. Samantham and D. Pradhan, *The de Bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI,* IEEE Trans. Comput., vol. 38, no. 4, pp. 567-581, 1989.

[18] G. Fox, N. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving problems on concurrent processors,* vol. 1, pp. 327-348.

[19] T. Naps and B. Singh, *Introduction to data structures with Pascal,* West Publ. Co., pp. 286-313, 1986.

[20] D. E. Knuth, *The art of computer programming,* Addison-Wesley Publ. Co., vol. 3, 1973.

BIOGRAPHY

Ato Y. Arkaah was born in Accra, Ghana, in 1970. He received the B.S. degree in computer science from Morehouse College, Atlanta, Georgia, in 1991. From 1990 to 1993, he has been a member of technical staff (MTS) at AT&T Bell Laboratories (Network Systems Organization), in Liberty Corner, New Jersey. During these periods, he has worked as both a hardware and software specialist for the development and maintenance of telecommunication network systems. He is a member of Phi Beta Kappa (Delta of Georgia), and the student branch of the IEEE.